In this article we explain the contruction and the source code of the LEμSR simulation using the Geant4 framework. After a insight of the directory structure, we will look at the special classes that were impemented for this simulation.

Chapter 1

G4 Installation

1.1 G4 Libraries

1.1.1 Downloading Geant4

To intall Geant4, one should download the source code from the G4 website http://geant4.web.cern.ch/geant4/

Geant4 *and* CLHEP codes must be downloaded in order to install Geant4. The compressed files can be extracted using the following commands:

- tar -zxvf geant4.*.tar.gz
- tar -zxvf clhep.*.tar.gz

It is recommanded to create a CLHEP directory in which one would extract the clhep.*.tar.gz file.

Once these files are unpacked, Geant4 can be installed.

1.1.2 Installation

Automatic way

The safest way is to execute the automatic installation by using the configure shell script. One should type the command $G4INSTALL/./Configure -install and follow the instructions. Each question must be answered *very carefully* in order to get a good configuration.

Manual way

The manual installation is a bit risky but is better when one need to know exactly about all installation parameters. First, some environement variables must be set (for example):

- export G4INSTALL=/afs/psi.ch/user/u/username/geant4.6.1 : path to geant4 principal directory

- export G4SYSTEM=Linux-g++ (for example)
- export CLHEP_BASE_DIR=/afs/psi.ch/user/u/username/CLHEP

Then the other variables should be set executing the $G4INSTALL/Configure file. One should answer various questions about the parameters of the installation.

This configuration can be done directly, editing the env.sh file. The file $G4INSTALL/env.sh has to be executed before the compilation

Answers will be registered in the config.sh file which is located in the $G4INSTALL/.config/bin/Linux-g++ directory .

Compiling geant4: The compilation is launched by the command $G4INSTALL/source/gmake. Wait...

1.1.3 First Run

Geant4 should be running now. and this can be checked by running an example, for example $G4INSTALL/examples/novice/N01/

1.2 Visualisation

The visualisation device has to be configured in order to get real time graphix of the geometry and simulation. Here is a summary of the configuration of the MesaGL visualization driver.

Fist, one would download the files mesa*lib*.tar.gz and mesa*demos*tar.gz from the website http://sourceforge.net/projects/mesa3d and uncompress them using the tar -zxvf command. Mesa installation is easy because it is auto configured. One just neeed to type "make" and choose the right extension (e.g. "make linux").

One last operation is to update the LD_LIBRARY_PATH environnement variable including the Mesa*/lib directory:

- export LD_LIBRARY_PATH=$HOME/Mesa*/lib

The MesaGL application should be working and this can be verified launching an example, as Mesa*/progs/demos/gears.

A difficulty can be encountered if the libraries linked to the demo file are obselete. Check using the ldd gears command: the libraries libglut.so.3 libGLU.so.1 and libGL.so.1 must be taken from the $HOME/Mesa*/lib directory.

Chapter 2

Introduction to Geant4 and LEμSR simulation

2.1 Geant4

Geant4 is a C++ toolkit for Monte-Carlo simultation. It contains informations on many physics processes and particles. For a simple simulation, one just has to define a geometry and the interactions one wants to consider. For non trivial experiments like LEμSR , one may have to personalize some physics processes.

The simulation can be interactive thanks to a User Interaction terminal already included in Geant4. One can also visualize the simulation and draw histograms.

2.1.1 A little vocabulary

In this presentation we will encounter the following words:

1. C++ words

   Class: a class is the definition of a new type of variable that is built upon the collection of other variables, functions and classes. They are called the *members* of the class. For example the class Particle has for members: mass, lifetime, spin ...

   Instance, object: from the previous lines one may retain that a class defines a familly. Consider class called A: an *object* or *instanciation* of the A class is a variable of type A. For example Particle mu is the declaration of an object of type Particle named mu. One may then assign the object's members to defined values.

   Inheritance: it is possible to build subclasses. For example, Lepton would be a subclass of Particle class. Instead of building a totally new class, one just have to give Lepton class all the caracteristics of a Particle and add the properties which make a particle a muon. We say that Lepton class *inherits* from Particle class.

2. G4 words: the simulation

   Run: this is the word for a the simulation process.

   Event: during a run, one can shoot as many particles as needed. An event is the simulation of all the primary particles. Before the run the user specifies the number of primary particles shoot per Event as well as the number of events.

   Track: this is the information collection of one particle tracking

   Step: step by step, a track is build. At each step the simulation engine is called to decide the process that the particle will suffer, this in accordance to cross-sections or priority orders defined by the user.

2.2 LEμSR tree

The main file LEMuSR.cc is located in the $LEMU directory. One will find all the header files and source codes in the $LEMU/src and $LEMU/include directories respectively.

2.2.1 The main file: LEMuSR.cc

The main file must contain the following classes.

The mandatory classes

The simplest Geant4 simulation requires four so-called mandatory classes. These classes provide the main information about the simulation fundamental parameters, like the detector description or the physical interaction taken in account. They are initialized at the beginning of LEMuSR.cc:

- runManager: it is the Geant4 mandatory class used to initialize the kernel.
- LEMuSRDetectorConstruction : mandatory user initialization class for the detector geometry.
- LEMuSRPhysicsList : mandatory user initialization class for the physical interactions.
- LEMuSRPrimaryGeneratorAction mandatory user action class for the simulation monitoring.

The visualization class

It ispossible to have visualization of the simulation. Many devices are available and they have to be set up using the LEMuSRVisManager class, that inherits form the G4VisManager class.

For example, using the OpenGL system, one can benefit of a real time visualitation of the simulation. Using to DAWNFILE system, one can get snapshots of the simulation into postscript files. Various visualization systems can be loaded simultaneously.

The Geant4 user interface manager

Geant4 provides a total interactivity tool as the user interface manager G4UIManager. It is a powerfull device that allow the user to change any parameter of the simulation entering command via a terminal. In addition to the modification possibilities included in Geant4 , one can implement its own commands in the user classes. This is explained at the end of this section. An example of building a messenger for the detector geometry can be found in **??**.

Optionnal user action classes

One can add optionnal classes to supervize the simulation at each level of its process. These are the run action, the event action, the tracking action etc. For the LEμSR simulation, we introduced the following LEMuSRAction classes:

- LEMuSREventAction
- LEMuSRTrackingAction
- LEMuSRSteppingAction

Messengers

We saw that it is possible to make the simulation interactive. This is done using messenger classes. One who wants to have a total interectivity shall build a messenger for each classs. The messenger can change the parameters of the gun, the particles, the geometry etc. It is linked to the G4UIdirectory, which registers all the commands. Commands are defined in the messenger.cc file and then add in a new command directory.

2.3 C++ language notion

2.3.1 Header files and source codes files

Geant4 is a C++ toolkit and one need to write both a header and a source files for each object class.

The header file contains the declaration of all methods and variables that are specific to the class that is being defined. One also have to specify all the other classes from which his class inherits. Finally, one should indicate if the methods and variables are public (can be seen and get their values modified by other methods)

or private (excusive appartenance to the class). Header files are located in the $LEMU/include directory.

The source file contains the code of each method. These files are located in the $LEMU/source directory.

2.3.2 A bit more about user classes: the inheritance philosophy

A base class (a class from which other class derive) can present so-called *virtual* methods. That means that these methods can be re-written in a daughter class definition: in C++ language, virtual methods can be overloaded. Virtual methods are the key of Geant4 framework modularity: a large amount of Geant4 classes can be personalized by the user. The latter just has to rewrite the virtual methods, making sure to keep their exact name because they might be called at other places of the framework. As an example, the method in charge of builing the geometry must be called Construct(), and the messengers' method in charge of modifying parameters must have the name SetNewValue().

Now let's have a more detailled description of the mandatory user classes we introduced before.

Chapter 3

Computing the geometry: the Detector class

3.1 The detector description

The description of this simulation of the LEμSR experiment begins at the trigger detector.

3.1.1 Experimental setting

After going through the trigger detector, muons enter an electrostatic Einzel lens which will accelerate them. Then a conical lens will focus the beam to the sample. The sample is mounted either on a cryostat, either on another holder. Instead of a sample. some experiments will use another multiple channel detector. This flexibility and all the other parameters must be part of the detector description using Geant4 .

3.1.2 Detector files

All informations about the detector are in the files LEMuSRDetectorConstruction.hh,cc. Before we looked at the code organization we have to define different kinds of volumes, which form a hierearchy in the Geant4 detector construction process.

Geant4 volumes hierarchy

There are three kinds of volumes in Geant4 , and one can arrange them in order to build a recursive hierarchy, or a tree, that defines the detector. They are organized as the creation process of a detector component.

- Base volume: it is the pure geometrical definition of a detector component. For example, a box, a cylinder, a polyhedra ... and the dimensions i.e. it is the component project on paper.

- Logical volume: it is the definition of the detector component obtained combining the geometric definition and the material, i.e. it is the component at the exit of the workshop.
- Physical volume: it is a logical volume that recieved a placement (center position vector and space transformation like rotation, translation ...) in a mother volume, i.e. it is the component introduced in the detector.

The mother volume of a physical volume is always a logical volume, expect for the main physical volume, which is the "World", the experimental hall volume. Therefore, a recursive hierarchy is possible. A physical volume C will be put in a logical volume B, at a certain position and with a certain orientation. The physical volume B will be the placement of the B logical volume (including the physical volume C) into a logical volume A, and so on.

Geant4 volumes attributes

Once one has defined his own geometry, one could set some specific attributes to the different volumes. These attributes can be graphical attributes, as the colors or the the drawing style, physical attribute, if one needs to set special cuts or step length to defined detectors, or attributes for simulation,like the enabling sensitive detector. We will see more about those latter attributes in section **??**.

