Università Degli Studi di Milano-Bicocca

Dipartimento di Fisica

Simulazione Montecarlo di Rivelatori di Radiazione

Martina Mozzanica

Docente
Prof. Gabriele Croci

Data
22 aprile 2022

Indice

Parte I

Geant 4

Geant4 è un insieme di strumenti software per la simulazione del passaggio di particelle attraverso la materia. Trova applicazioni nella fisica delle alte energie, nella fisica nucleare, negli acceleratori e in studi di fisica medica e astrofisica.

0.1 MakeFile e Main

La struttura di un progetto di Geant4 si compone di due cartelle: *include*, dove si includono alcune classi (file .hh) e *src* dove si includono i file principali (.cc). I file che includo (.hh) con il prefisso *G4* sono classi già definite (native) in Geant4 mentre quelle senza questo prefisso sono classi che deve definire (scrivere) l'utente.

Per runnare la simulazione si genera il MakeFile attraverso il file $CMakeList.txt$ dove si abilita l'utilizzo di ROOT, si includono le directories necessarie, si aggiunge e si installa il file eseguibile (.exe) e si possono aggiungere varie Macro (file .mac).

Nel main (.cc) viene creato $G4RunManager$ che gestisce il run (inizializza il Kernel di Geant4, ovvero il "cuore" della simulazione). All'interno del main è necessario chiamare e gestire con il RunManager tutte le Initialization classes (quelle obbligatorie sono $G4VUserDetectorConstruction$ e $G4VUserPhysicsList$) e le User Action classes (quella obbligatoria è $G4VUserPrimaryGeneratorAction$). Nel main viene inizializzato $G4VisManager$ per la visualizzazione e $UImanager$ per l'interfaccia con l'utente.

Creando la cartella *build*, si genera il MakeFile con il comando *cmake* e si compila il programma utilizzando *make*. A questo punto digitando $./MyFile$ si apre l'interfaccia GUI (Graphical User Interface) attraverso cui è possbile visualizzare la geometria del rivelatore e interagire con esso.

Capitolo 1

TASK 1

1.1 Task1b

File sorgente: Task1

Il detector che viene simulato è composto da un calorimetro elettromagnetico, delle strip in Silicio e un calorimetro adronico. Viene modificato il file Detector Construction (.cc) in cui si indicano i materiali e la geometria del rivelatore.
Nei file include viene inserito anche il Sitema di unità attraverso il comando *G4SystemOfUnits* in cui sono presenti le unità di misura comunemente usate. Ci sono unità di misura di base come Kelvin, MeV, mm, ns ecc mentre quelle mancanti vengono definite a partire da quest'ultime. Inoltre, Geant4 può scegliere autonomamente l'unità di misura più opportuna usando il comando $G4BestUnit$. Si possono definire nuove costanti con $G4UnitsDefinition$ che vengono inserite nel kernel in $G4UnitsTable$.

1.1.1 Materiali

I materiali usati per costruire il rivelatore vengono presi dal database NIST che risulta importato in Geant4. Nonostante ciò, si possono costruire manualmente i materiali definendo l'isotopo ($G4Isotope$), l'elemento ($G4Element$) e poi facendo uso del comando $G4Material$ si definisce il materiale che può contenere vari elementi nelle percentuali desiderate.
Per il calorimetro adronico si sono utilizzati: Vuoto, Aria, Silicio (Si), Argon liquido (lAr), Ferro (Fe) e tungstato di Piombo (PbWO4). Attraverso il comando $G4NistManager$ vengono importati i materiali presenti nel database NIST e, in seguito, usando il comando $FindOrBuiltMaterial$ si definiscono gli elementi che si desiderano utilizzare.

```
void DetectorConstruction::DefineMaterials()
{
    //Get Materials from NIST database
    G4NistManager* man = G4NistManager::Instance();
    man->SetVerbose(0);

    // define NIST materials
    vacuum  = man->FindOrBuildMaterial("G4_Galactic");

    //Tracker
    air     = man->FindOrBuildMaterial("G4_AIR");
    silicon = man->FindOrBuildMaterial("G4_Si");
    //Em calo
    pbw04   = man->FindOrBuildMaterial("G4_PbWO4");
    //Had calo

    lar = man->FindOrBuildMaterial("G4_lAr");
    fe = man->FindOrBuildMaterial("G4_Fe");
```

1.1.2 Geometria

Per definire la geometria del rivelatore si include la classe *DetectorConstruction*. Si richiama il metodo (o funzione membro) *ComputeParameters* in cui vengono definiti i valori di default che identificano la dimensione dei volumi e quindi del detector stesso. Il rivelatore è composto da un calorimetro elettromagnetico avente un cristallo al centro, da 600 strip di Silicio per tracciare le particelle e dal calorimetro adronico. Quindi si passa alla vera e propria costruzione della geometria del detector attraverso il metodo *Construct*. Qui vengono definiti i solidi, i volumi logici e i volumi fisici e come devono essere posizionati all'interno della geometria. In generale, un volume viene definito dalle tre caratteristiche menzionate sopra. Nel dettaglio, nel solido si indicano la forma e la dimensione, nel volume logico si indicano le proprietà come i materiali o l'eventuale presenza di campi elettrici o magnetici mentre nel volume fisico viene indicato come si desidera posizionare il mio volume nello spazio. Nella costruzione di un detector c'è una gerarchia da rispettare che riguarda il posizionamento dei volumi. In particolare si parla si volume madre e volume figlio. Il volume figlio viene sempre posizionato in un volume madre e il suo posizionamento e la sua rotazione vengono effettuati rispetto al sistema di riferimento del volume madre. L'origine del sistema di riferimento del volume madre è al centro del volume madre stesso. Inoltre, i volumi figli non possono n'e' fuoriuscire dal volume madre n'e' sovrapporsi.

World Volume Come primo step bisogna sempre definire il World Volume (volume mondo), un volume fisico unico che rappresenta l'area sperimentale in cui si implementa il setup di rilevazione, e che quindi contiene tutti i volumi madre. Il volume mondo deve contenere tutti gli altri volumi. Per questo setup viene definito un volume mondo come un box di 10x10x10 metri composto solamente da Aria, posizionato al centro del sistema di riferimento (0,0,0) ed esteso da -5 m a 5 m in ogni direzione degli assi cartesiani. Il volume mondo non viene mai ruotato. Inoltre, si imposta a 0 il suo volume madre per indicare che è il volume mondo e che quindi non deve essere contenuto all'interno di altri volumi.

Dopo aver definito il volume mondo si passa alla costruzione dei vari pezzi che compongono il setup sperimentale, che in questo caso sono: il telescopio, ovvero i tre

```
//------------------------------
// World
//------------------------------

G4GeometryManager::GetInstance()->SetWorldMaximumExtent(2.*halfWorldLength);
G4cout << "Computed tolerance = "
       << G4GeometryTolerance::GetInstance()->GetSurfaceTolerance()/mm
       << " mm" << G4endl;


G4Box * solidWorld= new G4Box("world",halfWorldLength,halfWorldLength,halfWorldLength);
logicWorld= new G4LogicalVolume( solidWorld, air, "World", 0, 0, 0);

//  Must place the World Physical volume unrotated at (0,0,0).
//
G4VPhysicalVolume * physiWorld = new G4PVPlacement(0,               // no rotation
       G4ThreeVector(), // at (0,0,0)
       logicWorld,      // its logical volume
       "World",         // its name
       0,               // its mother  volume
       false,           // no boolean operations
       0);              // copy number
```

piani di Silicio e i calorimetri elettromagnetico e adronico. All'interno di *Construct* si definiscono questi tre metodi.

Telescope Il telescope sono sostanzialmente i tre piani di Silicio composti da varie strip. Tutti e tre hanno le stesse dimensioni pari a 48x10x0.3 mm e vengono posizionati all'interno del volume mondo senza essere ruotati. Quello che differenzia i tre piani di Silicio è il volume fisico, ovvero come vengono posizionati all'interno del setup. Vengono posti a -190 mm, -100 mm e -10 mm sull'asse z rispetto al sistema di rifermento del volume mondo. Viene assegnato un numero identificativo unico per ogni volume fisico, detto copy number. In questo caso viene assegnato un copy number da 0 a 2 ad ognuno dei piani di silicio.
Dopo aver costruito e posizionato i tre piani di Silicio (volumi madre), vengono posizionate al loro interno 600 strip di Silicio (volumi figli), ognuna avente dimensione 0.08x10x0.3 mm. Viene utilizzata *G4PVReplica* per replicare il posizionamento di una strip di silicio 600 volte all'interno del piano di Silicio lungo l'asse X.
Per una visione migliore del detector si utilizzano vari colori, in particolare il giallo per distinguere i piani di silicio e il rosso per le strip.

Calorimetro Elettromagnetico Il calorimetro elettromagnetico viene definito come un box di dimensioni 110x110x230 mm composto da PbWO4. Viene posizionato nel volume mondo a 115 mm sull'asse z. La parte centrale del calorimetro è composta da tungstato di piombo con dimensioni di 22x22x230 mm.

Calorimetro adronico Il calorimetro adronico viene definito come un cilindro di raggio pari a 800 mm e lunghezza 2240 mm composto da strati alternati di Ferro (assorbitore) e LAr (materiale attivo). Si utilizza *G4Tubs* per costruire la geometria cilindrica e il materiale assegnato al calorimetro è il Ferro nel quale verranno inseriti cilindri di LAr con raggio pari al raggio del cilindro di Ferro ma molto più sottili. Si definiscono all'interno del solido anche l'angolo di partenza (0) e l'angolo finale (2π), per cui in questo caso abbiamo un cilindro completo. Viene posizionato all'interno del volume mondo a 1120 mm in direzione dell'asse z.

Questo task consiste nel creare un layer attivo di LAr. Questo viene definito come un cilindro con dimensioni in x e y identiche a quelle del calorimetro adronico, ma avente la terza dimensione pari a 8 mm. Gli angoli iniziali e finali sono gli stessi del calorimetro adronico.

```
G4double halfLayerHalfZ= hadCaloLArThickness/2;

G4Tubs* hadLayerSolid = new G4Tubs( "HadCaloLayerSolid", //its name
                                    0 , // inner radius
                                    hadCaloRadius , //outer radius
                                    halfLayerHalfZ, //lenght
                                    0, //start angle
                                    CLHEP::twopi);//Ending angle in phi,
```

Ora è necessario definire il volume logico in cui bisogna definire il solido, il materiale (LAr) e il nome.

```
G4LogicalVolume* hadLayerLogic = new G4LogicalVolume(hadLayerSolid,
                                                     lar,
                                                     "HadLayerLogic");
```

Il terzo step consiste nel posizionare vari layer di LAr nel volume del calorimetro adronico ($HadCaloLogic$) attraverso $G4PVPlacement$. In questo caso, al contrario delle strip di silicio, non viene usata la funzione di Geant 4 "Replica", ma gli 80 layer di LAr vengoni inseriti nel calorimetro adronico utilizzando un ciclo for.

```
        G4ThreeVector absorberLayer(0,0,hadCaloFeThickness);
        G4ThreeVector activeLayer(0,0,hadCaloLArThickness);
        G4int layerCopyNum = hadCaloCopyNum;
        for ( int layerIdx = 0 ; layerIdx < hadCaloNumLayers ; ++layerIdx )
        {
                G4ThreeVector position = (layerIdx+1)*absorberLayer +
(layerIdx+0.5)*activeLayer;
                position -= G4ThreeVector(0,0,halfHadCaloHalfZ);//Position is w.r.t.
center of mother volume
                new G4PVPlacement(0, //rotation
                                  position, //position
                                  hadLayerLogic,
                                  "HadCaloLayer",//a name
                                  hadCaloLogic,
                                  false,
                                  ++layerCopyNum);//The unique number
```

1.1.3 Physics List

Nella Physics List vengono indicate le particelle che si desiderano utilizzare nella simulazione con i loro rispettivi processi. Tutte le Physiscs Lists devono essere derivate dalla classe $G4VUserPhysicsList$. Qui viene indicato il valore di default dei Cuts di 10 μm, si pone il livello di verbose a 1 e si deriva la lista delle particelle

elettromagnetica di Geant 4. L'utente poi deve implementare i seguenti metodi: *ConstructParticle*, *ConstructProcess* e *SetCuts*.
Nel primo si scelgono le particelle da utilizzare nella simulazione, in questo caso sono abilitate le pseudo-particelle (Geantino e genatino carico) e si possono abilitare i gamma, elettroni, positroni, pioni e muoni carichi che sono stati gia' definiti utilizzando la lista delle particelle elettromagnetiche di Geant 4. Si sceglie di abilitare muoni e pioni per la simulazione.

Considerando le pseudo particelle geantino e geantino carico queste sono particelle fittizie senza massa. Geantino non risulta carico e quindi non interagisce, ma risulta utile per la diagnostica della geometria e della tracciatura, d'altra parte, geantino carico è una particella che non interagisce ma risulta carica, per cui può essere tracciata adeguatamente in un campo magnetico. A parte per il processo di trasporto, a nessuna di queste due particelle può essere assegnato un processo di interazione.

```
void PhysicsList::ConstructParticle()
{
  // pseudo-particles
  G4Geantino::GeantinoDefinition();
  G4ChargedGeantino::ChargedGeantinoDefinition();

  // define gamma, e+, e- and some charged Hadrons
  emPhysicsList->ConstructParticle();

// gamma
  //G4Gamma::Gamma();

// leptons
  //G4Electron::Electron();
  //G4Positron::Positron();
  G4MuonPlus::MuonPlus();
  G4MuonMinus::MuonMinus();

  // mesons
    G4PionPlus::PionPlusDefinition();
    G4PionMinus::PionMinusDefinition();
}
```

Dopo *ConstructParticle* si implementa il metodo *ConstructProcess* in cui vengono indicati, per ciascuna particella, tutti i processi fisici che risultano rilevanti nella simulazione. Bisogna sempre indicare il processo di trasporto per ogni particella.

L'ultimo metodo da definire è il *SetCuts*. Diversi processi fisici, ad esempio il bremsstrahlung, la ionizzazione da parte di tutte le particelle cariche o la produzione

```
void PhysicsList::ConstructProcess()
{
  AddTransportation();
  emPhysicsList->ConstructProcess();
}
```

di coppia e^+/e^- da muoni, hanno una sezione d'urto elevata a basse energie. Risulta dunque necessario implementare un taglio nella produzione (production cut), in modo tale che tutte le particelle al di sotto di questa soglia non vengano generate, ovvero non vengano identificate come secondarie. La loro energia viene considerata come deposito di energia. Geant4 usa tagli di produzione in range al posto di tagli in energia usati invece da Geant3 o da molti codici MonteCarlo. Per questa simulazione il production cut viene posto a 10 μm.

