Comparación entre generadores de números pseudoaleatorios calculando una integral múltiple

Gabriel Guimpelevich Agustín Farace

Junio 2025

1. Introducción

Este es un trabajo práctico en el que se busca estudiar la efectividad de varios métodos de generación de números pseudoaleatorios al compararlos con análisis estadístico y aplicarlos a un problema concreto de simulación.

1.1. Generadores usados

Los generadores elegidos son:

- Generador Congruencial Lineal (LCG) con parámetros:
  - $a = 16807$
  - $c = 0$
  - $m = 2^{31} - 1$
- XORShift
- Mersenne Twister MT19937

1.2. Problema a simular

Se desea estimar, mediante el método de Monte Carlo, el valor de la siguiente integral múltiple:

$$\int_{[0,1]^5} \left(\sum_{i=1}^{5} x_i \right)^2 dx_1 ... dx_5$$

Esta integral representa el valor esperado de la suma al cuadrado de cinco variables independientes uniformemente distribuidas en $[0, 1]$.
Para referencia, el valor teórico de la integral es $= \frac{20}{3}$.

2. Generadores

2.1. Generador Congruencial Lineal (LCG)

El más simple y antiguo de los algoritmos para generar números pseudoaleatorios es el LCG. Funciona aplicando una fórmula recursiva partiendo desde una *semilla* (X_0) :

$$X_{n+1} = (\ a \cdot X_n + c \ ) \quad mod \ m$$

Con un *multiplicador* (a), un *incremento* (c) y un *módulo* (m) para generar una secuencia de números los cuales son los que se tomarán (o una transformación de ellos para que se ajusten al intervalo deseado).

En nuestro caso se dice que es un LCG Multiplicativo porque $c = 0$, esto significa que la fórmula pasa a ser:

$$X_{n+1} = (\ a \cdot X_n \ ) \quad mod \ m$$

Este generador es fácil de implementar, eficiente, y tiene un *período* (la cantidad de números generados antes de que el patrón se repita) largo. El período tiene un límite teórico de $2^{31} - 1$, el cual es alcanzado si la semilla X_0 es coprima con m y $a - 1$ es divisible por todos los factores primos de m (el cual en nuestro caso es primo, así que sería sólo él).

Hablando de nuestro caso, los parámetros usados en nuestro LCG fueron dados por Stephen K. Park y Keith W. Miller en 1988, en donde el m es un *primo de Mersenne* y a es una raíz primitiva de m.

Este tipo de generador tiene problemas para simulaciones en varias dimensiones porque se detectan correlaciones (hiperplanos) como muestra la Figura 1. Si un LCG es usado para elegir puntos en un espacio n-dimensional, los puntos van a estar puestos en, como máximo, $m^{1/n}$ hiperplanos (Teorema de Marsaglia). Esto es por la correlacion serial entre valores sucesivos de la secuencia X_n.

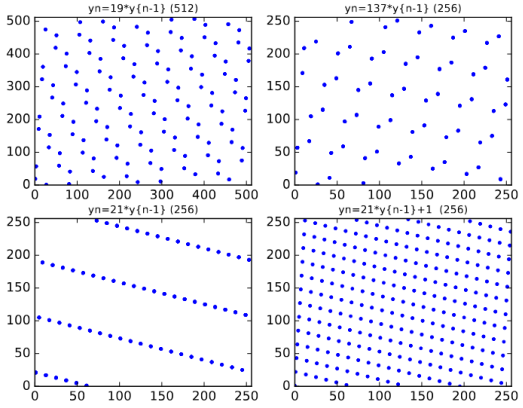

Figura 1: Problema de los hiperplanos

Además, también depende mucho de la semilla utilizada para funcionar a todo su potencial; incluso si es bien elegida, el período es predecible pasado cierto punto.

2.2. XORShift

El generador XORShift, propuesto por George Marsaglia en 2003, es un algoritmo de generación de números pseudoaleatorios basado en operaciones bit a bit, específicamente desplazamientos o *shifts* y operaciones XOR (OR exclusivo).

