Daniel Vázquez Lago

Simulación en Física de Partículas

Root, Geant4, CMake y Garfield++

Índice general

Capítulo 1

Instalación, configuración y ejecucción de archivos

1.1. Instalación

1.1.1. Linux

1.2. Ejecucción de ejemplo

1.3. Configuración de VSCode

Capítulo 2

Garfield++

2.1. Introducción

Garfield++ es una herramienta basada en la programación orientada a objetos que permite simulaciones detalladas de detectores de partículas basadas en ionización de gases o semiconductores. Para calcular los campos eléctricos, se ofrecen ls siguientes técnicas:

- Soluciones para hilos/cables finos para detectores basados en hilos y planos.
- Interfaces[1] para elementos finitos, que pueden calcular campos aproximados en configuraciones 2 y 3 dimensionales con materiales dieléctricos y conductores.

Para calcular las propiedades de tranpsorte de las partículas en mezclas de gases, usamos la interfaz de Magboltz. La ionización producida por partículas cargadas relativistas se estudia a través del programa Heed. Para la simulación de ionización producida por iones a baja energía, los resultados puede ser estudiado por el paquete SRIM. El programa Degrade simula la ionizacion producida por electrones.

Ejemplo 2.1 – Tubo de deriva

En est ejemplo vamos a considerar un tubo de deriva con un diámetro de 15 mm y un diámetro del hilo (cable) de 50 μm, similar a los tubos de derivas de muones del ATLAS (también con un diámetro pequeño) llamados sMDTs. Primero importamos los módulos:

```
#include "Garfield/MediumMagboltz.hh"
#include "Garfield/ViewMedium.hh"
```

Lo primero que tenemos que hacer es preparar la **tabla de gases**, es decir, la tabla que contiene los parámetros de transporte (velocidad de deriva, coeficientes de difusión, coeficiente de Townsend, coeficiente de captura) como funciones del campo eléctrico **E** (y en general, del campo magnético **B** y el ángulo entre **E** y **B**) para un gas a una temperatura y

[1] Conexión funcional entre dos sistemas, programas, dispositivos o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles, permitiendo el intercambio de información.

presión determinadas. En este ejemplo usaremos un gaz mezcla, a 3 atm y temperatura ambiente:

```
MediumMagboltz gas("ar", 93., "co2", 7);
// Set temperature [K] and preasure [Torr]
gas.SetPressure(3*760.);
gas.SetTemperature(293.15);
```

También debemos especificar el número de puntos de la malla campo eléctrico que vamos a usar en la tabla y el rango que va a ser cubierto. Usamos 20 puntos entre 100 V/cm a 100 kV/cm con un espaciado logaritmico:

```
gas.SetFieldGrid(100.,100.e3,20,true);
```

Ahora ejecutamos Magboltz para generar una tabla del gas para esta malla de campo eléctrico. Como un parámetro de entrada tenemos que especificar *el número de colisiones* (en múltiplos de 10^7) sobre el electrón cuya traza dibuja Magboltz:

```
const int ncolll=10;
```

Aunque tarde un rato, una vez este acabado podemos guardar los parámetros:

```
gas.WriteGasFile("ar_93_co2_7.gas");
```

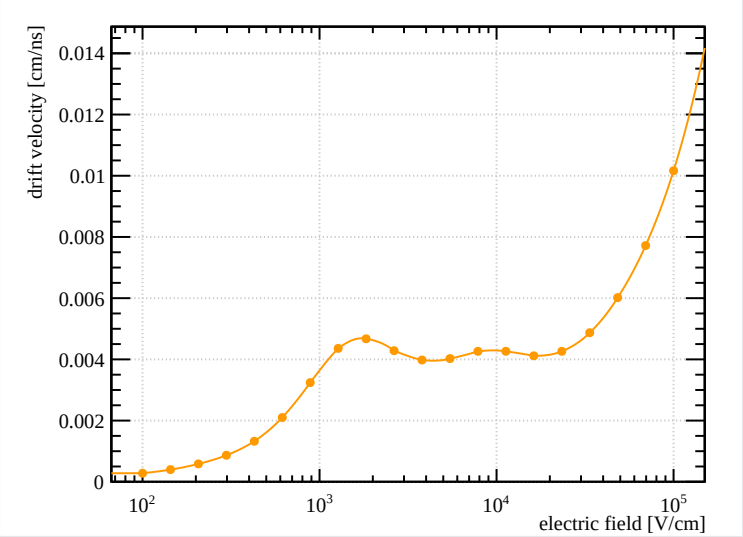
para luego poder importarlos cuando queramos, y no tener la necesidad de correr el programa cada vez que los queramos:

```
gas.LoadGasFile("ar_93_co2_7.gas");
```