3.1.3 The detector description: the code description

The LEMuSRDetectorConstruction class inherits from the G4UserDetectorDescription. That is, some functions *must* be overloaded by the user because they may be called by other methods of the framework. The main method of the detector description is the Construct() method. It is called by the G4RunManager during the initialization.

```
void G4RunManager::InitializeGeometry()
{
  if(!userDetector)
  {
    G4Exception
    ("G4RunManager::InitializeGeometry -
    G4VUserDetectorConstruction is not defined.");
  }

  if(verboseLevel>1)
  {G4cout << "userDetector->Construct() start."<< G4endl;}
  kernel->DefineWorldVolume(userDetector->Construct(),false);
  geometryInitialized = true;
}
```

The materials, the world and all subdetectors must be defined in this method. However, in order to simplify reading and further modifications, one may share the detector construction building other functions. That is what have been done, and one would read that the code consists of various functions as lemuCryo(), lemuMCP(), lemuLinse3() etc. The materials and attributes definitions have also been implemented in separate functions, MaterialsDefinition () and LoadAttributes respectively.

```
G4VPhysicalVolume* LEMuSRDetectorConstruction::Construct()
{
  return  lemuDetector();
}

G4VPhysicalVolume* LEMuSRDetectorConstruction::lemuDetector()
{


  // ++++++DEFINE THE MOTHER VOLUME:  THE LABORATOY++++++
  // solid
  G4double LABO_x = 2*m;
  G4double LABO_y = 2*m;
  G4double LABO_z = 2*m;

  LABO_box = new G4Box("World_box",LABO_x,LABO_y,LABO_z);
  // logical volume
  LABO_material = G4Material::GetMaterial("vacuum");
  lv_LABO  = new G4LogicalVolume(LABO_box,
                                 LABO_material,
                                 "lv_World",0,0,0);

  // physical volume
  pv_LABO  = new G4PVPlacement(0,// no rotation matrix
       G4ThreeVector(),// ()==(0,0,0)
       lv_LABO,// logical volume
       "pv_World",// name
       0, // no mother volume
       false, // always false
       0);
// !!! lv_LABO is the world logical,
// mother volume is 0 for the world!


  //SET VISUAL ATTRIBUTES
```

```
  lv_LABO->SetVisAttributes(G4VisAttributes::Invisible);

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


  //             END OF LABORATORY DEFINITION

  // &&&&&&&&&&  ADD THE OTHER DETECTOR VOLUMES  &&&&&&&&//



  // LOAD ATTRIBUTES AND USER LIMIT
  LoadAttributes();
  //-------------------------------


  lemuMCP2();

  lemuTrigger_Detector();
  lemuCGate();
  lemuLinse3();

  lemuANODE();
  lemuSCINT();

  return pv_LABO;

}
```

In this source listing we see that after the laboratory physical volume is built, other detector parts construction methods are called. Finally, one just has to return the laboratory physical volume.

After the muons decay, positrons are collected in scintillators for data analysis. The role of scintillators can be computed thanks to the *sensitive detector* facility. A detector that is made sensitive is able to store all datas about each particle that enters it. This is time saving because one would not have to build loops in order to check each particle at each step. This powerfull tool is the result of two classes, namely G4Hits and G4UserSensitiveDetector.

3.1.4 The Detector Messenger: modifying LEμSR geometry.

As seen before (ref), any class implemented can benefit of a messenger. We choosed to build a DetectorMessenger in order to change the geometry of the detector interactively. Hence, one is able to run the simulation and decide whether to

use the cryostat or the multiple channel detector. One can also modify the fields properties or have a planar cut visulization of the detector.

Building a messenger consists in three steps.

1. Declaring the interactive commands (usually in the header file)

```
#ifndef LEMuSRDetectorMessenger_h
#define LEMuSRDetectorMessenger_h 1

#include "G4ios.hh"
#include "globals.hh"
#include "G4UImessenger.hh"
#include "LEMuSRDetectorConstruction.hh"
#include "G4RunManager.hh"

#include "G4UIdirectory.hh"
#include "G4UIcmdWith3VectorAndUnit.hh"
#include "G4UIcmdWithADoubleAndUnit.hh"
#include "G4UIcmdWithAString.hh"
#include "G4UIcmdWithAnInteger.hh"
#include "G4UIcmdWithoutParameter.hh"
#include "G4UIcommand.hh"
#include "G4UImanager.hh"
#include "G4UIterminal.hh"
#include "G4UItcsh.hh"

class G4UIcommand;
class G4UIdirectory;
class G4UIcmdWithADoubleAndUnit;
class G4UIcmdWith3VectorAndUnit;
class G4UIcmdWithAnInteger;
class G4UIcmdWithAString;
class G4UIcmdWithoutParameter;
class LEMuSRDetectorConstruction; //modif

class LEMuSRDetectorMessenger : public G4UImessenger  {

public:
LEMuSRDetectorMessenger(LEMuSRDetectorConstruction*);
~LEMuSRDetectorMessenger();

public:
void SetNewValue(G4UIcommand* command, G4String newvalue);
```

```
//arguments
private:
LEMuSRDetectorConstruction* theDetector;

// commands
private:
G4UIdirectory*  DetMode;
G4UIcmdWithAString* SetMagField ;
G4VPhysicalVolume* newDetector;
};

#endif
```

It is important to notice that there are various kinds of user interactive commands: commands with a string, with an integer, without parameter ...

2. Creating a directory for these commands (in the constructor)

```
LEMuSRDetectorMessenger::LEMuSRDetectorMessenger
                         (LEMuSRDetectorConstruction
                          *lemuDetector)
{
  theDetector=lemuDetector;
  DetMode =  new G4UIdirectory("/Detector/");
  DetMode->SetGuidance("Set detector parameters");

  (...)

  SetDetVisualization  = new G4UIcmdWithAString
                             ("/Detector/View",this);
  SetDetVisualization->SetGuidance
("\n quarter: quarter cut view of the detector
  \n half: half cut view of the detector
  \n total: view of the total detector ");
  SetDetVisualization->SetParameterName("view",false);
  SetDetVisualization->SetDefaultValue("total");
  SetDetVisualization->AvailableForStates(G4State_PreInit,
                                          G4State_Idle);
}
```

The directory name and the command name in the terminal are defined. The user can also define some explanations of the command, which will be available entering the "help" command in the terminal. The directory and all the commands must be deleted in the destructor

```
LEMuSRDetectorMessenger::~LEMuSRDetectorMessenger()
{
  delete theDetector;
  delete DetMode;
  delete SetMagField;
  delete SetDetMode;
  delete SetDetVisualization;

}
```

3. Implementing the effect of each command (in the default SetNewValue() virtual method)

```
void LEMuSRDetectorMessenger::SetNewValue
                              (G4UIcommand* command,
                               G4String newvalue)
//MUST have the name  SetNewValue :
//inherited method from Messenger base class.
{
  G4UImanager* UI = G4UImanager::GetUIpointer();


  if(command == SetDetMode)
    {
      if(newvalue=="mcp")
{
  theDetector->mcdetector=1;
}
      else if(newvalue=="cryo")
{
  theDetector->mcdetector=0;
}
      else
{
  G4cout << "Unknown command: please check value."
         <<G4endl;
}

newDetector = theDetector->Construct();
G4RunManager::GetRunManager()
                     ->DefineWorldVolume(newDetector);
    }
```

```
  else if(command == SetMagField)
    {
      (...)
    }



  else if(command == SetDetVisualization)
    {
      (...)
    }

  else
    {
      G4cout << "Unknown command: please check value."
             <<G4endl;
    }
}
```

Each time the detector is modified, it is totally rebuilt and the run manager re-initiallized.

Chapter 4

How to detect muon decay particles? The Sensitive Detectors.

In the LEμSR simulation, some detector parts statistics may be modelized building sensitive detectors. These would collect every information needed about particles entering the corresponding detector. During an event, the ASensitiveDetector will pack all datas of a particle that enters A in an object called *hit*.

In a general case, sensitive detectors can be created for each logical volume of the detector. This is how we proceeded for each volume of the LEμSR simulation:

1. build a class for *Hits* and their collection
2. build a sensitive detector class
3. link the detector component to the sensitive detector class
4. allow changes in the detector messenger

4.1 The hits collection

A *hit* is an object that contains all information about a particle that enters a specific volume. It can be drawn on the visualization window, and printed into a file or a histogram. The Hit class defines the printing and drawing methods, as well as the variables one would stock in a hit (position, momentum, time of flight etc.). In addition, the Hit class contains the definition of the HitCollection class, which make possible to register all the hits. The G4THitsCollection class is a template used to build any hit colletion. One is encouraged to read some of the source files for more understanding.

4.2 The sensitive detector

The role of the sensitive detector is to fill the hits collection. The HitsCollection object is instanciated in the Initialize() method of the sensitive detector. When a hit is created, it is sent to the sensitive detector, which will act as a filter, or a HitsCollectionManager, deciding which hit has to be registered or not.

Let us have a look at some source listing. The outer scintillator's hit collection is defined as follow

```
LEMuSROScintSD::LEMuSROScintSD(G4String name)
:G4VSensitiveDetector(name)
{
  G4String HCname;
  collectionName.insert(HCname="OuterScintCollection");
 positionResolution = 5*mm;
}

LEMuSROScintSD::~LEMuSROScintSD(){;}

void LEMuSROScintSD::Initialize(G4HCofThisEvent* HCE)
{
  static int HCID = -1;
  ScintCollection = new LEMuSROScintHitsCollection
                    (SensitiveDetectorName,collectionName[0]);
  if(HCID<0) { HCID = GetCollectionID(0); }
  HCE->AddHitsCollection(HCID,ScintCollection);
}
```

It is very important to know that the hits collection object must be build only once. This will be more discussed in section **??**. Once this is done the next step is filling the hit collection with hits. In this example, a hit pointer is created, filled with desired parameters and finally added to the hits collection. All of these steps are in the ProcessHits() method.

```
G4bool LEMuSROScintSD::ProcessHits(G4Step* aStep,
                                   G4TouchableHistory*)
{
  LEMuSROScintHit* aHit;
  int nHit = ScintCollection->entries();
  // Get datas
  //a
  G4String p_name = aStep->GetTrack()->GetDefinition()
                                     ->GetParticleName();
  G4double spin=  aStep->GetTrack()->GetDefinition()
                                   ->GetPDGSpin();
```

```
  //b
  G4ThreeVector hitpos = aStep->GetPreStepPoint()
                              ->GetPosition(); // position
  G4ThreeVector hitmom = aStep->GetPreStepPoint()
                              ->GetMomentum(); // momentum
  //c
  // time since track creation
  G4double tof       = aStep->GetPreStepPoint()
                             ->GetLocalTime();
  // time since the event creation
  G4double globaltime= aStep->GetPreStepPoint()
                             ->GetGlobalTime();
  // proper time of the particle
  G4double proptime  = aStep->GetPreStepPoint()
                             ->GetProperTime();
  //d
  G4double edep      = aStep->GetTotalEnergyDeposit();

  // Define Hit
  aHit = new LEMuSROScintHit();

  //+++++++++++++++ set hit values _____________
  aHit->SetParticleName(p_name);
  aHit->SetSpin(spin);

  aHit->SetMomentum( hitmom );
  aHit->SetPosition( hitpos );

  aHit->SetTimeOfFlight( tof);
  aHit->SetEnergyDeposition( edep );

  ScintCollection->insert( aHit );
  aHit->Print();
  //  aHit->print("Statistics/SCOS.Hits");
  aHit->Draw();

  return true;
}
```

4.3 Enabling SensitiveDetectors; User Interaction

The sensitive detector is enabled in the detector's construction method as following

