musigSisfe documentation

Niccolò Ciavarelli

Contents

List of Figures

1 Introduction

This work focuses on developing the new grid geometry and simulating the stopping efficiency of muons ($\mu+$).
The key idea behind this new grid improvement is to generate muons through the lateral surfaces rather than the upper one, as was done in the old liquid helium pool design. The new design leverages the capillary properties of superfluid helium, creating a column of helium through this phenomenon between the silicon columns. The main differences can be seen in the Fig. 1.

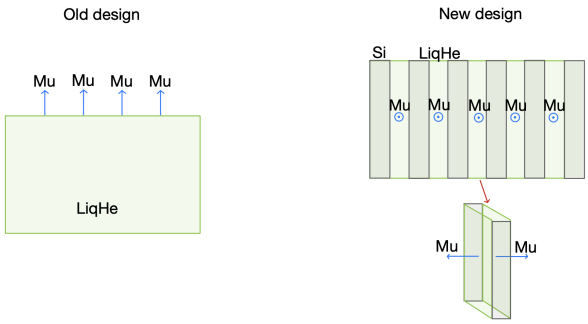

Figure 1: Schematic description of the difference between the new and old design for the liquid Helium target.

2 Silicon-Superfluid Helium grid class

The grid design has been implemented in the GEANT4 simulation using a standalone C++ class called **sisfeGeometry**. The class provides the following methods:

2.1 Constructor - Destructor

- `sisfeGeometry()`, default constructor **Recommended.*
- `sisfeGeometry(G4String nameID)` default constructor with nameID.
- `sisfeGeometry(G4LogicalVolume *, G4String, G4int nLiqHe, G4double LiqHeDimX, G4double LiqHeDimY, G4double LiqHeDimZ, G4double SiDimX, G4double SiDimY, G4double SiDimZ, G4ThreeVector position, G4RotationMatrix *rot)` default constructor where every parameter can be set.
- `~sisfeGeometry()` default destructor.

The constructor must be called in the *DetectorConstruction*.

2.2 Define materials

- `void DefineMaterials()`, material definition: new LiqHe, Si and Galactic materials will be created.
- `void DefineMaterials(G4String)` material definition, setting the nameID: new LiqHe, Si and Galactic materials will be created.

- `void DefineMaterials(G4Material* Vacuum, G4Material* LiqHe, G4Material* Si)`, material definition through already defined materials **Recommended.*

This method can be called in the constructor of *DetectorConstruction* or in the Detector-Construction *DefineMaterials()* for the first two points, and only in the DetectorConstruction *DefineMaterials()* for the third one.

2.3 Make the geometry

- `void MakeGeometry(G4LogicalVolume *, G4int nLiqHe, G4double LiqHeDimX, G4double LiqHeDimY, G4double LiqHeDimZ, G4double SiDimX, G4double SiDimY, G4double SiDimZ, G4ThreeVector position, G4RotationMatrix *rot)`, method that creates the geometry, ** Recommended.*
- `void MakeGeometry(G4LogicalVolume *, G4String ,G4int nLiqHe, G4double LiqHeDimX, G4double LiqHeDimY, G4double LiqHeDimZ, G4double SiDimX, G4double SiDimY, G4double SiDimZ, G4ThreeVector position, G4RotationMatrix *rot)`, method that creates the geometry and set the nameID.

This method must be called in the *Construct* method of DetectorConstruction. The columns are placed along the x axis.
An error will be raised if: the width (z axis size) or the height (y size) of the Helium columns are greater than the Silicon one.
Note that if the heights are not the same the helium columns are placed with their lower surface aligned with the silicon columns.

2.4 Set methods

- `void SetNameID(G4String nameID)`, set nameID, ** Recommended.*
- `void SetContainerColour(G4String colorContainer)`, set the colour of the container (that contains He and Si columns).
- `void SetLiqHeColour(G4String colorLiqHe)`, set colour of the Liquid Helium columns.
- `void SetSiColour(G4String colorSi)`, set colour of the Silicon columns.
- `void SetColours(G4String colorContainer, G4String colorLiqHe, G4String colorSi)`, set all the colours with one method, ** Recommended.*

They can be called in any part of the code.

2.5 Get methods

- `const G4String GetNameID()`.
- `const G4String GetNameSolidContainer()`.
- `const G4String GetNameSolidLiqHe()`.
- `const G4String GetNameSolidSi()`, Note: here the name of the solid silicon volume is given by the value of `nameID` + `"phySi"`.
- `const G4String GetNameLogicContainer()`.

