User's Manual

Guide for simulating phase contrast imaging setups and general understanding of the PEPI adaptation of Geant4 software

Authors
David López Osorio
Isabela Ávila Restrepo
Juan José Guzmán Mejía
Thomas Andrade Hernández

Contents

1 Introduction

This manual is intended not only for physics students at the Universidad de los Andes, but also for anyone interested in X-ray phase contrast simulations and in learning about aspects of high-energy physics—particularly the interaction of X-rays with matter. It covers the basic usage and configuration of Geant4, a C++ simulation platform designed to accurately describe various types of interactions—such as electromagnetic, weak, and strong—between particles and matter. Within this context, the manual presents applications of the program for simulating angiographic and mammographic scenarios, which can be adjusted through various parameters. We will explore how to modify a simplified version of a mammographic phantom, including the materials used in the setup, the distances between components, their geometry, and other settings in order to simulate specific cases.

To facilitate this process, a GitHub repository named Users_Manual was created. In this repository, you will find several folders, each with a specific purpose. The one that is fully explained is the `Phantom_Source` folder, which contains all of the code related to the simulation of the mammographic scenario, while the `Tube_Source` folder contains the model of a single PMMA tube, which serves as an example object that you can study independently if interested and may be understood as a simple scenario of the angiographic scenario. The code was developed using the source of the PEPI (Photon-counting Edge-illumination Phase-contrast Imaging) project [1], created at the INFN laboratories in Trieste, Italy, specifically for phase contrast imaging (PCI) simulations. Throughout this manual, we will cover the fundamentals of Geant4 and how to apply them within the structure of PEPI.

2 Getting Access to the Geant4 Framework:

The following chapter outlines the main ways to access the Geant4 framework and its associated tools. First, we will consider the scenario in which you are a student at Universidad de los Andes and have requested a server account for the institutional workstation, or if your institution provides access to a workstation where Geant4 is already installed. For users who are not on Unix/Linux distributions—commonly known as distros—we will explain how to install and use MobaXterm, a tool that allows Windows users to access Unix/Linux environments and perform remote administration and development tasks through a unified interface.

The second method is the use of a virtual machine (VM) provided and certified by the Geant4 development team.

The third method consists of a manual installation on an existing Unix/Linux system, either native or through a VM. This method is not explained in detail due to the inherent complexity arising from the large number of available distros. However, we will provide useful tools and links to existing tutorials in case you decide to try it.

That said, we recommend starting with MobaXterm if your institution has a workstation available, as it offers an accessible and user-friendly solution for those unfamiliar with the Unix/Linux environment. It is also possible to install Geant4 directly in a Windows environment, though this is not recommended due to several limitations compared to the original implementation.

2.1 MobaXterm

2.1.1 Downloading the Software:

To begin the download process, simply go to the official MobaXterm website. Click on "GET MOBAXTERM NOW!" and select the "Home Edition." On the next screen, it is important to download the "Portable edition" by clicking the blue button shown in 1.

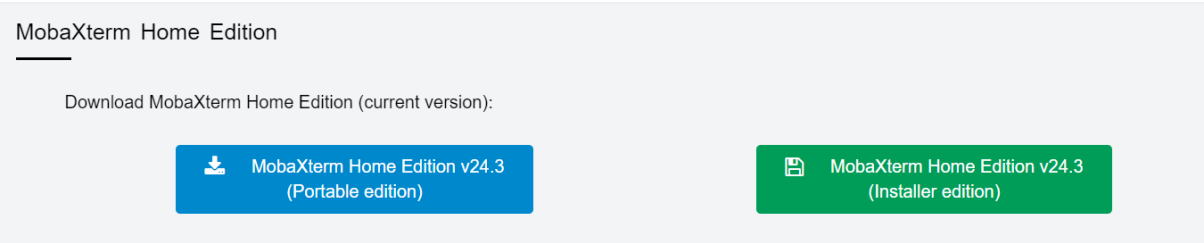

Figure 1: MobaXterm Home Edition download screen.

After completing these steps, a .zip file will be downloaded. Simply extract its contents, one of which is the MobaXterm application.

2.1.2 Logging into the Labhep Server:

At Universidad de Los Andes, we have access to a powerful server called Labhep, where all Geant4 simulations are to be run. Given this, we will see how to log in from the MobaXterm interface. First, users must request access to the server from the systems engineer, who will provide them with a username and password. There are two possible accounts that can be assigned: **labhep1** and **labhep**. If granted access to labhep1, the user only needs to follow the instructions below. However, if given a labhep account, additional steps are required, which will be mentioned at the end.

First, using a computer connected to any network, open MobaXterm and add a new session by clicking *Sessions* in the upper left corner. A pop-up window will appear; click **SSH** (Secure Shell), leading to the screen shown in figure 2. Here, you need to enter the *host* name. For labhep1, enter **labhep1.uniandes.edu.co**, and for labhep, enter **labhep.uniandes.edu.co**. Additionally, check *Specify username* and enter the assigned username. After these steps, click *OK*, and the program will prompt you for your password. Upon entering it correctly, another pop-up will appear asking you to create a *master password*; choose a password of your preference. Once done, you will have permanent access to the server.

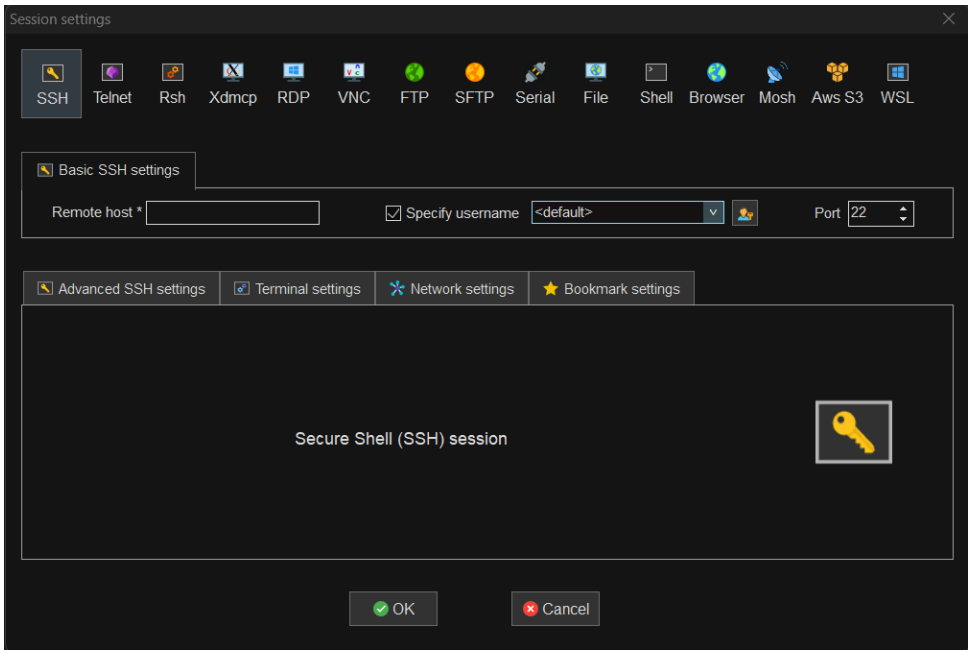

Figure 2: Pop-up in MobaXterm for adding a server.

As a result, users with access to **labhep1** will now be able to enter the server from any network. However, those with labhep accounts will only be able to log in while connected to the university network (Seneca). To access **labhep** and modify code or run simulations from any location, users will need to request access to a bridge server called **hep-server1**. This requires an additional account, which is different from the labhep account.

To remotely connect via **hep-server1**, follow these steps:

1. Repeat the previous steps but using the host name **hep-server1.uniandes.edu.co**.

2. At the **hep-server1** console, log into **labhep**:

   ```
   ssh -Y username@labhep.uniandes.edu.co
   ```

3. Enter the password provided for **labhep**.

With this, users will be able to modify code and run simulations on **labhep**. However, note that downloading files from **Labhep**, nor uploading them to a local computer via MobaXterm is possible, unless the device is connected to the university network if you are using **labhep**.

Warning!!

- Do not enter the wrong password more than three times, as this will block access to the server for everyone.
- The *master password* **does not** replace the password received when requesting the account. This is a kind of configuration exclusive of MobaXterm.

2.1.3 Preparing the Geant4 Framework:

After installing MobaXterm and logging into your account, the Geant4 package may not be available for immediate use. To gain access to the Geant4 framework, request the installation path from the workstation manager. If you already know the specified path, open the `.bashrc` file and add the path preceded by the `source` command to ensure that the Geant4 package is available every time you access a terminal. In the case of students at Universidad de los Andes, we have two servers, and therefore two directories to configure if you want to work on either of them:

Steps

1. First, start editting the `.bashrc` file by using the command `nano ~/.bashrc`.
2. Next, go to the lower line of the file and add the following commands. If you are using **labhep1** you must add:

```
# source geant4
source /usr/local/GATE/geant4.10.06.p01-install/bin/geant4.sh
# source /usr/local/HEP/GEANT4/geant4-v11/geant4-v11-install/bin/geant4.sh
```

   If you are using **labhep** you must add:

```
# source geant4
source /usr/local/HEP/GATE/geant4.10.06.p01-install/bin/geant4.sh
# source /usr/local/HEP/GEANT4/geant4-v11/geant4-v11-install/bin/geant4.sh
```

3. Once added the lines, save (**Ctrl+S**) and close (**Ctrl+X**) the file.
4. To complete this process, write the command `source ~/.bashrc` to ensure that Geant4 is completely added to your user.

As you may see from the previous lines, there are available two versions of the Geant4 package `v10.06.p01` and `v11`. It is recommended to use the first one for the purposes of this guide.

2.2 Geant4 Virtual Machine

If you want to use Geant4 on your own machine, you can use a virtual machine (VM) to emulate a Linux-based system. This section follows the content of the First Steps with Geant4 guide, which was created by CERN to teach an introductory course on Geant4. All the details regarding the recommended machine specifications and the steps you need to follow can be found in the guide mentioned above. Here, we summarize the main steps:

1. First, it is important to have a reliable VM player. The recommended one is VMware Workstation, as it is among the most optimized. VMware has recently made the Pro version free for personal use. Therefore, we recommend using VMware Workstation Pro.

2. Next, you must download the prepared ISO image, which is available on the Geant4 Virtual Machine official page. The ISO is approximately 19 GB in size, as it includes several pre-installed tools such as Visual Studio Code and Jupyter, along with all the packages required to run Geant4 properly.

3. The final step is to open VMware Workstation Pro and load the previously downloaded ISO. To follow the content of this manual, simply open a terminal inside the VM and replicate the commands and procedures described here.

Naturally, several details have been omitted for simplicity. If you are interested in this method, we encourage you to consult the guide mentioned before.

2.3 Installing Geant4 Natively

As mentioned in the introductory paragraph of this chapter, the installation of this framework can be somewhat messy. Since it falls outside the scope of this manual—which focuses on a specific application of the framework—we would still like to offer some support if you choose to follow this route. The complete installation process, including instructions for Windows systems, can be found in the official Geant4 Installation Guide.