1. Create the sensitive detector: the Sensitive Detector Manager registers the sensitive detector.

```
G4SDManager* SDMGR = G4SDManager::GetSDMpointer();

G4String iScintName = "/LEMuSR/Scintillator/Inner";

iScintSD = new LEMuSRScintSD(iScintName);

SDMGR->AddNewDetector(iScintSD);
```

2. Set the detector: the sensitive detector is assigned to the concerned detector's logical volume.

```
lv_SCIS->SetSensitiveDetector(iScintSD);
```

4.4 Risks of interactivity

Thanks to messengers classes, user interactivity is possible and one can modify the detector setup in the terminal before a run (cf.section **??**). As we saw that the hits collection must be build only once for a run, one should care about not to build twice a same collection while builiding a new detector. The following source listing show the implementation for the LEμSR simulation

```
  //SENSITIVE DETECTOR
  if(mcdetector==0)    //then use cryostat
    {
      lemuCRYO();
      if(cryo==0)
{
  G4SDManager* SDMGR = G4SDManager::GetSDMpointer();
  G4String CryoName = "/LEMuSR/Cryo_sample";
  CryoSD = new LEMuSRCryoSD(CryoName);
         SDMGR->AddNewDetector(CryoSD);
  cryo=1;
}
lv_SAPH->SetSensitiveDetector(CryoSD);
    }

  else if(mcdetector==1) //then use multiple channel detector
    {
      lemuMCPdet();
      if(mcp==0)
```

```
{
  G4SDManager* SDMGR = G4SDManager::GetSDMpointer();
  G4String McpName = "/LEMuSR/MCP";
  McpSD = new LEMuSRMcpSD(McpName);
         SDMGR->AddNewDetector(McpSD);
  mcp=1;
}
        lv_DMCP->SetSensitiveDetector(McpSD);
     }
```

mcp and cryo will act as a boolean variables which will notify the presence of a HitCollection for the mcp or the cryo. They are set to zero in the constructor. By default, mcdetector (boolean to indicates wheter one uses the mcp setup or not) is also set to zero and the simulation runs with a sample and its cryostat.

- Initially, the detector is built with a cryo setup. cryo is set to one after the first initialization. (cf main file)

  ```
  int main()
  {
    (...)
    runManager ->SetUserInitialization( lemuDetector );
    (...)
  }
  ```

- The user can choose to change for the mcp setup: as mcp==0, a hit collection is created for the mcp entering the command Detector/Mode mcp

- Then the user may run again the cryo setup. In this case, as cryo==1 the hit collection is known as already built, it won't be recreated.

Chapter 5

The Primary Generator Action Class

The primary generator action is the class where the user configures the initial vertex, i.e. where he sets up the initial conditions of a new event.

In the particular method named GeneratePrimaries(), the new event receives objects called primary particles, which define particles types and all the parameters like their positions, momenta, mass ...

5.1 GeneratePrimaries()

The primary particles can be generated using algorithms that recreate collision vertice, or using a object called particle gun, in order to recreates a beam. The latter case is the one of the LEMuSRsimulation. We will not detail collision vertice, but only mention that a well known vertex generator one can find is the Pythia algorithm, which is used in the CERN LHCb simulation.

When one uses a particle gun to generate primary particles, one has to specify the gun position, the number of particles to generate per event, and all the particle parameters.

The following source code is the LEMuSRimplementation of the Primary Generator Action GeneratePrimaries() method.

```
void LEMuSRPrimaryGeneratorAction::
                          GeneratePrimaries(G4Event* anEvent)
{
  G4RandGauss* iRndGauss =new G4RandGauss(theEngine,20.,10.);
  // the random energy
  energy=-1;
  do
  { rndenergy = iRndGauss->shoot(20.0, 0.50);
    energy= rndenergy*keV;//default unit is MeV
  }while(energy<=0);
```

```
  // the random time of the decay for muons
  G4double tau=0.000002197; // the muon mean lifetime
  G4double rnddecaytime = - tau*log(1-G4UniformRand());
  decaytime = rnddecaytime*s;
  // Get positive muons
  decaytime = tau*s;
  energy    = 20*keV;
  G4ParticleTable* particleTable=G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* particle=particleTable->FindParticle("mu+");
  lemuParticleGun->SetParticleDefinition(particle);
  lemuParticleGun->SetParticleEnergy(energy);
  lemuParticleGun->SetDecayTime(decaytime);
  lemuParticleGun->SetNumberOfParticles(1);
  (...)
  lemuParticleGun->SetParticlePolarization(G4ThreeVector(1.,0.,0.));
  (...scan...)
  lemuParticleGun->GeneratePrimaryVertex(anEvent);
}
```

The scan integer allows to switch the gun scanning mode. We the scan is enabled, one may set to one the number of particle per event and launch as many events as the bins in the scan plan.

The GeneratPrimaries() method also contains the random implementation of the energy or he lifetime of a particle. Moreover, initial positions and momenta can also be generated randomly.

As it appears clearly now, the primary generator action is just an intermediate and all the actions are performed by the particle gun.

5.2 The particle gun

We will see in this section how the particle gun is used to initialize the new event.

5.2.1 GeneratePrimaryVertex(G4Event*)

Here is the main method of the event initialization:

```
void G4ParticleGun::GeneratePrimaryVertex(G4Event* evt)
{
  if(particle_definition==0) return;
  // create a new vertex
  G4PrimaryVertex* vertex =
  new G4PrimaryVertex(particle_position,particle_time);
  // create new primaries and set them to the vertex
  G4double mass =  particle_definition->GetPDGMass();
  G4double energy = particle_energy + mass;
  G4double pmom = sqrt(energy*energy-mass*mass);
  G4double px = pmom*particle_momentum_direction.x();
  G4double py = pmom*particle_momentum_direction.y();
```

```
  G4double pz = pmom*particle_momentum_direction.z();
  for( G4int i=0; i<NumberOfParticlesToBeGenerated; i++ )
  {
    G4PrimaryParticle* particle =
    new G4PrimaryParticle(particle_definition,px,py,pz);
    particle->SetMass( mass );
    particle->SetCharge( particle_charge );
    particle->SetPolarization(particle_polarization.x(),
    particle_polarization.y(),
    particle_polarization.z());
    particle->SetProperTime(decaytime);
    vertex->SetPrimary( particle );
  }
  evt->AddPrimaryVertex( vertex );
}
```

This action can be summed up in three step:

1. Setting up primary particles properties
2. Adding primary particles to primary vertex object
3. Adding primary vertex to the new event.

Once the primary vertex is generated, the gun has no more role during the whole simulation.

5.2.2 Scnaning

We mentionned before that is possible to perform a scan of a region with the particle gun. This is very important when one wants to generate parallel beams, in order to collect uniform information of the geometry.The following listing completes the previous one

```
(...)
  //----------------- SCAN>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  if(scan==1)
  {
    //scan: position of the gun
    if(m_countery<=m_nbysteps)
    {
      if(m_counterx<=m_nbxsteps)
      {
px = -m_xrange+m_counterx*(2*m_xrange)/m_nbxsteps;
py = -m_yrange+m_countery*(2*m_yrange)/m_nbysteps;
m_counterx++;
      }
      if(m_counterx==m_nbxsteps)
      {
m_countery++;
```

```
m_counterx=0;
      }
    }
    else if(m_countery>m_nbysteps)
    {
      px=0;
      py=0;
    }

  }
  lemuParticleGun->SetParticlePosition
                    (G4ThreeVector(px*cm,py*cm,-115*cm));
  lemuParticleGun->GeneratePrimaryVertex(anEvent);
```

5.2.3 Primary Particles and Dynamic Particles

It is now very important to introduce a fundamental difference between Primary Particles objects and Dynamic Particles objects.

If the first one are generated by the GeneratePrimaries() method, using a gun or a vertex algorithm, and then given as initial conditions to the new Event, *they are not* the ones that are simulated during the simulation. In order to be simulated, a Primary Particle must be converted into a Dynamic Particle.

This becomes clear when we think about a collision vertex, or a spontaneous decay : all the primary particle do not have to be tracked, and the detector may not interact with all of them. This is translated in the running simulation by the conversion or not of the primary particles.

Chapter 6

The Physics List

6.1 The G4VProcess Class

6.1.1 Description

The G4VProcess class is the virtual class for any physics process. All the important methods are declared here and may be implemented by the user when he defines a new process. Usually, the user does not have to implement new processes because Geant4 physics interaction library is very large. All that one should do is to assign specific processes to specific particles in the PhysicsList. This is fast in general because many Geant4 examples create complete PhysicsList files.

In the next paragraph we will see the extent of physics processes that feature in Geant4 . But let's have a look at the principal methods.

Security bool In order to avoid mistakes and crashes because of processes assigned to the wrong particle, a boolean method called IsApplicable() return true only if the process is applicable to the regarded particle, as shown by its name. This method may testthe particle types and its properties, like its spin, energy ...

Interaction lengths Three methods provide the interaction lengths whether the particle is at rest, at the end of a step or along a step.

Process execution The process will be executed along a step, after a step or at rest according to which of the three "DoIt" methods (PostStepDoIt(), AlongStepDoIt(), AtRestDoIt()) is called. Each of these method returns a G4VParticleChange* object which contains the changes to perform on the track (momentum change, daughter particles...)

6.1.2 Different kinds of processes

Geant4 offers a large library of physical interactions and their associated cross sections per particle. Thanks to this, one may not have to compute his own processes

but to build a UserPhysicsList class (mandatory) with all the interaction that are relevant for his simulation.

It is however very usefull to look at the huge possibilities Geant4 offers in the Geant4 /source/processes directory. Electromagnetic, hadronic interactions, decay processes and more are implemented in the subdirectories. Specific processes can also be found in the Geant4 /source/particles/management directory.

6.2 Configuring the physics list

Building a physics list consits in three steps:

1. setting the particles that may be simulated
2. setting the physical processes
3. setting cuts

The physics list class inherits from the virtual G4VUserPhysicsList class. The three stepsmethods enumerated must be implemented by the user.

As seen in the case of the detector construction, a messenger can be built in order to change these settings from the simulation terminal.

