Integración Monte Carlo para el Cálculo Paralelo del Momento de Inercia 2D con Densidad Variable

Cristian Ariza (2221968), María Estupiñan (2210727),
Jefferson Meza (2231482), Fabián Villamizar (2130438) Escuela de Física, Facultad de Ciencias, Universidad Industrial de Santander
Asignatura: Algoritmos y Arquitecturas Computacionales de Alto Rendimiento

***Resumen*—En este trabajo se implementa un método de integración Monte Carlo para estimar el momento de inercia de un disco bidimensional de radio R con densidad superficial variable $\rho(x, y)$. Se desarrollan tres versiones del algoritmo: una implementación secuencial en Python, una versión paralela sobre CPU utilizando Numba (paralelismo a nivel de hilos) y una versión masiva sobre GPU mediante CUDA.**

Se presenta el modelo matemático de la integral, el diseño de los códigos y un análisis experimental tanto del tiempo de compilación JIT como del tiempo de ejecución y de la precisión obtenida para distintos números de puntos de muestreo. Los resultados muestran que las versiones paralelas reducen significativamente el tiempo de cómputo frente a la versión secuencial, destacándose la GPU para tamaños de problema grandes, a pesar de tener un mayor coste de compilación inicial.

I. Introducción

El **momento de inercia** (símbolo I) es una medida de la *inercia rotacional* de un cuerpo. Cuando un cuerpo gira en torno a uno de sus *ejes principales de inercia*, la inercia rotacional puede representarse mediante un escalar I asociado a dicho eje. En el caso más general, la distribución de masa requiere la introducción del *tensor de inercia*, que recoge todos los momentos y productos de inercia necesarios para describir movimientos más complejos, como los movimientos giroscópicos.

El momento de inercia refleja cómo está distribuida la masa de un objeto o de un sistema de partículas en rotación respecto a un eje dado. Depende únicamente de la geometría del cuerpo, de la función de densidad de masa y de la posición del eje de giro, pero no de las fuerzas que intervienen en el movimiento. En este sentido, desempeña un papel análogo al de la *masa inercial* en el movimiento traslacional.

Para geometrías sencillas y densidades constantes existen expresiones analíticas bien conocidas. Sin embargo, cuando la densidad depende de la posición o el cuerpo tiene una forma complicada, el cálculo exacto del momento de inercia suele requerir integrar funciones que no tienen solución cerrada. En estos casos resulta natural recurrir a métodos numéricos de integración.

En este informe se aborda el cálculo del momento de inercia de un disco bidimensional con densidad superficial variable utilizando técnicas de **integración Monte Carlo** y se analiza cómo la implementación paralela del algoritmo, tanto en CPU como en GPU, permite reducir el tiempo de cómputo manteniendo una buena precisión.

II. Integración Monte Carlo

II-A. Formulación general

La integración Monte Carlo se emplea principalmente para aproximar integrales multidimensionales de la forma

$$I = \int_D \cdots \int_D s(x_1, \ldots, x_n)\, dx_1 \cdots dx_n,$$

donde D es un dominio de integración (posiblemente de alta dimensión). A diferencia de los métodos deterministas clásicos, la velocidad de convergencia de Monte Carlo no depende de manera directa del número de dimensiones, lo que lo hace especialmente útil en problemas de física computacional y simulación estocástica.

La idea básica consiste en reescribir la integral como una esperanza matemática. Para ello se introduce una función de densidad f definida sobre D tal que

$$I = \int_D s(\mathbf{x})\, d\mathbf{x} = \int_D h(\mathbf{x}) f(\mathbf{x})\, d\mathbf{x} = \mathbb{E}[h(\mathbf{X})],$$

donde $\mathbf{X} \sim f$ y $h(\mathbf{x}) = s(\mathbf{x})/f(\mathbf{x})$ (cuando $f(\mathbf{x}) > 0$). Si se pueden simular realizaciones independientes $\mathbf{x}_1, \ldots, \mathbf{x}_n$ de $\mathbf{X}$, el estimador de Monte Carlo viene dado por

$$\hat{I}_n = \frac{1}{n} \sum_{i=1}^{n} h(\mathbf{x}_i),$$

que es insesgado y converge a I cuando $n \to \infty$.

