Daniel Vázquez Lago

Simulación en Física de Partículas

Root, Geant4, Degrad, 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

Geant4

Capítulo 3

Garfield++

3.1. 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.

Un resumen de Garfield, o de un programa de Garfield, sería el siguiente:

1. Primero tenemos que describir el medio (gas, silicio), información almacenada en un objeto clase `Medium`.
2. Luego tenemo que definir el campo eléctrico en toda la región de interés. Existen varias maneras de definir el campo, en general se hace a través de la clase `Component`, en alguna de sus diferentes variantes. Por ejemplo puede ser importada a través de documentos externos, definido para geometrías sencillas por programas propios de Garfield++, o incluso definido por el usuario a mano.
3. También está la clase `Sensor`, que se encarga de definir el detector de las partículas cargadas.
4. Una vez hemos creado el sistema, el detector propiamente dicho, pasamos a la parte de ionización. Lo primero es crear la clase `Track`, que se encarga vía diferentes modelos (Heed,

[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.

SRIM, Degrad) de crear los cluster de partículas productos de la ionización. En función de la partícula habrá que aplicar un modelo u otro (en el caso de iones pesados SRIM o TRIM, para otros Degrad o Heed). En casos extremos debemos enlazar Geant4 y Garfield++.

5. Una vez se han creado los diferentes clusters, podemos simular su movimiento hacia los sensores (en el campo eléctrico) a través de la clase `Drift` para varios modelos (Runge-Kutta-Fehlberg, Montecarlo, Seguimiento Micorscópico).

6. Una vez llega al sensor se puede visualizar la señal.

Ejemplo 3.1 – 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 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");
```

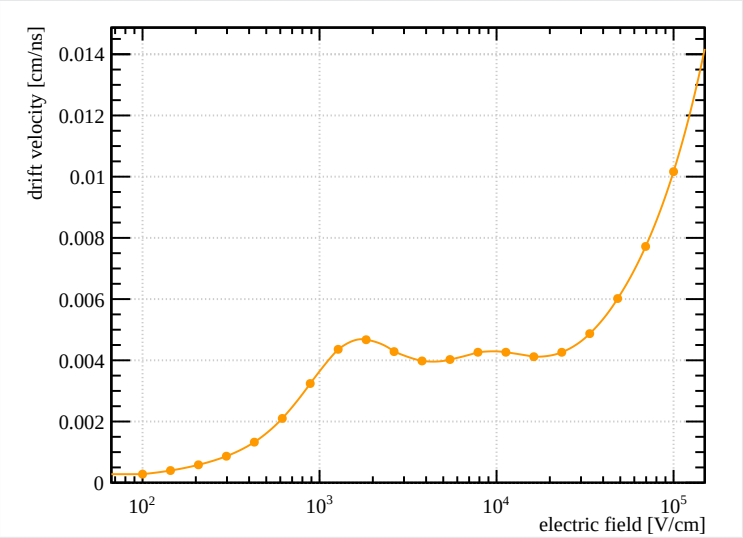
para luego poder importarlos cuando queramos, y no tener la necesidad de correr el programa cada vez que los queramos:

```
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:

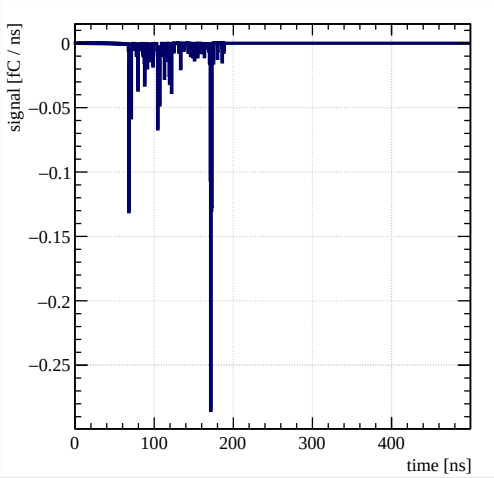
```
const double rTrack = 0.3;
const double x0=rTrack;
const double y0 = -sqrt(rTube * rTube - rTrack * rTrack)
track.NewTrack(x0,y0,0,0,0,1,0);
// Hacemos un bucle sobre los clusters producidos por el camino (track)
for (const auto& cluster: track.GetClusters()) {
    // Bucle alrededor de los electrones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

3.2. Medio

La clase `Medium` incorpora las propiedades del medio.

- El **coeficiente Townsend** α nos habla de cuantos pares de iones se generan por unidad de longitud para una partícula cargada en movimiento, tal que:

$$I = I_0 e^{\alpha d} \tag{3.1}$$

  siendo I la intensidad de corriente que circula por el dispositivo, I_0 la intensidad de corriente generada por efecto fotoeléctrico en el cátodo y d la distancia entre cátodo y ánodo.

- El **coeficiente de captura** o *attachment coefficient* nos habla de cuantos electrones/iones son capturados por el medio.

3.2.1. Parámetros de transporte

La clase `Medium` provee al usuario de las siguientes funciones para el cálculo de los parámetros de transporte de los electrones en función del campo eléctrico y magnético.

Visualización

Los parámtros de transporte se pueden graficar en función del campo eléctrico a través de:

