KSource: Fuentes de partículas por *Kernel Density Estimation*

Osiris Inti Abbate

Instituto Balseiro, UNCuyo - CNEA

7 de octubre de 2021

Resumen

KSource es una herramiente que asiste cálculos de blindajes por Monte Carlo, implementado una técnica de reducción de varianza. Procesa listas de partículas correspondientes a fuentes superficiales capturadas en posiciones intermedias de una simulación, estima su distribución por el método *Kernel Density Estimation*, y produce nuevas partículas respetando la densidad estimada. Esto permite producir más partículas que las presentes en la lista original, sin repeticiones, favoreciendo la propagación de la radiación hacia la región de interés. Además, posibilita el acople de distintos códigos de cálculo.

1. Introducción

El método Monte Carlo es una técnica poderosa y ampliamente utilizada en el transporte de radiación. Su principal ventaja radica en la ausencia de aproximaciones sobre la distribución angular de los flujos de partículas, lo cual asegura una representación fiel incluso en problemas relativamente difíciles, como aquellos con absorbentes fuertes o propagación en vacío.

Un problema de transporte de radiación por Monte Carlo queda esencialmente definido por tres elementos principales:

- Geometría: representación del sistema físico a modelar. Incluye los materiales involucrados y las condiciones de contorno.
- Fuente: región donde nacen las partículas, con una dada distribución energética, espacial y angular.
- Detector (*tally*): resultado a obtener de la simulación, usualmente asociado al nivel de flujo o corriente en una región de interés. El cómputo del valor a reportar se realiza en base a las partículas que llegan al detector.

La principal dificultad del método Monte Carlo es el relativamente alto costo computacional requerido para obtener resultados confiables. La confiabilidad de un resultado se mide por su varianza estadística, la cual a su vez se relaciona con la cantidad de partículas utilizadas para calcularlo. Para resolver dicha problemática existen diferentes estrategias conocidas como *técnicas de reducción de varianza*, las cuales buscan favorecer la propagación de la radiación hacia la región de interés, respetando la física del problema.

La herramienta KSource permite la implementación de una técnica de reducción de varianza, la cual se denominará método KSource. Los problemas de interés son entonces situaciones en las cuales no es posible alcanzar estadística suficiente en el detector en una única corrida, pues el costo computacional sería demasiado alto. Desde luego, siempre es posible determinar una superficie más cercana a la fuente en la cual sí es posible alcanzar buena estadística. Dicha situación se esquematiza en la Figura 1, y es en la que se basa la aplicación del método KSource.

Continuando con el ejemplo de la Figura 1, el método KSource consiste en registrar las partículas que atraviesan la superficie S1, obteniendo una lista de partículas (energía, posición, dirección) con estadística suficiente. Luego se utiliza dicha lista, también llamada lista de *tracks*, para estimar la distribución de corriente en S1, y se utiliza dicha distribución estimada como fuente en una nueva simulación. La cantidad de partículas producidas en esta segunda etapa puede superar el tamaño de la lista original, mejorando la estadística en la región del detector y reduciendo la varianza del resultado. Todo este proceso se muestra en la Figura 2.

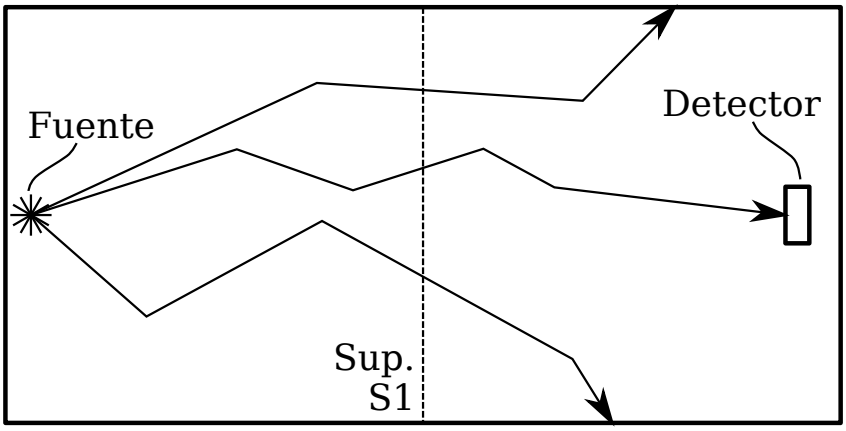

Figura 1: Esquema básico de una simulación por Monte Carlo. La superficie S1 permite la implementación de la técnica de reducción de varianza con la herramienta KSource.

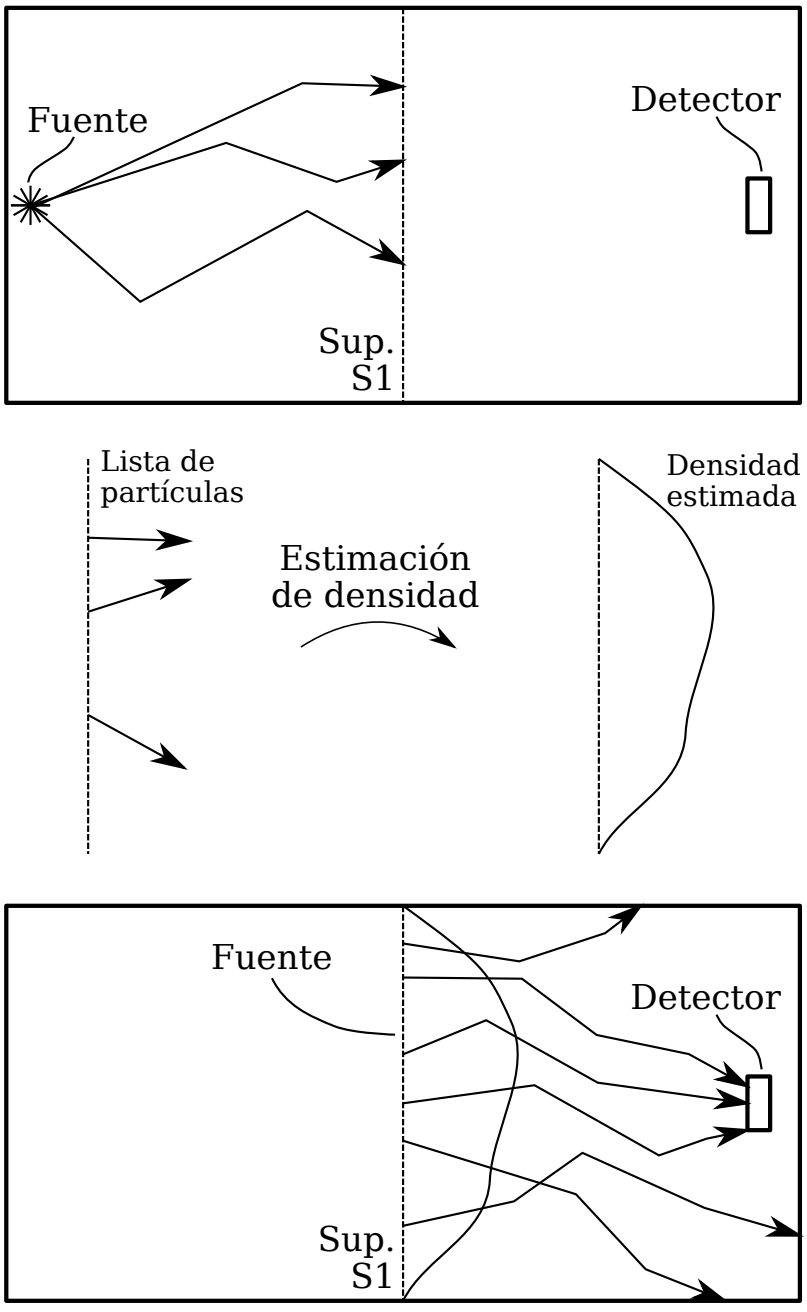

Figura 2: Esquema de la implementación del método KSource. Se utiliza una lista de partículas registrada en una superficie intermedia para estimar su distribución, y utilizar esta última como fuente en una segunda simulación.

1.1. *Kernel Density Estimation*

El método empleado para la estimación de densidad es *Kernel Density Estimation* (KDE). El mismo se basa en la siguiente expresión para la densidad estimada:

$$\tilde{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|H|}K(H^{-1}(x - x^{(i)})) \tag{1}$$

Donde:

- $X = \{x^{(i)} \in \mathbb{R}^D\}_{1\leq i\leq N}$ es el conjunto de muestras que representa la lista de partículas.
- $\tilde{p}(x)$ es la densidad estimada en función del vector $x = (x_1, ..., x_D)$.
- K es el *kernel* del método, que en KSource es la distribución normal (*gaussiana*).
- H es el ancho de banda (*bandwidth*). En el caso más general considerado en KSource $H = H^{(i)} = (h_1^{(i)}, ..., h_D^{(i)})$ es un vector, el cual puede ser diferente para cada partícula. Cuando $H^{(i)}$ es diferente para cada partícula se dice que el método KDE es adaptativo.

Considerando el *kernel gaussiano* y el ancho de banda vector la expresión resulta:

$$\tilde{p}(x) = \frac{1}{N(2\pi)^{D/2}}\sum_{i=1}^{N}\frac{1}{\prod_j h_j^{(i)}}exp\left(-\frac{1}{2}\sum_j\left(\frac{x_1 - x_1^{(i)}}{h_1^{(i)}}\right)^2\right) \tag{2}$$

El muestreo consiste en obtener nuevas muestras $\tilde{X} = \{\tilde{x}^{(i)} \in \mathbb{R}^D\}_{1\leq i\leq M}$ respetando la densidad estimada en 1. Para el método KDE, la misma se realiza con el siguiente algoritmo:

- Se toma un $x^{(i)} \in X$.
- Se obtiene la nueva muestra como $x = x^{(i)} + \Delta$, siendo Δ una perturbación aleatoria con la distribución del *kernel* (*gaussiana*) y los anchos de banda correspondientes a $x^{(i)}$.