1.1.4 Primary Generator Action

Mentre *DetectorConstruction* e *PhysicsList* sono classi di inizializzazione la *PrimaryGeneratorAction* è una classe di azione che viene invocata durante un loop di eventi. Questa è una delle classi obbligatorie che controlla la generazione delle particelle primarie; in particolare il numero e il tipo di particelle, l'energia, la posizione, la direzione, la polarizzazione ecc.. Questa classe viene usata solo per controllare la generazione di particelle primarie, per cui non deve generare le particelle primarie ma deve solo invocare il metodo $GeneratePrimaryVertex$, che selezione i generatori primari per costruire le particelle primarie, e $G4VPrimaryGenerator$, che fornisce i generatori di particelle primarie.

Nel costruttore si istanzia il generatore primario e si settano alcuni i valori di default. In questa simulazione il generatore primario è il GPS (General Particle Source): un' implementazione avanzata del generatore primario adatta sopratutto ad applicazioni nello spazio. Utilizzando il GPS ci sono molte funzioni disponibili: il Primary Vertex può essere posizionato casualmente con diverse opzioni (punto di emissione, piano ecc), l'angolo di emissione può avere diverse distribuzioni (isotropico, focalizzato ecc) con parametri aggiuntivi come le aperture angolari massime e minime, l'energia cinetica delle particelle primarie può essere anch'essa random con diverse opzioni (monoenergetica, legge di potenza), si possono avere sorgenti multiple definendo diverse intensità e infine ha la capacità di bias degli eventi (riduzione della varianza) migliorando il tipo di particella, la distribuzione del punto del vertice, l'energia e/o la direzione. Tutte le caratteristiche si possono controllare dai comandi UI o C++.

Nel programma principale vengono definiti i pioni, che sono le particelle primarie che si desiderano utilizzare. Si utilizza quindi un fascio monoenergetico di 2 GeV con centro a -80 cm sull'asse z con una deviazione standard sugli assi x e y di 0.1 mm. Si definisce la direzione del momento sull'asse z e si utilizza un fascio 2D con deviazione standard angolare sugli assi x e y di 0.1 mrad.

```
G4VPrimaryGenerator* PrimaryGeneratorAction::InitializeGPS()
{
  G4GeneralParticleSource * gps = new G4GeneralParticleSource();

  // setup details easier via UI commands see gps.mac

  // particle type
  G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* pion = particleTable->FindParticle("pi+");
  gps->GetCurrentSource()->SetParticleDefinition(pion);

  // set energy distribution
  G4SPSEneDistribution *eneDist = gps->GetCurrentSource()->GetEneDist() ;
  eneDist->SetEnergyDisType("Mono"); // or gauss
  eneDist->SetMonoEnergy(2.0*GeV);

  // set position distribution
  G4SPSPosDistribution *posDist = gps->GetCurrentSource()->GetPosDist();
  posDist->SetPosDisType("Beam");  // or Point,Plane,Volume,Beam
  posDist->SetCentreCoords(G4ThreeVector(0.0*cm,0.0*cm,-80.0*cm));
  posDist->SetBeamSigmaInX(0.1*mm);
  posDist->SetBeamSigmaInY(0.1*mm);

  // set angular distribution
  G4SPSAngDistribution *angDist = gps->GetCurrentSource()->GetAngDist();
  angDist->SetParticleMomentumDirection( G4ThreeVector(0., 0., 1.) );
  angDist->SetAngDistType("beam2d");
  angDist->SetBeamSigmaInAngX(0.1*mrad);
  angDist->SetBeamSigmaInAngY(0.1*mrad);
  angDist->DefineAngRefAxes("angref1",G4ThreeVector(-1.,0.,0.));
```

1.1.5 Output

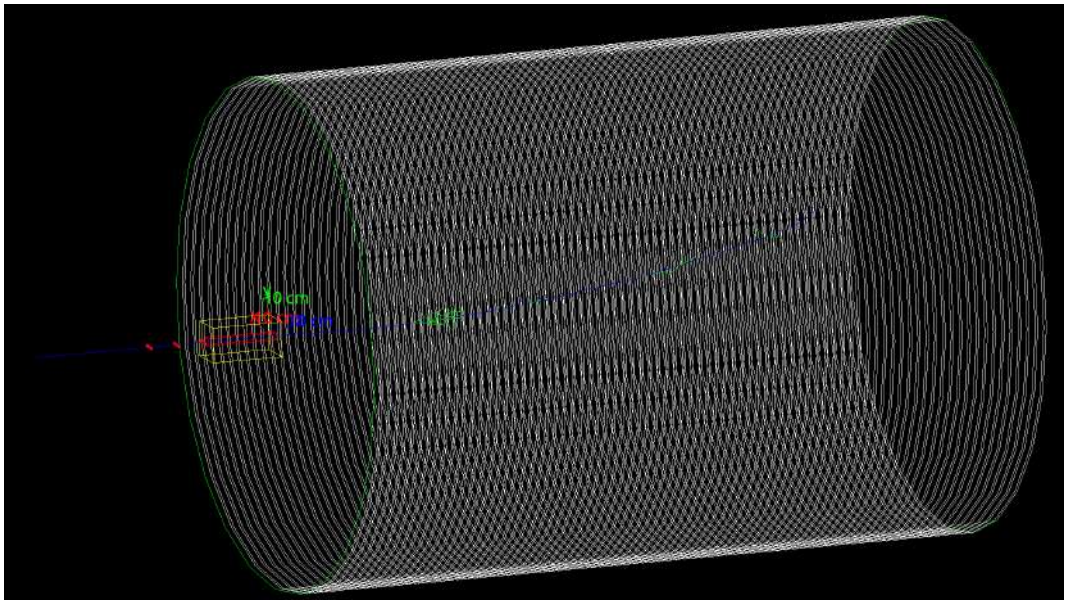


Figura 1.1: Setup sperimentale composto da Tracker di Silicio, calorimetro elettromagnetico e calorimetro adronico.

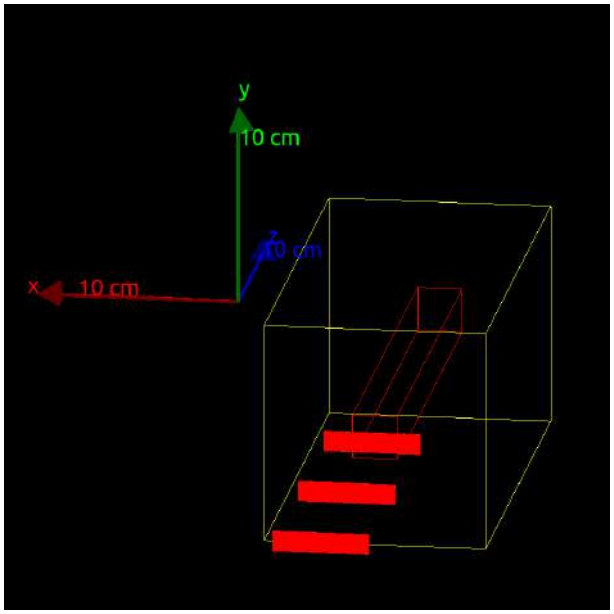


Figura 1.2: Zoom sul setup sperimentale sul Tracker di Silicio e il calorimetro elettromagnetico.

Capitolo 2

TASK 2

File sorgente: Task2

2.1 Task2a

2.1.1 Primary Generator Action

Geant4 ha la possibilità di provvedere ad alcune implementazioni concrete di $G4VPrimaryGenerator$. Una di queste è $G4GeneralParticleSource$ (vista del task1), mentre un'altra è $G4ParticleGun$. Quest'ultima è quella che si utilizza in questo task. $G4ParticleGun$ "spara" una particella primaria di una certa energia, da un punto definito nello spazio ad un certo istante di tempo e con una direzione definita. Si possono utilizzare anche i comandi UI per inizializzare alcune caratteristiche della particella.
Nel Costruttore della classe $G4VPrimaryGenerator$ viene definito $G4ParticleGun$ con il tipo di particella e la sua energia. In questo caso vengono utilizzate come particelle primarie i pioni positivi di 1 GeV.

```
PrimaryGeneratorAction::PrimaryGeneratorAction()
  : outfile(0)
{
  gun  = new G4ParticleGun(1);

  // complete particle name and energy  (do not forget the energy unit)
  G4ParticleDefinition* particle
            = G4ParticleTable::GetParticleTable()->FindParticle("pi+");
  gun->SetParticleDefinition(particle);
  gun->SetParticleEnergy(1.0*GeV);

}

PrimaryGeneratorAction::~PrimaryGeneratorAction()
{
  delete gun;
}
```

In *PrimaryGeneratorAction* non si creano le particelle primarie ma di definiscono solo le loro caratteristiche. La creazione di particelle primarie avviene nel metodo *GeneratePrimaries* (che prende in input *G4Event*). Questa funzione viene chiamata per generare ogni *G4Event*. Vengono riportati due esempi del codice implementato in questo metodo.
Nel primo viene generato solamente un pione positivo il cui punto d'origine è (0,0,0) mentre la direzione del momento è quella dell'asse z.

```
  G4double x0 = 0.*cm, y0 = 0.*cm, z0= 0.0*cm;
  G4cout<<"GeneratePrimaries : new event
"<<G4BestUnit(G4ThreeVector(x0,y0,z0),"Length")<<G4endl;

  gun->SetParticlePosition(G4ThreeVector(x0,y0,z0));
  gun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));
  gun->GeneratePrimaryVertex(anEvent);
```

Nel secondo viene generato un set di particelle uniformemente distribuite in un rettangolo 0.1x2 cm. La direzione del momento è sempre quella dell'asse z.

```
G4double z0 = 0.*cm, x0 = 0.*cm, y0 = 0.*cm;
x0 = -0.05 + 2*0.05*G4UniformRand();
y0 = -1.0+ 2*G4UniformRand();

gun->SetParticlePosition(G4ThreeVector(x0,y0,z0));
gun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));
gun->GeneratePrimaryVertex(anEvent);
```

2.1.2 Physics List

Nella physics list vengono abilitate le seguenti particelle: Geantino, Geantino carico, i pioni negativi e positivi e le particelle definite nella Physics List elettromegnetica di Geant4. L'unico processo che viene abilitato è quello di trasporto.

```
// pseudo-particles
G4Geantino::GeantinoDefinition();
G4ChargedGeantino::ChargedGeantinoDefinition();

// define gamma, e+, e- and some charged Hadrons
emPhysicsList->ConstructParticle();

// mesons
  G4PionPlus::PionPlusDefinition();
  G4PionMinus::PionMinusDefinition();
```

2.2 Task 2b

2.2.1 Primary Generator Action

Nel costruttore della classe *PrimaryGeneratorAction*, a differenza del task 2a (2.1), si implementa *G4PGeneralParticleSource* (GPS). Si sceglie una sorgente monoenergetica di 2 GeV posizionata nel punto (0,0,0) con momento diretto lungo l'asse z.

```
G4GeneralParticleSource *gps = new G4GeneralParticleSource();

gps->GetCurrentSource()->GetEneDist()->SetMonoEnergy(2.0*GeV);
gps->GetCurrentSource()->GetPosDist()->SetCentreCoords(G4ThreeVector(0.0*cm, 0.0*cm,
0.0*cm));
gps->GetCurrentSource()->GetAngDist()-
>SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));

gun = gps;
```

2.2.2 Comandi UI

Al posto di dichiarare la sorgente nel costruttore si può ottenere lo stesso risultato utilizzando i comandi UI (User Interface).
Per utilizzare la User Interface Session si include nel Main file la classe $G4UIExecutive$.
I comandi UI possono essere eseguiti sia in batch mode utilizzando i macro file sia in modalità interattiva (a terminale).
Un comando UI consiste in $/command\ directory/command/parameter(s)$. La command directory è la General Particle Source ($/gps/$) mnetre i command possono essere, ad esempio, il tipo di particella ($/gps/particle$), l'energia ($/gps/ene/$), la posizione ($/gps/position$). Il parametro spesso non è obbligatorio e, nel caso non venga definito dall'utente, Geant4 usa il parametro di default.
Dopo aver definito tutte le caratteristiche delle particelle si utilizza il comando $/run/beamOn$, con parametro il numero di particelle, per lanciare un dato numero di particelle nella simulazione.
Nella figura 1.1.5 si inizializza un raggio gamma da 2 GeV con sorgente nel punto (0,0,-1 m) e momento diretto lungo l'asse z. Si utilizza la modalità interattiva.

Nella figura 2.2.2, lavorando in batch mode con il file *gps.mac* (fig. 2.2.2), si inizializza un fascio monoenergetico di raggi gamma da 1 GeV con sorgente nel punto $(0,0,-1\ m)$ e momento diretto lungo l'asse z. I comandi σ_r,σ_x e σ_y (in unità di sigma) impostano la deviazione standard radiale o trasversale del profilo del fascio.

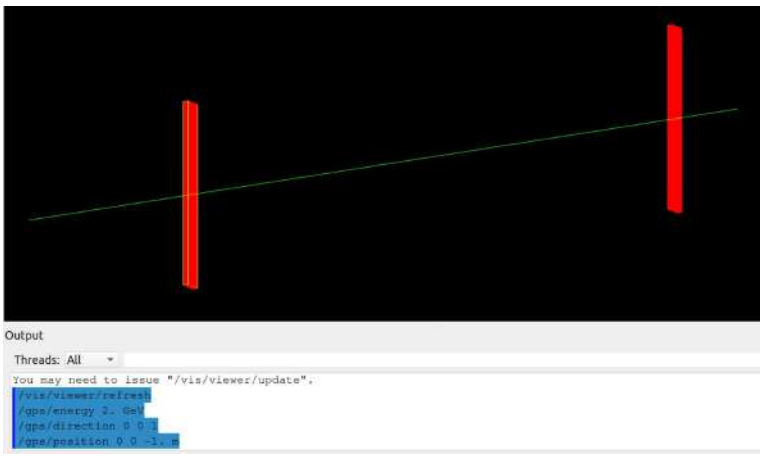


Figura 2.1: Raggio γ da 2 GeV inizializzato da terminale.

```
/gps/particle gamma

/gps/ene/type Mono
/gps/ene/mono 1. GeV

/gps/position 0 0 -1. m
/gps/pos/type Beam
/gps/pos/sigma_r 0.1 mm

/gps/direction 0 0 1
/gps/ang/sigma_x 0.15 mrad
/gps/ang/sigma_y 0.15 mrad


/run/beamOn 100
```

Figura 2.2: File *gps.mac* utilizzato in batch mode.

