Trabajo Especial
Modelos y Simulación

15 de Julio 2025, FaMAF

Tomás Achaval Berzero, Tomás Ezequiel Peyronel

Tema D

Resumen

El objetivo de este trabajo es analizar el desempeño de distintos generadores de números aleatorios al estimar una integral mediante el método de Monte Carlo.

Los generadores que analizamos son los siguientes:

- Generador congruencial lineal (**LCG**) con a = 16807, c = 0, m = $2^{31} - 1$
- Generador **XORShift** de 32 bits
- Generador **Xoshiro128**** de 32 bits

Y la integral que aproximamos mediante el método de Monte Carlo está dada por:

$$I_d = \int_{[0,1]^d} \prod_{i=1}^{d} \exp(-x_i^2)\, dx$$

Generadores

LCG

La fórmula de este generador es, partiendo de una semilla $\mathbf{y_0}$ distinta de 0:

$$\mathbf{y_{i+1} = (a * y_i) \bmod m}$$

donde **a** y **m** son constantes. Para este análisis utilizamos a = 16807 y m = $2^{31} - 1$. Estos parámetros son los recomendados por Stephen K. Park y Keith W. Miller[4] y son muy usados en la práctica.

Esta fórmula genera valores enteros entre 1 y m - 1, por lo que y_k / m son los valores de punto flotante uniformemente distribuidos en (0, 1) que utilizamos para estimar la integral.

Ventajas: velocidad, simplicidad, período máximo para un generador multiplicativo, estado pequeño: solo requiere guardar 32 bits.

Desventajas: no aprovecha los 32 bits, puesto que "solo" genera 2^{31} - 2 valores distintos. Otra desventaja de este generador es el ***problema de los hiperplanos***[1], pues está demostrado que los puntos generados para d dimensiones caen en no más de $\mathbf{(d! * m)^{1/d}}$ hiperplanos paralelos en $[0, 1]^d$.

XORShift de 32 bits

La fórmula de este generador es, partiendo de una semilla $\mathbf{y_0}$ distinta de 0:

$$\mathbf{x = y_i}$$
$$x = x \wedge (x << \mathbf{a})$$
$$x = x \wedge (x >> \mathbf{b})$$
$$x = x \wedge (x << \mathbf{c})$$
$$\mathbf{y_{i+1} = x}$$

donde **a**, **b**, **c** son constantes. En este caso, elegimos [a, b, c] = [13, 17, 5], que corresponde a la recomendación de Marsaglia en *Xorshift RNGs*[3] debido a la calidad empírica del generador obtenida con estos parámetros.

Esto genera valores enteros entre 1 y 2^{32} - 1, por lo que y_k / 2^{32} son los valores de punto flotante uniformemente distribuidos en (0, 1) que utilizamos para estimar la integral.

Ventajas: velocidad, simplicidad, estado pequeño: solo requiere guardar 32 bits, aprovecha el estado completo pues su período es de 2^{32} - 1
Desventajas: no pasa algunos de los test estadísticos de aleatoriedad más utilizados[2].

Xoshiro128** de 32 bits

En este caso, partiendo de una semilla y estado inicial $\mathbf{s = [s_0, s_1, s_2, s_3]}$ con al menos un $\mathbf{s_i}$ distinto de 0 y teniendo en cuenta que **rotl(x, k)** = **(x << k) | (x >> (32 - k))**,

La fórmula está dada por

$$\mathbf{y_{i+1}} = \text{rotl}(s[1] * 5 , 7) * 9$$

y luego se debe actualizar el estado con las siguientes operaciones:

```
t = s[1] << 9
s[2] =  s[2] ^ s[0]
s[3] = s[3] ^ s[1]
s[1] = s[1] ^ s[2]
s[0] = s[0] ^ s[3]
s[2] = s[2] ^ t
s[3] = rotl( s[3] , 11 )
```

Esto genera valores enteros entre 0 y 2^{32} - 1, por lo que y_k / 2^{32} son los valores de punto flotante uniformemente distribuidos en [0, 1) que utilizamos para estimar la integral.