Al repetir el muestreo, los $x^{(i)}$ se van tomando en orden tal como están en la lista de partículas. Cuando se llega al final, se vuelve al principio.

El ancho de banda es el parámetro más importante del método KDE. Para que la estimación de densidad tenga el mínimo error posible, los anchos de banda deben ser optimizados. Para ello KSource incluye tres métodos:

- Regla de Silverman: Estima el ancho de banda óptimo únicamente en base a la cantidad de partículas en la lista, su dimensionalidad, y la desviación estándar σ de cada variable. Es un método simple y rápido, aunque puede resultar en sobre-suavizado. Su fórmula es la siguiente:

$$h_{j,silv} = \left(\frac{4}{2+D}\right)^{\frac{1}{4+D}} N^{-\frac{1}{4+D}} \tag{3}$$

- K Vecinos Más Cercanos (KNN): Calcula el ancho de banda para cada variable como la distancia al K-ésimo vecino más cercano, dentro de la lista de partículas, donde K es un parámetro fijado por el usuario. Es una técnica relativamente sencilla y rápida para obtener un ancho de banda variable (KDE adaptativo).

- Validación Cruzada de Máxima Probabilidad (MLCV): Se basa en la maximización de la log-probabilidad media por validación cruzada, la cual funciona como factor de mérito de la estimación de densidad. Para ello se divide X en dos conjuntos X_{train} y X_{test}, y se calcula el factor de mérito como:

$$FM = \frac{1}{N_{test}} \sum_{X_{test}} \tilde{p}_{train}(x^{(i)}) \tag{4}$$

  Donde $\tilde{p}_{train}$ es el modelo KDE construido con X_{train}.

  Se construye una grilla de anchos de banda partiendo de una semilla y una grilla de factores. Para cada ancho de banda se calcula el factor de mérito, y se toma el ancho de banda que lo maximiza. Es un método costoso computacionalmente, pero robusto ante diferentes distribuciones. Se obtiene la máxima optimización cuando se utiliza como semilla un ancho de banda obtenido por KNN.

El carácter multidimensional del ancho de banda, es decir que $h_j^{(i)}$ sea diferente para cada variable j, puede simplificarse a un caso unidimensional, donde $h_1^{(i)} = ... = h_D^{(i)}$, mediante la normalización de datos. Usualmente, los anchos de banda para cada dimensión se toman como proporcionales a la desviación estándar de cada variable, es decir:

$$h_j^{(i)} = h^{(i)}\sigma_j \tag{5}$$

Introduciendo esta expresión en 2 se obtiene:

$$\tilde{p}(x) = \frac{1}{N(2\pi)^{D/2}} \sum_{i=1}^{N} \frac{1}{(h^{(i)})^D \prod_j \sigma_j} exp\left(-\frac{1}{2}\sum_j \left(\frac{x_1 - x_1^{(i)}}{h^{(i)}\sigma_j}\right)^2\right) = \frac{1}{\prod_j \sigma_j}\tilde{p}_n(x_n) \tag{6}$$

Donde $x_n = (\frac{x_1}{\sigma_1}, ..., \frac{x_D}{\sigma_D})$ es el vector normalizado, y $\tilde{p}_n$ es el modelo KDE construido con el conjunto de vectores normalizados, con anchos de banda $h^{(i)}$ unidimensionales.

La técnica consiste entonces en normalizar el conjunto X de datos, empleando los σ_j u otros factores de escaleo considerados apropiados, luego construir el modelo KDE normalizado $\tilde{p}_n(x_n)$, y finalmente obtener el modelo general de la ecuación 6. Los métodos de optimización de ancho de banda se aplican sobre el modelo normalizado.

1.2. KDE en simulaciones Monte Carlo: ¿Cuándo y por qué es útil el método KDE?

Considérese una simulación general por Monte Carlo con un único *tally* R y una fuente con distribución $S(x)$. En esta Subsección se analizará el impacto del error sistemático introducido por una fuente KDE sobre el valor de R, comparado con el muestreo directo de la lista de partículas, lo cual se denominará fuente de *tracks*. Se utilizará para ello la propiedad de las distribuciones de Poisson según la cual, al contar en un *tally* n partículas, se tiene un error estadístico de $\sqrt{n}$.

Desde el punto de vista de la función importancia, también llamada flujo adjunto o función de Green [5], el valor del *tally* puede expresarse como:

$$R = \int I(x)S(x)dx \approx I_0 \int_\Omega S(x)dx \tag{7}$$

Donde I es la función importancia, y la integración es sobre todo el espacio de fases. En la expresión aproximada se toma $I(x)$ como I_0 para $x \in \Omega$, y 0 afuera.

La fuente de *tracks* puede expresarse como una suma de deltas de Dirac:

$$S_t(x) = \frac{1}{N} \sum_i \delta \left(x - x^{(i)} \right) \tag{8}$$

El valor del *tally* resulta entonces:

$$R_t = \frac{1}{N} \sum_i I(x^{(i)}) \approx \frac{n_\Omega I_0}{N} \tag{9}$$

Donde n_Ω es la cantidad de partículas en la lista dentro de la región Ω.

Por lo tanto, el error del *tally* asociado a la fuente, en este caso, se puede estimar como:

$$\frac{\Delta R_t^{sist}}{R_t} = \frac{1}{\sqrt{n_\Omega}} \tag{10}$$

Para la fuente KDE, por simplicidad, se supondrá un ancho de banda único h, ya optimizado. Para la estimación de R con dicha fuente se distinguirán dos situaciones:

- $\mu(\Omega) >> h^D$
- $\mu(\Omega) << h^D$

Donde $\mu(\Omega)$ es la medida de la región Ω.

En el primer caso, al ser el ancho de banda mucho menor a la longitud característica de Ω, el resultado en el *tally* resultará muy similar al caso con fuente de *tracks*. También será similar el error asociado a la fuente, salvo por un *bias* dependiente del valor medio de la pendiente de la distribución de fuente en la frontera de Ω.

En este caso se puede concluir que el método KDE no resulta de gran utilidad, ya que no mejora el error del *tally* asociado a la fuente. Resulta más apropiado utilizar la fuente de *tracks*, si es necesario muestreando varias veces cada partícula en la lista, para reducir el error estadístico de la simulación.

En el segundo caso, al ser el ancho de banda mucho mayor a la longitud característica de Ω, se puede suponer a $S_{KDE}(x)$ como lineal dentro de Ω, y aproximar R como:

$$R_{KDE} = I_0 \mu(\Omega) S_{KDE}(x_0) \tag{11}$$

Donde S_{KDE} es la distribución de la fuente KDE, y x_0 es un vector de fase dentro de Ω.

El error de R asociado a la fuente, en este caso, se obtiene directamente del error puntual del método KDE. El mismo se compone de un error estadístico, que depende de la cantidad de partículas usadas en la estimación, y un *bias* por el suavizado [1].

$$\frac{\Delta R^{sist}_{KDE}}{R_{KDE}} = \frac{\sqrt{r}}{\sqrt{n_h}} + bias(x_0) \tag{12}$$

Donde n_h es la cantidad de partículas dentro de una hiper-esfera de radio h centrada en x_0, y r es la denominada rugosidad del *kernel*, que vale $1/2\sqrt{\pi}$ para el *kernel gaussiano*. Para otros *kernels* siempre es del orden de 1.

Es en este caso en el que el método KDE puede resultar útil, reduciendo el error en el *tally* asociado a la fuente. Debido a que $\mu(\Omega) << h^D$, se tiene que $n_\Omega << n_h$, y por lo tanto $1/\sqrt{n_\Omega} >> 1/\sqrt{n_h}$. Es decir que el método KDE reduce significativamente el error de R asociado al error estadístico de la fuente, aunque a costa de un error adicional asociado al *bias*. Si bien no es posible asegurar de forma general el resultado de dicho balance, es de esperar que, gracias a la optimización del ancho de banda, una mejora sea probable.

En conclusión, las situaciones en la que el método KDE resulta más útil son aquellas en las que la región de la fuente que afecta el *tally* es pequeña, con pocas partículas en su interior, y en particular con distancias características menores al ancho de banda optimizado. Al haber pocas partículas en la región de fuente "observada" por el *tally*, el error asociado a una fuente de *tracks* es grande, y el suavizado que realiza el método KDE, teniendo en cuenta un mayor número de partículas, resulta beneficioso. Estas situaciones pueden relacionarse con problemas de propagación en vacío, donde existen pocos caminos por los cuales las partículas pueden llegar de la fuente a la región de interés, en contraposición con los problemas de moderación.

La herramienta KSource, a pesar de estar esencialmente orientada a fuentes KDE, también permite implementar fuentes de *tracks*, y en ambos casos facilita el acople entre códigos Monte Carlo.