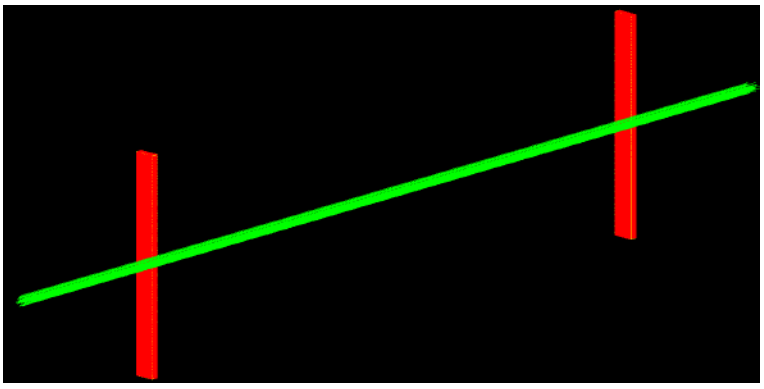


Figura 2.3: Fascio monoenergetico di raggi γ da 1 GeV inizializzato in batch mode.

Capitolo 3

TASK 3

File sorgente: Task3

3.1 Task3a

3.1.1 User Action classes

Nel Main vengono definite, oltre alla User Action class obbligatoria *PrimaryGeneratorAction*, altre quattro classi User Action facoltative: *StackingAction*, *SteppingAction*, *EventAction* e *RunAction*. Queste permettono direttamente all'utente di modificare il comportamento della simulazione e/o di ricavarne delle informazioni utili.

```
//Optional User Action classes
//Stacking Action
StackingAction* aStackingAction = new StackingAction();
runManager->SetUserAction(aStackingAction);
//Stepping Action
SteppingAction* aSteppingAction = new SteppingAction();
runManager->SetUserAction(aSteppingAction);
//Event action (handles for beginning / end of event)
EventAction* anEventAction = new EventAction();
runManager->SetUserAction( anEventAction );

//Run action (handles for beginning / end of event)
RunAction* aRunAction = new RunAction();
runManager->SetUserAction( aRunAction );
```

Analisys Oltre alle classi obbligatorie e alle quattro User Action classes facoltative viene implementata anche la classe *Analysis* che ci permette di memorizzare gli istogrammi, nTuple o altri oggetti come risultato della simulazione. Dato che Geant4 non ha uno strumento di analisi interno, Geant4 viene integrato con ROOT. Gli istogrammi di output saranno quindi file *.root*. Nella classe *Analysis* si implementano i seguenti metodi: $PrepareNewEvent$ (dove si resettano le variabili relative all'evento), $PrepareNewRun$ (dove si resettano le variabili relative alla Run e si creano i ROOT file e gli istogrammi), $EndOfEven$ (dove si riempiono i ROOT file) e $EndOfRun$ (dove si possono mostrare alcuni print outs, si scrivono e si chiudono i ROOT file). Gli istogrammi di output di questa simulazione mostrano l'energia

totale depositata normalizzata con l'energia del fascio, l'energia totale depositata nel cristallo centrale normalizzata con l'energia del fascio e il profilo energetico lungo il calorimetro.

Run Action Una Run è una collezione di eventi e parte con il comando $/run/BeamOn$. All'interno della Run l'utente non può modificare il detector setup e i processi fisici. Una Run è rappresentata dalla classe $G4Run$, processata da $G4RuManager$ e viene usato come optional user hook $G4UserRunAction$.
Nel $G4RuManager$ l'utente deve inizializzare obbligatoriamente la geometria e la fisica usando $SetUserInitializaton$ e la generazione di eventi usando $SetUserAction$. È sempre possibile inizializzare altri metodi facoltativi.
In $G4UserRunAction$ il metodo $GenerateRun$ istanzia gli user-customized run objects (in questo task non ce ne sono per cui non viene definito), $BeginOfRunAction$ definisce gli istogrammi e $EndOfRunAction$ analizza la Run e memorizza gli istogrammi.

```
void RunAction::BeginOfRunAction(const G4Run* aRun )
{
  G4cout<<"Starting Run: "<<aRun->GetRunID()<<G4endl;
  Analysis::GetInstance()->PrepareNewRun(aRun);
}

void RunAction::EndOfRunAction( const G4Run* aRun )
{
  Analysis::GetInstance()->EndOfRun(aRun);
}
```

Event Action Ogni Run è composta da un o più eventi. L'evento è l'unità di base per una simulazione in Geant4. All'inizio di ogni evento vengono generate le tracce primarie (mediante l'utilizzo della $PrimaryParticleGenerator$) che vengono inserite all'interno di uno stack. Le traccie delle primarie vengono prese nello stack, analizzate una ad una. In seguito vengono tracciate e le risultanti tracce secondarie vengono inserite nello stack. Questo processo continua fintanto che lo stack ha una traccia al suo interno. Quando lo stack risulta vuoto, l'evento in questione è terminato e si passa al successivo. La classe che rappresenta un evento è $G4Event$, che alla fine del processo (se andato a buon fine) ha informazioni riguardanti la lista di vertici e particelle primarie (input) e le collezioni di hits e traiettorie (output).
Il $G4EventManager$ processa l'evento mentre $G4UserEventAction$ è l'optional user hook. In $G4UserEventAction$ il metodo $BeginOfRunAction$ seleziona gli eventi o può collegare un $G4VUserInformation$ object mentre $EndOfRunAction$ analizza l'evento e riempio gli istogrammi.

```
void EventAction::BeginOfEventAction(const G4Event* anEvent )
{
  if ( anEvent->GetEventID() % 1000 == 0 )
    {
      G4cout<<"Starting Event: "<<anEvent->GetEventID()<<G4endl;
    }
  Analysis::GetInstance()->PrepareNewEvent(anEvent);
}

void EventAction::EndOfEventAction(const G4Event* anEvent)
{
  Analysis::GetInstance()->EndOfEvent(anEvent);
}
```

TrackingAction Ogni evento è composto da una serie di particelle (tracks). Una track è rappresentata da $G4Track$ è una foto di una particella che ha tutte le quantità fisiche di ciò che accade in quell'istante (non ha memoria di ciò che accade prima). La traccia non è una collezione di step, ma viene aggiornata dagli step. La traccia viene cancellata quando: risulta fuori dal volume mondo, scompare (decadimento), l'energia cinetica risulta nulla e non è richiesto il processo $AtRest$ o l'utente decide di ucciderla artificialmente. Non cè nessuna traccia che persiste alla fine dell'evento ma se risulta necessario memorizzarla si utilizza la classe $G4Trajectory$. Alla fine di ogni step l'user può cambiare lo stato della traccia nella classe $G4UserSteppingAction$. L'optional user hook è $G4UserTrackingAction$ in cui $PreUserTrackingAction$ decide se memorizzare la traiettoria e crea le trajectory definite dall'utente mentre $PostUserTrackingAction$ elimina le traiettorie non necessarie.

SteppingAction Una traccia è composta da diversi step. Uno step è uno snapshot dell'interazione di una particella con un volume. Lo step è rappresentato dalla classe $G4Step$ e può essere visto con un segmento delimitato da due punti, detti $PreStepPoint$ (inizio) e $PostStepPoint$ (fine), e contiene informazioni a "delta". Ogni punto dello step conosce il materiale ed il volume ad esso associato. Uno step è delimitato da processi fisici (decadimento) e/o confini geometrici (tra volumi); nel momento in cui il volume finisce lo step viene interrotto. Siccome il $PostStepPoint$ appartiene logicamente al volume successivo, per prendere le informazioni corrette da uno step si deve sempre utilizzare il $PreStepPoint$. L'optional user hook è $G4UserSteppingAction$ in cui in $UserSteppingAction$ cambia lo stato della traccia e disegna gli step. Nella classe $SteppingAction$ vengono rese disponibili all'utente le Touchables, cioè oggetti che identificano in modo univoco un elemento del detector. In particolare in questo task viene utilizzata la touchable per verificare se lo step viene effettuato nel calorimetro elettromagnetico, in caso positivo si registra l'evento attraverso la classe $Analysis$. Viene usato $GetVolume$ per fornire il volume fisico in cui mi trovo e $GetCopyNo$ per fornire il copy number del volume fisico (i copy-number del calorimetro EM sono 10 o 11). Si memorizza il deposito di energia in un punto casuale tra il punto iniziale e finale di uno step.

```
void SteppingAction::UserSteppingAction( const G4Step * theStep )
{
  // Check energy deposition
  G4double edep = theStep->GetTotalEnergyDeposit();
  if(edep == 0.0) { return; }

  //We need to know if this step is done inside the EM calo or not.
  //We ask the PreStepPoint the volume copy number.
  //Remember: EM calo has copy num 10 or 11
  //We could have asked the volume name, but string
  //comparison is not efficient
  const G4VTouchable* touchable = theStep->GetPreStepPoint()->GetTouchable();
  G4int volCopyNum = touchable->GetVolume()->GetCopyNo();
  if ( volCopyNum == 10 || volCopyNum == 11 ) //EM calo step
    {
      // Find out position along axis Z as a random point
      // between pre- and post step points.
      // This randomisation allows to smooth histogram profile independently
      // on histogram binning
      G4double z1 = theStep->GetPreStepPoint()->GetPosition().z();
      G4double z2 = theStep->GetPostStepPoint()->GetPosition().z();
      G4double z  = z1 + G4UniformRand()*(z2 - z1);

      // Save energy deposition
      Analysis::GetInstance()->AddEDepEM( edep, z, volCopyNum );

    }
}
```

Stacking Action Ogni traccia viene inserita all'interno di uno stack. L'ordine in cui vengono processate è del tipo "last-in-first-out", ovvero l'ultima che è stata messa nella stack sarà la prima ad essere processata. Nella classe $G4UserStackingAction$, si deriva da $G4ClassificationOfNewTrack$ il metodo $ClassifyNewTrack$ che permette di classificare le tracce come $fUrgent$, $fWaiting$ o $fPostponeToNextEvent$ oppure possono venire cancellate con $fKill$. Le tracce nell'$Urgent$ stack sono le prime ad essere processate, quando questo stack si svuota, le tracce nel $Waiting$ stack vengono trasferite nell'$Urgent$ stack e processate. A questo punto ho la fine dell'evento. All'evento successivo le tracce nel $PostponetoNextEvent$ stack vengono trasferite nell'$Urgent$ stack e processate. Di default tutte le tracce vengono inserite nell'$Urgent$ stack, sarà poi compito dell'utente modificare lo stato delle tracce dove risulta necessario. In questo task si chiama $Analysis$ e si vede se una particella è una secondaria attraverso il comando $GetParentID > 0$, se accade ciò si ricava il tipo di particella con $aTrack- > GetDefinition$ e si aggiunge alle particelle secondarie ($AddSecondary$), altrimenti, la particella è una primaria per cui non ha un ParentID (viene settato a -1), si chiama $SetBeam$ e si ricava il tipo di particella e la sua energia cinetica ($GetKineticEnergy$).

```
G4ClassificationOfNewTrack
StackingAction::ClassifyNewTrack( const G4Track * aTrack )
{
  // always "urgent" in current applications
  G4ClassificationOfNewTrack result( fUrgent );

  if ( aTrack->GetParentID() > 0 )//This is a secondary
    {
      Analysis::GetInstance()->AddSecondary(aTrack->GetDefinition());
    }
  else // This is primary
    {
      Analysis::GetInstance()->SetBeam(aTrack->GetDefinition(),
                                       aTrack->GetKineticEnergy());
    }

  return result;
```

3.1.2 Physiscs List

Nella Physics List viene utilizzata *G4EmStandardPhysics* che è la physics list elettromagnetica standard di Geant4 in cui sono definiti: γ, e^+, e^-, alcuni adroni e alcuni ioni fino a 100 TeV. Vengono sempre definite le pseudo-particelle Geantino e Geantino carico. I processi che vengono definiti sono quelli elettromagnetici standard della physics list a cui si aggiunge il processo di trasporto.

```
void PhysicsList::ConstructParticle()
{
  // In this method, static member functions should be called
  // for all particles which you want to use.
  // This ensures that objects of these particle types will be
  // created in the program.

  // pseudo-particles
  G4Geantino::GeantinoDefinition();
  G4ChargedGeantino::ChargedGeantinoDefinition();

  // define gamma, e+, e- and some charged Hadrons
  emPhysicsList->ConstructParticle();
}

void PhysicsList::ConstructProcess()
{
  AddTransportation();
  emPhysicsList->ConstructProcess();
}
```

3.1.3 Primary Generator Action

Nella *PrimaryGeneratorAction*, utilizzando la General Particle Source (gps), vengono definiti i pioni positivi da *G4ParticleTable*. Si sceglie un fascio monoenergetico di 2 GeV.

```
// particle type
G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
G4ParticleDefinition* pion = particleTable->FindParticle("pi+");
gps->GetCurrentSource()->SetParticleDefinition(pion);

// set energy distribution
G4SPSEneDistribution *eneDist = gps->GetCurrentSource()->GetEneDist() ;
eneDist->SetEnergyDisType("Mono"); // or gauss
eneDist->SetMonoEnergy(2.0*GeV);

// set position distribution
G4SPSPosDistribution *posDist = gps->GetCurrentSource()->GetPosDist();
posDist->SetPosDisType("Beam");  // or Point,Plane,Volume,Beam
posDist->SetCentreCoords(G4ThreeVector(0.0,0.0,-16.5*cm));
posDist->SetBeamSigmaInX(0.1*mm);
posDist->SetBeamSigmaInY(0.1*mm);

// set angular distribution
G4SPSAngDistribution *angDist = gps->GetCurrentSource()->GetAngDist();
angDist->SetParticleMomentumDirection( G4ThreeVector(0., 0., 1.) );
angDist->SetAngDistType("beam2d");
angDist->SetBeamSigmaInAngX(0.1*mrad);
angDist->SetBeamSigmaInAngY(0.1*mrad);
angDist->DefineAngRefAxes("angref1",G4ThreeVector(-1.,0.,0.));
```

3.1.4 Risultati della simulazione

Si mostra l'output della simulazione lanciando 10 π^+ e alcuni dati salvati.