Una buena idea podría ser, para asegurarse que el cálculo es correcto, graficar la velocidad de deriva en función del campo eléctrico:

```
ViewMedium view;
view.SetMedium(&gas);

// Dibujamos:
TCanvas* c1 = new TCanvas("c1", "Propiedades del gas", 800, 600);
view.PlotElectronVelocity();
c1->SaveAs("drift_velocity.pdf");
```



Ahora podemos calcular el **campo eléctrico** que se hace a través de `ComponentAnalyticField`, que básicamente maneja la disposición de cables, planos y tubos:

```
ComponentAnalyticField cmp;
```

Tenemos que introducir el medio que hemos definido en la región activa:

```
cmp.SetMedium(&gas);
```

Lo siguiente que tenemos que hacer es añadir los elementos que definen el campo eléctrico, i.e. el cable (denominado “s”) y el tubo:

```
// Radio del cable [cm]
const double rWire = 25.e-4;

// Radio del tubo externo [cm]
const double rTube = 0.71;

// Voltajes
const double vWire = 2730.;
const double vTube =    0.;

// Añadimos el cable en el centro de la disposición
cmp.Addwire(0,0,2 * rWire, vWire, "s");

// Añadimos el tubo
cmp.AddTube(rTube, vTube, 0);
```

Finalmente, creamos un `Sensor`, que es un objeto que actúa como interfaz en la clase transporte discutido más abajo:

```
// Calculamos el campo eléctrico usando el objeto Componente cmp.
Sensor sensor(&cmp);
// Hacemos una petición para que calcule la señal del electrodo llamado s
//     usando el campo dado por el objeto Componente cmp.
sensor.AddElectrode(&cmp,"s");
```

Ahora necesitadmos definir el intervalo temporal en el que la señal es guardada y la granularidad (ancho del bin). Podemos usar 1000 bins con un ancho de 0.5 ns

```
const double tstep = 0.5;
const double tmin = -0.5 * tstep;
const unsigned int nbins = 1000;
sensor.SetTimeWindow(tmin,tstep,nbins);
```

Ahora lo que nos queda es **simular la ionización producida** por la partícula en el tubo de carga usando Heed, de un muón, con por ejemplo, 170 GeV. *Track* significa camino o trayectoria en ingles.

```
TrackHeed track(&sensor);
track.SetParticle("muon");
track.SetEnergy(170.e9);
```

Las curvas (lineas) de deriva de los electrones se crean usando el método de integración Runge-Kutta:fehlberg (RKF), implementada en la clase `DriftLineRKF`. Este método usa las tablas previamente computadas de parámetros de transporte para calcular las líneas de deriva y su multiplicación:

```
DriftLineRKF drift(&sensor);
```

Consideremos ahora que la pista pasa a una distancia de 3 mm del centro del hilo. Después de simular el paso de la partícula cargada, nos tenemos que fijar en los “clusters” (agrupaciones de partículas cargadas producidas por la partícula primaria) y su movimiento en el dispositivo. Así pues, calculamos las líneas de deriva para cada electrón producido en el cluster:

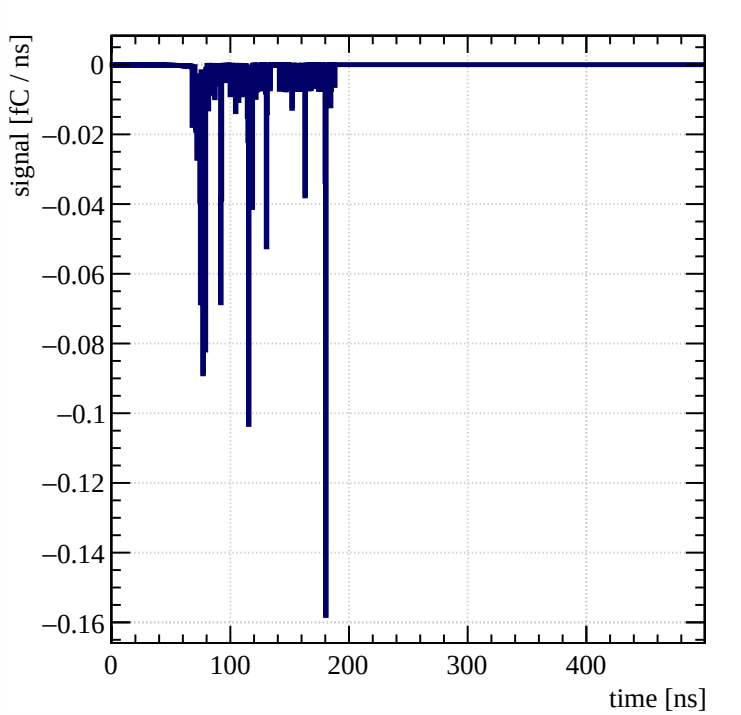
```
const double rTrack = 0.3;
const double x0=rTrack;
const double y0 = -sqrt(rTube * rTube - rTrack * rTrack)
track.NewTrack(x0,y0,0,0,0,1,0);
// Hacemos un bucle sobre los clusters producidos por el camino (track)
for (const auto& cluster: track.GetClusters()) {
    // Bucle alrededor de los electrones del cluster
    for (const auto& electron: cluster.electrons) {
        drift.DriftElectron(electron.x,electron.y,electron.z,electron.t)
    }
}
```

Como una comprobación de la simulación podemos visualizar las líneas de deriva. Antes de simular el recorrido de la partícula cargada y las curvas de deriva de los electrones, tenemso que decirle a `TrackHeed` y `DriftLineRKF` que pase las coordenadas de los clusters y los puntos de las líenas de deriva al objeto `ViewDrift`, que se encarga de graficarlas:

```
// Creamos un canvas
cD = new TCanvas ("cD"," ", 600, 600);
ViewDrift driftView;
drift.EnablePlotting(&driftView);
track.EnablePlotting(&driftView);
cellView.SetCanvas(cD);
cellView.Plot2d();
constexpr bool twod=true;
constexpr bool drawaxis = false;
driftView.Plot(twod,drawaxis);
cD->SaveAs("drift_view.pdf");
delete cD;
```

Podemos graficar la señal inducida en el hilo/cable por la deriva de los electrones simulados:

```
TCanvas* cS = new TCanvas("cS","",600,600);
sensor.PlotSignal("s",cS);
```



Si quisieramos considerar la contribución de los iones producidos en la avalancha necesitamos importar tablas de mobilidades de iones:

```
gas.LoadIonMobility("LoadIonMobility_Ar+_Ar.txt")
```

que, por defecto, `DriftLineRKF` las incluirá en la simulación.

2.2. Media

2.3. Components

2.4. Tracks

El propósito de las clases `Track`[2] son simular los procesos de ionización producidos por partículas cargadas atravesando el detector. Básicamente lo primero que hay que hacer es definir la partícula:

```
void SetParticle(std::string particle);
track.SetParticle(std::string particle);
```

como por ejemplo puede ser `"muon"`. La cinemática de la partícula cargada puede definirse de diferntes formas:

- Con la energía total en eV.
- Con la energía cinética en eV.
- Con el momento en eV/c.
- Con la velocidad β adimensional, el factor de Lorenzt γ o el producto $\beta\gamma$.

```
// Métodos de establecimiento de variables físicas
void SetEnergy(const double e);
void SetKineticEnergy(const double ekin);
void SetMomentum(const double p);
void SetBeta(const double beta);
void SetGamma(const double gamma);
void SetBetaGamma(const double bg);
```

El track se inicializa mediante:

```
void NewTrack(const double x0, cons double y0, const double z0, const double
    t0, const double dx0, const double dy0, const double dz0)
```

[2] *Track* se puede traducir como camino, trayectoria, pista, estela, senda...

Los marcadores `x0`, `y0` y `z0` marcan la *posición inicial* en cm, `t0` el *instante inicial* y `dx0`, `dy0`, `dz0` el *vector inicial*. El punto inicial del recorrido tiene que estra dentro del medio. Si la dirección del vector es nula, un vector aleatorio isótropo será generado. Dependiendo del tipo de clase `Track` que se use, mas restricciones podrán ser impuestas.