6.2.1 Setting particle types

The G4VUserPhysicsList::ConstructParticle() contains the list of the particles the user wants in his simulation. The following listing is an example that suits perfectly for LEμSR simulation:

```
void PhysicsList::ConstructParticle()
{
  G4Electron::ElectronDefinition();
  G4Positron::PositronDefinition();
  G4NeutrinoE::NeutrinoEDefinition();
  G4AntiNeutrinoE::AntiNeutrinoEDefinition();
  G4MuonPlus::MuonPlusDefinition();
  G4MuonMinus::MuonMinusDefinition();
  G4NeutrinoMu::NeutrinoMuDefinition();
  G4AntiNeutrinoMu::AntiNeutrinoMuDefinition();

  G4DecayTable* MuonPlusDecayTable = new G4DecayTable();
  MuonPlusDecayTable->Insert(new G4MuonDecayChannel("mu+",1.00));
  G4MuonPlus::MuonPlusDefinition()->
                      SetDecayTable(MuonPlusDecayTable);

  G4DecayTable* MuonMinusDecayTable = new G4DecayTable();
  MuonMinusDecayTable ->
              Insert(new G4MuonDecayChannelWithSpin("mu-",1.00));
```

```
  G4MuonMinus::MuonMinusDefinition() ->
                      SetDecayTable(MuonMinusDecayTable);
}
```

One may read that a decay table is built. This table registers all different decay channels of the particle and orders them. Decay channels are considered as *intrinsecs properties* of particle, that is why they are built in the ConstrucParticles() method.

6.2.2 Setting interactions and processes

To construct procesess and register them to particles the user hay to write a G4VUserPhysicsList::constructPhysics() method:

```
{
  // Define transportation process

  AddTransportation();

  theDecayProcess = new G4DecayWithSpin();

  G4ProcessManager* pManager;
  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    pManager = particle->GetProcessManager();

    if (theDecayProcess->IsApplicable(*particle)) {
      pManager->AddProcess(theDecayProcess);
      pManager ->SetProcessOrderingToLast(theDecayProcess, idxAtRest);
    }
  }

  theMuPlusIonisation = new G4MuIonisation();
  theMuPlusMultipleScattering = new G4MultipleScattering();
  theMuPlusBremsstrahlung=new G4MuBremsstrahlung();
  theMuPlusPairProduction= new G4MuPairProduction();

  // Muon Plus Physics
  pManager = G4MuonPlus::MuonPlus()->GetProcessManager();
  pManager->AddProcess(theMuPlusMultipleScattering,-1,  1, 1);
  pManager->AddProcess(theMuPlusIonisation,        -1,  2, 2);
  pManager->AddProcess(theMuPlusBremsstrahlung,    -1,  3, 3);
  pManager->AddProcess(theMuPlusPairProduction,    -1,  4, 4);

  (...)
}
```

Here, one sees how the decay process is introduced. For each registered particle that is meant to decay, the decay process is added. It is to see that the decay process *does not* means the decay channel, which has been built precedently.

The process manager finally add processes to particles through the AddProcess method. The three last numbers in

```
pManager$->$AddProcess(theMuPlusIonisation, x, y, z);
```

specify the order of execution of the process at rest (x), along the step (y), after the step (z). If one of this number is negative, the process will not be executed. Hence one can read that the theMuPlusBremsstrahlung process will not be called if the muon is at rest, and will be called in third position in the two other cases.

The AddTransportation() method is important to enable particles transportation in the detector. It must be called only once.

6.2.3 Setting Cuts

Cuts are important when the user wants to focus on particles energy range.They can be set in the G4VUserPhysicsList::SetCuts(). Usually the default cuts are used.

```
PhysicsList::SetCuts()
{
  //  " G4VUserPhysicsList::SetCutsWithDefault" method sets
  //   the default cut value for all particle types
  SetCutsWithDefault();

  SetCutValue(0., "e+");
  SetCutValue(0., "mu+");
}
```

say a bit more about cut units

6.3 Modular Physics List

G4VModularPhysicsList is a class that inherits from G4VPhysicsList. It makes the physics list code easier to change. Its particularitys is the RegisterPhysics method, which call G4VPhysicsConstructor objects.

A G4VPhysicsConstructor is nothing more than a physics list (whitout the SetCut() method) that constructs the physics for a specifiy kind of particle.

It is not necessary at all to choose this implementation of the physics list. However both were tried for LEμSR simulation. Here is a short listing showing the use of this G4VModularPhysicsList.

```
#include "LEMuSRGeneralPhysics.hh"
```

```
#include "LEMuSREMPhysics.hh"
#include "LEMuSRMuonPhysics.hh"
#include "LEMuSRHadronPhysics.hh"
#include "LEMuSRIonPhysics.hh"

LEMuSRPhysicsList::LEMuSRPhysicsList():G4VModularPhysicsList()
{
  defaultCutValue = 1.0*mm;
  // General Physics
  RegisterPhysics( new LEMuSRGeneralPhysics("general") );
  // EM Physics
  RegisterPhysics( new LEMuSREMPhysics("standard EM"));
  // Muon Physics
  RegisterPhysics(  new LEMuSRMuonPhysics("muon"));
  // Hadron Physics
  RegisterPhysics(  new LEMuSRHadronPhysics("hadron"));
  // Ion Physics
  RegisterPhysics( new LEMuSRIonPhysics("ion"));
}

LEMuSRPhysicsList::~LEMuSRPhysicsList()
{
}

void LEMuSRPhysicsList::SetCuts()
{
  SetCutsWithDefault();
}
```

6.3.1 Muon spin precession and decay

The spin precession at rest and the porarized muon decay did not feature Geant4 package when we built the LEμSR simulation. Thus, they were implemented and passed the asymmetry test of chapter **??**. One may look at the relative files and compare them to the geant code for more understanding.

Chapter 7

Introducing Electromagnetic Fields

In this chapter we will see which are the classes one needs to introduce a field in a simulation. Geant4 offers virtual classes for the fields and the user can introduce not only electromagnetic but any kind of field, just by implementing two classes:

1. The field definition class which would provide field values at different locations
2. The field equation class which would provide the derivatives of parameters such position, momentum, etc.

Then the user only has to link these classes to the concerned detector component.

Geant4 code is initially configured for magnetic fields. In the section **??** we will see how to link a magnetic field to a detector component, and in section **??** we will detail the introduction of an electric field given by a field map.

7.1 Inserting a magnetic field

The magnetic field in the mcp region has been simply modelized by a uniform field as in the source listing below. The way one adds a field is almost always the same and can be summed up this way.

First, one has to declare the field type and the field manager. The magnetic field in the mcp region is modelized by a 100 Gauss uniform field:

```
// magnetic field
 G4UniformMagField* mcField = new
                    G4UniformMagField(G4ThreeVector(0.,0.,100.*gauss));
 G4FieldManager* mcFieldMgr =
     // new G4FieldManager(); // 1
     // G4TransportationManager::GetTransportationManager()
                                 ->GetFieldManager(); // 2
```

We see that there is two ways of defining the field manager: the first one defines a local field manager, the second one defines a global field manager.

Then one should choose an appropriate equation of motion and a solver (stepper) and associate both to a chord finder. The G4ChordFinder's role is to determine the best straight line approximation of the particle's trajectory during one step. To get more precise values, one may reduce the maximum step length or change the accuracy parameter.

```
LEMuSRMag_SpinEqRhs *Mag_SpinEqRhs;
G4MagIntegratorStepper *pStepper;

Mag_SpinEqRhs = new LEMuSRMag_SpinEqRhs(mcField);
pStepper      = new G4SimpleHeum( Mag_SpinEqRhs,12 );
G4ChordFinder* pChordFinder = new
               G4ChordFinder(mcField,0.01* mm, pStepper );
```

The last step is the association of the field manager to the detector component

```
mcFieldMgr->SetDetectorField(mcField);
mcFieldMgr->SetChordFinder(pChordFinder);
lv_MCPV  = new G4LogicalVolume(MCPV_tube,Vacuum,"lv_MCPV",mcFieldMgr,0,0);
```

This is the procedure anyone would follow to introdue a field in the geometry. Only the field and the equation have to be personalized, the stepper, the chord finder and the field manager.

We are now going to look at the implementation of the electric field. For this we needed to build a electric field daughter class as well as equation of motion in the electric field.

7.2 Inserting an electric field

This section presents the insertion of an electric field in the third lens region, using a field map.

7.2.1 The electric field class

The field map was generated with the femlab3.0a solver, which output three files (one per field coordinate) with the position and the corresponding field value. All one has to do after this is reading the files and store the datas in an multiple dimensionnal array.

The reading of a field map is done in the constructor method. Two contructors have been created, one for the case of a field map in three parts, one for the case of a single file field map. Here is the listing of the single field map file constructor

```
LEMuSRElectricField::LEMuSRElectricField(const char* file,
```

```
     G4double Offset,
     G4double nbx,
     G4double nby,
     G4double nbz)
{
 // open files for reading
  std::ifstream fmap(file);
  // Ignore first blank line
  char buffer[256];
  fmap.getline(buffer,256);
  zOffset = Offset;
  nx=nbx;
  ny=nby;
  nz=nbz;
  int ix, iy, iz;
  // Read in the data
  double xval,yval,zval,bx,by,bz;
  for (ix=0; ix<nx; ix++) {
    for (iy=0; iy<ny; iy++) {
      for (iz=0; iz<nz; iz++) {
        fmap >> xval >> yval >> zval >>bx >>by >>bz;
        xField[ix][iy][iz] = bx*volt/meter;
yField[ix][iy][iz] = by*volt/meter;
zField[ix][iy][iz] = bz*volt/meter;
      }
    }
  }
  fmap.close();
  maxx = xval;
  maxy = yval;
  maxz = zval;
  minx =-maxx;
  miny =-maxy;
  minz =-maxz;
}
```