```
void PlotVelocity(const std::string& carriers, TPad* pad);
void PlotDiffusion(const std::string& carriers, TPad* pad);
void PlotTownsend(const std::string& carriers, TPad* pad);
void PlotAttachment(const std::string& carriers, TPad* pad);
void PlotAlphaEta(const std::string& carriers, TPad* pad);
```

donde la opción `carriers` indica la partícula cargada (ej: `''e''` indicaría electrones) para el cual vamos a garficar los parámetros. Por ejenmplo para visualizar las líneas de velocidad de deriva electrónica y de huecos en silicio así como el coeficiente Townsend y de captura:

```
MediumSilicon si;
TCanvas* c1 = new TCanvas("c1", "", 600, 600);
si.PlotVelocity("eh", c1);
TCanvas* c2 = new TCanvas("c2", "", 600, 600);
si.PlotAlphaEta("eh", c2);
```

3.2.2. Gases

Hay dos clases principales implementadas que se pueden usar para la descripción de medios gaseosos. La primera sería `MediumGas` y su hija `MediumMagboltz`. La primera solo interpola e importa tablas de gases, mientras que la segunda tiene una interfaz propieda con el programa Magboltz y puede ser usada para calcualr parámetros de transportes. Además, `MediumMagboltz` permite acceder a las secciones eficaces electrón-molécula, que puede ser usado para el seguimiiento microscópico de los electrones.

```
bool SetComposition(const std::string& gas1, const double f1 = 1.,
const std::string& gas2 = "", const double f2 = 0.,
const std::string& gas3 = "", const double f3 = 0.,
const std::string& gas4 = "", const double f4 = 0.,
const std::string& gas5 = "", const double f5 = 0.,
const std::string& gas6 = "", const double f6 = 0.);
```

donde podemos identificar gas1, gas2... con fracción f1,f2... Luego además podemos darle otros parámetros, como presión y temperatura:

```
void SetPressure(const double p);
void SetTemperature(const double t);
```

como por ejemplo en:

```
MediumMagboltz gas;
// Composición:
gas.SetComposition("ar", 80., "ch4", 20.);
gas.SetTemperature(293.15);
gas.SetPressure(760.)
```

Magboltz

Magboltz, escrito por Steve Biagi, es un programa qeu calcula las propiedades de transporte de los electrones en mezclas de gases usando una simulación semi-clásica Monte-Carlo, incluyendo las seccciones eficaces electrón-átomo/molécula para una alta variedad de gases.

La función

```
void GenerateGasTable(const int numCollisions, const bool verbose);
```

crearía una tabla de gases para cada valor de **E**, **B** y θ. Además, también devuelve las tasas de excitación e ionización del gas.

3.2.3. Semiconductores

3.3. Components

El cálculo de los campos eléctricos se hace a través de la clase abstracta `Component`. Sus funciones clave, usadas internamente ne la simulación del transporte de carga son:

```
void ElectricField(const double x, const double y, const double z, double& ex
    , double& ey, double& ez, Medium*& m, int& status);
void ElectricField(const double x, const double y, const double z, double& ex
    , double& ey, double& ez, double& v);
Medium* GetMedium(const double& x, const double& y, const double& z);
```

siendo `x,y,z` las posiciones donde el campo eléctrico debe ser definido, `ex,ey,ez,v` el campo eléctrico y potencial en una posición dada y `m` un puntero hacia el medio en la posición dada. El `status` nos indica donde el punto está localizado (en el medio de deriva, dentro de un cable/hilo...). Pudiendo ser devuelto tanto el potencial eléctrico como el campo eléctrico:

```
double ElectricPotential(const double x, const double y, const double z);
std::array<double, 3> ElectricField(const double x, const double y, const
    double z);
```

3.3.1. Definición de la geometría

Como hemos mencionado arriba, el propósito de la clase `Component` es proveer, para una localización dada, un campo eléctrico y magnético y un puntero al medio (objeto `Medium`). Para dicho propósito es necesario especificar la geometría del dispositivo. En el caso de mapas de campo precomputados (por ejemplo, tablas con valores del campo eléctrico o magnético en cada punto del espacio), la geometría ya está incluida dentro del propio mapa que se generó con un programa de resolución

de campos (field solver, como COMSOL, ANSYS o Garfield++), solo habría que asociar la geometría con el objeto `Medium`.

Para campos analíticos, en general la geometría viene dada por una celda. Para otras componentes la geometrías (como por ejemplo una parametrización de campo personal) es definida de manera separada. Estructuras simples pueden ser definida por `GeometrySimple`, qeu tiene un pequeño repertorio de formas (sólidos), como cajas, tubos, hueecos, anillos, cables, esferas...

Por ejemplo, un tubo lleno de gas de un diámetro de 1cm y un tamaño de 20 cm a lo largo del eje z se definiría:

```
// Create the medium.
MediumMagboltz gas;
// Create the geometry.
GeometrySimple geo;
// Dimensions of the tube
double rMax = 0.5, halfLength = 10.;
SolidTube tube(0., 0., 0., rMax, halfLength);
// Add the solid to the geometry, together with the gas inside.
geo.AddSolid(&tube, &gas);
```

Para estructuras más complejas, la calse `GeometryRoot` puede ser usada. Esta es una interfaz con la clase de ROOT `TGeo`.

3.4. 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);
```

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

```
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)
```

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.

3.4.1. 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`.

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

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.

Campos magnéticos

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

3.4.2. 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 la media recorrida, por energía.
- Straggling longitudinal y transversal para cada energía.

Se pueden visualizar usando las funciones:

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

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

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.
- 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.

3.4.3. 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.

3.4.4. 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 en Garfield++, al igual que no contiene las interacccioens 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`) producidas por los electrones ionizantes primarios y secundarios se guardan en el objeto `Cluster`.

3.5. Transporte de Carga

Desde un punto de vista fenomenológico, la deriva de los portadores de carga bajo un campo eléctrico y magnético está descrito por una ecuación de movimiento:

$$\dot{\mathbf{r}} = \mathbf{v}_d(\mathbf{E}(\mathbf{r}), \mathbf{B}(\mathbf{r})) \tag{3.2}$$

donde $\mathbf{v}_d$ es la velocidad de deriva. La solución de está ecuación viene dada por dos métodos diferentes:

- El método de integración Runge-Kutta-Fehlberg (`DriftLineRKF`).
- El método de integración por Monte Carlo (`AvalancheMC`).

Para simulaciones precisas en estrucutras pequeñas (con dimensiones similares al erecorrido libre medio del electrón) así como calculos detallados de proceos de ionización y excitación, el transporte de electrones a nivel microscópicos (i.e. basados en ecuaciones de movimiento de segundo orden) es el método que debería ser elegido, el cual se hace a través del método `AvalancheMicroscopic`.

3.5.1. Runge-Kutta-Fehlberg Integration

Una forma de garantizar la precisión en la solución de un problema de valor inicial (P.V.I.) es resolver el problema dos veces utilizando tamaños de paso h y $h/2$, y comparar las respuestas en los puntos de malla correspondientes al tamaño de paso mayor. Sin embargo, esto requiere una cantidad significativa de cálculo para el tamaño de paso más pequeño y debe repetirse si se determina que la concordancia no es suficientemente buena.

El método de Runge-Kutta-Fehlberg (denotado RKF45) es una forma de intentar resolver este problema. Tiene un procedimiento para determinar si se está utilizando el tamaño de paso adecuado h. En cada paso, se hacen y comparan dos aproximaciones diferentes para la solución. Si las dos respuestas están en estrecho acuerdo, la aproximación se acepta. Si las dos respuestas no coinciden dentro de una precisión especificada, el tamaño de paso se reduce. Si las respuestas coinciden en más cifras significativas de las requeridas, el tamaño de paso se incrementa [1].

Este método se implementa a través de la clase `DriftLineRKF`, y calcula la curva de deriva a través del siguiente proceso iterativo:

1. Dado un punto inicial $\mathbf{x}_0$, la velocidad en el pnto inicial y un Δt, calculamos dos estimaciones del salto al siguiente punto de la curva de deriva:

   - $\Delta v_{\mathrm{I}} = \sum_{k=0}^{2} C_{\mathrm{I},k}\mathbf{v}_d(\mathbf{x}_k)$ que es precisa a segundo orden.
   - $\Delta v_{\mathrm{II}} = \sum_{k=0}^{3} C_{\mathrm{II},k}\mathbf{v}_d(\mathbf{x}_k)$ que es precisa a tercer orden.