As this process can be overwhelming and not entirely clear, you might find it helpful to consult community-made tutorials. For example, an installation guide created by John Francis for Windows, and another by the Physics Matter channel for Linux, could provide additional guidance.

3 Linux Commands of Utility:

It is possible that you have no prior experience with Linux or any related operating system. In this section, we will introduce several essential commands that you need to understand to navigate freely through all the files we will be using.

3.1 Navigation Through Directories

Command `ls` (List Directory Contents):

The `ls` command is used to view the contents of a folder from the terminal, displaying all files and subdirectories.
Specific commands:

- `ls -l`: Displays file details (permissions, size, date, etc.).
- `ls -a`: Also shows hidden files (such as `.bashrc`).

Command `cd` (Change Directory):

The `cd` (short for `change directory`) command is used to change the current working directory in the terminal. In general, it is written as:

```
cd /path/to/your/project
```

where `/path/to/your/project` is the path of the directory you want to navigate to. It can be either an absolute path (starting from the root directory `/`) or a relative path (relative to the current directory).

Example: Suppose you have a project called `Phantom_Source` and you want to enter its build folder. Simply execute:

```
cd Phantom_Source
```

and then enter the build folder with:

```
cd build
```

Specific commands:

- `cd ..`: Moves to the parent directory of the current directory.
- `cd -`: Returns to the last directory you were working in.
- `cd` : Takes you to the user's home directory.
- `cd /`: Moves to the system's root directory.

Additional notes: The `cd` command does not produce any output if executed correctly. If you enter an invalid path, you will get an error message such as:

```
bash: cd: /invalid/path: No such file or directory
```

3.2 File Editing

Command `nano`:

The `nano` command is a simple text editor that runs directly in the terminal, allowing you to create and modify files.

Example: To open a file named `file.txt` in the `nano` editor, use:

```
nano file.txt
```

If you want to create a new file called `new.txt`, simply execute:

```
nano new.txt
```

Useful keyboard shortcuts: While using the `nano` editor, you can use the following shortcuts:

- `Ctrl + S`: Saves the changes to the file (`Write Out`).
- `Ctrl + X`: Closes the editor. If there are unsaved changes, `nano` will ask if you want to save them.
- `Ctrl + K`: Cuts a line of text.
- `Ctrl + U`: Pastes a previously cut line of text.
- `Ctrl + W`: Searches for text in the file (`Where Is`).
- `Ctrl + G`: Displays `nano`'s help menu.
- `Ctrl + C`: Shows the current cursor position (line and column).

3.3 General Process Management

Command `htop`:

The `htop` command is a visual and interactive tool for monitoring running processes and system resource usage. In this case, you can see the 32 cores used in the university server and the simulations that are running. If you want to check the created simulations, locate them using the uniandes username.

`htop` Interface:

- **Top bar:** Displays CPU, memory, and `swap` usage, along with system information such as processor load and uptime.
- **Process list:** Shows running processes with details like PID, CPU and memory usage percentage, runtime, and associated command.
- **Bottom bar:** Displays available keyboard shortcuts to interact with the tool.

Useful keyboard shortcuts:

- `F1`: Displays help.

- `F2`: Opens the settings menu.
- `F3`: Allows searching for a process by name.
- `F4`: Filters processes based on specific criteria.
- `F5`: Displays the process hierarchy.
- `F9`: Terminates a selected process.
- `F10`: Exits `htop`.

3.4 Searching Precise Information in Files

Command `grep`:

The `grep` command allows searching for lines in a file; this is especially useful if you are looking for a specific function or name.

Example: To search for all lines containing the word `Physics` in a file named `log.txt`, use:

```
grep Physics log.txt
```

Specific commands:

- `grep -i`: Ignores case sensitivity during the search.
- `grep -n`: Displays the line number where the pattern appears.
- `grep -v`: Displays lines that **do not** match the pattern.
- `grep -r`: Searches recursively in subdirectories.

4 World Editting:

4.1 General Understanding

Before modifying the simulation world, it is important to conduct a general review of the available files. Upon opening the `Phantom_Source` directory, we find five folders along with several other files. The user will notice that, except for the `README` text file, these files outside the folders are the same as those found within the build directory; later, we will see how this affects the code editing process. For now, we will focus on the contents of each folder separately (some files will not be mentioned as they are not intended to be edited by the user).

/build
The `build` directory contains everything necessary to execute the simulation. It serves as a redundancy barrier and includes final configurations for the beam, geometry, detection process, and parameters required to run the code. It is also where the files needed to compile the code are located. Let's briefly explore the relevant files:

- `detmaskconfig.in`: This file establishes a redundancy barrier to ensure the simulator recognizes the desired distances. It also enables and disables the use of the mask, as well as modifying its thickness and aperture. Additionally, it provides the option to generate up to two additional output files with different lower energy thresholds.

- `gps.in`: This file allows modifications to the nature of the beam, including its shape, position, opening angle, particle composition, and other parameters. To adjust the size of the focal spot, a sigma value must be set based on its Gaussian relation with the full width at half maximum (FWHM). It has been preconfigured so that the beam originates from the source and so that all other factors align with the setup at the Laboratorio de Altas Energías de la Universidad de los Andes.

- `scorers.mac`: This file is crucial for certain types of analyses, as it allows the creation of perfect detectors called primitive scorers. These scorers collect information regarding particle passage through the detection mesh. The topic of detection is covered in more detail in **Chapter 5**.

- `run.mac`: This file is responsible for running the entire simulation code. Modifying it allows the user to define the number of events, the source spectrum file, verbosity, and the names of the output files. Additionally, it is used to activate the meshes defined in scorers.mac.

/data This directory contains raw data files for the X-ray dispersion indices (δ) of different materials. As discussed in section 4.2, it is necessary to provide this information to the simulator when defining a new material. Additionally, this folder includes the `Energy.txt` file, which has a different purpose: it tells the simulator that the other files in this directory contain δ values for energies ranging from 1keV to 100keV, with increments of 1keV.

/spectra

In Geant4, it is not possible to directly specify the voltage applied to the X-ray source. Instead, the simulator receives a file in `run.mac` that contains the energy spectrum produced by the source being simulated. The spectra directory contains several such files. The spectra included in `Phantom_Source` correspond to a tungsten-anode source (the same as in the Laboratorio de Altas Energías de la Universidad de los Andes) at different voltages and without filters. If the user needs to apply a filter to the source or use a voltage not included in the spectra, they must provide a file with the appropriate spectrum (TASMICS is commonly used for this purpose).

/include
Those familiar with C++ programming will already understand the purpose of the include and src directories. In a C++ project, the include folder contains header files (`.hh`), which act as an index for the code, declaring the classes, functions, and variables used in the src directory. These files do not contain the full functional code but only the structure that will be implemented in the source files (`.cc`).

Due to the nature of the files in this directory, reviewing each one separately is unnecessary, as most users will not need to modify them. When working with Geant4, beginners should simply be aware that any changes made to functions or variable types in src must be reflected in the corresponding include files.

/src
This directory is arguably the most important, as it contains the source code that governs the simulation's world, relevant physics, and particle generation. Most files should not be modified by the user, as they ensure the correct functioning of the simulation, physics, and source behaviorswithout affecting the parameters of the angiography. We will briefly review the files relevant to the user and provide insight into the purpose of some files that should remain unmodified.

- `PepiDetectorConstruction.cc`: This is the file the user will work with most of the time. It contains the code responsible for creating and defining materials, setting the properties of the simulation world, defining object geometries, distances, detector properties, declaring objects for visualization, and other minor aspects.
- `PepiPrimaryGeneratorAction.cc`: Defines the initial generation of particles for each event, specifying the properties of the particles introduced into the detector. The provided code defines a `G4GeneralParticleSource`, a general particle source, which is then edited in `.../build/gps.in`. **This file should not be modified**.
- `PepiRunAction.cc`: Controls specific actions at the beginning and end of each "run" in a simulation. **This file should not be modified.**
- `PepiEventAction.cc`: Defines the specific actions to be executed at the start and end of each event in the simulation. An event, in the context of the `Phantom_Source` code, refers to an interaction between a particle and the solid detector. **This file should not be modified.**
- `PepiPhysicsList.cc`: Configures the physical properties (electromagnetic and X-ray processes) in the simulation. It defines the interaction processes by invoking

the *Livermore Physics* library for electromagnetic phenomena physics and calls `PepiPhysicsXrayRefraction.cc` to establish the physics behind X-ray refraction. Additionally, it provides cut settings for the particles produced by the source.

- `PepiPhysicsXrayRefraction.cc`: Creates the physics list for X-ray refraction in photons and is configured to receive the code from `PepiXrayRefraction.cc`. This allows gamma particles to specifically experience refraction. **This file should not be modified.**

- `PepiXrayRefraction.cc`: Contains the custom code describing the physics of X-ray refraction. It is then processed by `PepiPhysicsXrayRefraction.cc` and ultimately invoked in `PepiPhysicsList.cc`. **This file should not be modified.**

4.2 Materials

The code for creating and defining materials is located in `PepiDetectorConstruction.cc`, specifically starting at line 264, where the function `DefineMaterials()` begins. This function is responsible for handling the topic of this section.

The user will notice that multiple materials have already been predefined. These are the most commonly used in angiography at the Universidad de los Andes, including PMMA, aorta, blood, CdTe, silicon, air, and others. Thanks to this, most users will not need to add new materials or elements. However, for those who do, this chapter provides the necessary steps to properly integrate new materials into the code.

To add a new material or element correctly, several steps must be followed. In this section, we will go through each of them in order.

4.2.1 Summoning Materials and Elements from the NIST Catalogue

It may happen that the user is fortunate enough to find the required material in the Geant4 materials database. If this is the case, the initial invocation of the basic properties of the element, material, or compound is straightforward, as exemplified in Code 4.1, which is a fragment located at the beginning of the `DefineMaterials()` function in `PepiDetectorConstruction.cc`. This material's invocation follows the structure:

```
G4Material*"Name" = nist->FindOrBuildMaterial("Database_Name");
```

See the Geant4 materials database to check which materials, elements, and compounds are available.

Code 4.1

```
1  G4Material* CdTe      = nist->FindOrBuildMaterial("G4_CADMIUM_TELLURIDE");
2  G4Material* Air       = nist->FindOrBuildMaterial("G4_AIR");
3  G4Material* PlexiGlass = nist->FindOrBuildMaterial("G4_PLEXIGLASS");
4  G4Material* Water     = nist->FindOrBuildMaterial("G4_WATER");
5  G4Material* Nylon     = nist->FindOrBuildMaterial("G4_NYLON-6-6");
6  G4Material* PolyCarbonate = nist->FindOrBuildMaterial("G4_POLYCARBONATE");
7  G4Material* Silicon   = nist->FindOrBuildMaterial("G4_Si");
8  G4Material* Graphite  = nist->FindOrBuildMaterial("G4_GRAPHITE");
9  G4Material* Gold      = nist->FindOrBuildMaterial("G4_Au");
```

This method has its limitations. When invoking elements, they will have the density specified in the database, which cannot be modified. This means that isotopes cannot be invoked using this method. For materials and compounds, not only is the density unmodifiable, but also the mass fractions of the constituent elements cannot be edited.