2. Contenidos del paquete KSource

La herramienta KSource cuenta con los siguiente componentes:

- Biblioteca en Python, para estimación de densidad. Aquí se encuentras las herramientas necesarias para crear y optimizar una fuente KSource en base a una lista de partículas. Se puede además analizar su estadística, generar gráficos de las distribuciones, y exportar la fuente creada, para su uso en los otros componentes de KSource.
- Biblioteca en C, para muestreo. Aquí se encuentras las herramientas necesarias para muestrear nuevas partículas, en base a una fuente KSource previamente creada.
- Aplicación de línea de comando. Permite ejecutar un re-muestreo de partículas, en base a una fuente KSource previamente creada, obteniéndose una lista de partículas de longitud arbitraria. También permite acceder de forma simple a otras utilidades incuidas en el paquete.
- Plantillas y archivos útiles. Facilitan las operaciones más usuales con el paquete KSource. Se incluyen archivos Jupyter Notebook con las operaciones más usuales de la biblioteca de Python, así como *scripts* y otros componentes para ejecutar algunos códigos Monte Carlo en acople con KSource.

En la Sección 3 se describen los principales objetos definidos en las librerías, mediante los cuales se modelan y manipulan las fuentes KDE. En los Apéndices B, C y D se presenta la documentación detallada de la aplicación de línea de comando y de las librerías. Por otra parte, en la Sección 4 se describe el flujo de trabajo requerido para emplear la herramienta KSource en una simulación Monte Carlo.

3. El formato de fuente KSource

La herramienta KSource cuenta con librerías para modelar fuentes distribucionales de partículas en Python y C. En ambos casos se define la estructura `KSource`, la cual a su vez posee dos subestructuras: `PList` y `Geometry`. Éstas modelan los dos componentes fundamentales de una fuente KSource: La lista de partículas y su geometría. En Python, la técnica KDE se implementa a través de la librería `KDEpy` [2].

3.1. Listas de partículas

Las listas de partículas utilizadas en KSource utilizan el formato MCPL [3]. El mismo permite la comunicación con los siguientes códigos Monte Carlo:

- MCNP

- PHITS
- McStas
- GEANT4
- TRIPOLI-4

Esto quiere decir que es posible convertir las listas de partículas registradas por cualquiera de estos códigos a MCPL, y viceversa. El *software* asociado al formato MCPL está incluido en la distribución de KSource, incluyendo funcionalidades extra con respecto a la distribución original, como la posibilidad de comunicación con TRIPOLI-4.

La estructura `PList` empleada en las librerías administra la comunicación con el archivo MCPL. Además de la lectura y escritura, se incluye la posibilidad de aplicar una traslación y rotación a las partículas inmediatamente despues de leerlas, lo cual puede ser útil al acoplar simulaciones con distinto sistema de referencia.

3.2. Geometría

A pesar de su nombre, la estructura `Geometry` no sólo administra la geometría de fuente, sino también la manera en que se debe tratar la energía y dirección de las partículas. El objetivo fundamental de esta estructura es convertir el conjunto de parámetros que definen una partícula, es decir energía, posición y dirección, a un vector de parametrización (variables parametrizadas), más adecuado para la aplicación del método KDE. Dicho vector es el que se utilizará como x en las expresiones de la Subsección 1.1.

Para dar versatilidad al tratamiento energético, espacial y angular deseado, la estructura `Geometry` posee a su vez un conjunto de subestructuras de tipo `Metric`. Usualmente se cuenta con tres de éstas: una para la energía, una para la posición, y una para la dirección. Cada `Metric` define la métrica con la que se tratará cada conjunto de variables. Los objetos `Metric` pueden elegirse del conjunto de métricas implementadas:

- `Energy`: tratamiento simple de la energía, sin transformaciones.
- `Lethargy`: emplear letargía en lugar de energía.
- `Vol`: tratamiento espacial volumétrico.
- `SurfXY`: tratamiento espacial plano en XY.
- `Guide`: geometría de guía de sección rectangular, con tratamiento de los ángulos basado en la superficie de cada espejo.
- `Isotrop`: tratamiento simple de la dirección, basado en el versor unitario de dirección.
- `Polar`: tratamiento de la dirección en base a los ángulo θ (distancia angular a la dirección $\hat{z}$) y ϕ (acimut medido desde la dirección $\hat{x}$).

- `PolarMu`: igual a `Polar`, pero con $\mu = cos(\theta)$ en lugar de θ.

En la biblioteca de Python, la función principal tanto de `Geometry` como de `Metric` es la de transformar las partículas en el formato de partícula de MCPL al vector de parametrización, en formato de `numpy.array`. Esto se logra a través de las funciones `transform` e `inverse_transform`.

En la biblioteca de C, por su parte, considerando que ésta se enfoca en el muestreo de partículas, lo cual se logra mediante la perturbación (ver 1.1), la función principal de `Geometry` y `Metric` es la de perturbar partículas, respetando la métrica correspondiente y con los anchos de banda provistos. Esto se logra con las funciones `[MetricName]_perturb`, donde `[MetricName]` se debe reemplazar por el nombre de cada métrica.

El objeto `Geometry` también incluye (opcionalmente) una traslación y rotación espacial. Esto permite modelar fuentes ubicadas en dististas posiciones y con distintas orientaciones con respecto al sistema de coordenadas. Por ejemplo, permite modelar una fuente en el plano YZ mediante la métrica `SurfXY`, a través de una rotación de 90 grados en el eje Y.

3.3. Archivos de parámetros XML

Las fuentes creadas en Python, luego de la optimización, pueden guardarse en un archivo de parámetros en formato XML. En el mismo se registran los parámetros que definen las estructuras `PList` y `Geometry` que componen la fuente `KSource`, así como el *path* al archivo MCPL a utilizar. Este archivo de parámetros puede luego utilizarse para reconstruir la fuente mediante la biblioteca en C, o bien para re-muestrear directamente mediante la aplicación de línea de comando.

Si el ancho de banda del modelo es constante, su valor también se guarda en el mismo archivo XML. Si, por el contrario, el ancho de banda es variable (KDE adaptativo), es decir que se representa con un *array* de un valor por partícula, el mismo se guarda en un archivo separado, en formato binario. Para maximizar el ahorro de memoria, especialmente importante en listas de partículas largas, se utiliza el formato de punto flotante de simple presición (32 bits), sin separación entre valores.

4. Flujo de trabajo

El esquema del flujo de trabajo típico con la herramienta KSource se esquematiza en la Figura 3. Se parte de una lista de partículas inicial, la cual puede, o no, haber sido creada en una simulación anterior, y luego se ejecuta cierto número de simulaciones Monte Carlo. Cada una de ellas emplea una fuente de partículas KSource basada en la lista de partículas registrada inmediatamente antes, y registra una nueva lista de partículas para la etapa siguiente. Dependiendo del problema, pueden obtenerse resultados útiles en todas las simulaciones (por ejemplo, al construir un mapa de dosis), o bien sólo en la última (por ejemplo, al calcular el flujo a la salida de un haz). Cada simulación cubre una región de la

geometría modelada progresivamente más lejana a la fuente inicial, alcanzando distancias que no serían alcanzables en una sola corrida.

Cada construcción de una fuente KSource en base a una lista de partículas se realiza mediante la API en Python, y culmina al exportar un archivo de parámetros XML. Se provee una plantilla con el código requerido para dicha tarea en el archivo `preproc_tracks.ipynb`.

Es recomendable ejecutar dos veces cada etapa de simulación, una de ellas utilizando la fuente KDE de la manera usual, y en la otra muestreando partículas directamente de la

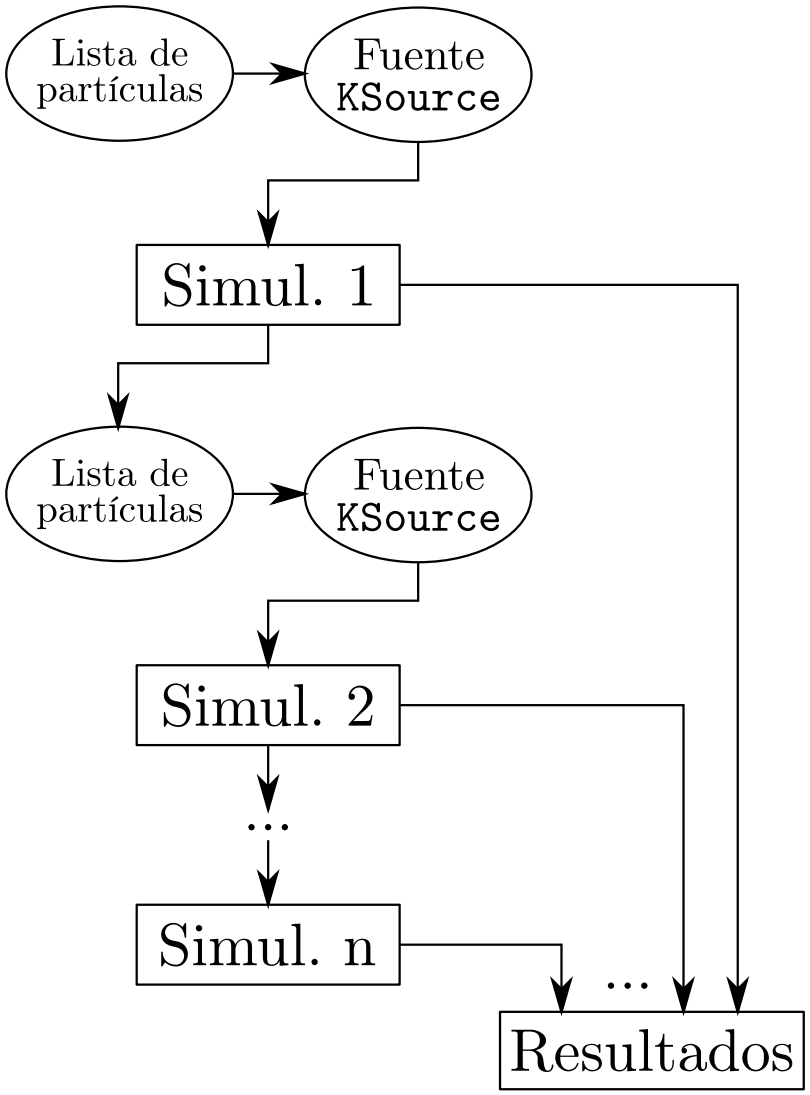

Figura 3: Esquema del flujo de trabajo típico.

lista de *tracks*. Esto puede lograrse de forma simple fijando el argumento `use_kde=0` en la función de muestreo, empleando la misma fuente `KSource`. El objetivo de dicha repetición del cálculo es verificar la compatibilidad de los resultados obtenidos con ambas fuentes.