Ventajas: Mayor período de 2^{128} - 1. Mejor desempeño en los tests estadísticos de calidad más utilizados[2].
Desventajas: Estado más grande que el resto de los generadores utilizados (128 bits). Más complejidad y cantidad de operaciones por generación. Más lento.

Problema a simular

Queremos estimar mediante simulación estocástica la siguiente integral definida sobre el hipercubo $[0, 1]^d$:

$$I_d = \int_{[0,1]^d} \prod_{i=1}^{d} \exp(-x_i^2)\, dx$$

cuyo valor exacto es conocido y es dado por la siguiente fórmula:

$$I_d = \left(\int_0^1 e^{-x^2} dx \right)^d = \left(\frac{\sqrt{\pi}}{2} \cdot \mathrm{erf}(1) \right)^d$$

Utilizaremos este valor de referencia para medir la calidad de nuestras estimaciones.

Metodología

A la hora de implementar los generadores en Python, tuvimos que aplicar en varios lugares máscaras con 0x00000000FFFFFFFF para simular enteros de 32 bits, pues en Python los números enteros tienen precisión arbitraria.

Para la elección de la semilla de Xoshiro128**, que es un arreglo de cuatro enteros, decidimos utilizar el generador XORShift para generar este arreglo a partir de un solo entero. Por esta razón se encuentra el generador XORShift_int, que devuelve enteros de 32 bits, y también el generador XORShift, que envuelve XORShift_int y devuelve valores en (0, 1).

Para estimar el valor de la integral utilizamos simulaciones mediante el método de Monte Carlo, promediando $\mathbf{f(U_1, \ldots, U_d)} = \prod_{i=1}^{d} e^{-U_i^2}$ sobre **N** puntos **d**-dimensionales uniformemente distribuídos en $(0, 1)^d$ generados por cada generador. Estas simulaciones generan valores que estiman la esperanza de $\mathbf{f(U_1, \ldots, U_d)}$ donde cada $\mathbf{U_i}$ es una variable aleatoria con distribución **U(0, 1)**. Esta esperanza es el valor exacto de la integral.