Tras la inicialización, los "cluster" se producen a lo largo del recorrido, que se pueden obtener con:

```
const std::vector<Cluster>& GetClusters();
```

Cuando hablamos de "cluster" nos refereimos a la energía cedida en una única interacción ionizante de la partícula primaria cargadam y los electrones secundarios producidos en este proceso. La implementación concreto de los objetos Cluster dependen de la clase `Track` de la que estemos hablando.

2.4.1. Heed

El programa Heed es una implementación del modelo de la ionización por foto-abosrción (también llamado modelo PAI, *photo-absorption model*, PAI), escrito por I. Smirnov. La interfaz Heed está disponible a través de `TrackHeed`.

Los objetos `Cluster` se obtienen a través de `TrackHeed::GetClusters` contiene la posición y tiempo de la colisión ionizante, la energía transferida y el vector de objetos `Electron` correspondientes a los electrones conductores/libres[3] asociados al cluster.

Transporte de electrones Delta

Heed simula el tiempo de degradación de los electrones δ y la producción de electrones secundarios (electrones conductores/libres) usando un modelo fenomenolóicos. `TrackHeed` recupera los parámetros de entrada necesarios (por ejemplo el el factor de Fano o el valor W) del objeto `Medium`. Si los parámetros son cero, usa los valores por defecto (por ej. $F = 0.19$).

Si los electrones delta son desactivados, el número de electrones devuelto por el `GetCluster` es el número de electrones primarios (electrones producidos por la ionización primaria), i.e. foto-electrones y electrones Auger. Las enerǵias cinéticas y posiciones de los electrones son acesibles vía `GetElectron`. Si el tranpsorte de electrones δ está activado (por defecto está activado), la función `GetElectron` devuelve la localización de los electrones de conducción calculados por el factor interno δ del algoritmo de Heed. Dado que este método no devuelve la energía cinética y dirección de los electrones secundarios, los parámetros de `GetElectron` no son significativos en este caso.

Transporte de fotones

Heed también puede simular la fotoabsorción de rayos-x.

[3] son simplemente los electrones libres en el gas.

Campos magnéticos

Si el `sensor` tiene un campo magnético nulo, `TrackHeed` lo tnedrá en cuenta para calcular la trayectoria de la partícula cargada.

2.4.2. SRIM

SRIM (*Stopping and Range of Ions in Matter*) es un programa que permite simular la perdida de energía iónica por la materia en la materia. Esto produce tablas de frenados energéticos, rangos y parámetros de *stragglings*[4] que pueden ser importados en Garfield a través de la clase `TrackSrim`. La función:

```
bool ReadFile(const std::string& file)
```

devuelve `true` si es leido correectamente. Los archivos SRIM contiene la siguiente información:

- Una lista de energías cinéticas en las que se han calculado pérdidas y rezagos;
- Energía promedio perdida por unidad de longitud vía proceos electromagnéticos, para cada enerǵia.
- Energía promedio perdida por unidad de longitud vía proceos nucleares, para cada enerǵia.
- Proyección de l4!a media recorrida, por energía.
- Straggling longitudinal y transversal para cada energía.

Se pueden visualizar usando las funciones:

```
void PlotEnergyLoss();
void PlotRange();
void PlotStraggling();
```

Además de estas tablas, el archivo también contiene la masa y carga del proyectil y la densidad del medio. Estas propiedades son también emportantes y guardadas por `TrackSrim` cuando leemos el archivo. Al contrario del caso Heed, el tipo de partícula no es especificada por el usuario, aunqeu si tenga que especificar la energía cinética del proyectil.

El `TrackSrim` genera reocrridos individuales que estadísiticamente representan las cantidades promedio calculadas por SRIM. Una vez se pasa la energía, `TrackSrim`, interativamente

- Calcula (interpolando las tablas) la energía electromagnética y nuclear perdida por unidad de distancia para dicha energía.

[4] Recordemos que el straggling es el fenómeno que recoge las fluctuaciones estadísticas asociadas a las pérdidas energéticas.