De acuerdo a lo explicado en la Subsección 1.2, las fuentes de *tracks* pueden añadir mayor "ruido" a la simulación, pero no introducen ningún sesgo. Las fuentes KDE, gracias a su mejor estadística, permiten obtener resultados más detallados en zonas lejanas a la fuente, y más suaves, pero corren el riesgo de introducir errores sistemáticos debidos al sesgo de suavizado. Si los resultados que es posible obtener con ambas simulaciones (cercanos a la fuente) coinciden con ambas fuentes, puede considerarse que el sesgo de la fuente KDE es suficientemente bajo, y "confiar" en los resultados más lejanos.

4.1. Simulaciones con McStas y TRIPOLI-4

Si el código utilizado fuera McStas o TRIPOLI-4, se incluyen además archivos plantilla para facilitar las tareas de interpretación de los resultados de las simulaciones. El archivo `postproc.ipynb` permite colectar los principales resultados, para su registro en una planilla de cálculo. Por otra parte, el archivo `doseplots.ipynb` facilita la creación de gráficos de los mapas de dosis registrados en TRIPOLI-4, utilizando un módulo de Python incluido para tal fin.

4.2. Acople entre óptica neutrónica y transporte con McStas

Entre los componentes de McStas incluidos en el paquete KSource se encuentra `Guide_shielding`, mediante el cual es posible efectuar cálculos de blindaje en la perfiferia de guías neutrónicas, respetando la física correspondiente a la óptica neutrónica. Esto se logra a través de un acople en un solo sentido (*one-way coupling*) entre McStas y un código de transporte de radiación, como MCNP, PHITS o TRIPOLI.

El componente `Guide_shielding`, en una simulación de McStas, funciona de igual manera que otros componentes que simulan guías neutrónicas. En particular modela guías de sección rectangular, con posibilidad de curvatura (similar a `Bender`). La diferencia radica en que, al mismo tiempo que propaga los neutrones en su interior, registra en un archivo MCPL los escapes a través de sus espejos. Es decir que por cada reflexión, se registra la energía, posición y dirección del neutrón incidente, y el peso estadístico de la fracción no reflejada. En McStas la simulación continúa con la porción reflejada.

La lista de partículas obtenida representa una fuente superficial de neutrones escapando a través de los espejos de la guía. Dicha fuente no es plana, sino que tiene forma de tubo, posiblemente curvado, de sección rectangular, y es la causante de la presencia de radiación en la periferia de la guía, la cual debe ser blindada. Este archivo MCPL debe utilizarse entonces para construir una fuente KDE mediante la herramienta KSource, con ayuda del archivo plantilla `preproc_tracks.ipynb`, y emplear esta última en un cálculo de blindajes de la región que rodea a la guía, tipicamente un búnker o *hall*.

El componente `Guide_shielding` registra, además, las partículas *gamma* emitidas por absorciones en el níquel o titanio de los espejos, de acuerdo a las relaciones presentadas en [6]. Las absorciones se descuentan del peso estadístico de los neutrones transmitidos, por lo que no afectan las curvas de reflectividad. Por la alta energía de los fotones *prompt* del níquel, el correcto modelado de dicho fenómeno es importante para no subestimar la dosis *gamma* alrededor de la guía, especialmente ante flujos neutrónicos con alta componente térmica y fría. Los fotones generados, con energías obtenidas de los espectros de emisión y direcciones isotrópicamente aleatorias, son registrados en otro archivo MCPL, el cual debe ser utilizado para generar una fuente KDE para un cálculo de blindajes, del mismo modo que con la fuente de escapes neutrónicos.

4.3. Fuentes de activación con TRIPOLI-4 (u otros códigos)

Una fuente de activación es una fuente de radiación gamma proveniente del decaimiento radiactivo de nucleidos inestables presentes en un material, producidos mediante reacciones nucleares (activación neutrónica). Dicha fuente no es superficial, sino volumétrica. Desde luego, la herramienta KSource es capaz de modelar este tipo de fuentes a través de la técnica KDE, pero, como siempre, necesita para ello una lista de partículas. En este caso dicha lista debe contener las propiedades de los fotones al momento de cada decaimiento, cuya interpretación es la siguiente:

- Energía: Tiene valores discretos, los cuales, junto con sus frecuencias de ocurrencia, conforman el denominado espectro de decaimiento.
- Posición: Se corresponden con los puntos en los que ocurrieron las reacciones de activación.
- Dirección: Cada gamma de activación tiene una dirección isotrópicamente aleatoria.

Esto quiere decir que, de contarse con una lista de fotones de activación, puede procederse a contruir una fuente KDE con las herramientas descritas anteriormente, en particular con la ayuda del archivo plantilla `preproc_tracks.ipynb`. Debe tenerse la precaución de fijar en cero los anchos de banda en energía, para respetar el carácter discreto del espectro de decaimiento.

Sin embargo, el modo más usual de registrar la activación en materiales es a través de *tallies* volumétricos. En conjunto con el espectro de decaimiento, es posible utilizar dichos *tallies* para crear una lista de gammas de decaimiento, y luego aplicar las herramientas de KSource. En particular, para el código TRIPOLI-4, la API en Python de KSource cuenta con un módulo que permite realizar de forma sencilla la conversión de *tally* a lista de *tracks*, mediante la clase `T4Tally`. Además, el archivo plantilla `preproc_tally.ipynb` cuenta con el conjunto de operaciones necesarias para construir una fuente KDE, en base a un *tally* de activación de TRIPOLI-4 y un espectro de decaimiento.

Referencias

[1] B. E. Hansen, "Lecture Notes on Nonparametrics", University of Wisconsin (2009), `https://www.ssc.wisc.edu/~bhansen/718/NonParametrics1.pdf`

[2] KDEpy (2018), `https://kdepy.readthedocs.io/en/latest/`

[3] T. Kittelmann, et al., "Monte Carlo Particle Lists: MCPL", *Computer Physics Communications*, vol. 218, pp. 17-42 (2017), `https://doi.org/10.1016/j.cpc.2017.04.012.`

[4] K. A. Olive, et al. (Particle Data Group), Chinese Physics C 38 (2014) 090001. DOI:10.1088/1674-1137/38/9/090001.

[5] Bell, G. I. and S. Glasstone. "Nuclear Reactor Theory", Van Nostrand Reinhold Co., New York, (1970).

[6] R. Kolevatov, C. Schanzer, P. Böni, "Neutron absorption in supermirror coatings: Effects on shielding", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 922, pp. 98-107, (2019). DOI: 10.3233/JNR-190123.

Apéndice A. Descripción del formato de archivo de parámetros XML

Un archivo de parámetros XML es el método por el cual se puede almacenar una fuente KDE en el disco. Usualmente dicho archivo es generado mediante la API en Python, luego de la optimización del ancho de banda. Este formato sirve además como método de comunicación entre los distintos componentes de KSource, ya que puede utilizarse para reconstruir una fuente tanto en Python como en C, y para aplicar re-muestreo por línea de comando.

Como se muestra en el ejemplo del Listing 1, un archivo de parámetros posee la siguiente información:

- Corriente total de la fuente, en unidades $[1/s]$.
- `PList`: Lista de partículas.
    - Tipo de partícula global.
    - Archivo MCPL con la lista de partículas.
    - Traslación (opcional).
    - Rotación (opcional).
    - Transformación $(x, y, z) \rightarrow (y, z, x)$ (opcional).

- `Geom`: Geometría y tratamiento de variables. El orden indica la cantidad de submétricas.
  - Submétricas. Para cada una, además de su nombre, se indica:
    - Dimensionalidad.
    - Parámetros (específicos para cada métrica).
  - Traslación (opcional).
  - Rotación (opcional).
- Factores de escaleo para cada variable de parametrización.
- Ancho de banda. Si es constante (`variable="0"`) se indica su valor, mientras que si es adaptativo (`variable="1"`) se indica el *path* del archivo conteniendo la lista de valores. Dicho archivo debe poseer una secuencia de valores en formato binario de punto flotante de simple precisión (32 bits), sin separación, de la misma longitud que la lista de partículas en el archivo MCPL.