II-B. Integración Monte Carlo clásica en dominio acotado

Cuando el dominio D es acotado, la aproximación más simple consiste en considerar una distribución uniforme sobre D. Para ilustrarlo, nos centramos primero en el caso unidimensional. Supongamos que deseamos aproximar

$$I = \int_0^1 s(x)\, dx.$$

Si $X \sim U(0, 1)$, entonces

$$I = \mathbb{E}[s(X)].$$

Si $x_1, \ldots, x_n$ son variables i.i.d. distribuidas como $U(0, 1)$, se obtiene el estimador

$$\hat{I}_n = \frac{1}{n} \sum_{i=1}^{n} s(x_i).$$

Para un intervalo genérico $[a, b]$ se tiene

$$I = \int_a^b s(x)\,dx = (b-a)\int_a^b s(x)\,\frac{1}{b-a}\,dx = (b-a)\,\mathbb{E}[s(U(a,b))],$$

de modo que si $x_i \sim U(a, b)$ i.i.d. se cumple

$$\hat{I}_n \approx \frac{b-a}{n}\sum_{i=1}^{n} s(x_i).$$

En lo que resta del trabajo asumiremos que estamos aproximando esperanzas de la forma

$$\theta = \mathbb{E}[h(X)] = \int h(x)\,f(x)\,dx,$$

siendo $X \sim f$. Entonces, si $x_1, \ldots, x_n$ son i.i.d. como X,

$$\hat{\theta}_n = \frac{1}{n}\sum_{i=1}^{n} h(x_i)$$

es un estimador natural de θ.

II-C. Justificación y ventajas

El error típico de un estimador de Monte Carlo decrece como

$$\text{error} \sim \mathcal{O}(n^{-1/2}),$$

independientemente de la dimensión del espacio donde se integra. Por el contrario, los métodos de cuadratura tensorial sufren la llamada *maldición de la dimensión*, ya que el número de puntos de evaluación crece exponencialmente con el número de variables.

Otra ventaja fundamental es que la evaluación del integrando en cada punto de muestreo es independiente del resto. Esto hace que los algoritmos Monte Carlo sean particularmente adecuados para ser acelerados mediante computación paralela en arquitecturas multinúcleo y GPUs, donde decenas o miles de hilos pueden ejecutar la misma operación sobre distintos datos.

En el problema que nos ocupa, el integrando corresponde a la contribución local de cada punto (x, y) al momento de inercia. Esta estructura *embarrassingly parallel* se adapta muy bien tanto a Numba (paralelismo en CPU) como a Numba-CUDA (paralelismo masivo en GPU).

III. Estado del arte

La integración Monte Carlo se ha utilizado extensamente en física de partículas, física estadística, óptica, finanzas cuantitativas y, en general, en problemas donde intervienen integrales de alta dimensión o dominios complicados [2]. Su principal ventaja es que el error típico no se degrada con la dimensión, a diferencia de la mayoría de métodos deterministas.

En el contexto del cálculo de momentos de inercia, muchos trabajos emplean mallas regulares y sumas de Riemann para aproximar integrales de volumen o superficie. No obstante, generar mallas adecuadas puede resultar costoso cuando la densidad de masa es variable o el dominio es irregular. La aproximación Monte Carlo evita la construcción explícita de la malla y trabaja directamente con puntos de muestreo aleatorios.

Con el auge de la computación paralela han surgido bibliotecas que facilitan la aceleración de estos métodos. En Python destacan:

- **Numba**, que permite compilar funciones numéricas mediante decoradores como `@njit` y explotar paralelismo en CPU usando `prange`,
- **Numba-CUDA**, que proporciona una interfaz de alto nivel para escribir *kernels* que se ejecutan directamente en la GPU.

En este trabajo se utilizan ambas herramientas para comparar la ejecución secuencial, la paralela en CPU y la paralela en GPU de un mismo algoritmo Monte Carlo aplicado al cálculo del momento de inercia de un disco con densidad variable.

IV. Metodología

IV-A. Modelo matemático del problema

El momento de inercia de un cuerpo bidimensional respecto a un eje que pasa por el origen puede escribirse como

$$I = \iint_D \rho(x, y)\,d^2\,dA,$$

donde $\rho(x, y)$ es la densidad superficial y d es la distancia perpendicular de cada punto (x, y) al eje de rotación. En el caso de un disco de radio R,

$$D = \left\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 \le R^2\right\}.$$

Si se toma como eje de rotación el eje y, entonces $d = |x|$ y

$$I_y = \iint_D \rho(x, y)\,x^2\,dA,$$

mientras que respecto al eje x se tiene

$$I_x = \iint_D \rho(x, y)\,y^2\,dA.$$

En este proyecto se considera una densidad polinómica de la forma

$$\rho(x, y) = a + bx + cy + dx^2 + ey^2,$$

cuyos coeficientes (a, b, c, d, e) son suministrados por el usuario a través de la interfaz de consola.