   Estas dos estimaciones están basasdas en la velocidad de deriva a la velocidad inicial en el punto inicital, y la velocidad a 3 nuevas localizaciones:

   $$\mathbf{x}_k = \mathbf{x}_0 + \Delta t \sum_{i=0}^{k-1} \beta_{k,i}\mathbf{v}_d(\mathbf{x}_i) \tag{3.3}$$

   con diferentes valores para $C_{\mathrm{I}}, C_{\mathrm{II}}, \beta_{k,i}$.

2. Se evalua el paso temporal Δt comparando la diferencia entre el segundo y tercer orden para una precisión determinada:

   $$\Delta t' = \sqrt{\frac{\varepsilon \Delta t}{|\Delta v_{\mathrm{I}} - \Delta v_{\mathrm{II}}|}} \tag{3.4}$$

3. Se vuelve a repetir la iteracción si:

   - El intervalo temporal se reduce más de un factor 5.
   - El intervalo temporal excede el máximo preestablecido (si está impuesto).

4. La posición se actualiza para la estimación de segundo orden.

5. La velocidad se actualiza para el punto final de la interacción, que es aquella en las que los tres factoes velocidade fueron calculados.

3.5.2. Integración Montecarlo

En la clase `AvalancheMC` la ecuación del movimiento está integrada de manera estocástica:

- Un paso/intervalo de longitud $\Delta s = v_d \Delta t$ en la dirección de la velocidad de deriva $\mathbf{v}_d$ para el campo eléctrico y magnético local es calculado (estado Δt o Δs especificados por el usuario).
- Un paso aleatorio es generado por una distribución gaussiana no correlacionada con una desviación estándar $\sigma_L = D_L\sqrt{\Delta s}$ en la componente paralela as la velocidad de deriva y una desviación estándar $\sigma_T = D_T\sqrt{\Delta s}$ en las dos direcciones transversales.
- Los dos pasos se añaden vectorialmente y la nueva posición de la partícula es actualizada.

3.5.3. Seguimiento microscópico (*microscopic tracking*)

Si el método elegido es el microscópico, impelementada hasta la fecha solo para electrones, la partícula es seguida colisión a colisión. Como valores de entrada, se requieren tablas de colisiones $\tau_i^{-1}(\epsilon)$ para cada proceso de dispersión i y como función de la energía del electrón. Para los gases, estos datos vienen dados por la clase `MediumMagboltz`. Entre las colisiones, se calcula el recorrido del

electrón acorde a la trayectoria de un electrón libre en un campo eléctrico y/o magnético (clásicamente). La duracción del tiempo de vuelo Δt está controlado por $\tau^{-1} = \sum_i \tau_i^{-1}$. El muestreo del tiempo de vuelo Δt se jhace a traŭes del método "null-collision", que básicamente tiene en cuenta el cambio de energía del electrón en la iteracción. Después de la iteracción, la energía, dirección, y posición del electrón es actualizado y el proceso de dispersión (*scattering*) es muestreado basado en las tasas de colisión a la nueva energía ϵ'. La energía y dirección del electrón dependerán del tipo de interacción.

En Garfield++ el seguimiento microscópico se implementa con `AvalancheMicroscopic`. Así:

```
void AvalancheElectron(const double x, const double y, const double z,
    const double t, const double e,
    const double dx = 0., const double dy = 0., const double dz = 0.);
```

siendo `x,y,z,t` las posiciones y tiempos iniciales, `e` la energía inicial en eV, y `dx,dy,dz` la dirección inicial. En el caso de no darse una dirección inicial está será aleatoria.

3.5.4. Visualización de las líneas de deriva

Para graficar las líneas de deriva y las trazas de las partículas usamos la clase `ViewDrift`. Después de relacionar `ViewDrift` con la clase que sea

```
void AvalancheMicroscopic::EnablePlotting(ViewDrift* view, const size_t nColl
    = 100);
void AvalancheMC::EnablePlotting(ViewDrift* view);
void DriftLineRKF::EnablePlotting(ViewDrift* view);
void Track::EnablePlotting(ViewDrift* view);
```

podremos dibuajr las trayectorias a travñes de:

```
void  ViewDrift::Plot();
```

En el caso de `AvalancheMicroscopic` el segundo argumento `EnablePlotting(nColl)` selecciona el número de colisiones que quieren saltarse entre puntos sucesivos de la gráfica (por derento se eligen cada 100 colisiones).

Capítulo 4

Nexo entre Geant4, Garfield++ y Degrad

Bibliografía

[1] John H. Mathews y Kurtis K. Fink. *Numerical Methods Using MATLAB*. 4th. Upper Saddle River, New Jersey, USA: Prentice-Hall Inc., 2004. ISBN: 0-13-065248-2. URL: http://vig.prenhall.com/.