The most important method of any field class is the GetFieldValue(G4double *point, G4double *E), where the first argument is the position-time vector and the second is the array where the user wants the field to be copied. In the case of a uniform field, this method will always assing the E array to the same value (which is the one set in the detector construction, like for the uniform magnetic field listed upper). In the case of a non uniform field, the user should return an analytical expression of the field with respect to the position, or, if he uses a field map, the user should build an interpolation method to approximate the field value at one position according to the field values at the neighbourgh grids. The interpolation algorithm was inspired from n existing example from Geant4 called "Purging Magnet"

```
void LEMuSRElectricField::GetFieldValue(const G4double point[4],
                                        G4double *Bfield ) const
{
  G4double x = point[0]/100;
  G4double y = point[1]/100;
  G4double z = (point[2] - zOffset)/100;
  //check that the point is within the defined region
  if ( x>=minx && x<=maxx && y>=miny &&
       y<=maxy && z>=minz && z<=maxz )
  {
    // Position of given point within region, normalized to the range
    // [0,1]
    G4double xfraction = (x - minx) / (2*maxx);
    G4double yfraction = (y - miny) / (2*maxy);
    G4double zfraction = (z - minz) / (2*maxz);
    // Need addresses of these to pass to modf below.
    // modf uses its second argument as an OUTPUT argument.
    G4double xdindex, ydindex, zdindex;
    // Position of the point within the cuboid defined by the
    // nearest surrounding tabulated points
    G4double xlocal = ( modf(xfraction*(nx-1), &xdindex));
    G4double ylocal = ( modf(yfraction*(ny-1), &ydindex));
    G4double zlocal = ( modf(zfraction*(nz-1), &zdindex));
    // The indices of the nearest tabulated point whose coordinates
    // are all less than those of the given point
    int xindex = static_cast<int>(xdindex);
    int yindex = static_cast<int>(ydindex);
    int zindex = static_cast<int>(zdindex);

    Bfield[0] =
    xField[xindex  ][yindex  ][zindex  ] * (1-xlocal) * (1-ylocal) * (1-zlocal) +
    xField[xindex  ][yindex  ][zindex+1] * (1-xlocal) * (1-ylocal) *    zlocal  +
    xField[xindex  ][yindex+1][zindex  ] * (1-xlocal) *    ylocal  * (1-zlocal) +
    xField[xindex  ][yindex+1][zindex+1] * (1-xlocal) *    ylocal  *    zlocal  +
    xField[xindex+1][yindex  ][zindex  ] *    xlocal  * (1-ylocal) * (1-zlocal) +
    xField[xindex+1][yindex  ][zindex+1] *    xlocal  * (1-ylocal) *    zlocal  +
    xField[xindex+1][yindex+1][zindex  ] *    xlocal  *    ylocal  * (1-zlocal) +
    xField[xindex+1][yindex+1][zindex+1] *    xlocal  *    ylocal  *    zlocal ;
    Bfield[1] =
    yField[xindex  ][yindex  ][zindex  ] * (1-xlocal) * (1-ylocal) * (1-zlocal) +
    yField[xindex  ][yindex  ][zindex+1] * (1-xlocal) * (1-ylocal) *    zlocal  +
    yField[xindex  ][yindex+1][zindex  ] * (1-xlocal) *    ylocal  * (1-zlocal) +
    yField[xindex  ][yindex+1][zindex+1] * (1-xlocal) *    ylocal  *    zlocal  +
    yField[xindex+1][yindex  ][zindex  ] *    xlocal  * (1-ylocal) * (1-zlocal) +
    yField[xindex+1][yindex  ][zindex+1] *    xlocal  * (1-ylocal) *    zlocal  +
    yField[xindex+1][yindex+1][zindex  ] *    xlocal  *    ylocal  * (1-zlocal) +
    yField[xindex+1][yindex+1][zindex+1] *    xlocal  *    ylocal  *    zlocal ;
    Bfield[2] =
    zField[xindex  ][yindex  ][zindex  ] * (1-xlocal) * (1-ylocal) * (1-zlocal) +
```

```
    zField[xindex  ][yindex  ][zindex+1] * (1-xlocal) * (1-ylocal) *    zlocal  +
    zField[xindex  ][yindex+1][zindex  ] * (1-xlocal) *    ylocal  * (1-zlocal) +
    zField[xindex  ][yindex+1][zindex+1] * (1-xlocal) *    ylocal  *    zlocal  +
    zField[xindex+1][yindex  ][zindex  ] *    xlocal  * (1-ylocal) * (1-zlocal) +
    zField[xindex+1][yindex  ][zindex+1] *    xlocal  * (1-ylocal) *    zlocal  +
    zField[xindex+1][yindex+1][zindex  ] *    xlocal  *    ylocal  * (1-zlocal) +
    zField[xindex+1][yindex+1][zindex+1] *    xlocal  *    ylocal  *    zlocal ;

  } else {
    Bfield[0] = 0.0;
    Bfield[1] = 0.0;
    Bfield[2] = 0.0;
  }
}
```

7.2.2 The equation of motion in an electric field

According to the electrostatic equation

$$\frac{dp}{dt} = qE$$

it is easy to get the derivatives of a particle's momentum once one got the field value at its position.

However, Geant4 developpers choosed to integrate over the path, and then needed the expression of the derivative along distance s. This leads to the following expressions:

$$\frac{d\vec{x}}{ds} = \frac{\vec{p}}{V}$$

$$\frac{d\vec{p}}{ds} = q\vec{E}\frac{1}{V}$$

where the velocity V is equal to $\frac{m}{||\vec{p}||}$. The drivers would then use these expression and combine them to the chord length in order to process a step in the field.

It is not the aim to describe the whole field integration, it can be easily accessed reading the code of related methods. We just mention here that the ChordFinder usually manipulates FieldTrack objects, which are nothing else than an array containing position, momentum and length of the chord.

Chapter 8

First Run and output

What happens during a run? It may be important for one to have a clear idea of when and where ones methods are called during a simulation. This is really time saving when the code contains some "if" loops, which sometimes have to be well positionned in the code. Geant4 code is light and well architectured, and to facilitate the desire one may have to read the code and find some interesting sources we decided to draw a sketch of the main Geant4 method that are called during a smiple run.

8.1 Running a simulation

8.1.1 beamOn

When a terminal UI-session is started, one can launch the simulation entering the command /run/beamOn, followed by the number of events, a macro file name and the number of event to run before the macro starts. This commands correspond to the G4RunMessenger beamOnCmd command, which calls the G4RunManager beamOn() method, which is listed below:

```
void G4RunManager::BeamOn(G4int n_event,
                          const char* macroFile,
                          G4int n_select)
{
  G4bool cond = ConfirmBeamOnCondition();
  if(cond)
  {
    numberOfEventToBeProcessed = n_event;
    RunInitialization();
    if(n_event>0) DoEventLoop(n_event,macroFile,n_select);
    RunTermination();
  }
}
```

The boolean condition is here to check if the mandatory classes are well defined. If this condition is verified, the RunInitialization() method will initialize the run parameters, and the DoEventLoop(..) method will be executed. Then the Run terminates.

We see that the method in charge of the simulation is this DoEventLoop() method, which will call the creation of a new event and send it to the event manager for simulation and finally register it:

```
void G4RunManager::DoEventLoop(G4int n_event,
                               const char* macroFile,
                               G4int n_select)
{
(...)
// Event loop
  G4int i_event;
  for( i_event=0; i_event<n_event; i_event++ )
  {
    currentEvent = GenerateEvent(i_event);
    eventManager->ProcessOneEvent(currentEvent);
    AnalyzeEvent(currentEvent);
    if(i_event<n_select) G4UImanager::GetUIpointer()->ApplyCommand(msg);
    StackPreviousEvent(currentEvent);
    currentEvent = 0;
    if(runAborted) break;
  }
(...)
}
```

The GenerateEvent() method calls the GeneratPrimaries() method of the UserPrimaryGeneratorAction. The event manager is a member of G4RunManager and is instanciated in the constructor. Let us see how it simulates the whole event from the new event generated.

8.1.2 The Event Manager

Directly after a new event is created for simulation, the ProcessOneEvent() method from the event manager is called. This method directly calls the DoProcessing() method after having reinitialized the track identification numbers. In the following listing, we simply kept the most important methods called by this DoProcessing(*anEvent) method

```
void G4EventManager::DoProcessing(G4Event* anEvent)
{
(...)
  G4Navigator* navigator =
  G4TransportationManager::GetTransportationManager()->
                           GetNavigatorForTracking();
```

```
 (...)
  trackContainer->PrepareNewEvent();
(...)
  if(sdManager)
  { currentEvent->SetHCofThisEvent(sdManager->PrepareNewEvent()); }
(...)
  if(userEventAction) userEventAction->BeginOfEventAction(currentEvent);
(...)
   trackManager->ProcessOneTrack( track );
(...)
    aTrajectory = trackManager->GimmeTrajectory();
(...)
}
```

We notice that the sensitive detector manager is called here to initialize the hit collections, as well as the user event action BeginOfEventAction() method, which set everything the user wants to execute at the beginning of an event.Then the trackManager is called to simulate a track.

8.1.3 The Tracking Manager

The hierarchy continues, the trackManager calls the user pre-tracking action initialize steps and send them to the stepping manager

```
////////////////////////////////////////////////////////////////
void G4TrackingManager::ProcessOneTrack(G4Track* apValueG4Track)
////////////////////////////////////////////////////////////////
{
(...)
  // Give SteppingManger the pointer to the track which will be tracked
  fpSteppingManager->SetInitialStep(fpTrack);
  // Pre tracking user intervention process.
  fpTrajectory = 0;
  if( fpUserTrackingAction != NULL ) {
     fpUserTrackingAction->PreUserTrackingAction(fpTrack);
  }
  // Give SteppingManger the maxmimum number of processes
  fpSteppingManager->GetProcessNumber();
  // Give track the pointer to the Step
  fpTrack->SetStep(fpSteppingManager->GetStep());
  // Inform beginning of tracking to physics processes
  fpTrack->GetDefinition()->GetProcessManager()->StartTracking();
  // Track the particle Step-by-Step while it is alive
  G4StepStatus stepStatus;
  while( (fpTrack->GetTrackStatus() == fAlive) ||
         (fpTrack->GetTrackStatus() == fStopButAlive) ){
    fpTrack->IncrementCurrentStepNumber();
    stepStatus = fpSteppingManager->Stepping();
(...)
```

```
    if(EventIsAborted) {
      fpTrack->SetTrackStatus( fKillTrackAndSecondaries );
    }
  }
  // Inform end of tracking to physics processes
  fpTrack->GetDefinition()->GetProcessManager()->EndTracking();
  // Post tracking user intervention process.
  if( fpUserTrackingAction != NULL ) {
     fpUserTrackingAction->PostUserTrackingAction(fpTrack);
  }
(...)
  }
}
```

8.1.4 The Stepping Manager

The Stepping manager Stepping() method is built in four step. After an initialization, the processes at rest are called. Then are the ones along step and after a step, and a final action updates the track, fills the sensitive detectors and performs final updates.

```
////////////////////////////////////////
G4StepStatus G4SteppingManager::Stepping()
////////////////////////////////////////
{
//--------
// Prelude
//--------
(...)
//-----------------
// AtRest Processes
//-----------------
   if( fTrack->GetTrackStatus() == fStopButAlive )
   {
        InvokeAtRestDoItProcs();
    }
(...)
//---------------------------------
// AlongStep and PostStep Processes
//---------------------------------
   else{
     (...)
     InvokeAlongStepDoItProcs();
     fStep->UpdateTrack();
     (...)
     InvokePostStepDoItProcs();
(...)
   }
//-------
```

```
// Finale
//-------
// Update 'TrackLength' and remeber the Step length of the current Step
   fTrack->AddTrackLength(fStep->GetStepLength());
   fPreviousStepSize = fStep->GetStepLength();
// Send G4Step information to Hit/Dig if the volume is sensitive
   fCurrentVolume = fStep->GetPreStepPoint()->GetPhysicalVolume();
   StepControlFlag =  fStep->GetControlFlag();
   if( fCurrentVolume != 0 && StepControlFlag != AvoidHitInvocation) {
      fSensitive = fCurrentVolume->GetLogicalVolume()->
                                   GetSensitiveDetector();
      if( fSensitive != 0 ) {
        fSensitive->Hit(fStep);
      }
   }

// User intervention process.
   fStep->SetTrack(fTrack);
   if( fUserSteppingAction != NULL ) {
      fUserSteppingAction->UserSteppingAction(fStep);
   }

// Stepping process finish. Return the value of the StepStatus.
   return fStepStatus;
}
```

With this method, we have drawn the ground structure of the simulation. The steps are updated through the InvokeProcess() methods. The next section will introduce the process method, and in particular the transportation process, which may take into account the presence of field in the geometry.