Listing 1: Ejemplo de archivo de parámetros XML.

```
<?xml version="1.0" ?>
<KSource>
    <J units="1/s">1.0</J>
    <PList>
        <pt>n</pt>
        <mcplname>/path/to/mcplfile.mcpl.gz</mcplname>
        <trasl>  0.   0. -10.</trasl>
        <rot/>
        <x2z>0</x2z>
    </PList>
    <Geom order="3">
        <Lethargy>
            <dim>1</dim>
            <params nps="1">10.0</params>
        </Lethargy>
        <SurfXY>
            <dim>2</dim>
            <params nps="5">-50. 50. -50. 50. 0.</params>
        </SurfXY>
        <PolarMu>
            <dim>2</dim>
            <params nps="0"/>
        </PolarMu>
        <trasl>10.  0. 50.</trasl>
        <rot>0.         3.14159265 0.        </rot>
    </Geom>
```

```
    <scaling>2.2339 14.1445 10.0147 0.2362 103.6779</scaling>
    <BW variable="1">/path/to/bwfile</BW>
</KSource>
```

Apéndice B. Documentación de aplicación `kstool`

La aplicación de linea de comando de KSource se accede a través del comando `kstool`. Sus instrucciones de uso se pueden obtener mediante el argumento `--help`, y se muestran en el Listing 2.

Listing 2: Instrucciones de uso del comando `kstool` (output de "`kstool --help`").

```
Usage: kstool [opciones]

KSource is a Monte Carlo calculations assistance tool. It implements particles
density estimation and sampling by means of Kernel Density Estimation method.

Options:
    resample:   Resample particles based on a ksource XML file.
    templates:  Copy templates for Monte Carlo calculations.
    [Any MCPL command]
    -h, --help: Display usage instructions.
```

La opción `resample` de `kstool` permite generar nuevas muestras en base a un modelo KSource guardado en un archivo de parámetros XML. Sus instrucciones de uso se pueden obtener mediante el argumento `--help`, y se muestran en el Listing 3.

Listing 3: Instrucciones de uso del comando `kstool resample` (output de "`kstool resample --help`").

```
Usage: kstool resample sourcefile [options]

Resample particles from source defined in XML file sourcefile, and save them in
a MCPL file.

Options:
    -o outfile: Name of MCPL file with new samples
                (default: \"resampled.mcpl\").
    -n N:       Number of new samples (default: 1E5).
    -h, --help: Display usage instructions.
```

La opción `templates` de `kstool` permite copiar los archivos plantilla (Jupyter Notebooks) de las operaciones más usuales con la API de Python al directorio de trabajo. Además, opcionalmente, permite copiar plantillas para ejecutar McStas o TRIPOLI-4 en acople con KSource. Sus instrucciones de uso se pueden obtener mediante el argumento `--help`, y se muestran en el Listing 4.

Listing 4: Instrucciones de uso del comando `kstool templates` (output de "`kstool templates --help`").

```
Usage: kstool templates dest [options]

Copy to dest templates for KSource usage in Python, or for interacting with
Monte Carlo codes.

Options:
    --mcstas:   Copy templates for using McStas.
    --tripoli:  Copy templates for using TRIPOLI-4.
    --all:      Copy all templates.
    -h, --help: Display usage instructions.
```

Por último, el comando `kstool` permite acceder a cualquiera de las aplicaciones de línea de comando de MCPL, en su versión extendida incluida en el paquete KSource.

Apéndice C. Documentación de API en Python

La API en Python de KSource, denominada `ksource`, se compone de los siguientes módulos:

- `ksource.py`: Módulo para el objeto `KSource`, que representa una fuente KDE.
- `kde.py`: Módulo para métodos de selección de ancho de banda, para la librería `KDEpy`.
- `plist.py`: Módulo para el objeto `PList`, y operaciones sobre listas de partículas.
- `geom.py`: Módulo para el objeto `Geometry`, la clase abstracta `Metric`, y todas sus implementaciones heredadas.
- `stats.py`: Módulo para análisis estadístico de listas de partículas, mediante el objeto `Stats`.
- `summary.py`: Módulo para colectar los principales resultados de simulaciones Monte Carlo, mediante el objeto `Summary`.
- `tally.py`: Módulo para lectura, gráficos, y conversión a lista de partículas, de *tallies* de TRIPOLI-4, mediante el objeto `T4Tally`.
- `utils.py`: Utilidades generales.

Todas las funcionalidades de los módulos mencionados se encuentran presentes en el *namespace* principal de la librería `ksource`.

C.1. Ejemplo de uso

Listing 5: Ejemplo básico de uso de API en Python.

```
import ksource as ks

# Define particle list
plist = ks.PList("surfsource", readformat="ssw")
# Define geometry
geom = ks.GeomFlat()
# Create KSource
s = ks.KSource(plist, geom, bw="mlcv")

# Fit KSource
s.fit(N=1E5)

# Save in XML file
s.save("source.xml")
```

En el Listing 5 se muestra un ejemplo básico de uso. Las principales etapas son:

- Creación de objeto `KSource` en base a objetos `PList` y `Geometry`.
- Ajuste de anchos de banda.
- Gráficos de distribuciones.
- Guardado de la fuente en archivo XML.

En los archivos plantilla (disponibles mediante "`kstool templates .`") se muestran más ejemplos de uso de los distintos módulos de KSource.

C.2. Módulo `ksource`

Este módulo se centra en la clase `KSource`, que modela una fuente de partículas KDE. Para su creación se debe contar previamente con objetos `PList` y `Geometry`. Mediante un modelo `KSource` se puede aplicar el método KDE sobre una lista de partículas, optimizar el ancho de banda, generar gráficos de las distribuciones estimadas, y guardar el modelo en un archivo XML, para su posterior uso con la misma u otras APIs.

El objeto `KSource` sirve como *wrapper* de un modelo KDE de la librería `KDEpy`, el cual guarda en el parámetro `kde`. Es posible modificar manualmente el ancho de banda modificando el parámetro `bw` de dicho objeto. Además `KSource` posee un parámetro `scaling`, el cual contiene los factores de normalización para cada variable (ver 1.1).

En el Listing 5 se muestra un ejemplo de uso básico del objeto KSource, mientras que en el Listing 6 se muestra el uso de las funciones de gráficos. Es posible realizar gráficos tanto 1D como 2D, y para cada uno de ellos existen 2 posibilidades:

- Gráficos integrados: Se grafica la densidad en función de 1 ó 2 variables parametrizadas, integrando sobre las demás (en un rango finito o en todo su dominio). Este es el caso de `plot_integr`, `plot_E` y `plot2D_integr`. Es el método de graficación recomendado. La función `plot_E` grafica el espectro en función de la energía independientemente de la parametrización elegida.
- Gráficos puntuales: Se grafica la densidad conjunta en función de 1 ó 2 variables no parametrizadas. Este es el caso de `plot_point` y `plot2D_point`.

Listing 6: Ejemplo de gráficos de distribuciones estimadas.

```
# Energy plot
EE = np.logspace(-9,1,50)
fig,[scores,errs] = s.plot_E(EE)
plt.show()

# Theta plot
tt = np.linspace(0,180,50)
fig,[scores,errs] = s.plot_integr("theta", tt)
plt.show()

# XY plot of epithermal neutrons
umin = 3  # Minimum lethargy
umax = 16 # Maximum lethargy
# Vector of min and max vals (parametrized)
#      [u   , x     , y     , theta, phi]
vec0 = [umin,-np.inf,-np.inf, 0    ,-180]
vec1 = [umax, np.inf, np.inf, 180  , 180]
xx = np.linspace(-10,10,30)
yy = np.linspace(-10,10,30)
s.plot2D_integr(["x,y"], [xx,yy], vec0=vec0, vec1=vec1)
plt.show()

# XY plot at fixed energy and angle
# Vector of fixed values (non parametrized)
#       [E   , x, y, z, dx, dy, dz]
part0 = [1E-3, 0, 0, 0, 0 , 0 , 1 ]
xx = np.linspace(-10,10,30)
yy = np.linspace(-10,10,30)
s.plot2D_point(["x,y"], [xx,yy], part0=part0)
plt.show()
```

C.3. Módulo `kde`

Este módulo contiene métodos de selección de ancho de banda que complementan la librería `KDEpy`. Debido a que esta no admite anchos de banda multidimensionales, y gracias

a la normalización de datos que aplica la clase `KSource`, los métodos en este módulo asumen que las dispersiones de los datos en cada variable son unitarios, y obtienen anchos de banda unidimensionales. Si se admite, desde luego, ancho de banda adaptativo, es decir uno por cada partícula. Se incluyen 3 técnicas de optimización:

- Regla de Silverman: Se obtiene el ancho de banda en función de N_{eff} y *dim* mediante el método `bw_silv`.
- K Vecinos Más Cercanos (KNN): Se obtiene un ancho de banda adaptativo como la distancia al K-ésimo vecino de cada partícula, mediante el método `bw_knn`. El cálculo se realiza por *batches*, y se debe especificar o bien la cantidad de vecinos por *batch* o bien la cantidad estimada total. Si la cantidad resultante de vecinos por *batch* no es entera se utiliza un factor de ajuste *f*.
- Validación Cruzada de Máxima Probabilidad (MLCV): Se evalúa el *score* de log-probabilidad media con un esquema de validación cruzada, para cada valor en una grilla de anchos de banda. Se toma el ancho de banda que maximiza dicho *score*. Es posible especificar tanto el ancho de banda semilla como la grilla de factores a utilizar para construir la grilla de anchos de banda. Se obtiene el ancho de banda óptimo con el método `bw_mlcv`.

El método recomendado es la MLCV, utilizando una semilla proveniente de KNN, aunque también es el más costoso. De todos modos, el método más adecuado puede variar según el problema. Nótese que, si se pretende guardar la fuente KDE y utilizarla para muestrear nuevas partículas, el método KNN, y el MLCV que lo utilice como semilla, debe realizarse utilizando todas las partículas en la lista (argumento "`N=-1`" en función `fit`), pues de lo contrario habría más partículas que anchos de banda para el KDE adaptativo.