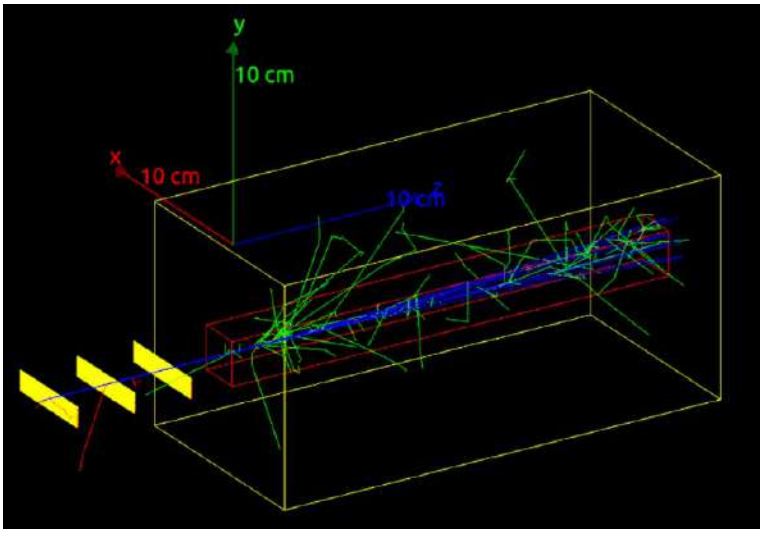


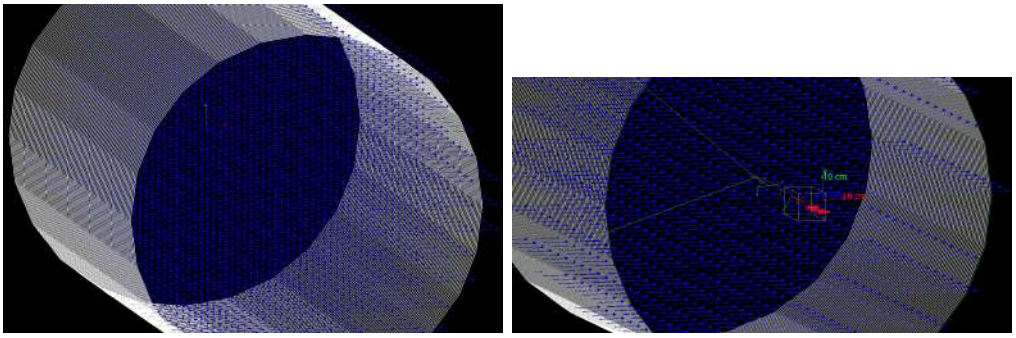
Figura 3.1: Simulazione di 10 pioni postivi.

```
Summary for run: 0
  Beam of pi+ kinetic energy: 2 GeV
  Event processed:          10
  Average number of gamma: 63.7
  Average number of e-    : 359.1
  Average number of e+    : 0.9
  Average energy deposition in EM calo: 273.102 MeV
  Normalized energy in EM calo:          0.136551  RMS: 0.0164604
  Normalized energy in central crystal: 0.133967  RMS: 0.0145377
  Ratio of central crystal to total:     0.981075
```

3.2 Task3b

3.2.1 Detector Construction

Rispetto agli esercizi precedenti vengono aggiunti al detector un calorimetro adronico e un campo magnetico uniforme. Il calorimetro adronico è un tubo di raggio pari a 800 mm e spesso 4 mm. È composto da 80 strati alternati di Ferro (Fe) e Argon liquido (LAr). I layer di Fe sono spessi 20 mm e svolgono la funzione di assorbire la radiazione incidente mentre i layer di LAr sono spessi 4 mm e sono la parte attiva di materiale. Nella classe *DetectorConstruction* è stato aggiunto un campo magnetico uniforme di $3.5 \cdot 10^{-3}$ T diretto nel verso positivo dell'asse x. Nella simulazione è stato possibile inserire la visualizzazione del campo magnetico attraverso il comando UI: $\backslash vis \backslash scene \backslash add \backslash magneticField$.



Dato che lo spin risulta proporzionale al momento della particella, con l'aggiunta di un campo magnetico uniforme si aggiunge il moto di precessione dello spin all'interno del campo magnetico.

$$\frac{d\vec{\mu}}{dt} = \gamma\vec{\mu} \times \vec{B} \tag{3.1}$$

$\vec{\mu}$ è il momento della particella, γ è il rapporto giromagnetico e $\vec{B}$ il campo magnetico

Lo spin precede attorno al campo magnetico alla frequenza di Larmor pari a $\omega_L = \gamma|\vec{B}|$.

In *DetectorConstruction*, dopo aver inserito il campo magnetico e definito l'equazione di moto, si utilizza Runge-Kutta 4 (RK4) per risolvere l'equazione differenziale ordinaria e infine si aggiunge il chord finder. Dopo aver risolto numericamente l'equazione differenziale con RK4, Geant4 divide la traiettoria reale in vari segmenti. Utilizzando chord finder si determinano i segmenti che approssimano al meglio la traiettoria reale. In generale uno step può consistere in più di un chord.

```
G4FieldManager* DetectorConstruction::GetLocalFieldManager()
{
  // pure magnetic field
  G4MagneticField* fMagneticField =
    new G4UniformMagField(G4ThreeVector(3.5e-3*tesla, 0., 0.));

  // equation of motion with spin
  G4Mag_EqRhs* fEquation = new G4Mag_SpinEqRhs(fMagneticField);

  // local field manager
  G4FieldManager* fFieldManager = new G4FieldManager();
  fFieldManager->SetDetectorField(fMagneticField );

  // default stepper Runge Kutta 4th order
  G4MagIntegratorStepper* fStepper = new G4ClassicalRK4( fEquation , 12); // spin needs
12 dof
  //  G4MagIntegratorStepper* fStepper = new G4SimpleRunge( fEquation , 12); // spin
needs 12 dof


  // add chord finder
  G4double fMinStep=1*mm;
  G4ChordFinder* fChordFinder = new G4ChordFinder( fMagneticField, fMinStep,fStepper);
  fFieldManager->SetChordFinder( fChordFinder );
  return fFieldManager;
}
```

3.2.2 Physics List

Nella Physics List sono stati aggiunti i decadimenti dei muoni positivi e negativi. In primo luogo si aggiunge la definizione di muoni positivi e negativi oltre a quella di tutte le altre particelle di interesse (γ, e^+, e^-, ν_e, $\overline{\nu_e}$, ν_μ, $\overline{\nu_\mu}$).

```
G4MuonPlus::MuonPlusDefinition();
G4MuonMinus::MuonMinusDefinition();
```

Per aggiungere il processo di decadimento dei Muoni si è include in *ConstructParticle* la *G4DecayTable* (tabella di decadimento) che include decadimenti forti, deboli ed elettromagnetici.

```
G4DecayTable* MuonPlusDecayTable = new G4DecayTable();
MuonPlusDecayTable -> Insert(new G4MuonDecayChannelWithSpin("mu+",0.986));
MuonPlusDecayTable -> Insert(new G4MuonRadiativeDecayChannelWithSpin("mu+",0.014));
G4MuonPlus::MuonPlusDefinition() -> SetDecayTable(MuonPlusDecayTable);

G4DecayTable* MuonMinusDecayTable = new G4DecayTable();
MuonMinusDecayTable -> Insert(new G4MuonDecayChannelWithSpin("mu-",0.986));
MuonMinusDecayTable -> Insert(new G4MuonRadiativeDecayChannelWithSpin("mu-",0.014));
G4MuonMinus::MuonMinusDefinition() -> SetDecayTable(MuonMinusDecayTable);
```

Ai muoni si aggiungono anche i processi di scattering multiplo, ionizzazione, bremsstrahlung e produzione di coppie. Questi vengono aggiunti al *ProcessManager* attraverso il comando *AddProcess*. Gli indici indicano l'ordine in cui i processi vengono eseguiti, più l'indice è alto più il processo ha priorità. Il primo indice si riferisce all'azione *AtRest*, il secondo all'azione *AlongStep* e il terzo all'azione *PostStep*. Se l'indice è -1 il processo non risulta attivo.

```
 G4ProcessManager* pmanager = particle->GetProcessManager();
else if( particleName == "mu+" ||
         particleName == "mu-"    ) {

pmanager->AddProcess(new G4MuMultipleScattering,-1, 1, 1);
pmanager->AddProcess(new G4MuIonisation,         -1, 2, 2);
pmanager->AddProcess(new G4MuBremsstrahlung,     -1, 3, 3);
pmanager->AddProcess(new G4MuPairProduction,     -1, 4, 4);
```

In *ConstructDecay* si costruiscono i processi di decadimento dei muone positivo e negativo e si aggiungono al *ProcessManager*.

```
G4Decay* muDecayProcess = new G4DecayWithSpin();

G4ParticleDefinition* muMinus= G4MuonMinus::MuonMinusDefinition();
G4ParticleDefinition* muPlus= G4MuonPlus::MuonPlusDefinition();

G4ProcessManager* muMinusManager = muMinus->GetProcessManager();
muMinusManager->AddProcess(muDecayProcess);
muMinusManager->AddProcess(muDecayProcess, idxPostStep);
muMinusManager->AddProcess(muDecayProcess, idxAtRest);

G4ProcessManager* muPlusManager = muPlus->GetProcessManager();
muPlusManager->AddProcess(muDecayProcess);
muPlusManager->AddProcess(muDecayProcess, idxPostStep);
muPlusManager->AddProcess(muDecayProcess, idxAtRest);
```

3.2.3 Output

Simulazione di 100 μ^-:

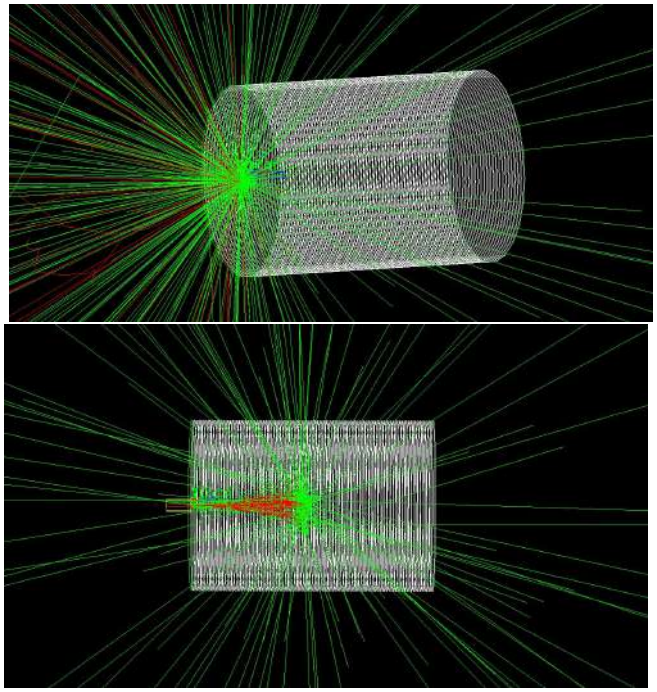


Figura 3.2: Simulazione di 100 muoni negativi.

Considerando il file *muon.mac* si sono simulati 2 000 000 muoni negativi di 1 GeV e si è osservata la posizione e il tempo di decadimento del muone e i tempi di decadimento degli elettroni forward e backward (figura 3.2.3). Considerando il momento lungo l'asse z: se il momento è positivo l'elettrone sarà forward mentre se è negativo sarà backward.

In figura 3.2.3 sono riportati due istogrammi in cui è stato svolto un fit del tempo di decadimento.

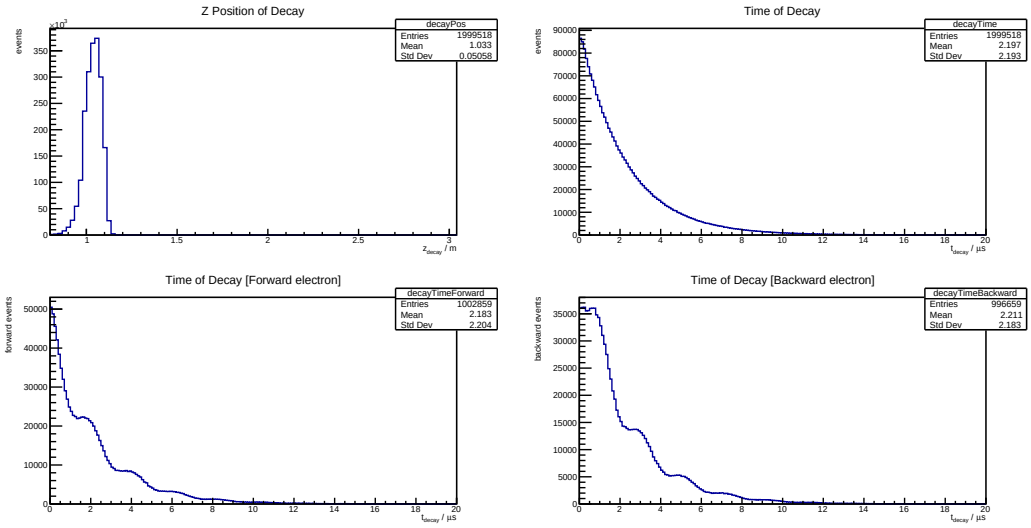


Figura 3.3: Posizione lungo l'asse z del μ (altro a sinistra), tempo di decadimento del μ (alto a destra), tempo di decadimento del e^- Forward (basso a sinistra), tempo di decadimento del e^- Backward (basso a destra).