Chapter 9

LEμSR Preliminary Asymmetry Test

9.1 Theoretical Background

As an introduction for the test of the asymmetry, we recall here some characteristics of the muon decay, especially the distribution of the positron emission direction with respect to the muon spin. We will then be able to predict some results and this would guide us in the verification of muons decay implementation.

9.1.1 Spin precession and asymmetry

Two effects can be observed if the simulation is well implemented.

First, we saw that in a muon decay the positron is more emitted in the muon spin direction. This leads to an *asymmetry* of the positron emission direction.

The second interesting effect is that in presence of a magnetic field, the muon spin precesses around the field's direction with the frequency

$$\omega = \gamma_\mu B$$

where

$$\gamma_\mu = f(1+a)$$

with $f = 8.5062e+7 * \frac{\text{rad}}{\text{s*kilogauss}}$ and the anomaly $a = 1.165922e-3$. For a magnetic field $B = 100$gauss, we then expect a period of about 740 nanoseconds.

Therefore, if one observes the number of positrons emitted in a certain direction along time, the muon laying in a magnetic field, one would notice that not only it decreases exponentially with the time-lifetime ratio, but also that it oscillates according to the spin precession frequency with a determined period $T = 7.38662$sec*kilogauss.

Hence, the number of positrons detected in a fixed direction is given by

$$N = N_0 e^{-\frac{t}{\tau}} (1 + A\cos(\omega t + \phi)$$

if the muon is in a magnetic field, and by

$$N = N_0 e^{-\frac{t}{\tau}}(1 + A)$$

if there is no magnetic field. τ is the lifetime of the muon, A is the asymmetry. It can be evaluated checking the front/back counts difference when the field $\vec{B} = \vec{0}$. In that case,

$$A = \frac{F - B}{F + B}$$

where F and B represent the total counts respectively in the forward and backward detectors according to the muon spin.

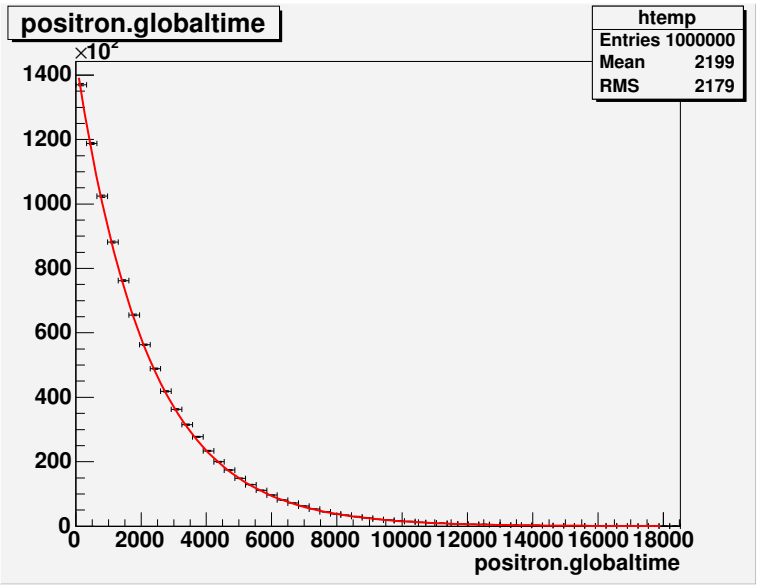

Figure 9.1: Muons time of decay distribution and comparison to the theoretical expectation. Results for a 1 million muons simulation.

9.1.2 Angular distribution, solid angle, asymmetry variation

The angular direction of the positron emission is in relation with the energy of the positron. The more energy the positron has, the smaller the angle of emission with respect to the muon spin is. The V-A theory predicts the positron rate to be

$$\mathrm{d}\Gamma^2(w, \theta) = \frac{1}{\tau n(w)\left[1 + D(w)cos\theta\right]}\mathrm{d}w\mathrm{d}(\cos\theta)$$

w being the ration between the energy of the emitted positron and the maximal energy, and θ the angle between the muon spin and the positron momentum. The distribution $n(w)$ along energy is given by the Michel's spectrum

$$n(w) = w^2(3 - 2w)$$

and the asymmetric factor is given by

$$D(w) = \frac{2w - 1}{3 - 2w}$$

. We assume, here, that the muons are fully polarized.

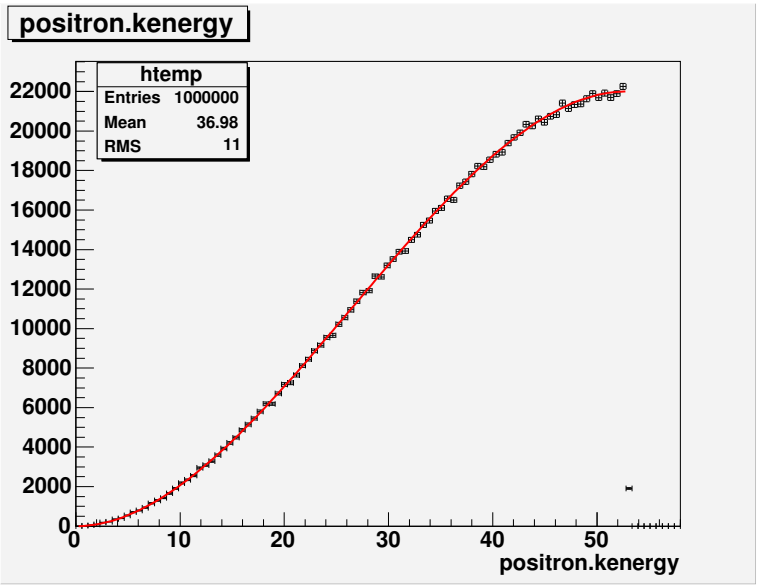

Figure 9.2: Energy distribution and Michel spectrum. Results for a 1 million muons simulation.

The distribution along energies becomes, for $\Theta = 0$,

$$n(w) + n(w)D(w) = 2w^2$$

and for $\Theta = \pi$,

$$n(w) - n(w)D(w) = 4(w^2 - w^3)$$

The asymmetry can be derived integrating the rate $d\Gamma^2$, and one should get

$$A(w_{min}, \Theta_0) = \frac{1 + \cos\Theta_0}{6} \frac{1 + 2w_{min}^3 - 3w_{min}^4}{1 - 2w_{min}^3 + w_{min}^4}$$

where Θ_0 is the opening angle of the solid angle. This means that if one select all positron energies, $w_{min} \simeq 0$

$A \simeq \frac{1}{3}$ for small solid angles

$A \simeq \frac{1}{6}$ for large solid angles

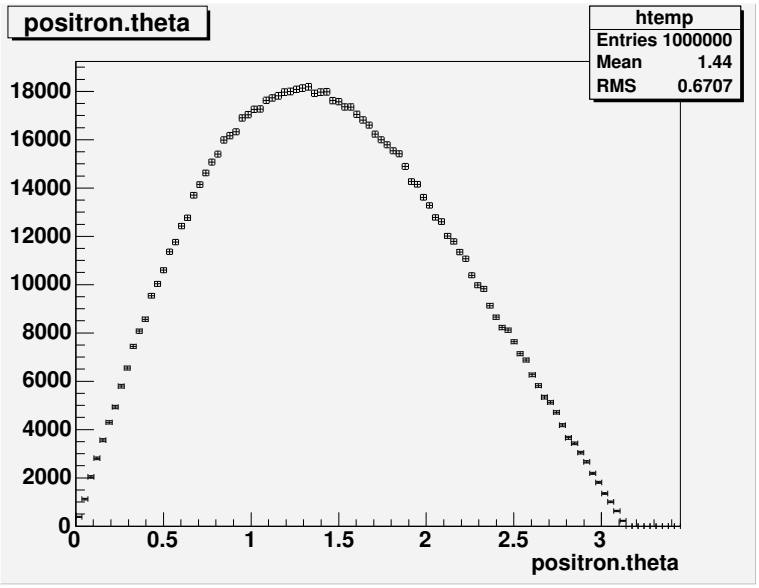

Figure 9.3: Theta distribution. Results for a 1 million muons simulation.

9.1.3 Test organisation

The previous properties were tested in the following way:

Geometry setup:

1. The geometry consists in a spherical detector with a uniform magnetic field (100 gauss) inside
2. The particle gun launch positive muons with no kinectic energy (at rest) and spin==(1.,0.,0.) at the center of the detector

The following parameters were registered for each positron produced by a muon decay process and entered the detector:

1. time of muon decay
2. kinetic energy of positron
3. positron position and momentum
4. muon spin and positron momentum angles

Output: datas were stored in a ROOT file as a tree object. The following histograms were built:

1. comparison of energy, decaytime and angles spectra with their theoretical curves
2. energy spectra for high or low opening angle and analytical comparison

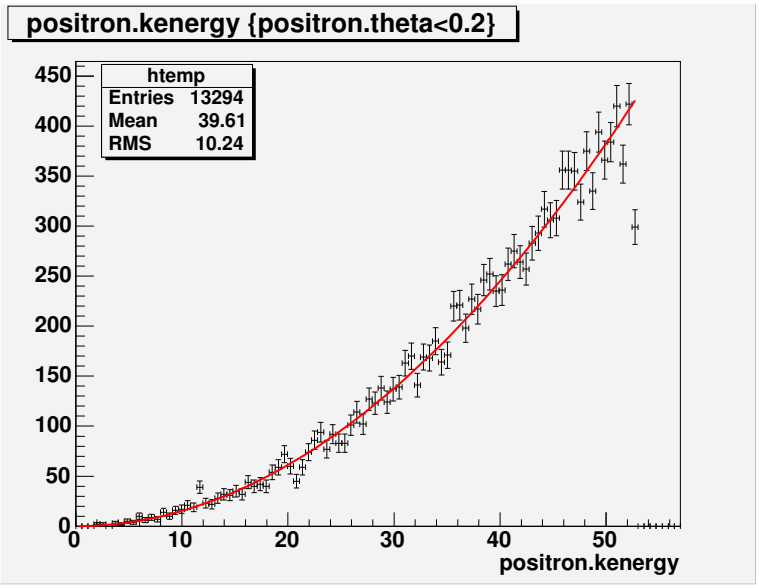

Figure 9.4: Energy distribution when theta close to 0

3. counts oscillation in forward/backward direction which should respect the precession frequency for large solid angle and small solid angle.