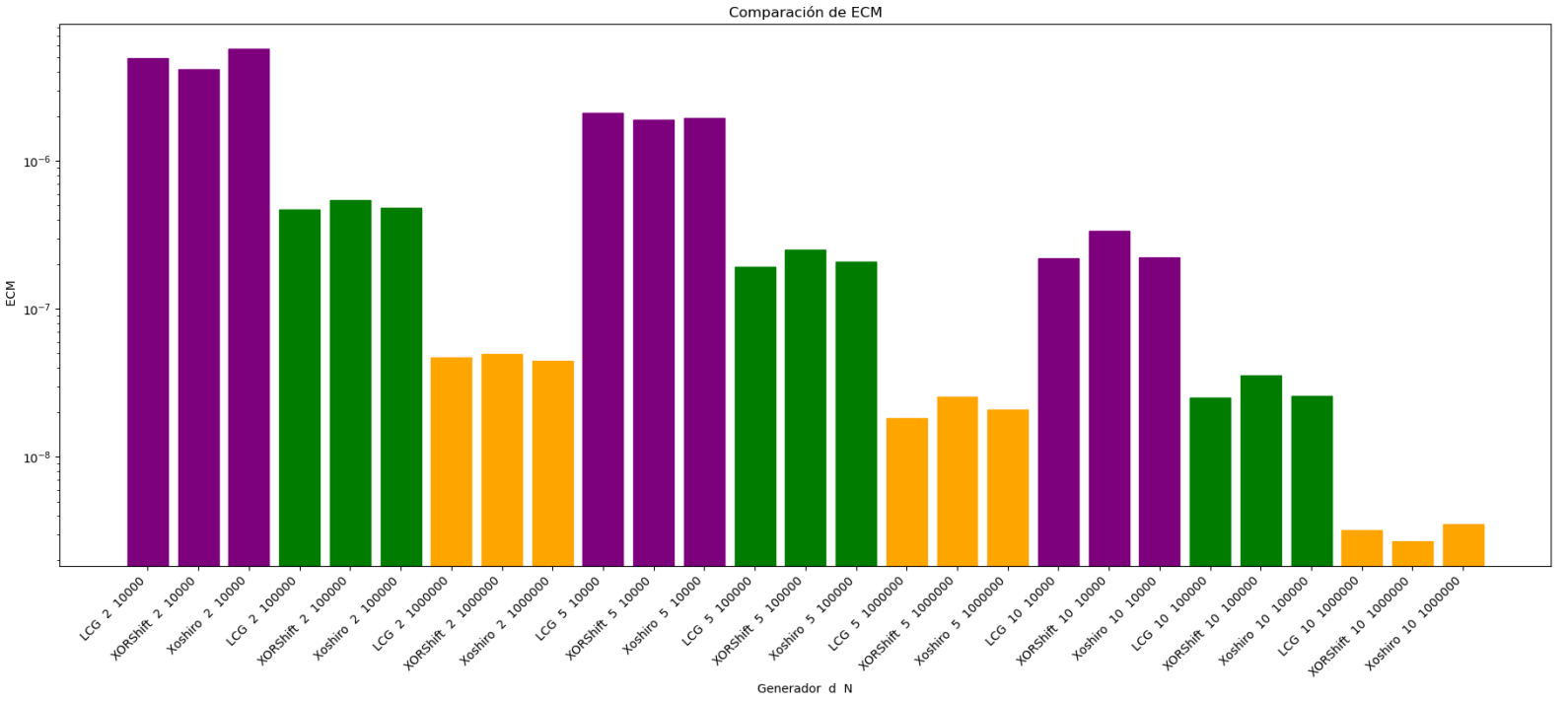
Para comparar los generadores, analizamos sus comportamientos en simulaciones con los valores **d** = 2, 5, 10 y **N** = 10^4, 10^5, 10^6. Para ello, tomamos **M** = 100

estimaciones para cada configuración (generador, d, N) y a partir de ellas analizamos las siguientes métricas:

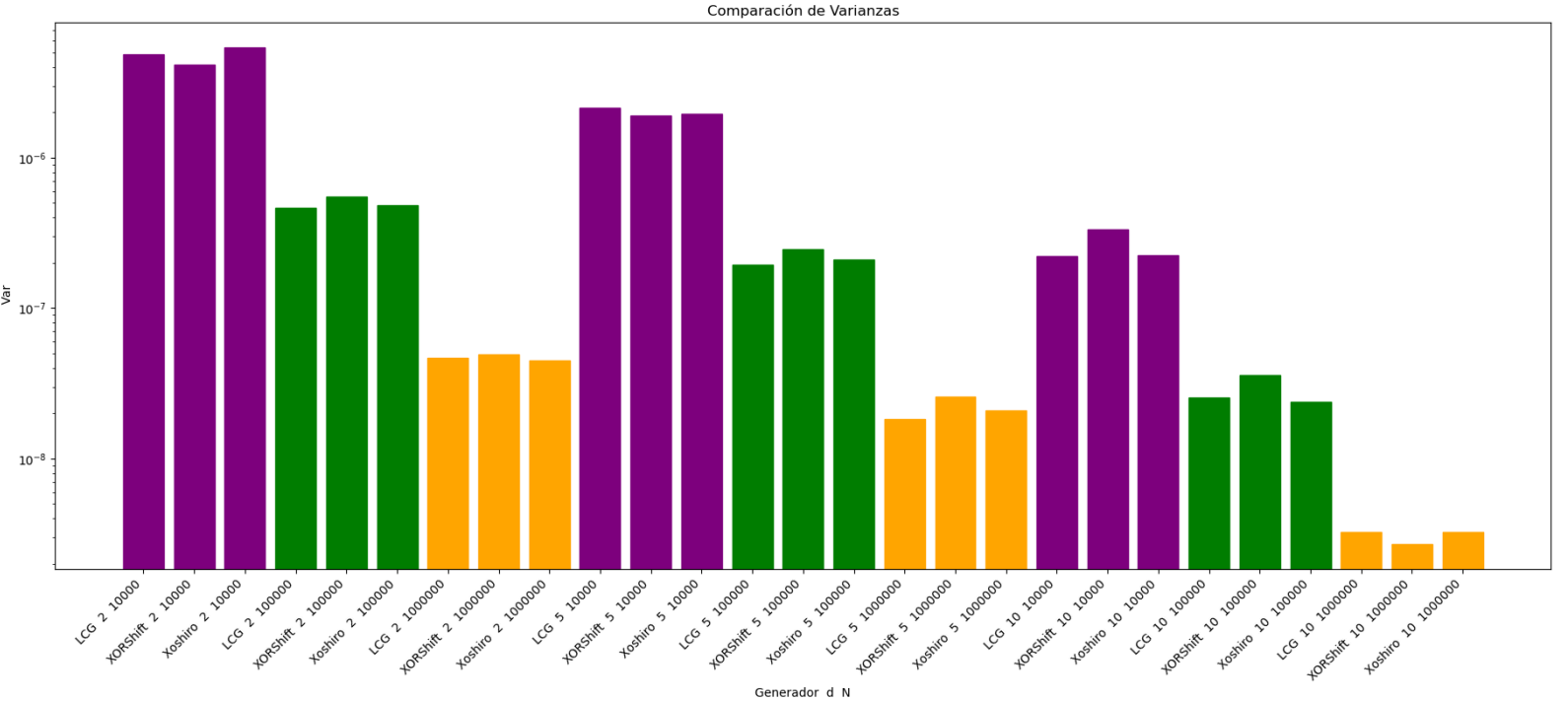
- El error cuadrático medio de la estimación respecto al valor teórico.
- La varianza de las estimaciones.
- El tiempo de ejecución de cada generador.
- La estabilidad de los resultados en función de la dimensión y del tamaño de la muestra.