It should be noted that the optical interaction of the material still needs to be specified, and this will be explained in the **Optical Properties** section.

4.2.2 Manual Implementation of Materials and Elements

If the desired material is not found in the database or its properties need to be modified, the material must be added manually. The process starts with the creation of the necessary elements, as shown in 4.2. In lines 1 and 2, class definitions are provided for the different parameters used in this section of the code: `G4double` for numerical values y `G4String` for text values. Once the parameters are defined, elements are created using the following syntax:

```
G4Element*"Name" = new G4Element("Name", "Symbol", Atomic_Number,
Molar_Mass);
```

As shown in Code 4.2, this method is used for elements such as nitrogen, oxygen, etc.

Code 4.2

```
1  G4double zN,aN,zO,aO,zH,aH,zC,aC, etc...;
2  G4String symbolN,nameN,symbolO,nameO,symbolH,nameH,symbolC,nameC, etc...;
3
4  aN = 14.01*g/mole;
5  G4Element* Nitrogen1 = new G4Element(nameN="Nitrogen",symbolN="N" , zN= 7.,
       aN);
6  aO = 16.00*g/mole;
7  G4Element* Oxygen1 = new G4Element(nameO="Oxygen" ,symbolO="O" , zO= 8., aO);
8  aH = 1.00*g/mole;
9  G4Element* Hydrogen1 = new G4Element(nameH="Hydrogen",symbolH="H" , zH= 1.,
       aH);
10 aC = 12.00*g/mole;
11 G4Element* Carbon1 = new G4Element(nameC="Carbon" ,symbolC="C" , zC= 6., aC);
```

Additionally, it is possible to add isotopes using this method. The code is simply modified to take the following form:

```
G4Isotope*"Name" = new G4Element("Name", Atomic_Number, Nucleon_Number);
```

With this in mind, we can see that creating more complex compounds or materials involves combining their constituent elements in the appropriate way. Geant4 offers two different methods for achieving this: (1) specifying the number of atoms per element or (2) defining the mass fraction of each element. An example of each method is illustrated in Code 4.3, where we use the elements previously defined in Code 4.2.

Code 4.3

```
1  G4double fractionmassA,densityA, DensityAl;
2  G4int ncomponentsA, ncomponentsAl,natomsAl;
3
```

```
4   //Air
5   densityA = 0.0013*g/cm3;
6   G4Material* AAir = new G4Material(nameA="AAir",densityA,ncomponentsA=4);
7   AAir ->AddElement(Carbon1, fractionmassA=0.0097*perCent);
8   AAir ->AddElement(Oxygen1, fractionmassA=23.1801*perCent);
9   AAir ->AddElement(Nitrogen1, fractionmassA=75.5301*perCent);
10  AAir ->AddElement(Argon1, fractionmassA=1.2801*perCent);
11
12  //Alumina
13  densityAl = 3.961*g/cm3;
14  G4Material* Alumina = new G4Material(nameAl="Alumina",densityAl,ncomponentsAl
        =2);
15  Alumina->AddElement(Aluminium1, natomsAl=2);
16  Alumina->AddElement(Oxygen1, natomsAl=3);
```

Starting from line 13, we observe that the first method is written as:

```
G4Material*"Name" = new G4Material("Element_Name", Density,
Component_Number, Matter_State, Temperature, Pressure);

"Name"->AddElement("Name", Mass_Fraction);
```

If it is necessary to specify the mass fraction of each element, the following code can be used, as seen from lines 5 to 11:

```
G4Material*"Name" = new G4Material("Element_Name", Density,
Component_Number, Matter_State, Temperature, Pressure);

"Name"->AddElement("Element_Name_1", Mass_Fraction_1);

"Name"->AddElement("Element_Name_2", Mass_Fraction_2);

...
```

It is evident that there is a certain difference between the general code just described and the one presented in Code 4.3, especifically the addition of the parameters `Matter_State`, `Temperature`, and `Pressure`. This is because these properties do not need to be explicitly specified. If they are not set, as is the case in the code, the temperature and pressure will take their normalized conditions: 293.15K and 1atm, respectively, while the state of matter will be solid or gaseous depending on the density.

In `PepiDetectorConstruction.cc`, the concepts discussed in this subsection can be observed starting around line 300, under the title **"Additional Materials"**.

4.2.3 Optical Properties

When creating a new material in Geant4, its properties are initially quite limited. One crucial missing property is the X-ray refractive index (n) at different energies, which is particularly important for angiography simulations. Therefore, it is necessary to provide the simulator with this information for the materials being used, and this is where the files in the `data` directory come into play.

Code 4.4

```
1  std::vector<double> AAirDelta = LoadDelta("../data/Air_delta.txt");
2  std::vector<double> AluminaDelta = LoadDelta("../data/Alumina_delta.txt");
3  std::vector<double> energies = LoadDelta("../data/Energy.txt");
```

The first step is to load the files that contain the dispersion data and the `Energy.txt`, which is straightforward using the code shown in Code 4.4. Here, we continue with the previous examples of `AAir` and `Alumina`. A data file for the specified material can be created using the delta coefficient (δ) for each energy value between 1 and 150 keV. These values can be obtained from resources such as the University of Melbourne and Berkeley Lab.

Next, we aim to create vectors for each material that will help calculate the refractive index. For this purpose, in Code 4.5, line 1 defines `NumEntries`, a variable representing the number of energy values in `Energy.txt` (100 in this case). Then, in lines 2 and 3, vectors are defined with the same number of elements as `NumEntries`, all initialized to zero by default.

With this setup, from lines 5 to 9, we can see how the refractive index information for each material is created and added to the corresponding `Rindex` vector, using the equation:

$$n = 1 - \delta + i\beta.$$

In the code presented in `Phantom_Source`, the value $\beta = 0$ is considered for all materials.

Code 4.5

```
1  G4int NumEntries = static_cast<int>(energies.size());
2  std::vector<double> AAirRindex(NumEntries);
3  std::vector<double> AluminaRindex(NumEntries);
4
5  for (G4int i = 0; i < NumEntries; ++i)
6  {
7    AAirRindex[i]           = 1 - AAirDelta[i];
8    AluminaRindex[i]        = 1 - AluminaDelta[i];
9  }
10
11 G4MaterialPropertiesTable* AAirMatPropTbl = new G4MaterialPropertiesTable();
12 AAirMatPropTbl->AddProperty("RINDEX",energies.data(),AAirRindex.data(),
       NumEntries);
13 AAir->SetMaterialPropertiesTable(AAirMatPropTbl);
14
15 G4MaterialPropertiesTable* AluminaMatPropTbl = new G4MaterialPropertiesTable
       ();
16 AluminaMatPropTbl->AddProperty("RINDEX",energies.data(),AluminaRindex.data(),
       NumEntries);
17 Alumina->SetMaterialPropertiesTable(AluminaMatPropTbl);
```

The process is finalized with what is observed from line 11 of Code 4.5. In this part of the code, we create a properties table for each material (`G4MaterialPropertiesTable`), as exemplified in lines 11 and 15. At the same time, we take the newly created refractive index data (`Rindex`) and add it to the corresponding material table, along with the energy data, under the key "RINDEX". With this, the materials are ready to be used.

4.2.4 Linking a Material with an Object

It is worth briefly mentioning where and how each object in the world is assigned a material. In the same file, `PepiDetectorConstruction.cc`, from line 724 to 736,

there is a section called **"Default Materials"**, seen in Code 4.6. Here, all objects that require material assignment are listed: `fWorldMaterial` is the material of the environment, `fIonMaterial` must always be the same as the previous one, `fDetectorMaterial` is the detector material, `fObject2Material` is the outer cylinder material of the vein phantom, and `fMuscleMaterial` is the inner cylinder material of the phantom. The remaining ones are not used in the base code of `Phantom_Source`, but they are included here so that the user can add more objects or enable the use of the mask if desired.

Code 4.6

```
1  // ========================================
2  //           DEFAULT MATERIALS
3  // ========================================
4
5  fWorldMaterial     = Vacuum;
6  fIonCMaterial      = Vacuum;
7  fDetectorMaterial = Silicon; // CdTe
8  fMaskMaterial      = Gold;
9  fSubMaterial       = Graphite;
10 fMuscleMaterial = PlexiGlass;
11 fMicroSphereMaterial = Alumina;
12 fMuscleMaterial1 = PlexiGlass;
13 fWaxInsertMaterial = Wax;
```

4.2.5 Important Considerations

- The material `CdTe`, although commented out in this example, can be easily activated for use in simulations requiring cadmium telluride detectors, which are often more effective.
- Each material must have the same name as the material definition of the object; otherwise, the code will not compile.

4.3 Geometry

The construction and definition of geometries is one of the fundamental aspects of working with Geant4, as it determines the volumes in which particles interact. The geometries are defined in the file `PepiDetectorConstruction.cc`, specifically in the `objects` function, which starts at line 1039 of this file.

The geometry consists of several predefined volumes that cover most necessary applications. In this section, we will explain how to create and modify existing geometries and how to add new volumes.

4.3.1 Mother Volume Definition

Geant4 geometry follows a hierarchy of nested volumes:

- **World Volume**: This is the mother volume, which contains all other volumes. It must be large enough to encompass the entire simulated environment.
- **Daughter Volume**: These are the volumes that represent physical objects within the `World Volume`, such as detectors or vein phantoms.

In the `DefineVolumes()` function, each volume is defined using the `G4Box`, `G4Sphere`, `G4Tubs` classes, among others, depending on the required geometry. Below is an example of how the world volume is defined.

World Volume Definition

The mother volume is where all other simulation volumes are placed. This volume is constructed as a non-rotated box (`G4Box`), located at the coordinate origin $(0, 0, 0)$.

Code 4.7

```
// ========================================
//                   WORLD
// ========================================

// - Build the WORLD as an unrotated Box in (0,0,0)

fWorldSolid = new G4Box("World",            // World's name.
                        fWorldSizeX/2,      // World's size.
                        fWorldSizeY/2,
                        fWorldSizeZ/2);

fWorldLogical = new G4LogicalVolume(fWorldSolid,      // Its solid.
                                    fWorldMaterial,   // Its material.
                                    "World");         // Its name.

fWorldPhysical =  new G4PVPlacement(0, // No rotation.
                    G4ThreeVector(),   // Positioning at (0,0,0).
                    fWorldLogical,     // Its logical volume.
                    "World",           // Its name.
                    0,                 // Its mother  volume.
                    false,             // No boolean operation.
                    0,                 // Copy number.
                    fCheckOverlaps);   // Checking Overlaps.
```

Code Description:

- `fWorldSolid`: Defines the solid volume as a box (`G4Box`) with dimensions specified by `fWorldSizeX`, `fWorldSizeY`, and `fWorldSizeZ`, which are divided by 2 because Geant4 defines volumes from their center.
- `fWorldLogical`: Assigns the material (`fWorldMaterial`) to the logical volume and gives it a name, in this case, (`"World"`).
- `fWorldPhysical`: Places the logical volume in space without rotation (angle `0`) and at the origin (`G4ThreeVector()`). An important detail is that it does not have a mother volume because it is the World, and it uses `fCheckOverlaps` to check for possible overlaps with other volumes.