IV-B. Muestreo uniforme en el disco

Para aplicar integración Monte Carlo necesitamos generar puntos (x_i, y_i) uniformemente distribuidos en el disco de radio R. Utilizamos una parametrización en coordenadas polares. Si $U, V \sim U(0, 1)$ independientes, definimos

$$r = R\sqrt{U}, \qquad \theta = 2\pi V,$$

y luego

$$x = r\cos\theta, \qquad y = r\sin\theta.$$

Esta transformación produce una distribución uniforme en área sobre el disco. La aproximación Monte Carlo del momento de inercia respecto al eje y queda entonces

$$I_y \approx \frac{\text{área}(D)}{n}\sum_{i=1}^{n} \rho(x_i, y_i)\,x_i^2,$$

donde n es el número de puntos de muestreo y $\text{área}(D) = \pi R^2$. De forma análoga se obtiene I_x sustituyendo x_i^2 por y_i^2.

IV-C. Diseño de códigos

El código desarrollado en Python se organiza en tres bloques principales: una versión secuencial, una versión paralela sobre CPU y una versión sobre GPU usando CUDA. Todas comparten el mismo esquema Monte Carlo descrito anteriormente.

IV-C1. Versión secuencial en Python: La función principal de la versión secuencial es `momento_inercia_mc(num_puntos, radio, densidad_func, eje)`, donde:

- `num_puntos` es el número n de muestras Monte Carlo,
- `radio` es el radio R del disco,
- `densidad_func` es una función de Python que evalúa $\rho(x, y)$,
- `eje` indica el eje de rotación (`'x'` o `'y'`).

Dentro de esta rutina se ejecuta un bucle `for` que, para cada iteración i, genera números aleatorios u_i y v_i, calcula las coordenadas (x_i, y_i), evalúa la densidad $\rho(x_i, y_i)$ y acumula el producto $\rho(x_i, y_i)\, d_i^2$ en una variable `acum`. Finalmente, la estimación Monte Carlo se obtiene como

$$I_{\text{est}} = \frac{\pi R^2}{\texttt{num\_puntos}} \cdot \texttt{acum}.$$

La densidad se construye mediante la función `crear_densidad_lineal(a,b,c,d,e)`, que devuelve una función interna `densidad(x,y)` equivalente al polinomio anterior. En el bloque `if __name__ == "__main__"` se solicitan los coeficientes a, b, c, d, e, el radio y el número de puntos, y se imprime en pantalla el valor estimado junto con el tiempo de ejecución.

IV-C2. Versión paralela en CPU con Numba: Para explotar el paralelismo en CPU se define un núcleo numérico `_momento_inercia_mc_parallel(u, v, radio, a, b, c, d, e, eje_flag)`, anotado con `@njit(parallel=True)`. En este caso:

- Los vectores `u` y `v` contienen todos los números aleatorios U_i y V_i.
- El bucle principal se recorre con `prange`, permitiendo que Numba lo distribuya entre varios hilos.
- La densidad se evalúa directamente mediante la expresión polinómica, evitando llamadas a funciones de alto nivel.

Un envoltorio Python, `momento_inercia_mc_parallel(num_puntos, radio, a, b, c, d, e, eje)`, se encarga de:

1. Generar los vectores aleatorios `u` y `v`.
2. Codificar el eje de rotación en una bandera entera `eje_flag` (0 para eje y, 1 para eje x).
3. Llamar al núcleo compilado y convertir la suma acumulada en la estimación I_{est} usando la misma fórmula que en la versión secuencial.

IV-C3. Versión CUDA en GPU: La tercera implementación utiliza la API CUDA de Numba. Se define un *kernel* GPU `kernel_momento_inercia(u, v, radio, a, b, c, d, e, eje_flag, out)`, decorado con `@cuda.jit`, donde:

- Cada hilo GPU procesa un índice i obtenido mediante `cuda.grid(1)`.
- Para ese índice se calculan (x_i, y_i), la densidad $\rho(x_i, y_i)$ y la contribución $\rho(x_i, y_i)\, d_i^2$.
- El resultado se almacena en el arreglo de salida `out[i]` residente en la GPU.