Por último, el método `optimize_bw` sirve como *wrapper* de los distintos métodos de optimización, redirigiendo a a cada uno correspondientemente. Este método es llamado por la clase `KSource` al momento del ajuste.

C.4. Módulo `plist`

Este módulo se centra en la clase `PList`, la cual sirve como *wrapper* de listas de partículas en formato MCPL. Su principal función es permitir acceder a las partículas almacenadas, convirtiéndolas a formato de `numpy`. Además incluye la posibilidad de aplicar una traslación y una rotación a las partículas apenas luego de leerlas, lo cual es útil al cambiar de sistema de referencia entre simulaciones. En el Listing 7 se muestra un ejemplo de uso.

Es posible emplear de manera conjunta más de una lista de partículas, siempre y cuando tengan todas el mismo formato. Los formatos posibles son `"mcpl"`, `"ssw"` (MCNP), `"phits"`, `"ptrac"` (MCPN), `"stock"` (TRIPOLI-4) y `"ssv"` (ASCII).

Listing 7: Ejemplo de uso de `PList`.

```
file1 = "surfsource1"
file2 = "surfsource2"
trasl = [0, 0, -20]
rot = [0, np.pi/2, 0]

# Create PList
pl = ks.PList([file1,file2], readformat='ssw', pt='n', trasl=trasl, rot=rot)

# Get particles and weights
parts,ws = pl.get(1E4)
```

Otras funcionalidad presentes en el módulo son:

- `convert2mcpl` y `join2mcpl`: Permiten convertir listas de partículas de cualquier formato compatible con MCPL a MCPL. Internamente ejecutan los comandos de conversión de línea de comando.
- `savessv` y `appendssv`: Permiten guardar *arrays* de partículas en formato de `numpy` en archivos ASCII SSV. Estos pueden luego ser convertidos a MCPL, por ejemplo mediante `convert2mcpl`.

C.5. Módulo `geom`

Este módulo se centra en las clases `Geometry` y `Metric`. La clase `Geometry` representa un conjunto de tratamientos a las variables que definen una partícula. La misma se compone de un conjunto de métricas, además de una posición y rotación que definen la ubicación espacial de la fuente. Las métricas definen el tratamiento de cada conjunto de variables (energía, posición y dirección) a través de una transformación de parametrización.

La clase `Metric` es una clase abstracta, con las siguientes herencias implementadas:

- `Energy`: Tratamiento simple para la energía, sin transformación.
- `Lethargy`: Métrica de letargía, definida como $u = log(E_0/E)$.
- `Vol`: Tratamiento simple para posición, para fuentes volumétricas.
- `SurfXY`: Tratamiento simple para posición, para fuentes planas en XY.
- `Guide`: Tratamiento conjunto para posición y dirección, para fuentes con geometría de guía.
- `Isotrop`: Métrica simple para la dirección, basada en la distancia angular entre direcciones.
- `Polar`: Métrica polar para la dirección, con ángulos θ (distancia angular al eje z) y ϕ (ángulo acimutal con respecto al eje x).

- `PolarMu`: Métrica polar, con $\mu = cos(\theta)$.

Además, se definieron las siguientes funciones para la creación rápida de las geometrías más usuales.

- `GeomFlat`: Fuente plana. Métricas: `Lethargy`, `SurfXY` y `Polar`.
- `GeomGuide`: Fuente sobre espejos de guía. Métricas: `Lethargy` y `Guide`.
- `GeomActiv`: Fuente volumétrica. Métricas: `Energy`, `Vol` e `Isotrop`.

En el Listing 8 se muestra un ejemplo de uso de `Geometry`.

Listing 8: Ejemplo de uso de `Geometry`.

```
# Metrics
m_E = ks.Lethargy(E0=20)
m_pos = ks.SurfXY(z=5)
m_dir = ks.Polar()

# Create Geometry
trasl = [0,0,15]
rot = [0,np.pi,0]
geom = ks.Geometry([m_E,m_pos,m_dir], trasl=trasl, rot=rot)

# Use Geometry
vecs_param = geom.transform(parts)
parts = geom.inverse_transform(vecs_param)
mean = geom.mean(parts=parts, weights=ws)
std = geom.std(parts=parts, weights=ws)
```

C.6. Módulo `stats`

Este módulo se centra en la clase `Stats`, la cual permite realizar un análisis de algunos indicadores estadísticos, y su relación con la cantidad de partículas. Sirve como complemento de las herramientas incluidas en la API de MCPL para tal fin, más precisamente las funciones `collect_stats`, `dump_stats` y `plot_stats`.

En el Listing 9 se muestra un ejemplo de uso de la clase `Stats`. La misma se inicializa con una lista de partículas obtenida de `PList`, y permite graficar la variación de parámetros estadísticos con el número de partículas. A través del argumento `steps` se regula la cantidad de subconjuntos, de tamaño creciente, para los cuales se evaluará el parámetro estadístico correspondiente. Por ejemplo, para `steps=2` la evaluación se realizaría para la mitad de la lista de partículas, y para la lista completa. Los gráficos obtenidos permiten observar el grado de convergencia de los parámetros de interés para el número de partículas en la lista.

Listing 9: Ejemplo de uso de `Stats`.

```
stats = ks.Stats(parts, ws)

N,I,err = stats.mean_weight(steps=100)
plt.show()
N,mn,err = stats.mean(var=1, steps=100)
plt.show()
N,std,err = stats.std(var=1, steps=100)
plt.show()
```

C.7. Módulo `summary`

Este módulo se centra en el objeto `Summary`, y está especialmente pensado para simplificar la transcripción a una planilla de cálculo de los resultados principales en una simulación. Actualmente, los códigos soportados para esta funcionalidad son McStas y TRIPOLI-4.

En el Listing 10 se muestra un ejemplo de uso de `Summary`. Además, se provee una plantilla para el mismo fin en el archivo `postproc.ipynb`.

Listing 10: Ejemplo de uso de `Summary`.

```
mccode = "TRIPOLI"            # "McStas" or "TRIPOLI"
folder = "outdir"             # Directory containing simulation outputs
bashoutput = "bash.out"       # Bash output from simulation
n_detectors = [tracks1]       # Neutron list detectors
t4output = "simul.out"        # TRIPOLI output (only if mccode="TRIPOLI")
tallies = ["activ-fe","dose"] # Tally names

summary = ks.Summary(mccode,
                     folder,
                     bashoutput=bashoutput,
                     n_detectors=n_detectors,
                     t4output=t4output,
                     tallies=tallies)

summary.save("summary.txt")  # Saves inside summary.folder
```

C.8. Módulo `tally`

Este módulo está destinado al procesamiento de *tallies* volumétricos. Actualmente sólo está implementado para TRIPOLI-4, a través de la clase `T4Tally`. El mismo tiene dos funciones principales:

- Graficar mapas de dosis.
- Convertir *tallies* de activación a listas de partículas, para generar fuentes de activación.

En el Listing 11 se muestra un ejemplo de uso de la clase `T4Tally` para un *tally* de activación. El tratamiento de *tallies* de activación, y su uso para generación de fuentes KDE de fotones de decaimiento, se facilita a través del archivo plantilla `preproc_tally.ipynb`. Por otra parte, el uso de `T4Tally` para gráficos de mapas de dosis se muestra en `doseplots.ipynb`.

Listing 11: Ejemplo de uso de `T4Tally`.

```
t4output = "simul.out" # Simulation output file
tallyname = "activ-fe" # Tally name
spectrum = "decay-fe"  # Energy decay spectrum

tally = ks.T4Tally(t4output, tallyname, spectrum=spectrum)

# Plot tally
[fig, [scores,errs]] = tally.plot2D(['z', 'x'])

# Save as particle list
mcplname = tally.save_tracks("activsource.mcpl")
```

C.9. Módulo `utils`

Este módulo posee utilidades auxiliares. Las mismas se describen a continuación:

- Funciones de conversión entre formatos de tipo de partícula (`pt2pdg` y `pdg2pt`): Convierten entre el Código PDG de tipo de partícula [4], y la notación con caracteres (`"n"`: neutrón, `"p"`: fotón, `"e"`: electrón).
- Funciones de *weighting* (`H10`): Sirven para aplicar un peso a las partículas, y así dar más importancia a las que son de más interés para una aplicación en particular. Actualmente sólo está implementado el *weighting* por factor dosimétrico.
- Funciones de *masking* (`Box`): Sirven para restringir la región en el espacio de fases en el que pueden encontrarse las partículas, eliminando las que se encuentren por fuera. Sirve para focalizar el análisis u optimización en una región de interés. Actualmente sólo está implementado el *masking* tipo hipercubo, es decir fijando un rango para cada variable.

Apéndice D. Documentación de API en C

La API en C consiste en la librería compartida `ksource`, cuya interfaz de usuario está definida en cuatro archivos de cabeceras, los cuales se describen a continuación:

- `ksource.h`: Se definen las estructuras `KSource`, que modela una fuente KDE, y `MultiSource`, que modela un conjunto de `KSource`'s, además de sus funciones de utilización.

- `plist.h`: Se define la estructura `PList`, *wrapper* de listas de partículas MCPL, y sus funciones de utilización.
- `geom.h`: Se definen las estructuras `Geometry` y `Metric`, que definen el tratamiento de variables, y sus funciones de utilización.
- `utils.h`: Utilidades generales.