Resultados Obtenidos

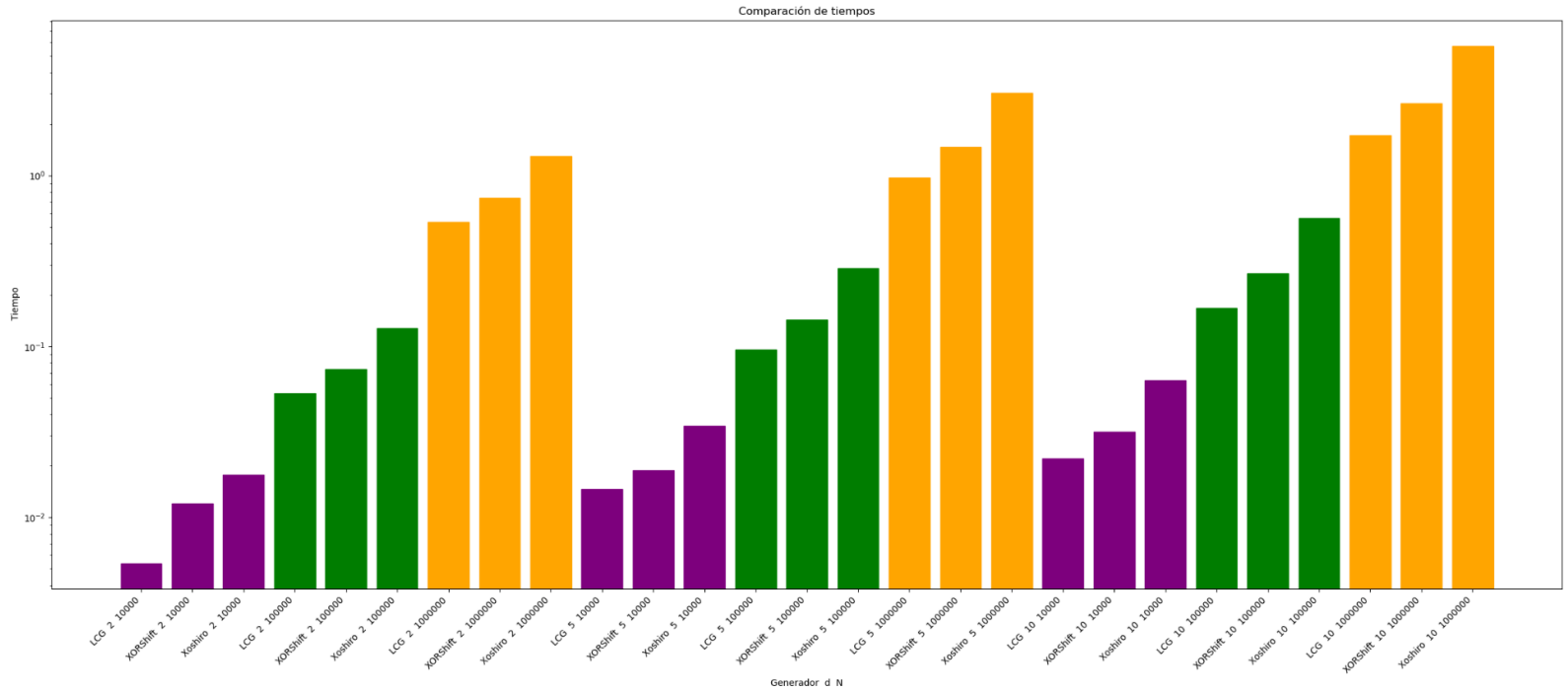
A continuación se encuentra el gráfico del **Error Cuadrático Medio** estimado del estimador con respecto al valor teórico de la integral. Como se puede ver, el **ECM** disminuye al aumentar el tamaño de la muestra usada en Monte Carlo, lo cual es de esperar y significa que al aumentar el **N** mejora la estimación. Lo interesante es que no existe ningún patrón que indique que algunos generadores sean mejores que otros a la hora de estimar la integral.



El siguiente gráfico nos muestra la **varianza muestral** del estimador, y se puede ver que es casi idéntica al ECM del gráfico anterior. Esto se debe a que el estimador de la integral es insesgado (y por lo tanto ECM = Var).



Como las mediciones anteriores no dieron resultados relevantes en cuanto a la calidad de cada generador, decidimos analizar el tiempo de ejecución para cada uno de ellos y los resultados obtenidos se pueden ver en el siguiente gráfico. En él se observa que el **LCG** tiene el mejor rendimiento, seguido del generador **XORShift** y por último el generador **Xoshiro128****.



Conclusiones

En nuestro caso, para calcular la integral mediante Monte Carlo, la calidad del generador de números aleatorios usado no tuvo ningún impacto apreciable en la estimación de la integral, por lo que la elección del mismo debería basarse principalmente en su velocidad de cómputo. En ese sentido, el mejor fue el **LCG**.

Sin embargo, por el teorema de Marsaglia[1], es comúnmente desaconsejado utilizar generadores congruenciales lineales si lo que se busca es una aproximación de alta calidad de una integral utilizando Monte Carlo debido al problema de los hiperplanos mencionado anteriormente. Por lo tanto, consideramos importante revisar si es necesario un generador de alta calidad estadística al estimar una integral, o si uno más simple pero más rápido es suficiente para lo que se busca (como en este caso).

Quizás si tomáramos un valor de **d** para el cuál los puntos generados caigan en menos hiperplanos, y/o un tamaño de muestra **N** más grande, entonces el problema de los hiperplanos se manifestaría y veríamos que utilizando el generador congruencial la integral no converge al valor exacto.

Referencias

1. https://en.wikipedia.org/wiki/Marsaglia%27s_theorem
2. https://en.wikipedia.org/wiki/TestU01, https://en.wikipedia.org/wiki/Diehard_tests
3. https://www.jstatsoft.org/article/view/v008i14/916
4. https://www.firstpr.com.au/dsp/rand31/p1192-park.pdf