Additional Notes:

- The material of the `World Volume` is defined in the materials section and, by default, is usually `Vacuum`, as this makes simulations faster.
- The size of the `World Volume` must be large enough to contain all other volumes.

4.3.2 Adding Sub-Volumes:

Here we can find several examples of 3-dimensional objects that you could build in the simulation's world. To see more examples of more sophisticated structures, you may find very useful the **Geometry** section of the general Geant4 Documentation. **Note: The lengths given to any of the following objects represent half of the full dimensions, so, in order to establish a certain length of 1m, you must write 0.5m.**

Cylinder

To include a new volume within the `World Volume`, you must first define its shape, its logical volume, and its position in space. In the case of a hollow cylinder, it can be defined as a `G4Tubs`, and in Geant4 code, it is defined as follows:

Code 4.8

```
G4double Rint = 3*cm;
G4double Rext = 5*cm;
G4double h = 10*cm;
cylinder = new G4Tubs("Cylinder",    // Its name.
                      Rint,          // Its intern radius.
                      Rext,          // Its extern radius.
                      h/2,           // Its half-height.
                      0,             // The start angle.
                      2*pi);         // The end angle. 2*pi
                                     // to create a full cylinder.
```

In this code, a hollow cylinder was defined with an inner radius `Rint`, an outer radius `Rext`, and a height `h`. If a solid cylinder is desired, `Rint` should be set to zero. After defining the shape of the cylinder, materials are assigned to a logical volume, and its position in the simulation space is set.

Box

A cube is defined using `G4Box`, a useful class for constructing solid barriers. The definition of a box is done by specifying its size in the three dimensions **X**, **Y**, and **Z**. Below is how the definition looks in Geant4 code:

Code 4.9

```
G4double a = 10*cm;
G4double b = 5*cm
G4double c = 15*cm
cube = new G4Box("Cube",     // Its name.
                 a/2,        // Its X half-lenght.
                 b/2,        // Its Y half-lenght.
                 c/2);       // Its Z half-lenght.
```

Sphere

The definition for a sphere in Geant4 is `G4Sphere`, which allows for the creation of either a solid or a hollow sphere. Additionally, the ϕ and θ angles can be specified in case a full sphere is not desired. This solid requires an inner radius and an outer radius as input. Below is the definition of the sphere:

Code 4.10

```
G4double r = 10*cm;                 // Sphere radius.
sphere = new G4Sphere("Sphere",     // Its name.
                      0,            // Its intern radius (0 for a solid sphere).
                      r,            // Its extern radius.
                      0,            // Start angle for phi.
                      2*pi,         // Finish angle for phi.
                      0,            // Start angle for theta.
                      pi);          // Finish angle for theta.
```

Definition in the Header File

As previously observed, the `include` folder contains the functions where the classes, functions, and variables to be used in the `src` directory are declared. Therefore, if an object needs to be modified or defined, the `PepiDetectorConstruction.hh` file in `include` must also be edited. In general, the class definition of the object is given by the following line of code:

```
G4(Solid_Definition)*"Code_Name";
```

`Phantom_Source` contains certain initial definitions that the user can modify. Some of them are presented in Code 4.11.

Code 4.11

```
G4Box*  fWorldSolid;
G4Trd*  fObject1Solid;
G4Sphere* fSphereSolid;
G4Box* fMuscleSolid;
G4Tubs* fSphere3Solid;
```

Application to the Phantom_Source Code

The `Phantom_Source` code was developed with the High Energy Physics Laboratory at Universidad de los Andes in mind. For this reason, the example solid chosen for the study is a simplified reconstruction of the CIRS015 Mammographic Accreditation Phantom, and schematic of this can be seen in Figure 3. While this model is a significantly simplified version of the original, it serves as a starting point for understanding how to create complex volumes within the simulation world.

This mammographic phantom consists of three main parts: a primary region made of PMMA, a secondary subregion filled with wax, and a third part containing objects embedded in the wax that mimic microcalcifications and foreign bodies. These microcalcifications were limited to four spheres with diameters of 250, 500, 750, and 1000 micrometers, made of alumina, just to keep this example model as simple as possible. You could modify it and add other type of structures.

In Code 4.12 we can see the definition of the first mentioned region as an example of adding sub-volumes to the simulation world. Here we can differentiate the following objects:

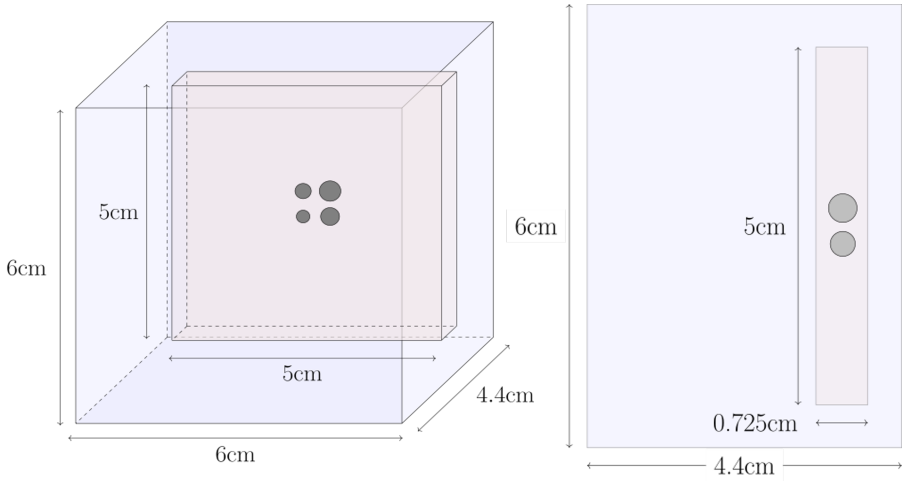

Figure 3: Schematic of the CIRS015 Mammographic Accredation Phantom. The dimensions were taken from its manual.

- **Solid Volume (`fMuscleSolid`)**:
  - Defined using the `G4Box` class, which represents a cube.
- **Logical Volume (`fMuscleLogical`)**:
  - Associated with the solid `fMuscleSolid`, it is important to use the same name that is being referenced.
  - Material used: `fMuscleMaterial`, which is PlexiGlass (PMMA).
- **Position (`objectPositionCube`)**:
  - Uses a translation vector defined as `fSourcePosZ + fSrcObjDistance`; if an object is to be placed in the same position, the same formula should be used.
- **Physical Volume (`fMusclePhysical`)**:
  - The logical volume is physically placed in the world with the previously specified parameters.
  - Additional parameters:
    - `fWorldLogical`: Defines the `World Volume` as the mother volume.
    - `fCheckOverlaps`: Checks for overlaps with other volumes.

Code 4.12

```
1  // Phantom Base:
2
3  // Muscle Mimic
4
5  fMuscleSolid = new G4Box("Cube", 6*cm/2, 6*cm/2, 4.4*cm/2);
6  fMuscleLogical = new G4LogicalVolume(fMuscleSolid, fMuscleMaterial, "CubeLV");
7  G4ThreeVector objectPositionCube = G4ThreeVector(0*mm, 0, fSourcePosZ+
       fSrcObjDistance);
8  fMusclePhysical = new G4PVPlacement(0, objectPositionCube, fMuscleLogical, "
       Cube", fWorldLogical, false, 0, fCheckOverlaps);
```

4.3.3 An Object inside Another

All volumes in Geant4 must be children of another volume, and the World Volume serves as the universal mother volume. If an object needs to be placed inside another, the mother volume must be specified accordingly in the parameter definition, specifically in `G4PVPlacement` function. If this is not changed, the volumes will overlap, potentially causing errors in the simulation. Note that the coordinate system is relative to the mother volume, meaning the position (0,0,0) refers to the center of the parent volume.

Application to the Phantom_Source Code

As mentioned earlier, the phantom includes several distinct regions. In this example, the wax and microcalcifications will be added. First, Code 4.13 shows how the wax region is created inside the PMMA volume. Notice it is slightly off from the center of the PMMA region.

Code 4.13

```
// Wax

fWaxInsertSolid = new G4Box("Wax", 5*cm/2, 5*cm/2, 7.25*mm/2);
fWaxInsertLogical = new G4LogicalVolume(fWaxInsertSolid, fWaxInsertMaterial, "
    WaxLV");
G4ThreeVector objectPositionWax = G4ThreeVector(0*mm, 0, 3.375*cm/2);
fWaxInsertPhysical = new G4PVPlacement(0, objectPositionWax, fWaxInsertLogical,
     "Wax", fMuscleLogical, false, 0, fCheckOverlaps);
```

This concept of a "mother volume" is not limited to the simulation world and its immediate sub-volumes; it's also possible to create nested volumes—volumes within volumes that themselves are inside other volumes. This is the case for the microcalcifications, which are daughter volumes of the wax region, as shown in Code 4.14:

Code 4.14

```
//Microcalcifications:

// Sphere
fMicroSphereSolid = new G4Sphere("Sphere", 0, 1000*um/2, 0, 2*pi, 0, pi);
fMicroSphereLogical = new G4LogicalVolume(fMicroSphereSolid,
    fMicroSphereMaterial, "SphereLV");
G4ThreeVector objectPositionSphere = G4ThreeVector(1, 1*mm, 0);
fMicroSpherePhysical = new G4PVPlacement(0, objectPositionSphere,
    fMicroSphereLogical, "Sphere", fWaxInsertLogical, false, 0, fCheckOverlaps);

// Sphere1

fMicroSphereSolid1 = new G4Sphere("Sphere1", 0, 750*um/2, 0, 2*pi, 0, pi);
fMicroSphereLogical1 = new G4LogicalVolume(fMicroSphereSolid1,
    fMicroSphereMaterial, "SphereLV1");
G4ThreeVector objectPositionSphere1 = G4ThreeVector(1, -1*mm, 0);
fMicroSpherePhysical1 = new G4PVPlacement(0, objectPositionSphere1,
    fMicroSphereLogical1, "Sphere1", fWaxInsertLogical, false, 0, fCheckOverlaps
    );

// Sphere2
```

```
17
18 fMicroSphereSolid2 = new G4Sphere("Sphere2", 0, 500*um/2, 0, 2*pi, 0, pi);
19 fMicroSphereLogical2 = new G4LogicalVolume(fMicroSphereSolid2,
       fMicroSphereMaterial, "SphereLV2");
20 G4ThreeVector objectPositionSphere2 = G4ThreeVector(-1*mm, -1*mm, 0);
21 fMicroSpherePhysical2 = new G4PVPlacement(0, objectPositionSphere2,
       fMicroSphereLogical2, "Sphere2", fWaxInsertLogical, false, 0, fCheckOverlaps
       );
22
23 // Sphere3
24
25 fMicroSphereSolid3 = new G4Sphere("Sphere3", 0, 250*um/2, 0, 2*pi, 0, pi);
26 fMicroSphereLogical3 = new G4LogicalVolume(fMicroSphereSolid3,
       fMicroSphereMaterial, "SphereLV3");
27 G4ThreeVector objectPositionSphere3 = G4ThreeVector(-1*mm, 1*mm, 0);
28 fMicroSpherePhysical3 = new G4PVPlacement(0, objectPositionSphere3,
       fMicroSphereLogical3, "Sphere3", fWaxInsertLogical, false, 0, fCheckOverlaps
       );
```

4.3.4 Important Considerations

If you wish to define more complex geometries, such as combined or boolean shapes, refer to the documentation on `G4BooleanSolid` in the Geant4 Documentation. These G4 shapes allow for the creation of volumes by combining other shapes through intersections, unions, and subtractions.