Además, se incluye el archivo de cabecera `KSourceConfig.h`, donde se define la versión de `KSource` mediante las siguientes constantes:

```
#define KSource_VERSION_MAJOR 1
#define KSource_VERSION_MINOR 0
#define KSource_VERSION_PATCH 0
```

En el archivo `ksource.h` se incluyen los archivos `mcpl.h`, `KSourceConfig.h`, `plist.h`, `geom.h` y `utils.h`, por lo que sólo es necesario incluir `ksource.h` para utilizar la librería `ksource`. Para compilar un programa que utiliza `ksource` se deben utilizar los siguientes comandos:

```
KSOURCE=/path/to/ksourceinstall
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$KSOURCE/lib
gcc example.c -lksource -lmcpl -lm -I$KSOURCE/include -L$KSOURCE/lib
```

Donde `/path/to/ksourceinstall` es el *path* al directorio donde se instaló el paquete `KSource`.

D.1. Estructuras `KSource` y `MultiSource`

La estructura `KSource` modela una fuente de partículas KDE. Usualmente se crea en base a un archivo de parámetros XML generado mediante la API en Python, aunque es posible crearla solamente mediante la API en C. La funcionalidad principal de esta estructura es el muestreo de partículas.

La estructura `MultiSource`, por su parte, modela un conjunto de fuentes `KSource` superpuestas. En cada muestreo se elige aleatoriamente la fuente a emplear, respetando las intensidades relativas de las fuentes. Es posible implementar *source biasing*, fijando las frecuencias de muestreo por separado las intensidades relativas, lo cual implica un ajuste en los pesos de las partículas nacientes.

A continuación se describen las definiciones, estructuras de datos y funciones declaradas en el archivo `ksource.h`.

```
#define MAX_RESAMPLES 1000
#define NAME_MAX_LEN 256
```

`MAX_RESAMPLES` define el máximo número de intentos al muestrear, es decir que si no se obtiene una partícula válida luego de `MAX_RESAMPLES` intentos se termina la simulación. `NAME_MAX_LEN` define la máxima longitud posible del nombre de un archivo a ser leído con el paquete `KSource`.

```
typedef double (*WeightFun)(const mcpl_particle_t* part);
```

Definición de función de *weighting*. Puede utilizarse en algunas funcionalidades para aplicar un pesado o *biasing* basado en los parámetros de las partículas.

```
typedef struct KSource{
    double J;       // Total current [1/s]
    PList* plist;   // Particle list
    Geometry* geom; // Geometry defining variable treatment
} KSource;
```

Estructura `KSource`. Modela una fuente KDE, la cual se compone de una lista de partículas (`PList`) y un tratamiento de variables (`Geometry`). Incluye además el valor de corriente total, en unidades $[1/s]$. Dicho valor es utilizado para definir las intensidades relativas en fuentes múltiples `MultiSource`.

```
KSource* KS_create(double J, PList* plist, Geometry* geom);
```

Crear estructura `KSource` en base a estructuras `PList` y `Geometry` previamente creados. Debe definirse también un valor de corriente total (unidades $[1/s]$). Dicho valor es utilizado para definir las intensidades relativas en fuentes múltiples `MultiSource`, por lo que puede fijarse en 1 si no se planea crear dicha estructura.

```
KSource* KS_open(const char* xmlfilename);
```

Cargar fuente `KSource` en base al archivo de parámetros XML de nombre `xmlfilename`. El mismo usualmente es creado mediante la API en Python.

```
int KS_sample2(KSource* ks, mcpl_particle_t* part, int perturb, double w_crit,
    WeightFun bias, int loop);
```

Función principal para muestreo de partículas con una fuente `KSource`. La partícula muestreada se guarda en `part`. Incluye los siguientes argumentos para configurar el muestreo:

- `perturb`: Si es 0, las partículas se muestrean directo del archivo MCPL sin modificación. Sino, se aplica una perturbación con la distribución del *kernel* y el ancho de banda correspondiente, de acuerdo a la técnica de muestreo con KDE.
- `w_crit`: Si es menor o igual a 0, se fija el peso estadístico de la partícula muestreada como $w = w_0$, siendo w_0 el peso de la partícula original en el archivo MCPL. Si es mayor a 0, se normaliza w a 1, utilizando la siguiente técnica:
  - Si $w_0 < w_{crit}$: Se usa w_0/w_{crit} como probabilidad de tomar la partícula, en lugar de descartarla y avanzar en la lista.
  - Si $w_0 > w_{crit}$: Se usa w_{crit}/w_0 como probabilidad de avanzar en la lista luego del muestreo.

  De este modo, en promedio se utilizará w_0 veces cada partícula en la lista. Se recomienda fijar `w_crit` como el peso medio en la lista.
- `bias`: Función de *weighting* para aplicar *biasing* durante el muestreo. Será ignorada si `w_crit<=0`.
- `loop`: Si es 0, se llama a `exit(EXIT_SUCCESS)` al llegar al final de la lista, terminando la simulación. Sino, al llegar al final de la lista se vuelve al inicio.

```
int KS_sample(KSource* ks, mcpl_particle_t* part);
```

Función de muestreo simple de partículas con una fuente `KSource`. Redirige a `KS_sample2`, con los argumentos `perturb=1`, `w_crit=1`, `bias=NULL` y `loop=1`.

```
double KS_w_mean(KSource* ks, int N, WeightFun bias);
```

Computar peso medio de las partículas en la lista utilizada por la fuente `ks`. Se utilizan `N` partículas para el cómputo. Si se fija `bias` distinto de `NULL`, se incluye la función de *weighting* `bias`.

```
void KS_destroy(KSource* ks);
```

Destruir fuente `KSource`, liberando toda la memoria asociada.

```
typedef struct MultiSource{
    int len;     // Number of sources
    KSource** s; // Array of sources
    double J;    // Total current [1/s]
    double* ws;  // Frequency weights of sources
    double* cdf; // cdf of sources weights
} MultiSource;
```

Estructura `MultiSource`. Modela un conjunto de fuentes KDE superpuestas. Los valores en el *array* `ws` definen las frecuencias de muestreo de cada fuente, mientras que sus intensidades se obtienen del parámetro `J` de cada fuente `KSource`.

```
MultiSource* MS_create(int len, KSource** s, const double* ws);
```

Crear estructura `MultiSource` en base a la cantidad de fuentes, el *array* de estructuras `KSource`, y las frecuencias de muestreo deseadas. Durante la creación se computa la corriente total `J` y la función acumulativa de densidad `cdf`.

```
MultiSource* MS_open(int len, const char** xmlfilenames, const double* ws);
```

Cargar un conjunto de fuentes `KSource` de los archivos de parámetros XML definidos en `xmlfilenames`, y construir estructura `MultiSource`.

```
int MS_sample2(MultiSource* ms, mcpl_particle_t* part, int perturb, double
    w_crit, WeightFun bias, int loop);
```

Función principal para muestreo de partículas con una fuente `MultiSource`. Se elige aleatoriamente una fuente usando las frecuencias definidas en `ms->ws`, y se le redirige el muestreo pasándole los mismos parámetros. Luego del muestreo se multiplica el peso de la partícula por el factor:

$$f_{SB} = \frac{J_i/J_{tot}}{w_i/w_{tot}} \tag{13}$$

Donde el subíndice i representa la fuente elegida para el muestreo, y *tot* la suma sobre todas las fuentes. De este modo se corrige la eventual discrepancia entre la intensidad relativa y la frecuencia relativa de muestreo, de acuerdo con la técnica de *source biasing*.

```
int MS_sample(MultiSource* ms, mcpl_particle_t* part);
```

Función de muestreo simple de partículas con una fuente `MultiSource`. Redirige a `MS_sample2`, con los argumentos `perturb=1`, `w_crit=1`, `bias=NULL` y `loop=1`.

```
double MS_w_mean(MultiSource* ms, int N, WeightFun bias);
```

Computar peso medio de las partículas en las listas de todas las fuentes. Computa los pesos medios de cada fuente mediante la función `KS_w_mean` con los mismos parámetros `N` y `bias`, y computa el peso medio global como el promedio pesado mediante los valores en `ms->ws`.

```
void MS_destroy(MultiSource* ms);
```

Destruir fuente `MultiSource`, liberando toda la memoria asociada.

D.2. Estructura `PList`

La estructura `PList` modela una lista de partículas, actuando como *wrapper* de un archivo MCPL. Permite acceder a las partículas, e incluye la posibilidad de aplicarles una traslación y una rotación luego de la lectura.

A continuación se presentan las estructuras y funciones declaradas en el archivo `plist.h`.

```
typedef struct PList{
    char pt;                      // Particle type ("n", "p", "e", ...)
    int pdgcode;                  // PDG code for particle type

    char* filename;               // Name of MCPL file
    mcpl_file_t file;             // MCPL file

    double* trasl;                // PList translation
    double* rot;                  // PList rotation
    int x2z;                      // If true, apply permutation x,y,z -> y,z,x

    const mcpl_particle_t* part; // Pointer to selected particle
} PList;
```

Definición de la estructura `PList`. La misma tiene fijado el tipo de partícula según `pt`. Incluye la estructura correspondiente para la lectura de un archivo MCPL, además de (opcionalmente), parámetros que definen una transformación espacial a aplicar luego de la lectura de partículas. El parámetro `part` apunta a la ultima partícula leída, en todo momento.

```
PList* PList_create(char pt, const char* filename, const double* trasl, const
    double* rot, int switch_x2z);
```

Crear estructura `PList`. Se debe definir el tipo de partícula (`"n"` para neutrón, `"p"` para fotón, `"e"` para electrón) y el nombre del archivo MCPL. Opcionalmente se puede definir una traslación (*array* 3D) y una rotación (*array* 3D, formato eje-ángulo), o fijar en `NULL` dichos argumentos en caso contrario. La rotación se aplica luego de la traslación. Si `switch_x2z` es distinto de 0, luego de aplicar la rotación y traslación (de haberlas), se aplica la transformación $(x, y, z) \to (y, z, x)$.

```
int PList_get(const PList* plist, mcpl_particle_t* part);
```

Obtener partícula, aplicar transformaciones (de haberlas), y guardarla en `part`. La partícula se obtiene de `plist->part`, y no se modifica dicha variable luego de la lectura.

```
int PList_next(PList* plist, int loop);
```

Avanzar en la lista, actualizando la variable `plist->part`, hasta la siguiente partícula válida. Se considera una partícula como válida si tiene peso estadístico mayor a cero y su código PDG (tipo de partícula) coincide con el de la `PList`.

```
void PList_destroy(PList* plist);
```

Destruir estructura `PList`, liberando toda la memoria asociada.