- `const G4String GetNameLogicLiqHe().`
- `const G4String GetNameLogicSi().`
- `const G4String GetNamePhysContainer().`
- `const G4String GetNamePhysLiqHe().`
- `const G4String GetNamePhysSi().`
- `const ThreeDimensions GetContainerDimensions().`
- `const ThreeDimensions GetLiqHeDimensions().`
- `const ThreeDimensions GetSiDimensions().`
- `const G4Box* GetSolidContainer().`
- `const G4Box* GetSolidLiqHe().`
- `const G4Box* GetSolidSi().`
- `const G4LogicalVolume* GetLogicalContainer().`
- `const G4LogicalVolume* GetLogicalLiqHe().`
- `const G4LogicalVolume* GetLogicalSi().`
- `const G4VPhysicalVolume* GetPhysicalVolumeContainer().`
- `const G4VPhysicalVolume* GetPhysicalVolumeLiqHe().`
- `const G4VPhysicalVolume* GetPhysicalVolumeSi().`

ThreeDimensions is a struct which contains a three variables, G4double x, y, z, representing the three Cartesian coordinate axes.
The nameID, that is necessarry for the correct working of the class object, will be also the name of the Helium column physical volumes.

2.6 Working principle

The working principle behind the geometry construction is as follows: first, create a container made from the same material as the world material. This container will then be rotated, shifted, and placed in position. Afterward, the container is filled along the x axis with helium and silicon columns. The container's dimensions are calculated based on the dimensions and quantities of each column. The input only requires the number of helium columns, as the number of silicon columns is determined accordingly (number of helium columns + 1).

2.7 Messengers

To set through the *.mac* file the following command can be used.

Superfluid Helium - Silicon grid object

```
/setup/sisfe  [name] [number of LiqHe columns] [LiqHe column size x]
 [LiqHe column size y] [LiqHe column size z] [unit of size] [Si column size x]
 [Si column size y] [Si column size z] [unit of size] [pos x] [pos y] [pos z]
 [unit of position] [rotation angle around X] [around Y] [around Z] [mother vol]
```

Colours

/setup/color/sisfe `[container colour] [LiqHe colour] [Si colour]`

Please note that the color is independent of the number of sisfeGeometry objects; each objects will always use the set of colors specified in /setup/color/sisfe.

2.8 Example

in the *muStopping2024gridNew.mac* there is an example of implementation:

```
    ############## GRID TARGET HELIUM AND SILICON ##############
/setup/sisfe SfHeTarget 1000 0.04 28.999 0.07 mm  0.01 28.999 0.07 mm
 0.035 0. 0. mm 0 90 0 World
#### Set visualization: red, green, blue, yellow, magenta, invisible
/setup/color/sisfe invisible green blue
```

In this case, we are creating the object sisfe, representing the new grid design. As mentioned in the section 2.6, the grid is initially created and filled along the x-axis, then rotated 90 degrees around the y-axis, aligning the columns along the z-axis. The liquid helium columns have a width of 0.04 millimeters, while the silicon columns have a width of 0.01 millimeters.

2.9 Usage in the MuSiG

The added library consists of *musigSisfe.h* and *musigSisfe.cpp*. The *musigSisfe.h* file is included in *musigDetectorConstruction.h*. These files contain the main class for the geometry, named **sisfeGeometry**. Additionally, they define and utilise the ThreeDimensions struct, which is used to retrieve dimensional information about the container, liquid helium, and silicon.

Two typedefs have been added to *musigDetectorConstruction.h*: **SisfeGeometryDefinition**, to handle the definition of the **sisfeGeometry** object, and **SisfeColDefinition**, to manage the colour definition of the object.

Two methods have been added to the **DetectorConstruction** class:
`void SetSisfe(const SisfeGeometryDefinition &)` and
`void SetSisfeColour(const SisfeColDefinition &)`, in order to handle the settings of the parameters via messengers.