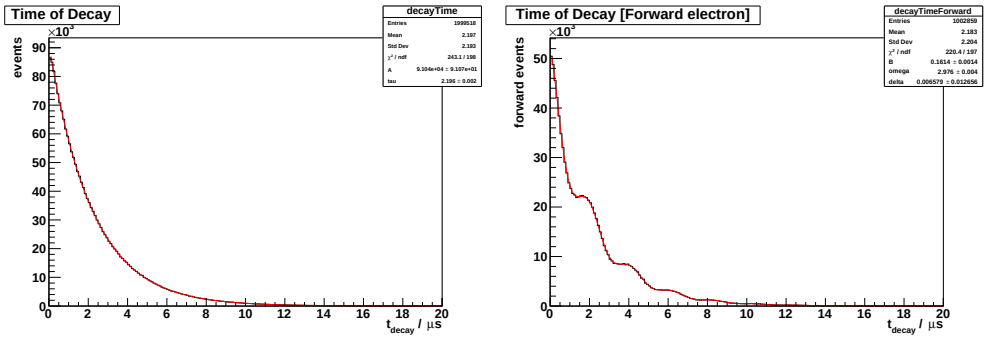


Figura 3.4: Fit del tempo di decadimento.

Il valore teorico del tempo di decadimento del muone è $\tau_{teorico} = 2.197\ \mu s$. Il decadimento del muone segue un andamento esponenziale e dal fit si è ricavata la vita media del muone pari a $\tau = 2.196 \pm 0.002\ \mu s$ che dista 0.5σ dal valore teorico. Per cui si conclude che il valore ricavato dalla simulazione è compatibile con il valore esatto. Nel fit del tempo di decadimento dell'elettrone forward si è tenuto conto anche della dipendenza dallo spin. La funzione di fit utilizzata è:

$$A \cdot e^{-t/\tau} \cdot (1 + B \cdot \cos(\omega t + \delta)) \tag{3.2}$$

Capitolo 4

TASK 4

File sorgente: Task4

4.1 Task4a

4.1.1 Physics List

QGSP è la Physics List di base in cui si usa il Quark Gluon String Precompuond Model per interazioni ad alte energie di protoni, neutroni, pioni, kaoni e nuclei. QGSP BERT è simile alla QGSP ma viene usata la cascata Bertini per i protoni, neutroni, pioni e kaoni primari sotto i 10 GeV. Il modello Bertini produce più protoni e neutroni secondari rispetto al modello LEP, portando a una concordanza migliore con i dati sperimentali. Questa Physics List è la più raccomandata per studiare la fisica delle alte energie.

```
G4VUserPhysicsList* physics = new QGSP_BERT();//new PhysicsList();
runManager->SetUserInitialization(physics);
```

Nel caso si voglia vedere, per un dato tipo di particella, quali processi sono attivi si possono utilizzare i comandi UI sotto riportati. Si sono scelti neutroni e pioni positivi e vengono riportati i risultati di output solamente di due processi, ovviamente la lista sarebbe più lunga.

```
  /particle/select neutron
  /particle/process/dump
G4ProcessManager: particle[neutron]
[0]=== process[Transportation :Transportation] Active
  Ordering::             AtRest             AlongStep          PostStep
                   GetPIL/    DoIt    GetPIL/    DoIt    GetPIL/    DoIt
  Ordering::
  index                 -1:      -1:       0:       0:       5:       0:
  parameter             -1:      -1:       0:       0:       0:       0:
[1]=== process[Decay :Decay] Active
  Ordering::             AtRest             AlongStep          PostStep
                   GetPIL/    DoIt    GetPIL/    DoIt    GetPIL/    DoIt
  Ordering::
  index                  0:       0:      -1:      -1:       4:       1:
  parameter           1000:    1000:      -1:      -1:    1000:    1000:
```

```
/particle/select pi+
/particle/process/dump
G4ProcessManager: particle[pi+]
[0]=== process[Transportation :Transportation] Active
  Ordering::              AtRest              AlongStep           PostStep
                    GetPIL/    DoIt     GetPIL/    DoIt     GetPIL/    DoIt
  Ordering::
  index                 -1:       -1:        2:        0:        7:        0:
  parameter             -1:       -1:        0:        0:        0:        0:
[1]=== process[msc :Electromagnetic] Active
  Ordering::              AtRest              AlongStep           PostStep
                    GetPIL/    DoIt     GetPIL/    DoIt     GetPIL/    DoIt
  Ordering::
  index                 -1:       -1:        1:        1:       -1:       -1:
  parameter             -1:       -1:        1:        1:       -1:       -1:
```

4.1.2 Detector Construction e Sensitive Detector

In Detector Construction si utilizza il Sensitive Detector (SD) per dichiarare gli elementi geometrici che si vuole che siano sensibili al passaggio delle particelle. Il SD da all'utente una "maniglia" (handle) per interagire con il detector e ricavarne le quantità fisiche necessarie. Il rivelatore è composto da un calorimetro adronico cilindrico di strati alternati di assorbitore (Fe) e materiale attivo (LAr). Si dichiarano come SD i layers di Argon liquido.
Per creare un Sensitive Detector si scrive la classe $HadCaloSensitiveDetector$ ereditata da $G4VSensitiveDetector$. I metodi presenti in questa classe sono:
$Initialize$ (chiama il SD all'inzio dell'evento), $EndOfEvent$ (chiama la fine dell'evento) e $ProcessHits$ (dice come processare le hits ad ogni step). Quest'ultimo metodo prende in input il $G4Step$ e la $G4TouchableHistory$ e viene chiamato ad ogni step del volume logico a cui il SD viene associato. La $G4TouchableHistory$ viene utilizzata per identificare dove si trova lo step e quindi ricavare il $CopyNumber$ del volume ad esso associato (ogni layer di LAr ha un copy number unico). Poichè ogni step contiene informazioni sul volume e sul materiale ad esso associati, per ottenere le quantità fisiche d'interesse si interroga il $PreStepPoint$ del $G4Step$; in questo caso di ricava l'energia depositata. A questo punto viene chiamata la classe $Analysis$ per memorizzare l'energia depositata nel layer di LAr associato allo step. Viene interrogato lo step anche per sapere se l'energia depositata appartiene ad una particella primaria o secondaria.

```
G4bool HadCaloSensitiveDetector::ProcessHits(G4Step *step, G4TouchableHistory *)
{
        //This method is called every time a G4Step is performed in the logical volume
        //to which this SD is attached: the HAD calo.

        //To identify where the step is we use the touchable navigation,
        //remember we need to use PreStepPoint!
        G4TouchableHandle touchable = step->GetPreStepPoint()->GetTouchableHandle();
        G4int copyNo = touchable->GetVolume(0)->GetCopyNo();
        //Hadronic layers have number from 1001 to 1080. The index is from 0 to 79:
        G4int layerIndex = copyNo-1001;
        //We get now the energy deposited by this step
        G4double edep = step->GetTotalEnergyDeposit();

        //This line is used to store in Analysis class the energy deposited in this layer
        //The Analysis class will summ up this edep to the current event total energy in this layer
        Analysis::GetInstance()->AddEDepHad(layerIndex,edep);

        //check if edep is from primary or secondary:
        G4String isPri = step->GetTrack()->GetTrackID() == 1 ? "Yes" : "No";
        //--------------
        //Exercise 2 task4a
        //-------------
        G4cout<<"Layer: "<<layerIndex<<" (volume CopyNo: "<<copyNo<<") Edep="<<G4BestUnit(edep,"Energy")<<" isPrimary? "<<isPri;
        G4cout<<" (name="<<step->GetTrack()->GetDefinition()->GetParticleName()<<")"<<G4endl;
        return true;
}
```

Si nota che gli step che attraversano materiali non sensibili al passaggio delle particelle vengono ignorati, mentre quando la particella attraversa uno dei materiali sensibili viene richiamato il metodo *ProcessHits*. In *Detectorconstruction.cc* si utilizza il SD *HadCaloSensitiveDetector* e si associa al volume logico degli strati di LAr. *G4SDManager* rappresenta il database che Geant4 crea per memorizzare cosa si sta facendo.

```
//Step 1
HadCaloSensitiveDetector* sensitive = new HadCaloSensitiveDetector("/HadClo");

//Step 2
G4SDManager* sdman = G4SDManager::GetSDMpointer();
sdman->AddNewDetector( sensitive );

//Step 3
hadLayerLogic->SetSensitiveDetector(sensitive);
```

Nelle ultime righe di codice sopra riportate abbiamo l'output che viene mostrato a terminale, ovvero Copy Number, energia depositata, nome della particella e se quest'ultima è primaria o secondaria. Di seguito si riportano un paio di righe presenti in output.

```
Layer: 79 (volume CopyNo: 1080) Edep=1.42877 MeV isPrimary? Yes (name=mu-)
Layer: 77 (volume CopyNo: 1078) Edep=1.12465 MeV isPrimary? No (name=e-)
```

4.1.3 Output

In figura 4.1.3, seguendo il file *sd*2.*mac* si è simulato un muone negativo di 20 GeV.

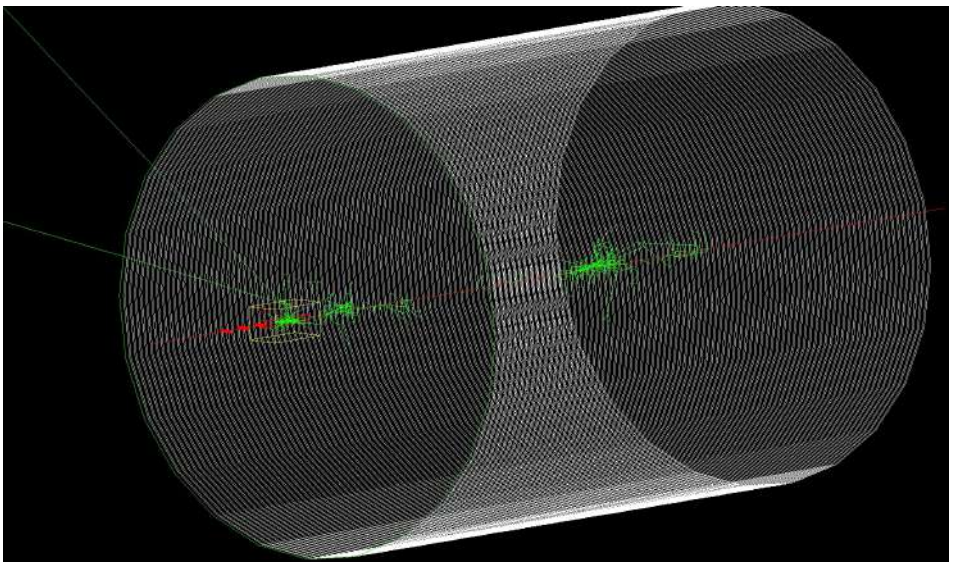


Figura 4.1: Simulazione di $1\mu^-$ di 20 GeV.

L'output riporta l'energia depositata nel rivelatore nel calorimetro elettromagnetico e nel calorimetro adronico, e il numero medio di particelle secondarie generate. Si noti che il muone rilascia poca della sua energia nei calorimetri, in totale solo 1,21 GeV vengono depositati nei calorimetri che corrisponde a circa il 6 % dell'energia iniziale del muone.

```
Event processed: 1
Average number of secondaries: 2476
Average energy in EM calo: 806.911 MeV
Average energy in Had calo: 214.916 MeV
```

4.2 Task4b

4.2.1 Stacking Action

Nella classe Stacking Action l'utente può cambiare lo stato di una traccia, dopo che questa viene creata. Si utilizza il metodo $ClassifyNewTrack$ che ha in input la traccia della particella. In questo esercizio viene posta una soglia (threshold) a 30 MeV. Si trova il tipo di particella dalla sua traccia comparando la variabile $ParticleType$ con la definizione statica di $Gamma$ o $Neutrone$. Si verifica se la particella (gamma o neutrone) ha un'energia cinetica superiore alla soglia. L'energia cinetica della particella si ricava dalla traccia con $aTrack- > GetKineticEnergy()$.Se accade ciò, attraverso la classe Analysis, si aumenta il contatore di 1. In questo task vengono poste di default le particelle nello stack $fUrgent$. Si sceglie di "uccidere"" i gamma che, se troppo energetici, farebbero produzione di coppie e quindi darebbero origine ad uno sciame elettromagnetico con altri γ come secondarie. Per uccidere i γ si utilizza il comando $fKill$.

```
G4double thresh = 30*MeV;

if ( particleType == G4Gamma::GammaDefinition() ){
  if (aTrack->GetKineticEnergy() > thresh){

    analysis->AddGammas(1);
  }

}

 if ( particleType == G4Neutron::NeutronDefinition() ){
  if (aTrack->GetKineticEnergy() > thresh){

    analysis->AddNeutrons(1);
  }
}
if ( particleType == G4Gamma::GammaDefinition() ){

    result = fKill;
}
```

4.2.2 Output

Nella simulazione vengono lanciati rispettivamente un'elettrone negativo da 2 GeV e un muone negativo da 20 GeV. Per entrambe le simulazioni si sono mostrati a terminale i risultati con e senza il processo di "uccisione" dei gamma. Come ci si aspetta, il numero medio di gamma cambia, i conteggi dei gamma diminuiscono utilizzando il comando $fKill$, mentre il numero di neutroni rimane circa lo stesso.

Output dell'elettrone senza il comando $fKill$:

```
Event processed: 1
Average number of secondaries: 1759
Average energy in EM calo: 0 eV
Average energy in Had calo: 131.911 MeV
Average number of gammas: 3
Average number of neutrons: 10
```

Output dell'elettrone con il comando $fKill$:

```
Event processed: 1
Average number of secondaries: 376
Average energy in EM calo: 0 eV
Average energy in Had calo: 53.1696 MeV
Average number of gammas: 0
Average number of neutrons: 8
```

Output del muone senza il comando $fKill$:

```
Event processed: 1
Average number of secondaries: 30536
Average energy in EM calo: 0 eV
Average energy in Had calo: 1.35374 GeV
Average number of gammas: 179
Average number of neutrons: 4
```

Output del muone con il comando $fKill$:

```
Event processed: 1
Average number of secondaries: 664
Average energy in EM calo: 0 eV
Average energy in Had calo: 181.675 MeV
Average number of gammas: 4
Average number of neutrons: 12
```

Nella simulazione del muone negativo è più chiaro il grande cambiamento nel conteggio dei raggi gamma in cui si passa da 179 a 4.