The results are presented in the two next sections. First for the case of a field equal to zero, secondly for a field $B = 100$ gauss

9.2 The $\vec{B} = 0$ case

In the case of a magnetic field equal to zero, we expect to have no precession. Therefore the asymmetry can be measured precisely. For small solid angles the latter should be equal to $\frac{1}{3}$, and for large solid angles, to $\frac{1}{6}$. The muon lifetime distribution and the relation between the positron emission angle and energy can also be checked. Here is a sum up of what we obtained after a one million muon simulation:

9.2.1 Lifetime, energy and angle distributions

Lifetime The lifetime was in accordance with the theoretical distribution normalized with the initial number of muons.

The theoretical curve $N = N_0 e^{-\frac{t}{\tau}}$ with $N_0 =$ is well reproduced on figure **??**

Energy and angle distribution The energy distribution according to Michel's spectrum was generated by the Von Neumann improved rejection method. It fits very well the analytical expression, as shown on the figure **??**

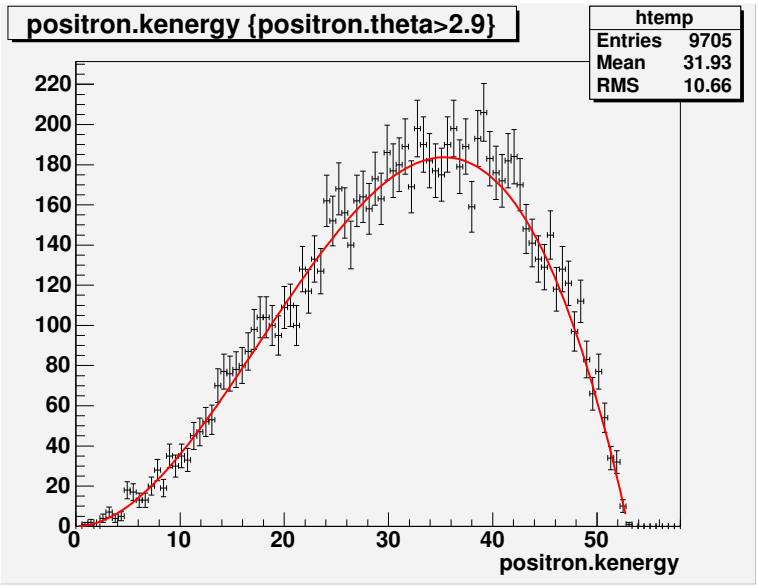

Figure 9.5: Energy distribution when theta close to π

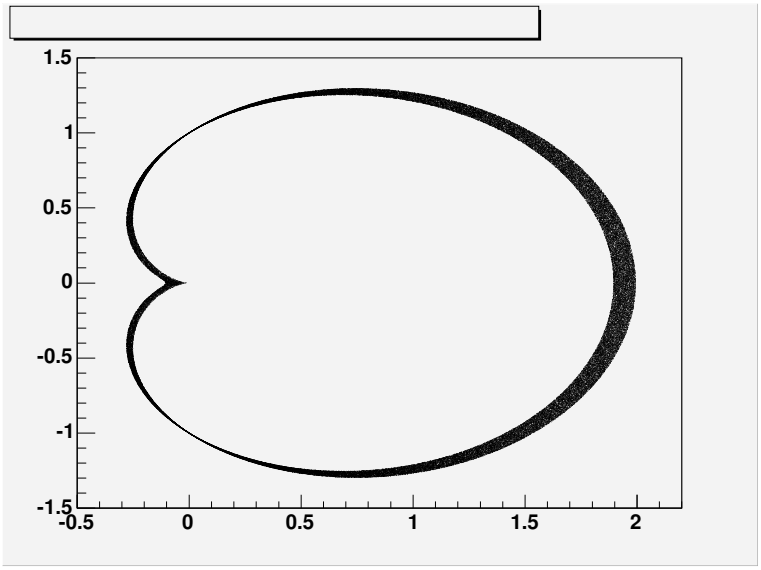

Figure 9.6: Cardioidal representation of positron theta/energy relation.

The angle distribution, given by $\sin\theta\,(1 + D(w)\cos\theta)$ was generated using the inversion method:

Let's

$$N\kappa = \int_0^t \sin\theta + \sin\theta\,\frac{D(w)}{2}\mathrm{d}\theta$$

be a uniformely distributed random number between 0 and 1, and N, is the normalisation

$$N = \int_0^\pi \sin\theta + \sin\theta\,\frac{D(w)}{2}\mathrm{d}\theta = 2$$

We have

$$2\kappa = \left(-\cos t - \frac{D(w)}{4} + 1 + \frac{D(w)}{4}\cos 2t\right) = 1 - \cos t + \frac{D(w)}{2}\left[1 - \cos^2 t\right]$$

what leads to the quadratic equation

$$\frac{D(w)}{2}X^2 + X + \left(2\kappa - \frac{D(w)}{2} - 1\right) = 0 \quad \text{where } X = \cos t$$

and we retain the solution such as $-1 < \cos\theta < 1$,

$$X = \frac{-1 + \sqrt{1 - 2D(w)\left(2\kappa - 1 - \frac{D(w)}{2}\right)}}{D(w)}$$

The curve of this distribution is represented on figure **??**.

While selecting special opening angles, the energy distribution is supposed to changed as in **??**. The curves in figure **??** and figure **??**, and the cardioid representation **??** are the verification that the energy/angle distributions are well implemented.

9.3 The $\vec{B} \neq 0$ case

For a non zero magnetic field, the muon spin precession creates an oscillation of the number of positrons detected in one direction with respect to the time. The theoretical expression predicts that one should obtain $\sin\theta\left(1 + \frac{1}{3}\cos\theta\right)$ for small solid angles (figure **??**) and $\sin\theta\left(1 + \frac{1}{6}\cos\theta\right)$ for large solid angles (figure **??**), what is perfectly reproduced by the simulation

These results show us that the implementation of random distribution is well interpreted by the running simulation. It would now be interesting to have an idea of the electric field simulation.

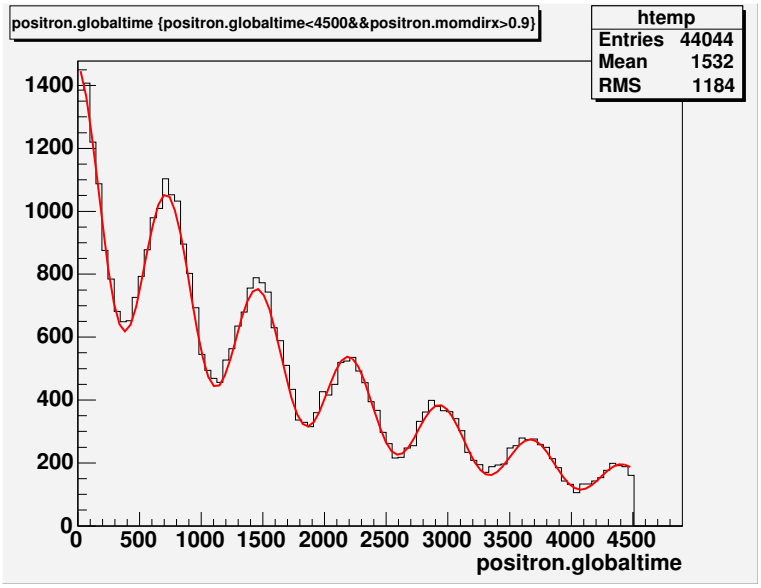

Figure 9.7: Positrons counted in forward direction when muon spin precesses under a 100 gauss magnetic field. Detector solid angle close to zero

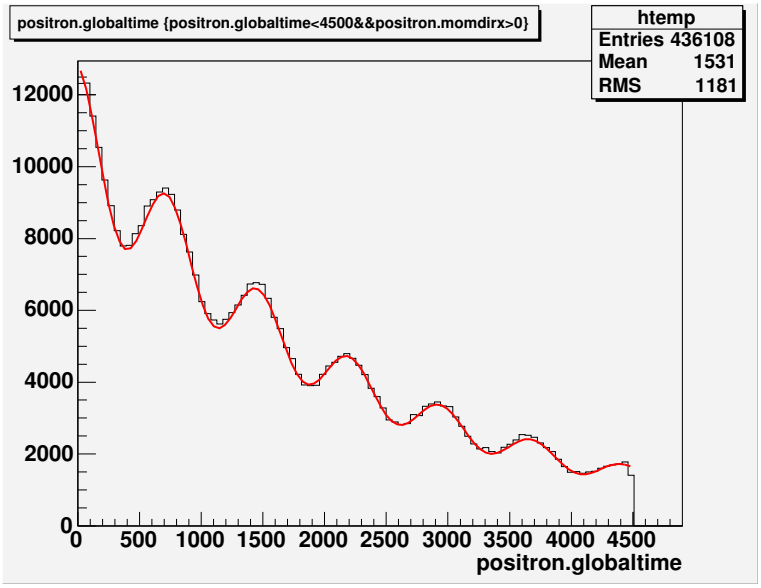

Figure 9.8: Positrons counted in forward direction when muon spin precesses under a 100 gauss magnetic field. Detector with large solid angle

Chapter 10

LEμSR Preliminary Electric Field Test

One final preliminary test has been performed in order to verify the electric field implementation. We will here give a short description of the test and present the main results.

10.1 Third lense electric field description

10.1.1 Setup

To analyse the electric field in the third lense, 44 dummy planes were created and inserted in the third lense vacuum volume. These dummy planes are thin cylinder, filled with vacuum. Their radius is equal to the inner radius of the lense cylinders and the space between each plane is one centimeter. Proceeding this way is usefull because each time the muons will enter one of these dummy planes it will start a new step. Hence the stepping action is called and one can obtain any information about the muons. Morever this can provide sliced information of the field in the lense.

10.1.2 Tests operated

The following test were operated:

1. Test of the field value: this test was to check the interpolation of the field and to see if the value provided by the field map are coherent
2. Radial field: the radial field was drawn for different planes in order to see how it changes according to our expectations
3. Physical extents: position and energy along the position on the z axis were drawn

4. Focal length: as a final test, the focal length of the lense was calculated and compared to theoretical values.

10.2 Results

10.2.1 Field value

To verify the field value, we first expressed it according to the formula $E = V/d$ which we applied between the lense and the wall of the lense vacuum chamber at the middle of the high voltage cylinder. Hence we obtained

$$E = \frac{V}{d} = \frac{1[\text{kV}]}{1.75[\text{cm}]} \simeq 58823[\text{V/m}]$$

which is in accordance with the field map value, $(5,5 \pm 0.4)10^4[\text{V/m}]$.