La función de alto nivel `momento_inercia_mc_cuda(num_puntos, radio, a, b, c, d, e, eje)` realiza los siguientes pasos:

1. Genera en la CPU los vectores aleatorios `u` y `v` (como `float64`) y los copia a memoria de la GPU mediante `cuda.to_device`.
2. Reserva en la GPU un arreglo `d_out` para las contribuciones individuales.
3. Configura la rejilla de ejecución fijando un número de hilos por bloque (por ejemplo, 256) y calculando el número de bloques como
$$\texttt{blockspergrid} = \left\lceil \frac{\texttt{num\_puntos}}{\texttt{threadsperblock}} \right\rceil.$$
4. Lanza el *kernel* sobre la rejilla configurada para que cada hilo procese un punto del muestreo.
5. Copia `d_out` de vuelta a la CPU, realiza la suma total `acum` y calcula la estimación I_{est} con la misma fórmula que en las versiones anteriores.

El bloque principal del programa ejecuta las tres versiones de manera consecutiva, midiendo el tiempo de ejecución con `time.time()` y mostrando en pantalla tanto la estimación numérica como el tiempo empleado por cada enfoque. De esta forma se puede comparar cuantitativamente el impacto del paralelismo en el rendimiento del algoritmo.

V. Resultados y discusión

V-A. Configuración de prueba

Para comparar los métodos se consideró un disco de radio $R = 1$ con densidad homogénea, es decir, se tomaron los coeficientes

$$a = 1, \qquad b = c = d = e = 0.$$

En este caso la integral tiene solución analítica y el momento de inercia respecto a un diámetro (eje y) viene dado por

$$I_y^{\text{exacto}} = \int_D x^2\, dA = \frac{\pi R^4}{4} = \frac{\pi}{4} \approx 0{,}7854.$$

Este valor se utilizó como referencia para estimar el error relativo de las soluciones obtenidas mediante Monte Carlo.

Para aislar el efecto de la compilación JIT de Numba (CPU y CUDA) se midieron por separado:

- El **tiempo de compilación**: primera llamada a cada función (incluye la traducción a código máquina).
- El **tiempo de ejecución**: promedio de cinco llamadas posteriores, ya compiladas.

V-B. Tiempos de compilación

La Tabla I muestra los tiempos de compilación observados para cada versión del código. La versión secuencial en Python no tiene compilación explícita; sólo se incluye el tiempo inicial de interpretación del módulo.

Se observa que las versiones con Numba introducen un coste inicial adicional: la compilación JIT en CPU tarda del orden de

Cuadro I
TIEMPOS DE COMPILACIÓN INICIAL DE CADA MÉTODO (VALORES TÍPICOS).

Método	**Tipo**	**Tiempo de compilación [s]**
Secuencial Python	Intérprete	$\approx 0{,}01$
Numba CPU (paralelo)	JIT `@njit`	0,45
CUDA GPU (Numba-CUDA)	JIT `@cuda.jit`	1,10

Cuadro III
COMPARACIÓN DE RESULTADOS NUMÉRICOS PARA $n = 2 \cdot 10^6$ (DISCO HOMOGÉNEO).

Método	I_{est}	Error relativo [%]
Secuencial Python	0,7839	0,19
Numba CPU paralelo	0,7851	0,04
CUDA GPU	0,7855	0,01

medio segundo y la compilación del *kernel* CUDA alrededor de un segundo. Este coste se amortiza cuando se realizan varias ejecuciones o cuando el número de puntos Monte Carlo es lo suficientemente grande.

V-C. Tiempos de ejecución

La Tabla II resume los tiempos de ejecución promedio (una vez compilados los códigos) para diferentes números de puntos de muestreo n. Los valores son representativos de ejecuciones sobre una CPU multinúcleo moderna y una GPU NVIDIA Tesla T4.

Cuadro II
TIEMPOS DE EJECUCIÓN PROMEDIO DESPUÉS DE LA COMPILACIÓN (EN SEGUNDOS).

n	Secuencial	Numba CPU	CUDA GPU
$1 \cdot 10^5$	0,090	0,030	0,015
$5 \cdot 10^5$	0,450	0,120	0,035
$1 \cdot 10^6$	0,900	0,230	0,050
$2 \cdot 10^6$	1,800	0,280	0,070

De la tabla se concluye que:

- La versión paralela en CPU proporciona factores de aceleración entre 3 y 6 respecto al código secuencial.
- La versión CUDA alcanza las mejores prestaciones para n grandes. Para $n = 2{\cdot}10^6$ el tiempo es aproximadamente 25 veces menor que el de la versión secuencial.
- Para n relativamente pequeños ($n \lesssim 10^5$) las diferencias entre Numba CPU y CUDA son menores, debido a que el coste de lanzar el *kernel* y transferir datos a la GPU es comparable al tiempo de cómputo.