4.4 Physics of the Simulation

In Geant4, it is possible to describe the physics of particle interactions with matter through libraries, which are listed in the Physics Reference Manual. Below, the two most relevant physics models for angiography are described.

Livermore

All of the folders in the GitHub page use the Livermore physics model, which is designed to simulate particle interactions at both high and low energies. High-energy processes refer to particles with energies above 1 GeV. Since this is not the case for X-ray imaging, we will not explore it further.

For low energies, which are relevant to our simulations, a set of interaction models is used to simulate various physical processes occurring at low energies (typically below 1 GeV), such as the photoelectric effect, Compton scattering, Rayleigh scattering, gamma conversion, Bremsstrahlung, and ionization. Livermore is defined in the `PepiPhysicsList.cc` file as illustrated in Code 4.15.

Compared to high energies, the atomic shell structure is more important in most cases at low energies. Therefore, these processes make direct use of the cross-section data of the shells. The standard processes, which are optimized for high-energy physics applications, often rely on parameterizations of these data. **Code 4.15**

```
#include "G4EmLivermorePhysics.hh"
#include "G4VModularPhysicsList.hh"
#include "G4EmStandardPhysics.hh"

PepiPhysicsList::PepiPhysicsList()
: G4VModularPhysicsList()
{
  fMessenger = new PepiPhysicsListMessenger(this);

  SetVerboseLevel(-1);

  // Electromagnetic Physics
  fEmPhysicsList = new G4EmLivermorePhysics();
```

Penelope

Another model that specializes in low energies is Penelope. The difference lies in the fact that, instead of obtaining energy information from a database as Livermore does, it uses a Monte Carlo process to better model interaction events. These events are generated randomly according to the physical probabilities of the model. If you wish to use Penelope, you must import the appropriate library to enable its use.

The library that the user wishes to use must be specified in the `PepiPhysicsList.cc` file. Line 50 contains the function that defines the library in use, so if the physics model needs to be changed, this is where the name should be modified, but not before including the corresponding library in the code.

Code 4.16

```
#include "G4EmPenelopePhysics.hh"
#include "G4VModularPhysicsList.hh"
#include "G4EmStandardPhysics.hh"

PepiPhysicsList::PepiPhysicsList()
: G4VModularPhysicsList()
{
  fMessenger = new PepiPhysicsListMessenger(this);

  SetVerboseLevel(-1);

  // Electromagnetic Physics
  fEmPhysicsList = new G4EmPenelopePhysics();
```

4.5 Visualization

Visualization in Geant4 is an essential component for understanding and verifying all of the previously defined geometry. The visualization settings are managed in the `PepiDetectorConstruction.cc` file at line 971. This section controls how the geometries are rendered in PEPI's graphical viewer.

An example of how a visualization code appears is shown in 4.17.

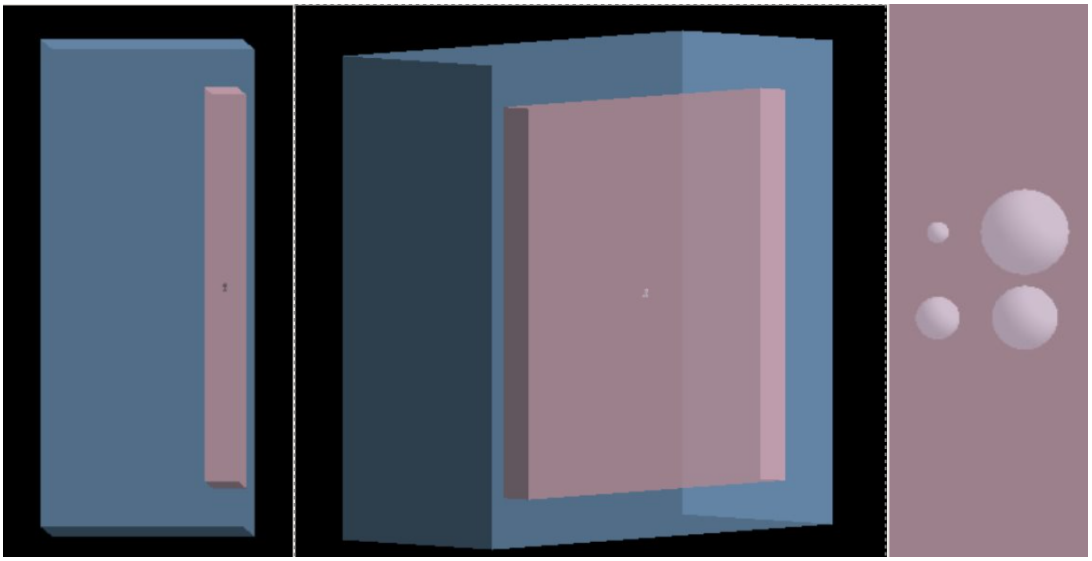

Figure 4: Phantom created in Geant4 using the CIRS015 model as a reference. The blue block corresponds to PMMA material, the pink block is made of wax and the spheres are the microcalcifications made of alumina.

Code 4.17

```
G4VisAttributes* visAttributes = new G4VisAttributes(G4Colour(1.0, 0.0, 0.0));
    // The defined colour in this case was red.
visAttributes->SetVisibility(true);          // Make the volume visible.
visAttributes->SetForceSolid(true);          // Show the volume as a solid.
fVolume->SetVisAttributes(visAttributes);    // Apply the attributes to the
   volume itself.
```

The color definition follows an RGB scale expressed in percentages, where 1.0 corresponds to 255. This is why the visualization attributes in Code 4.17 (Red = 1.0, Green = 0.0, Blue = 0.0) define a red object. An optional fourth parameter can be added to the G4Colour constructor to set the object's opacity. In `Phantom_Source`, the visualization section is divided into three parts: the visualization of the world, the PMMA and wax regions, and finally, the microcalcifications. Code 4.18 below shows how the visualization properties for each object are defined:

Code 4.18

```
// ========================================================================
//                      VISUALIZATION OF THE PHANTOM
// ========================================================================

  G4VisAttributes* cubeVisAtt = new G4VisAttributes(G4Colour(0.6,0.8,1.0,0.5));
  cubeVisAtt->SetForceSolid(true);
  fMuscleLogical->SetVisAttributes(cubeVisAtt);

  G4VisAttributes* waxVisAtt = new G4VisAttributes(G4Colour(1,0.75,0.79,0.5));
  waxVisAtt->SetForceSolid(true);
  fWaxInsertLogical->SetVisAttributes(waxVisAtt);

  // ------------------------------------------------------

```

```
15  G4VisAttributes* sphereVisAtt = new G4VisAttributes(G4Colour(1, 1, 1.0, 1.0))
        ;
16  sphereVisAtt->SetForceSolid(true);
17  fMicroSphereLogical->SetVisAttributes(sphereVisAtt);
18
19  // ------------------------------------------------------
20
21  G4VisAttributes* sphereVisAtt1 = new G4VisAttributes(G4Colour(1, 1, 1.0, 1.0)
        );
22  sphereVisAtt1->SetForceSolid(true);
23  fMicroSphereLogical1->SetVisAttributes(sphereVisAtt1);
24
25  // ------------------------------------------------------
26
27  G4VisAttributes* sphereVisAtt2 = new G4VisAttributes(G4Colour(1, 1, 1.0, 1.0)
        );
28  sphereVisAtt2->SetForceSolid(true);
29  fMicroSphereLogical2->SetVisAttributes(sphereVisAtt2);
30
31  // ------------------------------------------------------
32
33  G4VisAttributes* sphereVisAtt3 = new G4VisAttributes(G4Colour(1, 1, 1.0, 1.0)
        );
34  sphereVisAtt3->SetForceSolid(true);
35  fMicroSphereLogical3->SetVisAttributes(sphereVisAtt3);
```

If you followed the phantom construction steps, or if you simply used the `Phantom_Source` code available in the GitHub Repository linked at the end of this manual, you should see a structure like the one shown in Figure 4. Of course, the previous explanation omitted the need to include all of these bodies in the header file `PepiDetectorConstruction.hh`. It is essential to declare all the corresponding variables there, as shown in Code 4.19.

Code 4.19

```
1   G4Material* fMicroSphereMaterial;
2   G4Material* fWaxInsertMaterial;
3   G4Material* fMuscleMaterial1;
4   G4Sphere* fMicroSphereSolid;
5   G4Sphere* fMicroSphereSolid1;
6   G4Sphere* fMicroSphereSolid2;
7   G4Sphere* fMicroSphereSolid3;
8   G4Box* fMuscleSolid1;
9   G4Box* fWaxInsertSolid;
10  G4LogicalVolume* fMicroSphereLogical;
11  G4LogicalVolume* fMicroSphereLogical1;
12  G4LogicalVolume* fMicroSphereLogical2;
13  G4LogicalVolume* fMicroSphereLogical3;
14  G4LogicalVolume* fMuscleLogical1;
15  G4LogicalVolume* fWaxInsertLogical;
16  G4VPhysicalVolume* fMicroSpherePhysical;
17  G4VPhysicalVolume* fMicroSpherePhysical1;
18  G4VPhysicalVolume* fMicroSpherePhysical2;
19  G4VPhysicalVolume* fMicroSpherePhysical3;
20  G4VPhysicalVolume* fMusclePhysical1;
21  G4VPhysicalVolume* fWaxInsertPhysical;
```

Then they were initialized in the `PepiDetectorConstruction.cc` as seen in Code 4.20:

Code 4.20

```
    fMicroSphereMaterial(0),
    fWaxInsertMaterial(0),
    fMuscleMaterial1(0),
    fMicroSphereSolid(0),
    fMicroSphereSolid1(0),
    fMicroSphereSolid2(0),
    fMuscleSolid1(0),
    fWaxInsertSolid(0),
    fMicroSphereLogical(0),
    fMicroSphereLogical1(0),
    fMicroSphereLogical2(0),
    fMuscleLogical1(0),
    fWaxInsertLogical(0),
    fMicroSpherePhysical(0),
    fMicroSpherePhysical1(0),
    fMicroSpherePhysical2(0),
    fMusclePhysical1(0),
    fWaxInsertPhysical(0),
```

4.6 Preparing a Flat Field

A flat field (FF) is a reference image generated using the same parameters as the actual simulation with the phantom. In this way, the FF captures the background intensity, which is later used to normalize the pixel intensity on the detector.