Inside `DetectorConstruction::DetectorConstruction()` the object of **sisfeGeometry** is instantiated, with the default constructor: `sisfe = sisfeGeometry()`.
Inside `void DetectorConstruction::DefineMaterials()` the materials for the object **sisfe** are setted, with: `sisfe.DefineMaterials(Galactic, LiqHe, SiM)`; note that **SiM** (Silicon material) has been defined as `SiM = manager->FindOrBuildMaterial("G4_Si")` and that here the materials are **passed** to the object.
Inside `G4VPhysicalVolume *DetectorConstruction::Construct()` the grid of **sisfe** is constructed:

```
//------------------Sisfe grid construction -----------------------
        for(const auto &fSisfeParams : fSisfeParamsV)
        if(fSisfeParams.isPlaced){
            auto sisfeMother =
```

```
                G4LogicalVolumeStore::GetInstance()->
                GetVolume(fSisfeParams.mother);
            auto gridRot = new G4RotationMatrix();
            gridRot->rotateX(fSisfeParams.rot.x() * deg);
            gridRot->rotateY(fSisfeParams.rot.y() * deg);
            gridRot->rotateZ(fSisfeParams.rot.z() * deg);
            sisfe.SetNameID(fSisfeParams.name);
            if(fSisfeColParams.isInv){
                sisfe.SetColours(fSisfeColParams.ContainerCol,
                fSisfeColParams.LiqHeCol, fSisfeColParams.SiCol);
            }
            sisfe.MakeGeometry(sisfeMother, fSisfeParams.nLiqHe,
            fSisfeParams.sizeLiqHe.x(),  fSisfeParams.sizeLiqHe.y(),
            fSisfeParams.sizeLiqHe.z(), fSisfeParams.sizeSi.x(),
            fSisfeParams.sizeSi.y(),  fSisfeParams.sizeSi.z(),
            fSisfeParams.pos, gridRot);
        }
//-------------------------------------------------------------------
```

Is important to notice that the methods `void SetNameID(G4String nameID)`, `void SetColours(G4String colorContainer, G4String colorLiqHe, G4String colorSi)` and `void MakeGeometry(G4LogicalVolume *, G4int nLiqHe, G4double LiqHeDimX, G4double LiqHeDimY, G4double LiqHeDimZ, G4double SiDimX, G4double SiDimY, G4double SiDimZ, G4ThreeVector position, G4RotationMatrix *rot)` have been used. With this logic the user is able to create more grids (each one with a different name) and place in different positions, with different rotations.

The two setups methods have been implemented in the following way:

```
        void DetectorConstruction::SetSisfe(const SisfeGeometryDefinition &params){
        SisfeGeometryDefinition fSisfeParams;
        fSisfeParams.name = params.name;
        fSisfeParams.nLiqHe = params.nLiqHe;
        fSisfeParams.sizeLiqHe = params.sizeLiqHe;
        fSisfeParams.sizeSi = params.sizeSi;
        fSisfeParams.pos = params.pos;
        fSisfeParams.rot = params.rot;
        fSisfeParams.mother = params.mother;
        fSisfeParams.isPlaced = params.isPlaced;
        fSisfeParamsV.push_back(fSisfeParams);
    }
    void DetectorConstruction::SetSisfeColour(const SisfeColDefinition &params){

        fSisfeColParams.ContainerCol = params.ContainerCol;
        fSisfeColParams.LiqHeCol = params.LiqHeCol;
        fSisfeColParams.SiCol = params.SiCol;
        fSisfeColParams.isInv = params.isInv;

    }
```

Is important to notice that: **isPlaced** and **isInv** are always initialised as `false` to ensure a logic safety and to avoid errors in case the user does not want to use the class.

Inside *musigDetectorMessenger.cpp* two new messengers **fSisfeDefCmd** and **fColorSisfeDefCmd** have been added to handle the parameters input via .mac files. The first one is for the class object parameters and the second one to set the colours.

3 Encountered issues

The following list contains some of the issues encountered during the integration and development of the sisfeGeometry class:

Main issues

1. **Geant4 version:** currently, the MuSig code produces inconsistent results between version 11.2.0 (and higher) and previous versions. Results from version 11.1.3 and earlier are consistent with each other.

2. **Duplicated muons stopping**: Currently, the MuSig code produces a *TThree* with duplicated stopping entries. This issue arises from the logic used in the stopping condition:
   `(step->GetTrack()->GetTrackStatus() == fStopButAlive)` ||
   `(step->GetTrack()->GetTrackStatus() == fStopAndKill)`.
   When a track status is fStopButAlive, the code executes one additional step (without changing the current position, only updating the step number and global time) before transitioning to fStopAndKill. As a result, particles that transition to fStopAndKill after being in fStopButAlive will create duplicate entries. Particles that directly reach the fStopAndKill status (without passing through fStopButAlive) do not produce duplicates.