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");
```

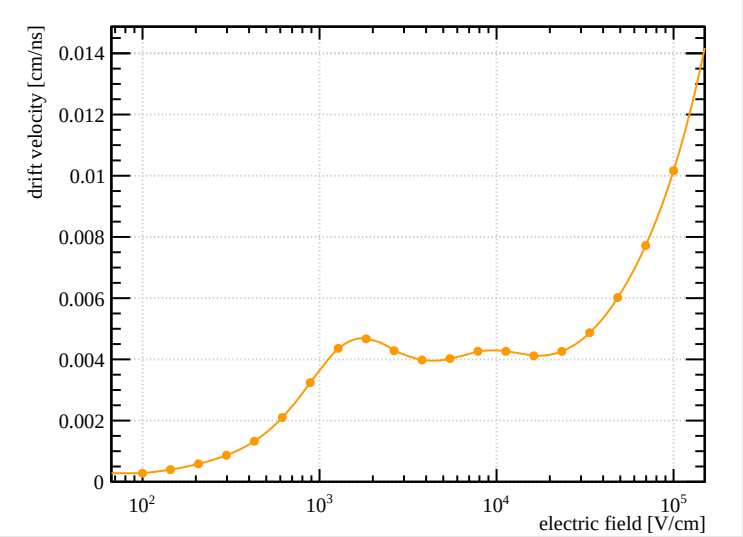
para luego poder importarlos cuando queramos, y no tener la necesidad de correr el programa cada vez que los queramos:

```
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:

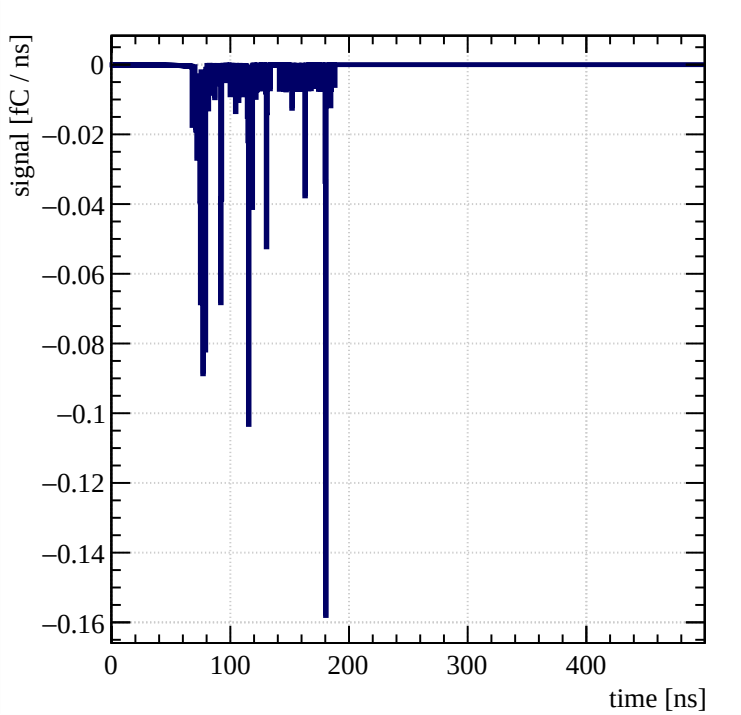
```
const double rTrack = 0.3;
const double x0=rTrack;
const double y0 = -sqrt(rTube * rTube - rTrack * rTrack)
track.NewTrack(x0,y0,0,0,0,1,0);
// Hacemos un bucle sobre los clusters producidos por el camino (track)
for (const auto& cluster: track.GetClusters()) {
    // Bucle alrededor de los electrones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