Typically, a flat field simulation replicates the original setup but without the sample. By dividing the data matrix by the FF, the background noise is compensated for, resulting in a corrected image with improved intensity uniformity and visual quality. This process enhances contrast and resolution of important features.

There are several ways to generate a flat field. The first is to simply remove all definitions of the object being studied. However, this approach requires maintaining two separate folders: one with the object and one without it.

Code 4.21

```
  // ========================================
  //           DEFAULT MATERIALS
  // ========================================

  fWorldMaterial     = Vacuum;
  fIonCMaterial      = Vacuum;
  fDetectorMaterial  = Silicon; // CdTe
  fMaskMaterial      = Gold;
  fSubMaterial       = Graphite;
  fMuscleMaterial = PlexiGlass;
  fMicroSphereMaterial = Wax;
  fMuscleMaterial1 = PlexiGlass;
  fWaxInsertMaterial = Wax;
```

A simpler method is to assign the object the same material as its surroundings. For example, if the object is placed directly inside the simulation world, you can make it out of the same material as the world volume. This eliminates refraction and effectively simulates the absence of any object in the scene. By doing so in the `Phantom_Source` phantom we just have to modify the structure of Code 4.6 as shown in Code 4.21.

Note that in our case, we are specifically interested in studying the microcalcifications within the phantom. For this reason, their material is changed to match that of the surrounding wax.

Note: It is crucial that the flat field is not run simultaneously with the simulation, as this may cause interference and result in incorrect data correction.

5 Detector Bins:

In Geant4, there are two main objects that can be created to register detections: a *mesh* of *primitive scorers* or a **sensitive detector**. This chapter will focus on these detectors; however, many users are only interested in retrieving the 2D pixel-by-pixel intensity image of the detector. For them, this section will not be very relevant, as `Phantom_Source` already includes the necessary code to create a pixel-by-pixel detector that replicates the one in the High Energy Laboratory; this is briefly explained in Section 5.2. On the other hand, *primitive scorers* will be useful for various types of detection beyond intensity, while the **sensitive detector** is a more specialized tool that is not essential for angiography simulations and, therefore, will not be covered in this manual.

5.1 Primitive Scorers

A primitive scorer in Geant4 is a specialized tool for registering physical quantities (such as deposited energy, particle flux, or the number of events) in volumes of interest in a structured and automated manner. It is part of the `G4MultiFunctionalDetector` system, which allows multiple scorers to be associated with a single logical volume to collect specific data without the need to implement a complete sensitive detector. Primitive scorers simplify data collection by using predefined structures, such as hit maps (`G4THitsMap`), and allow users to extend their functionality by overwriting key methods, such as `ProcessHits`.

5.1.1 Detection Mesh

Evidently, primitive scorers have many layers to be explored; however, in an applied case for a new user, it is appropriate to focus on the creation of a mesh. A mesh is a user-defined volume that is divided and positioned as needed, but it is not made of any material; it acts as a perfect counter for physical quantities. In other words, these objects can be instructed to register the accumulation of a wide range of physical properties of the particles that pass through them.

It is worth considering that meshes can only have a cylindrical or box shape. Additionally, their dimensions are also measured from the center of the figure, just like in `G4Box` and `G4Tubs`, so they are divided in half.

In `scorers.mac` of `Phantom_Source`, the user has two example meshes available; however, both are dedicated to the same type of detection. One of them is shown in code 5.1 to illustrate how it is implemented.

Code 5.1

```
/score/create/boxMesh Mesh1              // Name and shape of the mesh
/score/mesh/boxSize 4 13 4 mm            // Dimensions of the box
/score/mesh/translate/xyz 0 0 -36 cm     // Location
/score/mesh/nBin 1 1 1 // x y z          // Number of bins
/score/quantity/doseDeposit dose         // Physical quantity to measure
/score/quantity/passageCellFlux passageCellFlux // Physical quantity to measure
/score/filter/particle protonFilter proton  // Particle filter
/score/close
```

Code Description

- **Mesh shape (`boxMesh`)**: A box-shaped mesh is defined.
- **Box size (`boxSize`)**: The mesh dimensions are specified as $4\,\text{mm} \times 13\,\text{mm} \times 4\,\text{mm}$. This defines the size of the box along the x, y, and z axes, respectively.
- **Mesh position (`translate/xyz`)**: The mesh is translated to position $(0, 0, -36\,\text{cm})$ in space, where the x, y, and z coordinates are specified in centimeters in this case.
- **Number of bins (`nBin`)**: The number of bins (subdivisions) along each axis x, y, and z is set to $1, 1, 1$, respectively. This means that there are no additional subdivisions within the mesh.
- **Physical quantities to measure (`doseDeposit` and `passageCellFlux`)**:
  - `doseDeposit`: Specifies that the deposited dose in the box will be measured.
  - `passageCellFlux`: Measures the volumetric flux of particles passing through the mesh bins.
- **Particle filter (`particle`)**: A filter named `protonFilter` is applied, which restricts the measurements to particles of type `proton` (protons) only.
- **Configuration closure (`close`)**: This command finalizes the mesh configuration, indicating that it is ready for use in simulations.

Scorers

As seen in lines 4 and 5 of 5.1, a mesh can act as a detector for multiple physical quantities. Below is a list of all available scorers in Geant4:

- **energyDeposit**: Measures the amount of energy deposited in a specific volume due to interacting particles.
- **cellCharge**: Records the accumulated charge in a cell due to charged particles passing through it.
- **cellFlux**: Measures the flux of particles crossing a specific cell over a given time period.
- **passageCellFlux**: Measures the flux of particles passing through a cell in terms of the distance traveled within the volume.
- **doseDeposit**: Records the radiation dose deposited in a region due to particle interactions.
- **nOfStep**: Counts the number of steps taken by particles within a given volume or space.
- **nOfSecondary**: Counts the number of secondary particles produced by interactions of primary particles.
- **trackLength**: Measures the total path length traveled by particles within a given volume.

- **passageCellCurrent**: Records the current generated by particles passing through a specific cell.
- **passageTrackLength**: Measures the total path length traveled by particles while passing through a cell.
- **flatSurfaceCurrent**: Measures the current generated on a flat surface due to passing particles.
- **flatSurfaceFlux**: Measures the flux of particles on a flat surface over a period of time.
- **nOfCollision**: Counts the number of collisions occurring within a given volume during particle interactions.
- **population**: Records the number of particles in a specific region during the simulation.
- **nOfTrack**: Counts the number of particle tracks generated or processed in the simulation.
- **nOfTerminatedTrack**: Counts the number of particle tracks that have ended, either by absorption, stopping, or exiting the system.

Filters

Special attention is given to line 7 of 5.1, where an important property of the mesh is applied: a **filter**. Each `/score/quantity` can have only one filter, which helps increase the specificity of the type of detection being sought. Below is a list of all available filters:

- **charged**: Filter for charged particles, allowing selection of only electrically charged particles.

```
1    /score/filter/charged "Name"
```

- **neutral**: Filter for neutral particles, allowing selection of only electrically neutral particles.

```
1    /score/filter/neutral "Name"
```

- **kineticEnergy**: Filter based on the kinetic energy of particles. A range of energy is defined between `eLow` and `eHigh`, in the specified unit.

```
1    /score/filter/kineticEnergy "Name" *eLow* *eHigh* *Units*
```

- **particle**: Filter based on particle type. It allows selection of specific particles, such as protons, electrons, etc., indicated by `p1` to `pn`.

```
1    /score/filter/particle "Name" *p1* *p2* ... *pn*
```

- **particleWithKineticEnergy**: A combined filter that selects particles based on both their type and kinetic energy. A range of energy is defined between `eLow` and `eHigh` with the specified unit, and the included particles (`p1` to `pn`) are specified.

```
/score/filter/particleWithKineticEnergy "Name" *eLow* *eHigh* *Units* *p1*
    *p2* ... *pn*
```

5.1.2 Creating an Energy Spectrum

A type of information that is often highly useful for attenuation analysis and similar studies is the energy spectrum. In this section, we will see how we can use detection meshes to retrieve the spectrum.

Code 5.2

```
/score/create/boxMesh Mesh1
/score/mesh/boxSize 14080/2 14080/2 300/2 um
/score/mesh/translate/xyz 0 0 *detector location* cm
/score/mesh/nBin 1 1 1 // x y z
/score/quantity/flatSurfaceFlux flux0
/score/filter/particleWithKineticEnergy range0 0. 5. keV gamma
/score/quantity/flatSurfaceFlux flux1
/score/filter/particleWithKineticEnergy range1 5. 10. keV gamma
/score/quantity/flatSurfaceFlux flux2
/score/filter/particleWithKineticEnergy range2 10. 15. keV gamma
/score/quantity/flatSurfaceFlux flux3
/score/filter/particleWithKineticEnergy range3 15. 20. keV gamma
/score/quantity/flatSurfaceFlux flux4
/score/filter/particleWithKineticEnergy range4 20. 25. keV gamma
/score/quantity/flatSurfaceFlux flux5
/score/filter/particleWithKineticEnergy range5 25. 30. keV gamma
/score/quantity/flatSurfaceFlux flux6
/score/filter/particleWithKineticEnergy range6 30. 35. keV gamma
/score/close
```

In 5.2, a base script is shown that the user can use in `scorers.mac`. Regarding size and positioning, the mesh is placed above the detector that includes `Phantom_Source`, using the same dimensions. Additionally, for detection, multiple surface flux scorers are used, which measure the photon flux per cm^2, each with an energy and particle filter applied. Note that the spectrum will have energy bins of 5 keV, ranging from 0 keV to 35 keV, and will only register gamma particles (high-energy photons). Clearly, this script will likely require user intervention to adjust the mesh position, the number of bins, and the energy range.

5.1.3 Retrieval of the Mesh's Results

To activate the functionality of a mesh, it must be invoked in the `run.mac` file. If we want to retrieve the measurements from a single scorer in the mesh, we write the following:

```
/score/dumpQuantityToFile "Mesh_Name" "Scorer_Name" "Output_File_Name"
```

Similarly, if we are interested in retrieving the results from all the scorers in a mesh, as in the case of the energy spectrum, we write:

```
/score/dumpAllQuantitiesToFile "Mesh_Name" "Output_File_Name"
```

Data Format

The output format of a mesh is always similar, regardless of the scorers used. To illustrate, let us specifically examine what a resulting file would look like for an energy spectrum. Below is a section of a file retrieved for a spectrum mesh with bins of $1keV$ up to $80keV$.