10.2.2 Radial field

The radial field was drawn for the planes 1 15 22 and 38 (cf. figure **??** **??** and **??**) in order to analyse field behaviour in the lense. This was done for a 15 kilovolt central cylinder voltage.

10.2.3 Physical extent

On figure **??** one can see the evolution of some parameters as position, field component and agree about their relevance. We can see on figure **??** particles trajectories and then beam thickness modification with respect to its progression in the lense.

10.2.4 Focal length

The focal length was approximated using the momentum direction of particles at the exit of the lense. It was been compared to values in tables (cf E:Harting and F.H. Read, *Electrostatic Lenses*, Elsevier, pp. 8, 10 and 138). For a 15kV einzel lense, with the dimensions of the one of LEMuSRexperiment, we are supposed to find a $F \simeq 30[\text{cm}]$. The figure **??** show the focal length distribution

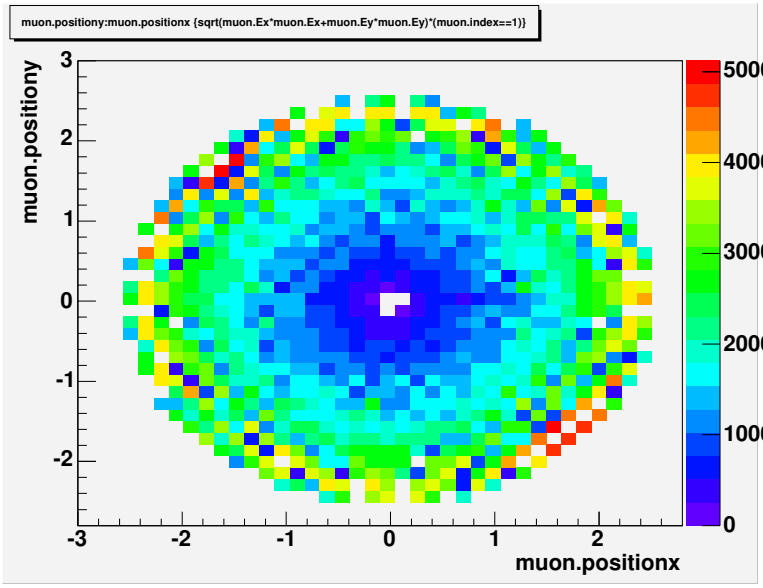

Figure 10.1: Radial field at the entrance of the lense $z = -77.7$cm

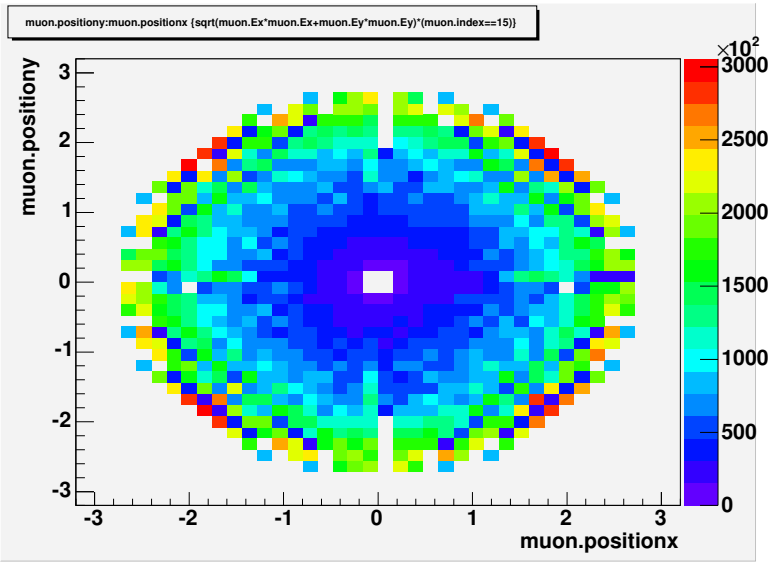

Figure 10.2: Radial field at $z = -63.7$cm

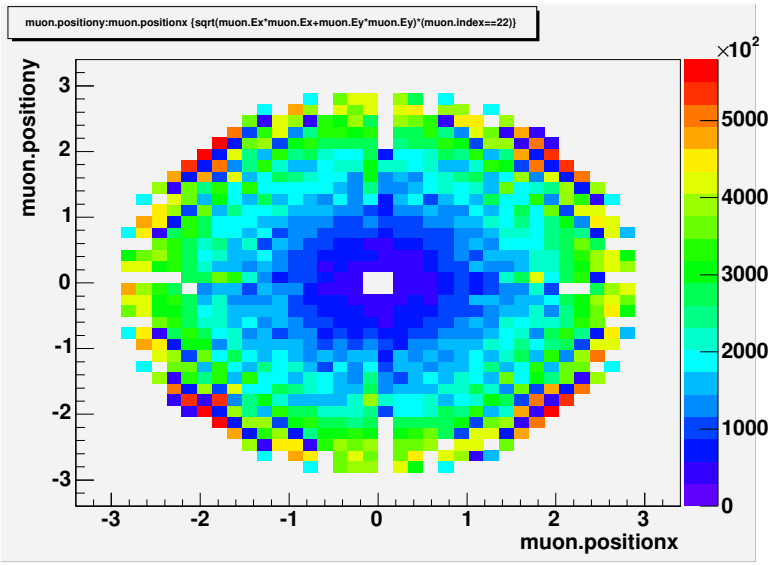

Figure 10.3: Radial field at the entrance of the lense $z = -56.7$cm

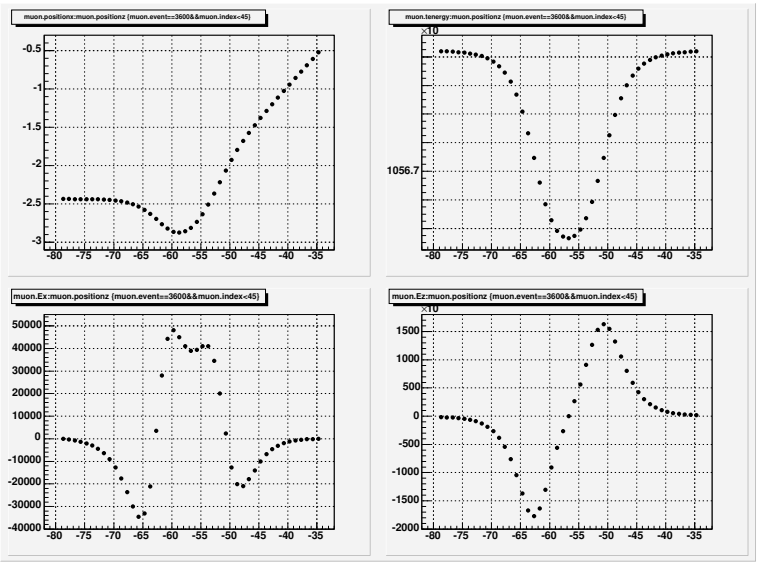

Figure 10.4: a) x position of a muon in the lense. b) corresponding kinetic energy. c) the x-component of the lelectric field. d) the field component along beam axis

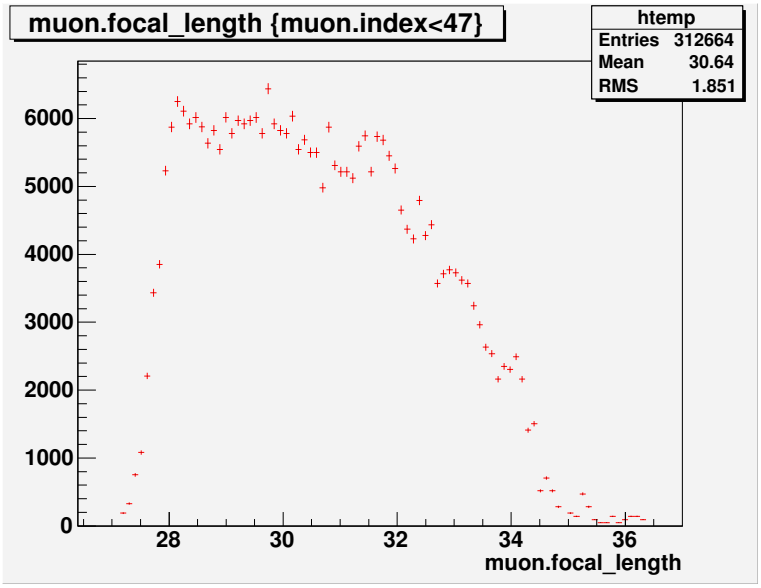

Figure 10.5: Focal length distribution for a 5cm diameter monoenergetic beam (20 keV) in a 15kV einzel lense

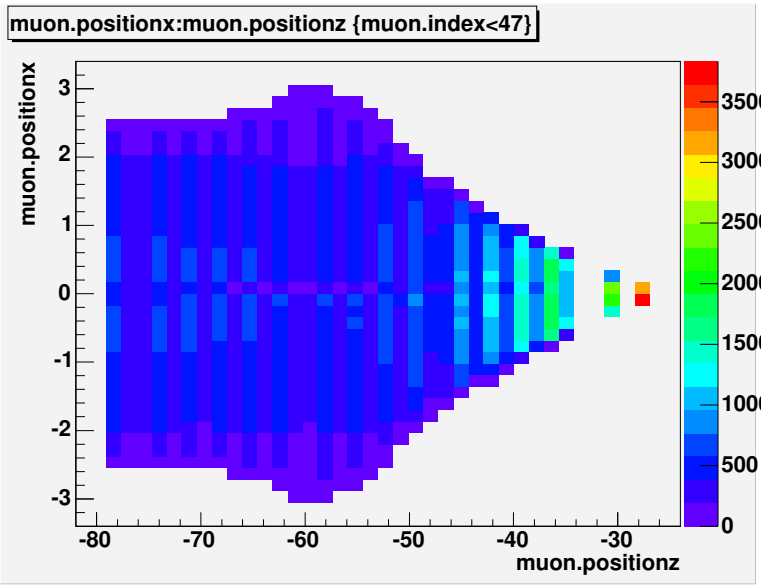

Figure 10.6: Focal length distribution for a 5cm diameter monoenergetic beam (20 keV) in a 15kV einzel lense

Chapter 11

Conclusion

As a conclusion we may say that Geant4 is a powerfull tool for LEMuSRsimulation. The above all advantage is the possibility offer to the user to personnalize the whole geant code.

This simulation passed the preliminary physical tests successfully, and this is an indication that it will be usefull for monitoring further experiment based on the designed LEμSR device.

Finally this simulation would not be difficult to extend or modify, and one is strongly encouraged to read the code carefully before proceeding to some changes.[1]

[1] Further explaination about geant code can be found on the www.geant4.cern.ch website or by mailing taofiq.paraiso@bluewin.ch