D.3. Estructuras `Geometry` y `Metric`

La función principal de la estructura `Geometry` es perturbar partículas siguiendo la distribución del *kernel*. Lo logra redirigiendo dicha tarea a las métricas correspondientes a cada conjunto de variables, las cuales utilizan una función de perturbación específica para cada tipo de métrica. `Geometry` también se encarga de administrar los anchos de banda y normalización de variables.

A continuación se presentan las definiciones, estructuras y funciones declaradas en `geom.h`.

```
typedef struct Metric Metric;

typedef int (*PerturbFun)(const Metric* metric, mcpl_particle_t* part,
    double bw);

struct Metric{
    int dim;            // Dimension
    float* scaling;     // Variables scaling
    PerturbFun perturb; // Perturbation function
    int nps;            // Number of metric parameters
    double* params;     // Metric parameters
};
```

Definición de la estructura `Metric`, en conjunto con la definición de función de perturbación. La función principal de dicha estructura es perturbar un conjunto de variables, lo cual realiza llamando a la función disponible en `perturb`, la cual a su vez utilizará, además del ancho de banda provisto como argumento, los escaleos de variables (`scaling`) y los parámetros de la métrica (`params`). El significado de dichos parámetros depende según el tipo de métrica, pudiendo representar tamaños de la fuente, valores mínimos, máximos o de referencia de variables, etc. Algunas métricas no poseen parámetros.

```
Metric* Metric_create(int dim, const double* scaling, PerturbFun perturb, int
    nps, const double* params);
```

Crear estructura `Metric`. Se debe proveer la dimensionalidad de la métrica (`dim`), los factores de escaleo (`scaling`), y la cantidad y valores de parámetros de la métrica (`nps` y `params`).

```
void Metric_destroy(Metric* metric);
```

Destruir estructura `Metric`, liberando toda la memoria asociada.

```
typedef struct Geometry{
    int ord;          // Number of submetrics
    Metric** ms;      // Submetrics
    char* bwfilename; // Bandwidth file name
    FILE* bwfile;     // Bandwidth file
    double bw;        // Normalized bandwidth

    double* trasl;    // Geometry translation
    double* rot;      // Geometry rotation
} Geometry;
```

Definición de la estructura `Geometry`. La misma contiene una cantidad `ord` de métricas almacenadas en `ms`. En el caso de que el ancho de banda sea adaptativo, `Geometry` administra la lectura del archivo donde se almacenan sus valores. En cualquier caso el parámetro `bw` contiene el ancho de banda actual. `KSource` debe encargarse de que el ancho de banda presente en `bw` siempre se corresponda con la partícula presente en el parámetro `part` de la `PList`. Los parámetros `trasl` y `rot` representan la ubicación y orientación espacial de la fuente.

```
Geometry* Geom_create(int ord, Metric** metrics, double bw, const char*
    bwfilename,
    const double* trasl, const double* rot);
```

Crear estructura `Geometry`. Se debe indicar el orden `ord` de la misma (cantidad de métricas), proveer la ubicación de las métricas previamente creadas (`metrics`), y la posición y rotación de la fuente (`trasl` y `rot`). Para modelos con ancho de banda constante éste se debe proveer en el argumento `bw` y se debe fijar `bwfilename=NULL`, mientras que para ancho de banda adaptativo se debe indicar el archivo que contiene los anchos de banda (en formato binario de secuencia de puntos flotantes de simple precisión), y `bw` es ignorado.

```
int Geom_perturb(const Geometry* geom, mcpl_particle_t* part);
```

Perturbar partícula, siguiendo la distribución del *kernel* (*gaussiano*), y el ancho de banda presente en `geom->bw`.

```
int Geom_next(Geometry* geom, int loop);
```

Avanzar una posición en la lista de anchos de banda, en el caso de ancho de banda adaptativo. Para ancho de banda constante esta función no realiza ninguna acción.

```
void Geom_destroy(Geometry* geom);
```

Destruir estructura `Geometry`, liberando toda la memoria asociada.

```
#define E_MIN 1e-11
#define E_MAX 20
```

Valores mínimo y máximo de energía. Si luego de una perturbación se obtiene un valor de energía por fuera de este rango, se repite la perturbación.

```
int E_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar energía, con métrica simple de energía. En este caso `bw` multiplicado por el elemento de `scaling` correspondiente tiene unidades de energía (MeV).

```
int Let_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar energía, con métrica de letargía. En este caso `bw` multiplicado por el elemento de `scaling` correspondiente tiene unidades de letargía (adimensional).

```
int Vol_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar posición en sus 3 dimensiones, con métrica simple de posición. En este caso `bw` multiplicado por cada elemento de `scaling` correspondiente tiene unidades de posición (cm).

```
int SurfXY_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar posición en sus dimensiones x e y, con métrica simple de posición. En este caso `bw` multiplicado por cada elemento de `scaling` correspondiente tiene unidades de posición (cm).

```
int Guide_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar posición y dirección, con métrica de guía neutrónica. En este caso `bw` multiplicado por los dos primeros elementos de `scaling` tiene unidades de posición (cm), mientras que para los últimos dos tiene unidades de ángulo (grados). La métrica en dirección es polar, relativo a la normal de cada espejo. Internamente se transforma a las variables de guía z, t, θ, ϕ, se perturba dichas variables, y se antitransforma.

```
int Isotrop_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar dirección, con métrica isotrópica. En este caso `bw` multiplicado por el elemento de `scaling` correspondiente tiene unidades de ángulo (grados). La perturbación sigue la denominada distribución de Von Mises-Fischer.

```
int Polar_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar dirección, con métrica polar relativa a z. En este caso `bw` multiplicado por cada elemento de `scaling` correspondiente tiene unidades de ángulo (grados). Internamente se transforma a θ, ϕ, se perturba dichas variables, y se antitransforma.

```
int PolarMu_perturb(const Metric* metric, mcpl_particle_t* part, double bw);
```

Perturbar dirección, con métrica polar relativa a z, utilizando $\mu = cos(\theta)$. En este caso `bw` multiplicado por el primer elemento de `scaling` tiene unidades de μ (adimensional), mientras que por el segundo elemento tiene unidades de ángulo (grados). Internamente se transforma a μ, ϕ, se perturba dichas variables, y se antitransforma.

```
static const int _n_metrics = 8;
static const char *_metric_names[] = {"Energy", "Lethargy", "Vol", "SurfXY", "
    Guide", "Isotrop", "Polar", "PolarMu"};
static const PerturbFun _metric_perturbs[] = {E_perturb, Let_perturb,
    Vol_perturb, SurfXY_perturb, Guide_perturb, Isotrop_perturb, Polar_perturb,
    PolarMu_perturb};
```

Variables estáticas conteniendo la cantidad, nombres y funciones de perturbación de las métricas implementadas.

D.4. Utilidades generales

Además de las estructuras descritas previamente, se incluye un conjunto de utilidades generales para resolver problemas específicos. Esto incluye funciones matemáticas, conversión entre formatos de partícula y factores dosimétricos.

A continuación se describen las funciones declaradas en `utils.h`.

```
double rand_norm();
```

Obtener valor aleatorio con distribución normal, centrada en cero y con dispersión unitaria. Internamente utiliza el método de Box-Muller.

```
double *traslv(double *vect, const double *trasl, int inverse);
double *rotv(double *vect, const double *rotvec, int inverse);
```

Trasladar y rotar vector tridimensional. Realiza la transformación *in-place* sobre `vect` y lo retorna. Si `inverse` es distinto de 0 se aplica la transformación inversa. `trasl` es el vector de traslación, mientras que `rot` es el vector de rotación en formato ángulo-eje.

```
int pt2pdg(char pt);
char pdg2pt(int pdgcode);
```

Convertir de partícula en formato *char* (`"n"`: neutrón, `"p"`: fotón, `"e"`: electrón) a código PDG [4], y viceversa.

```
double interp(double x, const double *xs, const double *ys, int N);
```

Función de interpolación. Los *arrays* `xs` e `ys`, de longitud `N`, poseen los valores a interpolar. Se devuelve el valor interpolado en la posición `x`.

```
double H10_n_ARN(double E);
double H10_p_ARN(double E);
double H10_n_ICRP(double E);
double H10_p_ICRP(double E);
```

Factores dosimétricos en función de la energía, en unidades $[pSv\ cm^2]$. `n` indica neutrón y `p` indica fotón. `ARN` y `ICRP` indican la referencia para la tabla de interpolación que se utilizará. La interpolación se realiza en escala logarítmica.