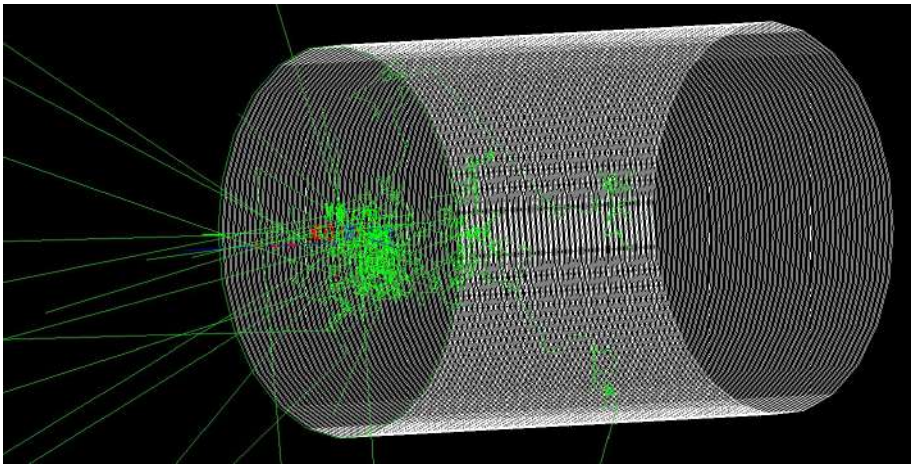


Figura 4.2: Simulazione di un e^- da 2 GeV.

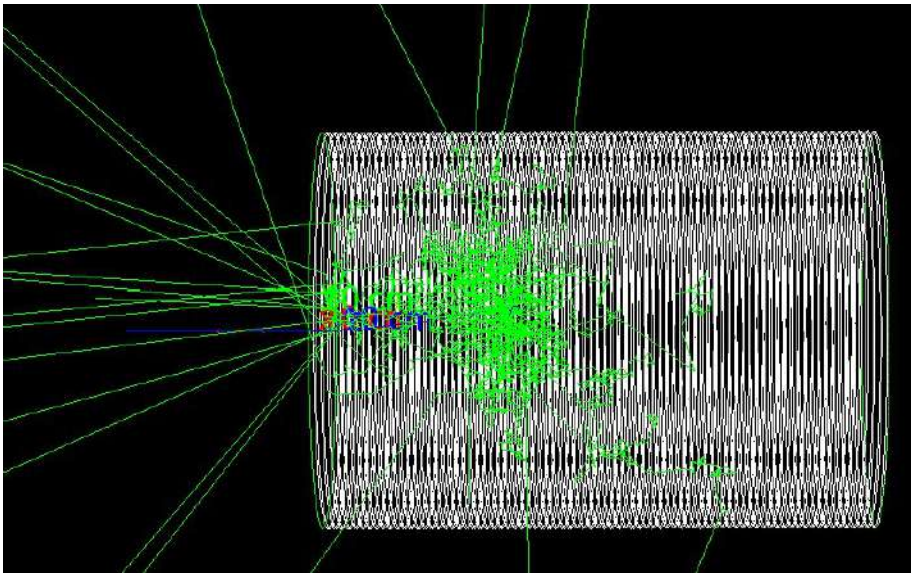


Figura 4.3: Simulazione di un μ^- da 20 GeV.

4.3 Task4c

4.3.1 Hits

Le Hits sono uno snapshot dell'interazione fisica di uno step (traccia) oppure un'accumulazione di traccie nella regione sensibile del detector. L'output della simulazione basico è chiamato Hit (una classe definita dall'utente da $G4VHit$): un deposito di energia nello spazio e nel tempo. Di solito non si è interessati alle Hit di tutti gli elementi del detector, ma invece si vuole memorizzare le informazioni solo per alcuni componenti rilevanti. In Geant4 questo scopo viene raggiunto con i concetti di hits e Sensitive Detector (SD). Le Hit vengono create nel SD per salvare delle quantità fisiche che dipendono dal tipo di rivelatore. Un rivelatore di tracciamento genera una hit per ogni step di ogni singola traccia. Una Hit di questo tipo tipicamente contiene posizione, tempo, deposito di energia e ID della traccia della particella. Invece, una hit di un calorimetro accumula il deposito di energia in ogni "cella" per tutti gli step di tutte le tracce e tipicamente la Hit contiene la somma dell'enrgia depositata e l'ID della cella.
Le Hit devono essere univocamente identificate e raccolte in una collezione ($HitContainer$). Geant4 provvede a creare un database che tiene conto di tutte le Hit create.

Nel file $SensitiveDetector.cc$ si implementa il SD $HadCaloSensitiveDetector$. Nel costruttore si dichiara il nome della Hit Collection utilizzando $myCollectionName$ e lo aggiungo con $myCollectionName.insert$.

```
HadCaloSensitiveDetector::HadCaloSensitiveDetector(G4String SDname)
  : G4VSensitiveDetector(SDname)
{
        G4cout<<"Creating SD with name: "<<SDname<<G4endl;

        G4String myCollectionName = "HadCaloHitCollection";

        collectionName.insert(myCollectionName);
}
```

In generale il Sensitive Detector può avere più di una collection, in questo caso possiamo aggiungere più nomi per ogni Hit Collection che si vuole utilizzare.
La Hit Collection viene creata nel metodo $Initialize$ del SD. Il costruttore della Hit Collection vuole due parametri in input: il nome del SD ($GetName$) e il nome della Collection che, avendone solo una, si trova nella posizione 0 di $CollectionName$. La coppia nome del SD più nome della Hit Collection è unica. Per aggiungere la Hit Collection ($AddHitsCollection$) bisogna prima definire un'indice univoco e attraverso il metodo $GetCollectionID$. Con $hitMap.clear()$ resetto la mappa delle hits.

```
void HadCaloSensitiveDetector::Initialize(G4HCofThisEvent* HCE)
{
        hitCollection = new HadCaloHitCollection(GetName(), collectionName[0]);

        static G4int HCID = -1;
        if (HCID<0) HCID = GetCollectionID(0); // <<-- this is to get an ID for collectionName[0]
        HCE->AddHitsCollection(HCID, hitCollection);

        //Reset map of hits
        hitMap.clear();
}
```

Il metodo fondamentale della classe $SensitiveDetector$ è $ProcessHits$. Questo metodo viene implementato ogni volta che viene eseguito il $G4Step$ nel volume logico

associato al SD; in questo caso i layers di LAr. Quindi si ricavano l'energia depositata e il copy number del layer associati ad un *G4Step*. La Hit che viene creata dovrà memorizzare queste due informazioni.

Se la particella non deposita energia nel calorimetro adronico ($edep = 0$), per cui se si tratta di γ o neutroni, non si fa nulla. Dato che il SD è associato agli strati attivi del calorimetro adronico e, in generale, un calorimetro misura solo il deposito di energia totale all'interno del layer, ci interessa creare solamente una hit per ogni layer. Si inizializza la Hit a zero e si vede se è già stata creata una hit per uno specifico layer con $hitMap$. Nel caso in cui la hit per un certo layer non esiste, viene creata e inserita nella Hit Collection. Il valore che otteniamo è la somma dell'energia depositata dalla particella nei layers.

```
G4bool HadCaloSensitiveDetector::ProcessHits(G4Step *step, G4TouchableHistory *)
{
        G4TouchableHandle touchable = step->GetPreStepPoint()->GetTouchableHandle();
        G4int copyNo = touchable->GetVolume(0)->GetCopyNo();
        //Hadronic layers have number from 1001 to 1080. The index is from 0 to 79:
        G4int layerIndex = copyNo-1001;
        //We get now the energy deposited by this step
        G4double edep = step->GetTotalEnergyDeposit();

        //If the step does not have energy depostion (for example because the particle is
        //a gamma or neutron, then we do not have to do anything
        if ( edep == 0 )
        {
                return true; }

        hitMap_t::iterator it = hitMap.find(layerIndex);
        HadCaloHit* aHit = 0;
        if ( it != hitMap.end() )
        {
                //Hit for this layer already exists
                //remember the hit pointer
                aHit = it->second;
        }
        else
        {
                //Hit for this layer does not exists,
                //we create it
                aHit = new HadCaloHit(layerIndex);
                hitMap.insert( std::make_pair(layerIndex,aHit) );
                hitCollection->insert(aHit);
        }
        aHit->AddEdep( edep );
        return true;
}
```

Nel metodo $EndOfEvent$ del Sensitive Detector viene implementato il print-out delle hits.

```
void HadCaloSensitiveDetector::EndOfEvent(G4HCofThisEvent*)
{

        hitCollection->PrintAllHits();
}
```

4.3.2 Output

È stato simulato un muone negativo di 5 GeV. Di seguito si riportano i risultati ottenuti a terminale per i primi 14 layers.
Nei primi layer abbiamo rilasci di energia variabili mentre negli ultimi layer il rilascio è circa nullo. Per cui si conclude che il calorimetro adronico funziona nell'arrestare la particella.

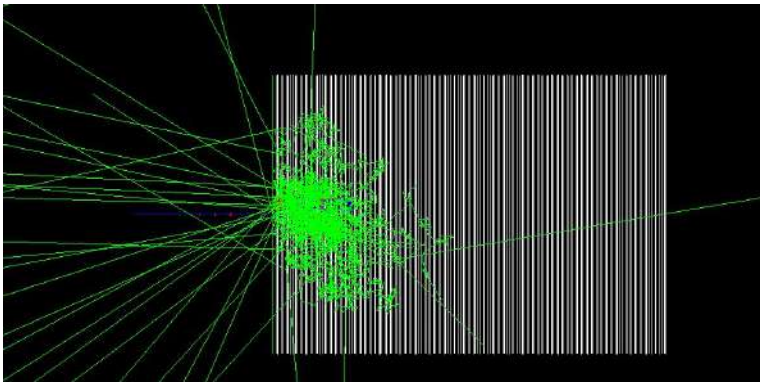


Figura 4.4: Simulazione di un μ^- di 5 GeV.

```
Event processed: 1
Average number of secondaries: 6011
Average energy in EM calo: 0 eV
Average energy in Had calo: 292.105 MeV
    Average energy in Layer 0: 1.31758 MeV
    Average energy in Layer 1: 29.0362 MeV
    Average energy in Layer 2: 3.96121 MeV
    Average energy in Layer 3: 6.73335 MeV
    Average energy in Layer 4: 84.8918 MeV
    Average energy in Layer 5: 15.337 MeV
    Average energy in Layer 6: 15.9296 MeV
    Average energy in Layer 7: 21.7769 MeV
    Average energy in Layer 8: 13.1168 MeV
    Average energy in Layer 9: 6.87002 MeV
    Average energy in Layer 10: 41.9304 MeV
    Average energy in Layer 11: 19.2868 MeV
    Average energy in Layer 12: 12.8675 MeV
    Average energy in Layer 13: 737.484 keV
    Average energy in Layer 14: 1.0141 MeV
```

Capitolo 5

TASK 6

File sorgente: Task6

5.0.1 Detector Construction

Nella classe *DetectorConstruction* si definiscono i materiali dal database NIST da cui si ricava il polietilene (CH_2) e l'Argon in forma gassosa. Il Detector è composto solo da uno strato di polietilene spesso 50 μm e uno strato di materiale attivo (Ar) spesso 3 mm. L'Argon emette luce di scintillazione alla passaggio della radiazione, per cui ha il compito di rivelare l'energia depositata dalle particelle. Il polietilene invece ha il compito di far iniziare la shower adronica.
La simulazione viene eseguita lanciando neutroni al variare del materiale assorbitore. In particolare, oltre al polietilene, vengono usati: oro, rame, carbonio e alluminio. I neutroni possono interagire con la materia attraverso: scattering elastico, scattering anelastico, cattura neutronica e fissione. Utilizzando materiali assorbitori più leggeri (come l'idrogeno) i neutroni vanno maggiormente incontro a scattering elastico per cui vengono moderati. Rallentandoli i neutroni perdono gradualmente la loro energia. Con materiali pesanti (Au,Cu) si aggiungo i processi di scattering inelastico e cattura neutronica. Variando l'energia dei neutroni e il materiale assorbitore l'energia depositata cambia.
Per osservare ciò, si aggiunge il *SensitiveDetector* al rivelatore per estrarre l'energia depositata e si mostrano gli istogrammi al variare del numero di neutroni lanciati e del materiale assorbitore.

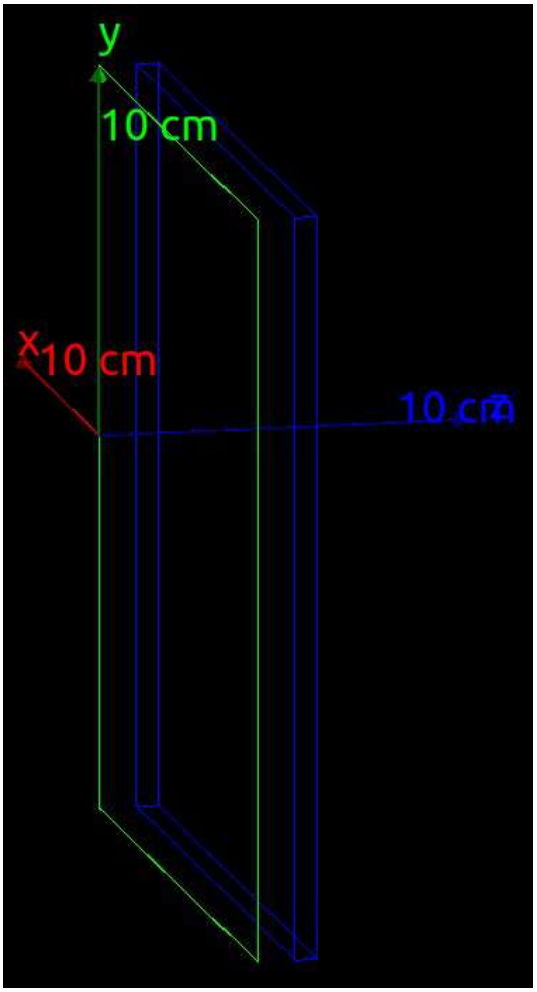


Figura 5.1: Setup del detector con un layer di polietilene e un layer di Argon.

Parte II

Garfield

In questa parte non viene utilizzato *Geant4* ma si utilizzano *Ansys* e *Garfield* per le simulazioni. In particolare si vuole simulare un rivelatore GEM.