- Calcula la longitud del paso/intervalo (*step*) en el cual la energía producirá una cantidad de electrones promedio. .
- Actualiza la trayectoria basada en el la dispersión longitudinal y transerval para la energía de la partícula.
- Calcula una pérdida energía electromagnética aleatoria sobre el intervalo y actualiza la energía cinética.

repitiendo el proceso hasta que la partícula ya no tiene maś energía o deja de estar en la geometría (dispositivo). Se pueden elegir varios modelos por los cuales se aleatoriza la péridda de energía en cada paso

```
void SetModel(const imt m)
```

En función de m tendremos un modelo u otro. Los modelos disponibles son:

Modelo	Descripción
0	Sin fluctuaciones
1	Distribución de Landau no truncada
2	Distribución de Vavilov (siempre que los parámetros cinemáticos estén dentro del rango de aplicabilidad de lo contrario, las fluctuaciones se desactivan)
3	Distribución gaussiana
4	Combinación de los modelos de Landau, Vavilov y Gauss, cada uno aplicado en su supuesto dominio de aplicabilidad

Para samplear las pérdidas energéticas, `TrackSrim` necesita la densidad delectrónica del material, que por defecto se recupera del objeto `Medium` (y escalada con la densidad de masa del archivo SRIM). Tambień se puuede usar un número Z efectivo y el número másico A usando

```
{TrackSrim::SetAtomicMassNumbers}.
```

Para calcular el número de electrones generados para una energía depositada, `TrackSrim` necestia la función de trabajo W en eV y el factor de Fano del material, qeu se pedirán al objeto `Medium`, aunque puede ser dado a mano. El objeto `Cluster` devuelto por `TrackSrim::GetClusters` contiene la localización y tiempo inicial del cluster, la energía gastada para producir el cluster, la energía del ion cuando el cluster fue creado y el número de elctrones por cluster.

2.4.3. TRIM

TRIM (*TRansport of Ions in Matter*) es una simulación montecarlo del mismo conjunto de programas que SRIM, que simula la trayectoria individual de un ión en en el medio y su proceso de pérdida de energía (cascadas de retroceso, daños por desplazamiento...). TRIM produce típicamente un número determinado de archivos de salida, entre los cuales EXYZ.txt que contiene la lista de posición y pérdidas de energía electrónicas para cada ion simulado en pasos regularess.

2.4.4. Degrade

La clase `TrackDegrade` simula las ionizaciones por electrones primarios en el gas y la subsecuente degradación en electrones δ (electrón que puede recorrer cierta distancia antes de perder energía y provocar ionización secundaria a lo largo de su camino.), electrones Auger (surgen de la reorganización interna del átomo cuando la energía de una transición electrónica se transfiere a otro electrón que es expulsado) y fotoelectrones (fotoelectrones se originan por la absorción de un fotón, mientras que los electrones Auger), usando una interfaz al programa Degrade, desarrollado por S. Biagi. Degrade tiene muchos puntos en común con Magboltz, en particular la base de datos de secciones eficaces de electrón-átomo/molécula.

Mientras el programa Degrade puede ser usado para la simulación de rayos X, los decaimientos β y doble β, estas características aún no han sido implementadas, al igual que no contiene las interaccioens en presencia de campo eléctrico y magnético.

Los objetos `Cluster` devuletos por `TrackDegrade::GetClusters` contienen la posición y tiempo de las colisiones ionizantes (`x,y,z,t`) y un `vector` de objetos `Electron` correspondientes a electrones termializados asociados al cluster. Además, contiene un vector de electrones delta y Auger.

```
TrackDegrade track;          // Se crea un objeto de tipo TrackDegrade

// Variables iniciales de posición y tiempo:
double x0 = 0., y0 = 0., z0 = 0., t0 = 0.;

// Dirección inicial del movimiento:
double dx0 = 1., dy0 = 0., dz0 = 0.;

// Se genera una nueva trayectoria con las condiciones iniciales anteriores:
track.NewTrack(x0, y0, z0, t0, dx0, dy0, dz0);

// Bucle sobre los "clusters" (agrupaciones de ionización o excitación) a lo
   largo de la trayectoria.
for (const auto& cluster : track.GetClusters()) {

    // Bucle sobre los electrones termalizados dentro del cluster.
    for (const auto& electron : cluster.electrons) {

        // Se obtienen las coordenadas y la energía cinética del electrón.
        double xe = electron.x;
        double ye = electron.y;
        double ze = electron.z;
        double te = electron.t;
        double ee = electron.energy;
    }
}
```

Por defecto, los electrones están *trackeados* hasta que su energía cinética cae hasta 2eV. Esto peude ser modificado. Si la función

```
void StoreExcitations(const bool on=true, const double ethr);
```

se llama antes de `NewTrack`, las excitaciones (con una energía de excitación superior a `ethr`) pro-

ducidas por los electrones ionizantes primarios y secundarios se guardan en el objeto `Cluster`.

2.5. Transporte de Carga