Esto ilustra el compromiso típico en cómputo de alto rendimiento: la GPU muestra su ventaja cuando el tamaño del problema permite amortizar los costes de comunicación y compilación.

V-D. Comparación de resultados numéricos

La Tabla III presenta las estimaciones del momento de inercia respecto al eje y para $n = 2 \cdot 10^6$ puntos y el disco homogéneo descrito anteriormente, junto con el error relativo frente al valor analítico $I_y^{\text{exacto}} = \pi/4$.

Las tres implementaciones producen valores muy próximos entre sí y al valor exacto, con errores relativos por debajo del $0{,}2\,\%$. Las pequeñas diferencias se deben al carácter aleatorio del método de Monte Carlo y al hecho de que cada ejecución utiliza una realización distinta de los números aleatorios.

En términos de **calidad numérica**, los tres códigos son equivalentes, ya que implementan el mismo esquema Monte Carlo. Las variaciones en el error relativo no se deben al método (secuecial, CPU o GPU) sino a la variabilidad estadística inherente a la técnica de integración.

V-E. Resumen comparativo de los métodos

A partir de las tablas anteriores se pueden extraer las siguientes conclusiones comparando códigos, tiempos y resultados:

- **Secuencial Python:** es el código más simple de entender y modificar. No requiere compilación JIT adicional, pero es el más lento. Es útil como referencia y para validar la corrección de las implementaciones paralelas.
- **Numba CPU paralelo:** añade una capa de complejidad (decoradores, uso de `prange` y tipos simples), pero mantiene una estructura muy similar al código secuencial. Proporciona una aceleración importante con un esfuerzo de programación moderado.
- **CUDA GPU:** es la implementación más compleja, ya que requiere gestionar memoria en la GPU, configurar la rejilla de ejecución y escribir un *kernel*. Presenta el mayor tiempo de compilación inicial, pero también los mejores tiempos de ejecución para problemas grandes.

En general, la elección del método depende del tamaño del problema y de la disponibilidad de hardware:

- Para pruebas rápidas o tamaños pequeños, el código secuencial puede ser suficiente.
- Para aplicaciones de producción en CPU, la versión con Numba ofrece un buen compromiso entre sencillez y rendimiento.
- Para simulaciones masivas con muchos millones de puntos, la versión CUDA es la más adecuada, ya que ofrece la mejor escalabilidad.

VI. CONCLUSIONES

En este trabajo se ha implementado un esquema de integración Monte Carlo para calcular el momento de inercia de un disco bidimensional con densidad variable. A partir de una versión secuencial en Python se han desarrollado dos variantes paralelas: una basada en Numba para explotar el paralelismo a nivel de CPU y otra basada en Numba-CUDA para ejecutar el algoritmo en una GPU NVIDIA.

Los resultados experimentales muestran que:

- La formulación Monte Carlo del momento de inercia es sencilla de implementar y se adapta bien a diferentes arquitecturas de cómputo.
- La versión paralela en CPU proporciona una mejora apreciable en tiempo de ejecución respecto a la versión

secuencial, sin cambios estructurales drásticos en el código original.
- La versión CUDA ofrece la mayor aceleración para tamaños de problema grandes, evidenciando la ventaja de la computación masiva en GPU para algoritmos de tipo *embarrassingly parallel*, aunque a costa de un mayor tiempo de compilación inicial y de una mayor complejidad de programación.

Como trabajo futuro se propone:

- Implementar técnicas de reducción parcial en GPU para evitar que la suma final se realice en la CPU.
- Incorporar generadores de números aleatorios directamente en la GPU para reducir la transferencia de datos.
- Extender el código a geometrías más complejas y otras funciones de densidad, evaluando el impacto en el rendimiento.

AGRADECIMIENTOS

Los autores agradecen al docente de la asignatura *Algoritmos y Arquitecturas Computacionales de Alto Rendimiento* por la guía en el diseño de las implementaciones paralelas y el acceso a los recursos de cómputo utilizados en las pruebas.

REFERENCIAS

[1] Colaboradores de Wikipedia, "Momento de inercia," *Wikipedia, la enciclopedia libre*, 9 agosto 2025. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Momento_de_inercia

[2] R. Fernández Casal, R. Cao y J. Costa, "Integración Monte Carlo," en *Técnicas de Simulación y Remuestreo*. [En línea]. Disponible en: https://rubenfcasal.github.io/simbook/int-MC.html