La GEM (Gas Electron Multiplier) si compone di un foglio GEM formato da una layer di materiale isolante, che nel nostro caso è il Kapton di spessore 50 μm. Su entrambe le facce del layer di Kapton ci sono degli elettrodi in rame spessi 5 μm. Si perfora il foglio di rame e kapton al fine di ottenere una matrice molto fitta di fori di diametro 70 μm distanti 140 μm (pitch). Ogni foro si comporta come un contatore proporzionale. Applicando una differenza di potenziale tra gli elettrodi in rame si ottiene un campo elettrico tra due fori. In questo modo, quando un'elettrone lo attraversa da luogo ad una moltiplicazione a valanga. Per formare una GEM si impilano più fogli GEM distanti circa 1 mm racchiusi tra un catodo e un anodo. Si applica una differenza di potenziale tra ogni coppia di fogli GEM, tra catodo e il primo foglio GEM e tra anodo e l'ultimo foglio GEM. Tra catodo e anodo si inserisce un gas, che viene ionizzato al passaggio della radiazione creando coppie elettrone-ione. Questi vengono guidati, grazie alla differenza di potenziale, dal catodo sul primo foglio GEM, dove passando all'interno dei fori vengono moltiplicati. Così poi vengono guidati sui successivi fogli GEM fino ad arrivare all'anodo. La carica depositata all'anodo da luogo al segnale che poi verrà amplificato e letto dall'elettronica. Si andrà a simulare un singolo foglio GEM racchiuso tra catodo e anodo e si assume un potenziale elettrostatico. Per semplicità si assume una disposizione di fori simmetrica e quindi si utilizza un foglio GEM con 7 fori mostrato in figura 5.2.

Figura 5.2: Foglio GEM con 7 fori.

Capitolo 6

GEM THICK

File sorgente: GEMThick

Dal programma *Ansys* vengono generati i file *.lis* che inseriamo nel codice della simulazione per definire la field Map (ovvero la geometria del sistema, l'associazione con materiali e la generazione di campi). Inoltre, grazie al fatto che è presente una simmetria nella disposizione dei fori, vengono definite periodicità sia lungo l'asse x che lungo l'asse y.
Utilizzando la classe $ViewField$ si riesce a visualizzare ciò che viene creato come il campo elettrico o il potenziale.
Il mezzo (gas) viene definito tramite la classe $MediumMagboltz$. Magboltz è un database che contiene tutte le sezioni d'urto dell'interazione tra un'elettrone e un particolare gas. Sostanzialmente questo database contiene le proprietà fisiche della GEM. Viene definito il trasporto in funzione di temperatura e pressione. Un altro parametro importante è il fattore di pening, sopratutto per l'Argon che, avendo anche livelli di eccitazione intermedi, può diseccitarsi emettendo un fotone nel range UV. Questo potrebbe avere l'energia giusta (13,7 eV) per eccitare un atomo di CO_2 (con una certa probabilità). Quindi nel guadagno in questi casi devo contare un elettrone in più nella valanga.
Dopo aver creato il nostro gas dobbiamo associarlo al materiale corrispondente nella Field Map. Si può fare in più modi ma in questo caso si è cercata la permeabilità di ogni elemento e si è scelta quella corrispondente alla parte con il gas (permeabilità =1).
Il prossimo step è costruire attraverso la classe *Sensor* il sensore, ovvero il volume all'interno della geometria nel quale faremo muovere i nostri elettroni.
Ora si deve driftare le cariche. Nel metodo *Avanlanche MC* si definisce un certo distance step, per cui, ad ogni step, tramite un metodo montecarlo, si decide cosa accade alla mia particella. Questo è un procedimento veloce anche se non preciso, nel nostro caso è corretto utilizzarlo solo prima di arrivare al foro della GEM poichè la successiva moltiplicazione a valanga è processo locale e si ha necessità di un metodo più preciso. Il metodo utilizzato è *Avalanche Microscopic* che fa i calcoli collisione per collisione. Qui, dopo che l'elettrone percorre circa un libero cammino medio, interagisce con il mezzo e si tiene conto di questa interazione. Riusciamo a tenere conto di ciò che succede a ogni singolo elettrone. *Avanlanche MC* viene utilizzato per gli ioni mentre *Avalanche Microscopic* per gli elettroni.

Il guadagno che viene calcolato corrisponde al numero di elettroni che sono arrivati all'anodo rispetto al numero di elettroni iniziali. Gli elettroni possono interagire e produrre elettroni secondari o andare direttamente sulla GEM. Attraverso il comando *GetElectronEndPoint* posso tenere traccia di tutti gli elettroni; in particolare di dove nascono, dove muoiono e delle loro energie.

Vengono utilizzati 10 elettroni iniziali che attraversano un gas di $ArCO_2$ con due composizioni diverse: la prima 90% Ar e 10% CO_2 mentre la seconda 70% Ar e 30% CO_2. Nei grafici seguenti vengono mostrate le linee di drift degli elettroni (arancione) e degli ioni (rosso). Si nota come i rettangoli verdi sono le proiezioni dei fori della GEM che vengono attraversati dagli elettroni/ioni.

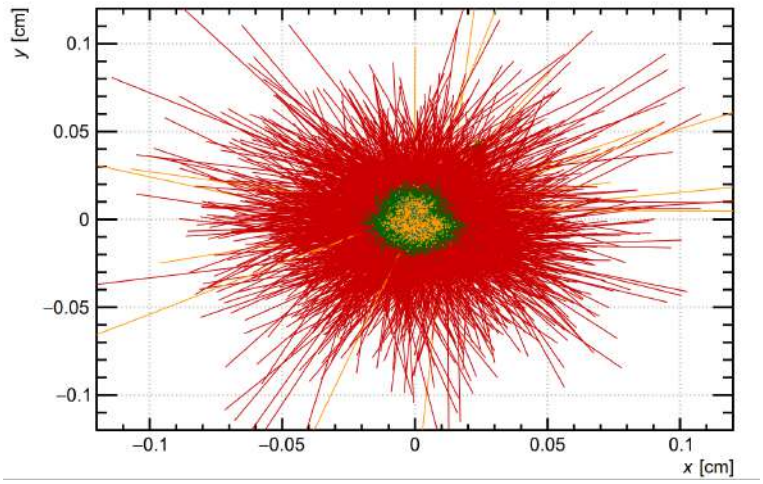


Figura 6.1: Proiezione delle linee di drift nel piano x-y per la miscela 90% Ar e 10% CO_2

6.1 Risultati della simulazione

Vengono riportati i risultati delle simulazioni con le due miscele di gas. Si mostrano gli istogrammi relativi al numero di elettroni secondari, numero di ioni e numero di elettroni e di ioni che hanno impattato sulla parte in plastica della GEM. Per la composizione 70% Ar e 30% CO_2 vengono simulati 100 elettroni primari mentre per 90% Ar e 10% CO_2 vengono simulati 30 elettroni primari.

Si nota come la composizione del gas incida molto sulla produzione di elettroni secondari. Per la miscela 90% Ar e 10% CO_2 si ha infatti un guadagno di 797.067 che nel caso della miscela 70% Ar e 30% CO_2 si riduce a solamente 15.21. Questo è dovuto alla maggior presenza di CO_2 (30% al posto che 10%) che è un gas *quencher*, ovvero riduce la probabilità di estrarre gli elettroni dagli elettrodi in rame e quindi riduce la probabilità di creare elettroni secondari.

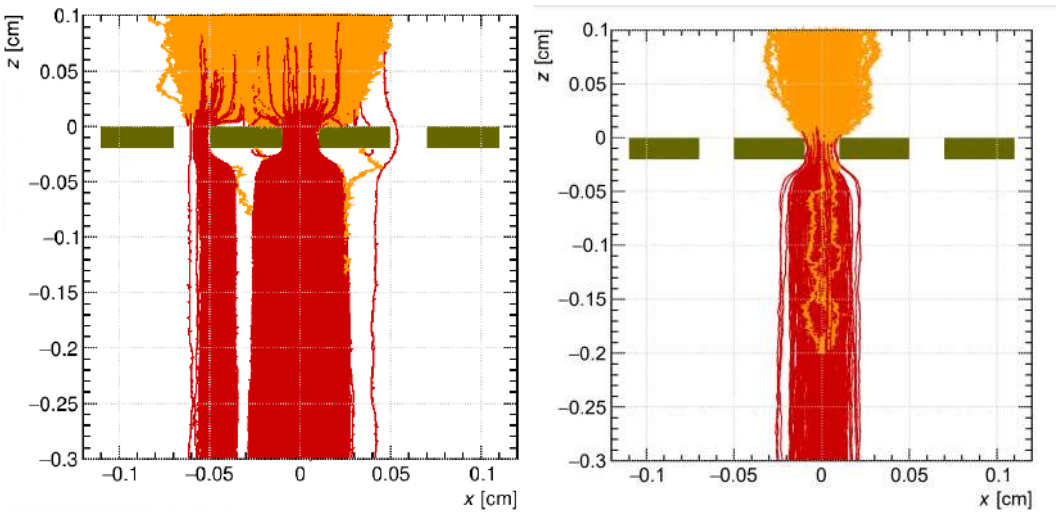
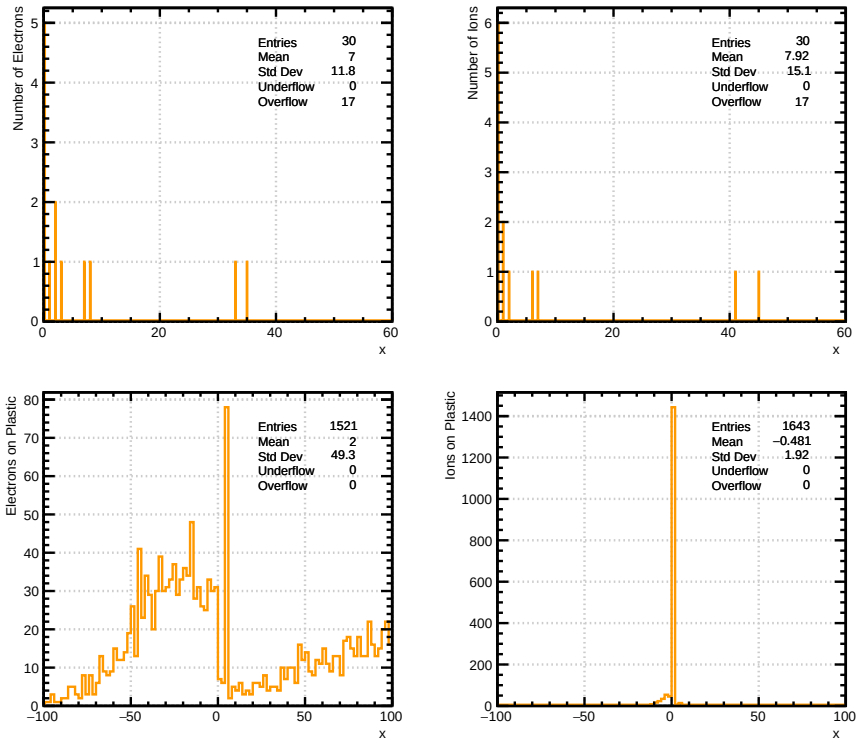
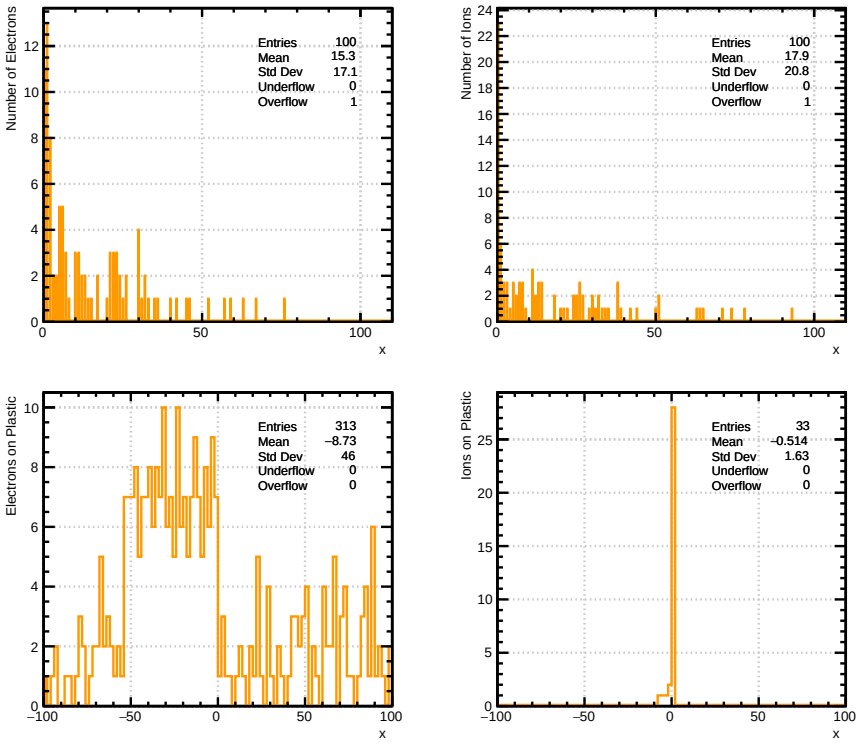


Figura 6.2: Proiezione delle linee di drift nel piano x-z per la miscela 90% Ar e 10% CO_2 a sinistra e per la miscela 70% Ar e 30% CO_2 a destra




```
Mean number of electrons: 7
Mean number of ions: 7.92308
Mean number of electrons on plastic: 50.7
Mean number of ions on plastic: 54.7667
Electron endpoints:
    upper metal: 0.171821%
    plastic:     5.33347%
    lower metal: 0.0140262%
    transfer:    0.189354%
    other:       94.2913%
Total Number of multiplied electrons:0
Gain:797.067
```

Figura 6.3: Output per la miscela 90% Ar e 10% CO_2




```
Fraction of ions drifting back: 0
Mean number of electrons: 16.36
Mean number of ions: 19.1
Mean number of electrons on plastic: 2.97
Mean number of ions on plastic: 0.26
Electron endpoints:
    upper metal: 0.248756%
    plastic:     14.7761%
    lower metal: 0.149254%
    transfer:    2.63682%
    other:       82.1891%
Total Number of multiplied electrons:0
Gain:15.21
```

Figura 6.4: Output per la miscela 70% Ar e 30% CO_2

Capitolo 7

INTERAZIONE DEI RAGGI X

File sorgente: RaggiX

Viene utilizzato lo stesso rivelatore GEM (6) per simulare gli elettroni primari creati dall'interazione di un raggio X con il gas. Il codice in $Garfield$ utilizzato è molto simile a quello visto prima con qualche differenza. In questo esercizio non si utilizza il fattore di Pening in quanto si è interessati alla simulazione a valanga e quindi al successivo trasporto degli elettroni. Viene creato un volume ($Solidbox$) che parte dal catodo fino all'inizio del foglio GEM con un campo elettrico di 3 kV per centimetro. Questa è la regione in cui si inserisce un sensore per tracciare l'assorbimento del fotone (raggio X) da parte del gas, con successiva emissione di elettroni primari. Se il raggio X ha interagito con il gas, attraverso il comando $Track.GetElectron$ ottengo tutte le informazione come energia, posizione, momento dell'elettrone primario che è stato generato.