# primitive scorer name: flux1					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	1.532626229	0.425124904	7
# primitive scorer name: flux10					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	4886.316783	619.4940665	38665
# primitive scorer name: flux11					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	2290.01467	290.3764093	18117
# primitive scorer name: flux12					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	1483.855131	213.7454094	11681
# primitive scorer name: flux13					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	3124.053971	396.8287125	24710
# primitive scorer name: flux14					
iX	**iY**	**iZ**	**total(value) [percm2]**	**total(valˆ2)**	**entry**
0	0	0	5826.131469	758.1260093	46061

The first thing that stands out is the titles marked with #, which indicate that the scorers appear in an unordered manner, jumping from flux1 to flux10. The user must take this into account to ensure the spectrum is plotted in the correct order. Next, the columns iX, iY, and iZ indicate the coordinates of the bin where the data in that row was recorded. Since only one bin was used in the mesh definition in 5.2, the data appears only in the $(0, 0, 0)$ cell. Additionally, see below:

- **total(value) [percm2]**: This column shows the accumulated value of the physical quantity being measured in that cell. The quantity may vary depending on the type of *primitive scorer*. In this case, it measures the surface particle flux.
- **total(valˆ2)**: This column reports the accumulated value of the squared measured quantities. This is used to calculate the variance and standard deviation of the measurements, which is essential for estimating statistical uncertainty.
- **entry**: This value indicates how many times events or contributions were recorded in that cell during the simulation. That is, if a particle passes through or interacts with the cell, it is counted as an entry.

5.2 Pixel to Pixel Detection

The pixel-by-pixel detector editing included in `Phantom_Source` is not essential, but it is worth understanding it at a basic level.

A file named `PepiPSPixiRad.cc` is included in the `src` directory. In general, this is where the pixel detection capability is assigned to the intensity detector pixels. The main purpose of this code is to simulate how particles interact with a detector composed of a pixel matrix and how the energy deposited in those pixels can be measured. Instead of using a standard scorer that simply measures the deposited energy, a class has been implemented that performs several additional calculations. For example, it ensures that the deposited energy is recorded only in pixels that have received a significant amount of energy and also groups the energy of several neighboring pixels to simulate how a real detector signal might behave.

At the end of each event, the class also has a method called `EndOfEvent`, which is responsible for final processing of the results, summing the energy of nearby pixels if necessary and applying some adjustments to simulate effects such as energy dispersion.

6 Getting Started: Running a Simulation

To execute a simulation, it is necessary to properly prepare all key system components, as the simulation will not compile if they are not correctly configured. This includes defining geometries, involved materials, the particle source, and visualization attributes. Additionally, it is crucial to properly configure the physics model that describes how particles interact with materials.

6.1 Understanding detmask_config.in

Before running the simulation, the user should review the `detmask_config.in` file. This part of the code is responsible for several important aspects. One of them is acting as a confirmation of certain code parameters: source position, source-object distance, object-detector distance, mask thickness, distance between mask apertures, and aperture width. It serves as a redundancy check where these conditions, previously specified in `PepiDetectorConstruction.cc`, are entered again. In 6.1, an example is shown in `Phantom_Source`, between lines 2 and 7. The other two functions of this file deserve further exploration.

Code 6.1

```
1  /Pepi/det/setBidimensional true     // Configures the detector to operate in 2D.
2  /Pepi/det/setSourcePosZ -85 cm              // Source position.
3  /Pepi/det/setSrcObjDistance 49 cm           // Source-object distance.
4  /Pepi/det/setObjectDetDistance 49 cm        // Object-detector distance.
5  /Pepi/det/setMaskThickness 250 um           // Mask thickness.
6  /Pepi/det/setM2Pitch 62 um                  // Distance between apertures.
7  /Pepi/det/setM2Aperture 15 um               // Aperture width.
8  /Pepi/det/setAcquisitionType conventional   // Acquisition type.
9  /Pepi/det/setDetType 1COL                   // Detector type.
10 /Pepi/det/setThreshold1 9 keV               // Threshold 1.
11 /Pepi/det/setThreshold2 20 keV              // Threshold 2.
```

Acquisition Type and Introduction of Masks

In line 8 of 6.1, the acquisition type is specified, a parameter that determines the use of masks. Regarding this, for certain phase retrieval strategies, such as edge illumination, filters (usually called masks) are necessary for analysis. Here, in `detmask_config.in`, the user can select `conventional` for use without masks, `singlemask` to activate mask 1, and `doublemask` to activate both masks.

The PEPI implementation of phase contrast imaging focuses on the edge illumination technique. This code allows the simulation of both the conventional setup for this method—using two masks—and a single-mask variation. The masks are constructed using two functions: `CreateMask`, which generates the absorbing blocks between the slits, and `CreateSubstrate`, which creates the structural support of the mask.

If you read Code 6.1 carefully, you will notice that the only mask characteristics explicitly defined in the code correspond to the second mask, the one placed in front of the detector. This might raise a question: how is Mask 1 (the pre-sample mask) created if

its dimensions are not specified?

Let's take a look at the code responsible for the mask creation process described in lines 1424 to 1605 of `Phantom_Source`. A relevant fragment is shown in Code 6.2:

Code 6.2

```
// - Build the MASK APERTURE UNIT as a Box -:
G4Box* MSolid = new G4Box(name,  // The name of the absorbing block.
                    ((pitch-aperture)/mag)/2,    // Its X-halfsize.
                    1.1*fPixiRadSizeY/2,         // Its Y-halfsiz.
                    thickness/2);                // Its half thickness.

G4String lvname = name+"LV";
G4LogicalVolume* MLogical = new G4LogicalVolume(MSolid, // Its solid.
                                material,               // Its material.
                                lvname);                // Its name.

// - Build the MASK ENVELOPE -:
G4String envname = "Envelope"+name;
G4Box* EnvelopeSolid = new G4Box(envname,       // The mask's envelope name.
                      (1.1*fPixiRadSizeX/mag)/2,    // Its X-halfsize.
                      1.1*fPixiRadSizeY/2,          // Its Y-halfsize.
                      thickness/2);                 // Its half thickness.

G4String lvenvname = "Envelope"+name+"LV";
EnvelopeLogical =  new G4LogicalVolume(EnvelopeSolid,     // Its solid.
                                       fWorldMaterial, // Its material.
                                       lvenvname);     // Its name.
```

First, the code creates the thin absorbing regions of the mask. Their dimension along the x-axis is not exactly what one might expect at first—it is calculated as the pitch minus the aperture, but this value is scaled according to the distances chosen for the mask configuration (the mask's magnification value).

For the pre-detector mask, this scaling factor typically ends up being close to 1, so the written configuration matches the simulated one. However, in the single-mask setup, where the mask is positioned before the sample under study, the effective dimensions can become a bit more complex to interpret.

This situation is not explicitly handled in the original PEPI implementation. That is why, in our `Phantom_Source` adaptation, we establish that if the `AcquisitionType` is set to `singlemask`, the written dimensions are used directly to build the mask with the exact configuration defined by the user.

Detector Type

This parameter allows the user to receive more than one output file from the pixel-by-pixel detector simulation. By setting `0COL`, only a single file is obtained, this is the case of an ideal detector. With `1COL`, two files are generated: one with a lower energy threshold defined by `Threshold1` and another normal file. Finally, using `2COL`, three files are produced: one with the lower threshold `Threshold1`, another with `Threshold2`, and a normal one.

6.2 Understanding geometry_config.in

The `geometry_config.in` file is based on three parameters that are quite useful in techniques such as Edge Illumination. The first one corresponds to the lateral position of the sample mask (stepping), the second one to the lateral position of the sample (dithering), and the third one to the rotation of the sample. An example is shown in 6.3.

Code 6.3

```
-4.2 23.25 0
```

6.3 Understanding run.mac

The `run.mac` file is a Geant4 command script that controls the execution of the simulation, defines simulation parameters, configures physics aspects, and specifies how results are stored. The contents of this file can be seen in Code 6.4:

Code 6.4

```
/run/numberOfThreads 4
/control/verbose 0
/control/saveHistory
/run/verbose 0
/control/execute gps.in
/control/execute detmask_config.in
/Pepi/cont/loadConfig geometry_config.in
#/control/macroPath ../spectra
/control/execute ../spectra/28kV.in
#/control/macroPath ../build
/Pepi/cont/setBaseName Frame_1
/run/initialize
/control/execute scorers.mac

#/run/printProgress 1000000
/Pepi/cont/beamOn 2000000000
/Pepi/cont/beamOn 2000000000
/Pepi/cont/beamOn 1000000000

/score/dumpQuantityToFile Mesh1 dose dose_28kVInLine2_.out
```

Below, we define the possible modifications that can be made before running the simulation.

Source Spectra

The energy spectrum generated by the source depends on the selected initial voltage. If a filter is used, it must also be considered when determining the corresponding spectrum. Generally, predefined spectra are available in the `spectra` file. However, if a different energy spectrum is required, a `.in` file must be added in the `spectra` folder, using the TASMICS platform to generate it.

To define the spectrum, the `/control/execute` line must be edited with the name of the spectrum file used.

Modifying the File Name

In Geant4, it is crucial to modify the file name before saving it, as new files will not be stored in the `build` folder if another file with the same name already exists. This change can be made using the `/Pepi/cont/setBaseName` command.

Verbosity Levels

Verbosity refers to the amount of information required from a simulation; the possible verbosity levels are organized as follows:

Level	Description
0	Silent (basic information)
1	Normal (more information)
2	Detailed (process and step details)
3	Very Detailed (advanced and deep information)
4	Extreme (all available information)

Table 2: Verbosity Levels in Geant4

Verbosity can be configured in the `run.mac` file. The global verbosity level is controlled with `/control/verbose <level>`, while execution-phase verbosity is managed with `/run/verbose <level>`.

Event Count

The number of events represents specific instances within a modeled process, which may include particle interactions with materials, secondary particle generation, and their detection in the detectors. A sufficient number of events is essential to ensure that the simulation provides statistically significant and complete information.

The number of events is defined using the `/Pepi/cont/beamOn` command. A value of 1×10^9 events is recommended to ensure adequate statistical quality in both the simulation and the resulting images.

If you wish to simulate more events, you must consider that a single process is limited by the number of bits available for indexing. In our case, we are working with a 64-bit system, which imposes a per-process limit of 2,147,483,647 events. To overcome this limitation, you can add multiple `/Pepi/cont/beamOn` commands. For example, Code 6.4 illustrates how to simulate a total of 5×10^9 events using two commands with 2×10^9 events each, and one with 1×10^9.

6.4 Running Code

Once everything is set up and the user is ready to run the code, the following commands should be executed in the `build` directory:

- `make`: Compiles the code.
- `./pepi run.mac`: Runs the simulation.

On the first attempt, the user may encounter an error due to **permissions**. Initially, it is not possible to run a simulation using code from another user due to permission issues, as is the case with `Phantom_Source`. To resolve this, the build folder must be deleted and recompiled using the following steps:

1. Execute the command `rm -r *` inside the folder.
2. Then, run `cmake ..`
3. Finally, after compilation, execute `make -j30`

This step is also necessary if a new folder containing the same information as the previous one has been created.

Note: There are two `run.mac` files, one inside the `build` folder and one outside it. It is recommended to edit both files simultaneously to avoid compilation errors.

6.4.1 Analysis of a 2-Dimensional Image

The `build` folder generates `.raw` files that must be processed for analysis. The **Fiji** application is recommended for this purpose, at least for previewing the image and ensuring everything went well.

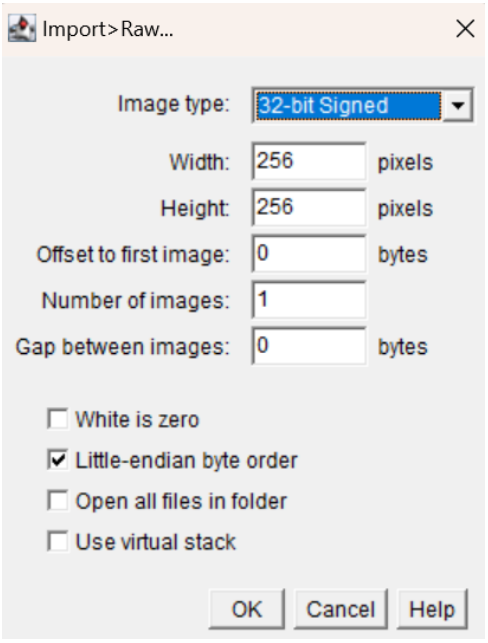

Figure 5: General configuration for displaying a raw image in the Fiji software. It is important to note that this is only a preview tool. The software is not capable of displaying a meaningful grayscale, so it is recommended to use a Python script for rigorous analysis.

For more accurate reconstruction and detailed image analysis, it is recommended to use `Image_Analysis`, a script that complements the use of `Phantom_Source`. This code is provided as a Jupyter Notebook, where each step is documented. To access it, visit

the GitHub Repository where this manual is hosted. Make sure to install all of the packages used in the beginning of the notebook to ensure good functionality of the code.

6.5 Common Issues

- If unexpected information is obtained or if the simulation does not seem to compile correctly, Geant4 might be using a different `build` folder. To fix this, delete the build folder following the same steps as before.
- The files *run.mac*, *detmask_config*, and *geometry_config* are present both inside and outside the `build` folder. To avoid inconsistencies, any modifications should be made to the versions outside the `build` folder. Afterwards, update the build by running `make` inside the `build` folder.

7 Implementation of Several PCI Method

7.1 In-Line Method

To simulate the in-line method, also known as the free propagation or propagation-based method, it must be specified in the detmask config.in file using the command:

```
/Pepi/det/setAcquisitionType conventional
```

7.2 Edge-Illumination Method

7.2.1 Double Mask Configuration

To simulate the edge illumination method using a double-mask setup, it must be explicitly specified in the detmask config.in file using the command:

```
/Pepi/det/setAcquisitionType doublemask
```

As previously explained, it is important to note that the parameters `setM2Pitch` and `setM2Aperture` define the geometric characteristics of the detector mask. The parameters corresponding to the sample mask are not entered directly; instead, they are automatically determined based on the values of the detector mask, scaled by the geometric magnification factor of the mask.

A little theoretical background

The edge-illumination method is an X-ray phase contrast imaging (XPCI) technique that commonly uses two periodic absorbing masks. In this method, a sample mask (M1) is placed upstream of the sample to segment the incident X-ray beam into narrow and independent beamlets. Each opening in M1 acts as an individual source, illuminating a small portion of the sample.

As the beamlets pass through the sample, they undergo attenuation (a reduction in intensity due to absorption), refraction (a slight deviation caused by changes in the refractive index at material interfaces), and scattering (resulting from small-scale inhomogeneities, which can broaden the beamlet and contribute to the dark-field signal).

Immediately before the detector, a second mask (M2) is positioned. This detector mask, with a pitch scaled by the geometric magnification of the system relative to the sample mask, modulates the transmitted beam by creating insensitive regions between adjacent detector pixels. The final image is formed from the recorded intensities of the beamlets, which reflect the cumulative effects of attenuation, refraction (or phase shifts) and scattering.

7.2.2 Single Mask Configuration

To simulate the edge-illumination method using a single-mask setup, it must be explicitly specified in the `detmask_config.in` file using the command:

```
/Pepi/det/setAcquisitionType singlemask
```

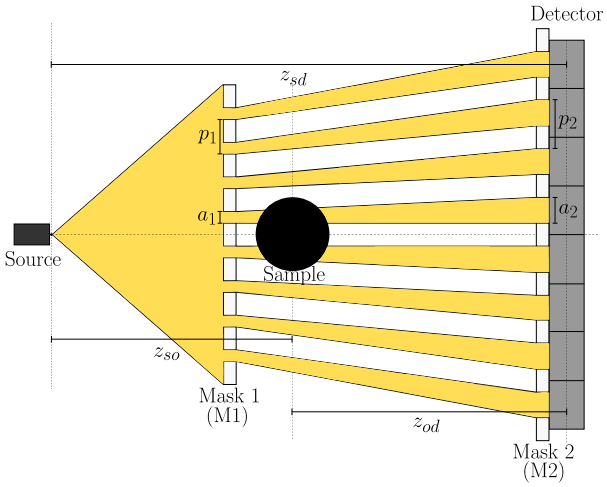

Figure 6: Schematic diagram of the DM-EI setup. A divergent X-ray beam from the source passes through two periodic masks: Mask 1 (M1), located at a distance z_{so} from the source, and Mask 2 (M2), positioned just before the detector at a distance z_{od} from M1. The sample is placed between the masks, and partially obstructs the beam. The periodic structures of M1 and M2, with pitches p_1 and p_2 and apertures a_1 and a_2, define the illumination pattern at the detector plane. The relative alignment of the masks modulates the detected intensity, enabling phase and absorption contrast retrieval.

As previously explained, it is important to note that the parameters `setM2Pitch` and `setM2Aperture` now define the geometric characteristics of the sample mask, and these parameters are not scaled by the magnification.

A little theoretical background

The Single-Mask Edge Illumination (SM-EI) technique is a simplified version of the conventional EI method, using only one periodic absorbing mask. This mask, known as the sample mask (M1), is placed immediately in front of the sample.

The sample mask structures the incident X-ray beam into an array of beamlets, similar to the double-mask method. However, in SM-EI, the detector mask is omitted. Instead, the edges between adjacent detector pixels are used to analyze phase shift.

The parameters of the sample mask are chosen so that the projected beamlets are aligned with the boundaries between adjacent columns (or rows) of pixels on the detector. Ideally, each beamlet is split evenly, with approximately 50% of its intensity falling on each of the two neighboring pixels.

Precise alignment of the mask with the detector is crucial. It is essential to ensure that the beamlets are incident exactly on the edges of the pixels to maximize the sensitivity to refraction. This technique is primarily sensitive to attenuation and refraction. It does not provide a direct extraction of the dark-field signal.

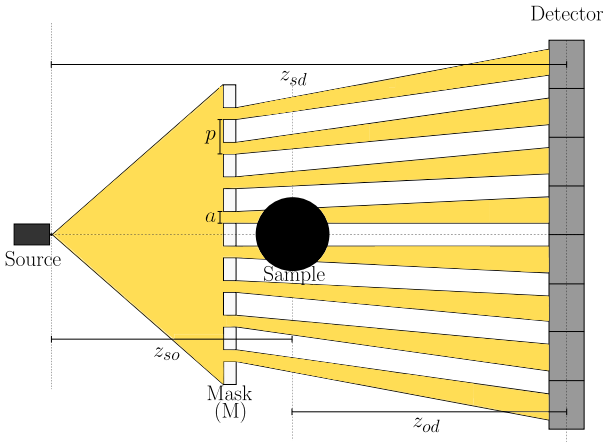

Figure 7: Schematic of the experimental setup for the SM-EI configuration. A divergent X-ray beam emitted by the source passes through a periodic mask (M1), placed at a distance z_{so} from the source. This mask modulates the beam, generating illumination fringes that pass through the sample before reaching the detector. The mask has a pitch p_1 and aperture a_1. The distance between the mask and the detector is z_{od}. This setup allows phase and attenuation contrast retrieval without the need for a second mask.

7.3 Image Construction

To obtain reliable spatial resolution in the reconstructed image, it is essential to properly adjust parameters such as stepping and dithering. The stepping should be configured so that the density distributions of even and odd pixels are correctly aligned, avoiding mismatches that could affect image quality. This analysis can be made in our code `Image_Analysis` given in the GitHub Repository.

Dithering, on the other hand, allows the sample to be moved in order to capture areas that would otherwise be blocked by the absorbing lines of the mask, thus preventing discontinuities in the final image. These parameters can be modified in the `geometry_config.in` file, as previously explained. The number of dithering positions is given by:

$$N_{\text{dith}} = \frac{\rho_{\text{detector}}}{M\Delta x}, \tag{1}$$

where ρ_{detector} is the pitch of the detector, M is the geometric magnification between the sample and the detector, and Δx is the dithering step—usually taken as 10μm.

7.4 Phase and Attenuation Maps

In `Image_Analysis` code, phase gradient image is given by:

$$P = \left(\frac{I_{\text{odd}} - I_{\text{even}}}{I_{\text{odd}} + I_{\text{even}}}\right) \frac{M \cdot o \cdot a}{2}, \tag{2}$$

where I_{odd} is the normalized image for the odd pixels, I_{even} is the normalized image for the even pixels, M is the system magnification, and o is a scale factor, and a is the mask

aperture.

The absorption is given by:

$$A = \frac{I_{\text{odd}} + I_{\text{even}}}{I_{ff,\text{odd}} + I_{ff,\text{even}}}, \tag{3}$$

where $I_{ff,\text{odd}}$ and $I_{ff,\text{even}}$ are the flat fields for the odd and even pixels, respectively.

You can go to the `Image_Analysis` folder and open the Jupyter Notebook with the same name, where it is explained in detail how the code works and the way in which you can retrieve these maps.

References

[1] L. Brombal, *PEPILab: Geant4-based simulation toolkit*, INFN. Available at: https://baltig.infn.it/brombal/PEPILab.git.

[2] D. Cullen, J.H. Hubbell, and L. Kissel. Epdl97: the evaluated photon data library, 97 version. *UCRL–50400*, 6(Rev.5):, 1989.

[3] J. Allison et al., Recent Developments in Geant4, *Nucl. Instrum. Meth. A*, 835, 186-225, 2016.

[4] J. Allison et al., Geant4 Developments and Applications, *IEEE Trans. Nucl. Sci*, 53, 270-278, 2006.

[5] Geant4 - A Simulation Toolkit, S. Agostinelli et al., *Nucl. Instrum. Meth. A*, 506, 250-303, 2003.

Appendix

- Below is the repository where the `Original_Source` folder can be found. https://github.com/ThomasAnkhe/Users_Manual.git
- And the official Geant4 page. https://geant4.web.cern.ch/