7.1 Produzione di elettroni primari

Sono stati simulati 100 000 Raggi X inizialmente con energia pari a 6 keV con la miscela 90% Ar e 10% CO_2. Il grafico in figura 7.1 mostra il numero di raggi X che hanno interagito con il gas ($entries = 13128$), in questo caso ho un'efficienza pari al rapporto tra raggi X che hanno interagito con il gas e raggi X iniziali (100 000):

$$efficienza(\%) = \frac{13\,128}{100\,000} \cdot 100 = 13\% \tag{7.1}$$

Ognuno dei raggi X che ha interagito con il gas ha prodotto un certo numero di elettroni. Nel grafico viene riportato un valore medio di elettroni prodotti ($mean = 221$) con una certa deviazione standard ricavata dal fit gaussiano.

La simulazione viene fatta variando l'energia iniziale dei raggi X e variando il gas all'interno del rivelatore. In tabella 7.1 si può notare come per diverse miscele di gas, la presenza del gas quencher (CO_2) riduca l'efficienza del rivelatore.

Di seguito si riportano gli istogrammi in cui la media del fotopicco rappresenta gli elettroni prodotti (asse x).

Gas = $KrCO_2$

ENERGIA RAGGI X	EFFICIENZA
6 keV	13%
10 keV	3.5%
14 keV	1.3%
18 keV	4.8%
22 keV	2.9%

Energia = 6 keV

GAS	EFFICIENZA
$KrCO_2$	13%
Kr	18%
$ArCO_2$	8.6%
Xe	64%
$XeCO_2$	51%

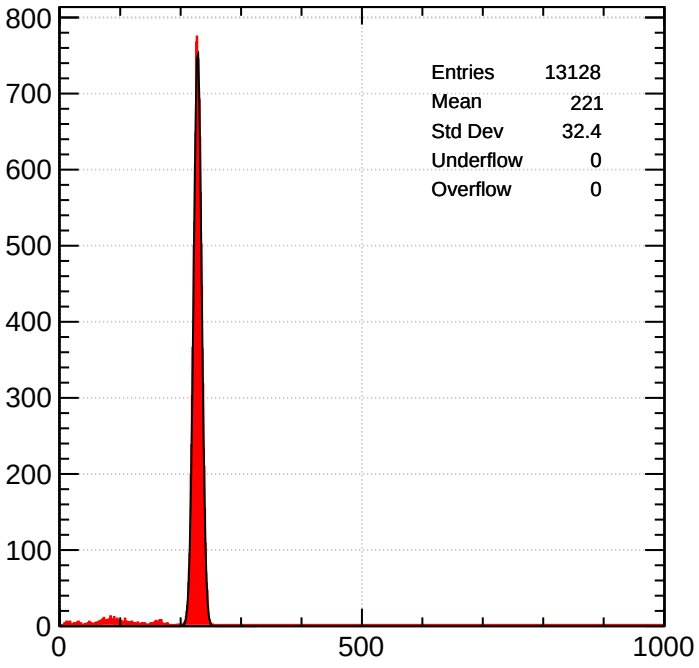


Figura 7.1: KrCO2 6 keV.

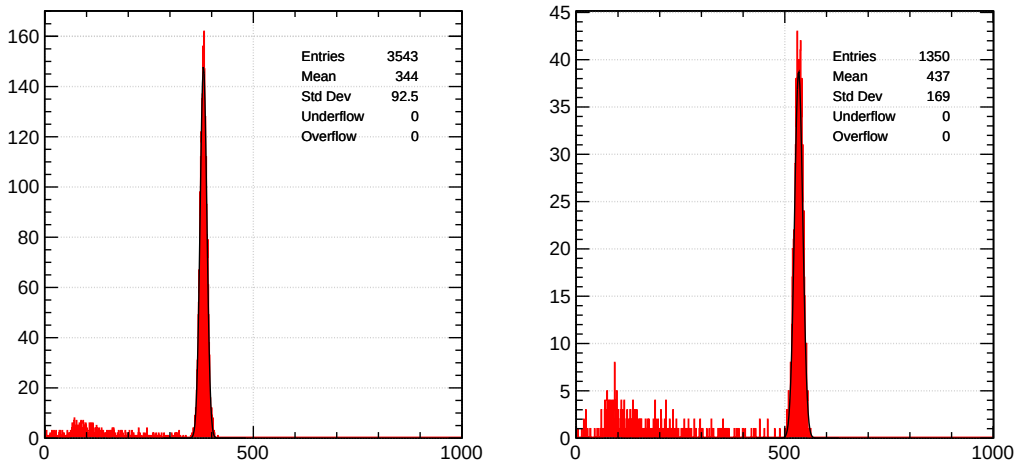


Figura 7.2: KrCO2 10keV e KrCO2 14keV

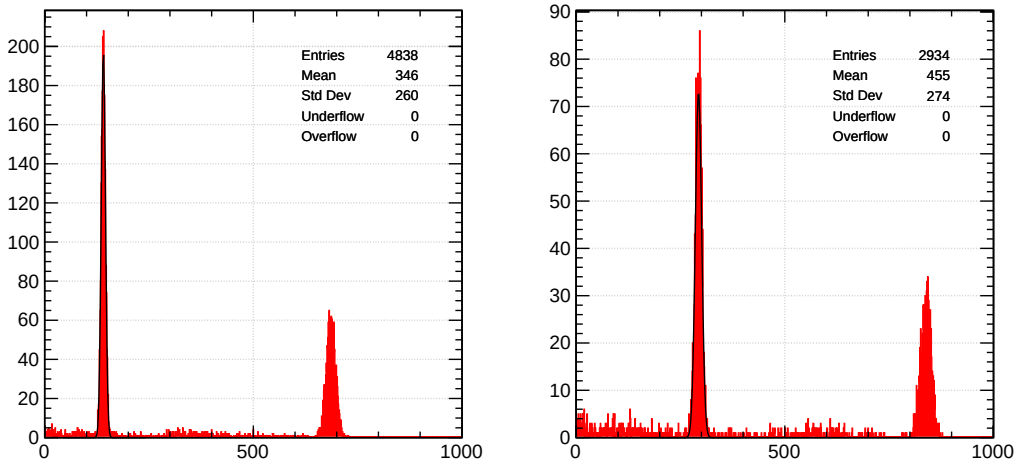


Figura 7.3: KrCO2 18keV e KrCO2 22keV

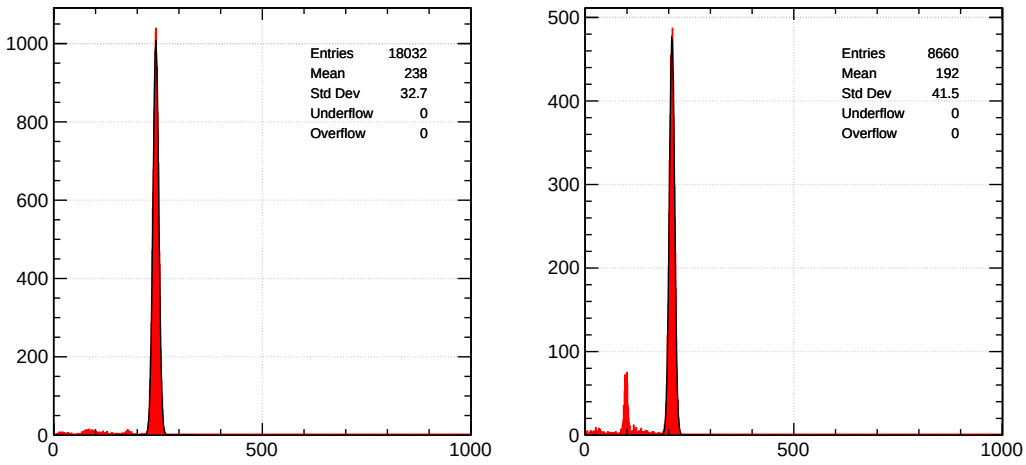


Figura 7.4: Kr 6keV e ArCO2 6keV

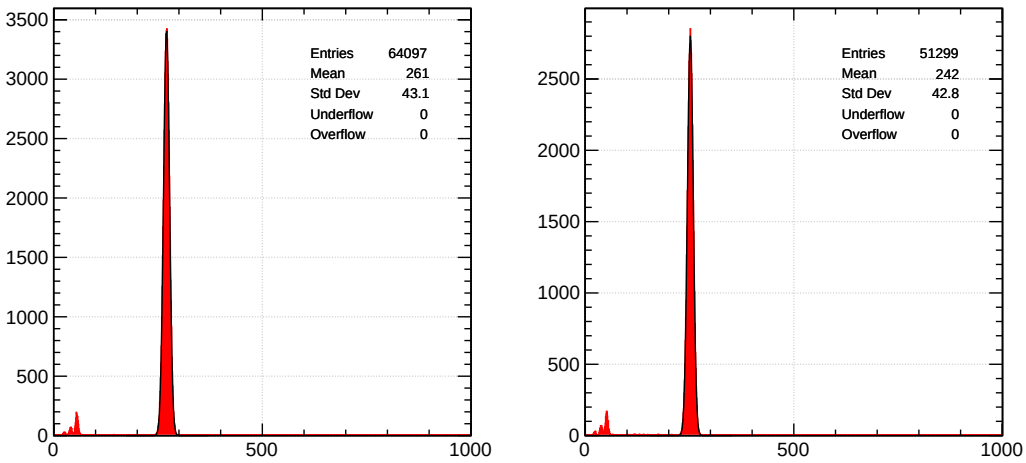


Figura 7.5: Xe 6keV e XeCo2 6keV

In alcuni grafici mostrati si nota un secondo picco con altezza minore rispetto a quello principale: si tratta dell'$ArgonEscapePeak$. Può accadere che il fotone interagisca con l'atomo di Argon tramite effetto fotoelettrico di conseguenza l'elettrone emesso proviene da una shell interna dell'atomo di Argon. Il fotoelettrone emesso ha energia pari a $EnergiaRaggioX - BindingEnergy$ (circa 3 keV). Quando l'atomo si riassesta (cioè si diseccita tornando allo stato ground) può emettere un fotone (fluorescenza X) che esce dal rivelatore. Questo fotone, che causa il picco più spostato a sinistra in quanto genera un minor numero di elettroni, può uscire dal rivelatore, in questo caso sarà perso, o rinteragire.
Questo effetto dipende molto dal tipo di gas in quanto è legato alle binding energy. Ovviamente l'*escape peak* si nota con gas pesanti perchè altrimenti avrei una binding energy troppo bassa e di conseguenza il fotone emesso verrà subito riassorbito dal gas. Si nota come alcune volte l'altezza dell'*escape peak* risulti maggiore rispetto a quella del fotopicco, questo può portare a un problema di calibrazione della GEM.

7.2 Curve di assorbimento

L'ultimo esercizio consiste nel mostrare la curva di assorbimento dei fotoni per le miscele $KrCO_2$ e $ArCO_2$. In particolare si studia come varia il numero di elettroni primari prodotti al variare dell'energia iniziale dei raggi X.

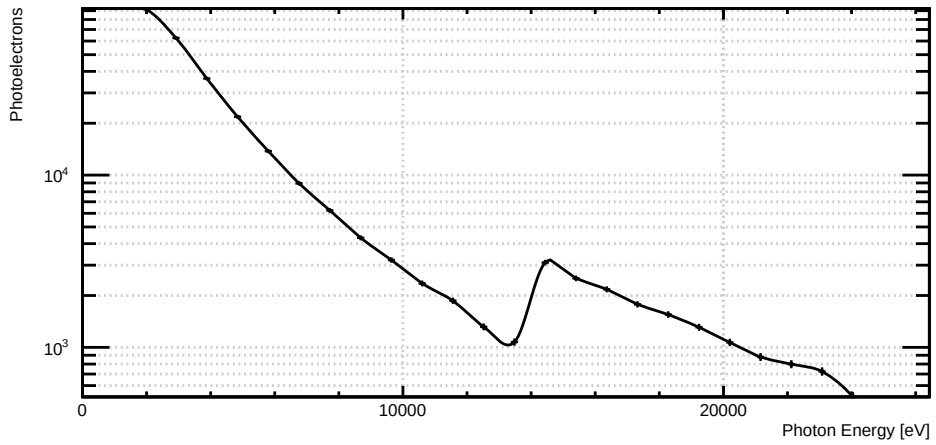


Figura 7.6: KrCO2

Si vede come all'aumentare dell'energia dei raggi X il numero di elettroni prodotti diminuisce. Questo è dovuto al fatto che si ha energia sufficiente per produrre l'escape peak.

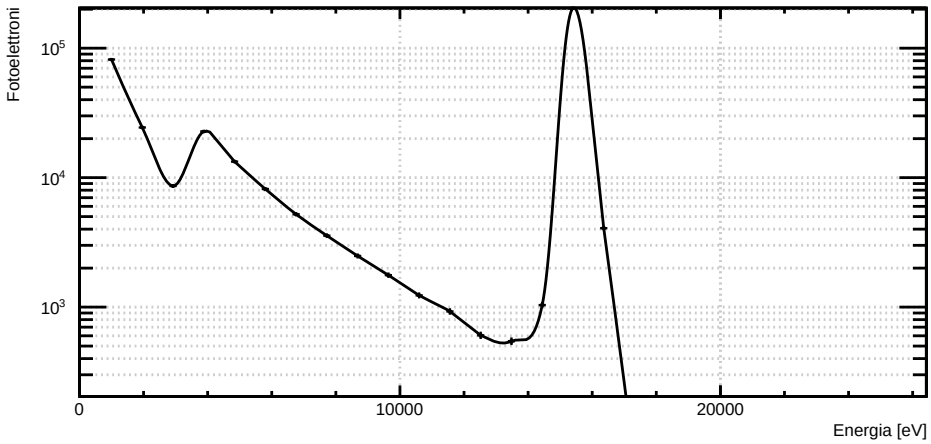


Figura 7.7: ArCO2

Elenco delle figure