El algoritmo XORShift genera una secuencia de periodo $2^{32}-1$ o una secuencia de $2^{64}-1$ pares x,y, o una secuencia de $2^{96}-1$ triplas x,y,z, etc. a partir

de una semilla inicial X_0 utilizando una combinación de desplazamientos a la izquierda $\ll$, desplazamientos a la derecha $\gg$ y operaciones XOR $\oplus$. Por ejemplo, una implementación común de XORShift es de esta forma:

$$X_{n+1} = X_n \oplus (X_n \ll a) \oplus (X_n \gg b) \oplus (X_n \ll c)$$

Donde X_n es un entero sin signo y a, b, c son la cantidad de bits a desplazar en la dirección correspondiente.
Este algoritmo es muy rápido en arquitecturas modernas debido a que usa estas operaciones en bits. Aunque menos que el LCG, también puede tener correlaciones notables en simulaciones de varias dimensiones.

2.3. Mersenne Twister MT19937

El Mersenne twister es un Generador de números pseudoaleatorios desarrollado en 1997 por Makoto Matsumoto y Takuji Nishimura, reputado por su calidad.

Su nombre proviene del hecho de que la longitud del periodo corresponde a un Número primo de Mersenne, es decir, un número de la forma

$$M_p = 2^p - 1$$

donde p es un número primo. Existen al menos dos variantes de este algoritmo, distinguiéndose únicamente en el tamaño de primos Mersenne utilizados. El más reciente y más utilizado es el Mersenne Twister MT19937, con un tamaño de palabra w de 32 bits, que es la versión que usaremos. Existe otra variante con tamaño de palabra w de 64 bits, el MT19937-64, la cual genera otra secuencia.

El algoritmo comienza inicializando un estado interno (al que llamaremos mt) de $n = 624$ enteros de 32 bits. El estado es la información interna que tiene el generador para producir la próxima salida aleatoria. Este estado se inicializa **una única vez** mediante una semilla *seed* y una formula recursiva:

$$mt[0] = seed$$
$$mt[i] = (f * (\, mt[i-1] \oplus (\, mt[i-1] \gg (w-2))) + i) \mod 2^w, \quad para\ i = 1, ..., 623$$

Donde:

- $f = 1812433253$: Constante de inicialización
- $w = 32$: Tamaño de palabra en bits
- $\gg$: Desplazamiento de bits a la derecha
- $\oplus$: XOR bit a bit

Una vez inicializado el estado, se establece el *indice* del estado en 624, forzando así un *twist* y volviendo a poner el *indice* en 0. Este proceso toma combinaciones de los valores actuales del estado y genera 624 nuevos valores, reemplazando completamente el arreglo mt. Para cada indice i del estado:

$$x = (mt[i] \ \& \ UPPER\_MASK) + (mt[(i+1) \ mod \ n] \ \& \ LOWER\_MASK)$$

$$xA = x \gg 1$$

$$si \ (x \ mod \ 2 \neq 0) \Rightarrow xA = xA \oplus a$$

$$mt[i] = mt[(i+m) \ mod \ n] \oplus xA$$

Donde:

- $a =$ 0x9908B0DF (una constante mágica usada por MT19937)
- $m = 397$ (esta constante representa el desplazamiento del valor que se mezcla con $mt[i]$)
- $UPPER\_MASK$ y $LOWER\_MASK$ dividen el número de 32 bits en dos partes: los bits superiores e inferiores.

Esto logra que el generador tenga un período, es decir la longitud máxima de la secuencia de números que puede generar antes de que comience a repetirse, de $2^{19937} - 1$

Al tomar un valor indicado por el *indice* (llamemoslo y) se produce el proceso de *tempering*, este proceso transforma el valor del estado antes de devolverlo como salida, con el objetivo de garantizar una distribución uniforme a nivel de bits. El algoritmo hace lo siguiente:

$$y = y \oplus (y \gg u)$$

$$y = y \oplus ((y \ll s) \& b)$$

$$y = y \oplus ((y \ll t) \& c)$$
$$y = y \oplus (y \gg l)$$

Donde:

- $u = 11$ (desplazamiento a la derecha inicial)
- $s = 7$ (desplazamiento a la izquierda 1)
- $t = 15$ (desplazamiento a la izquierda 2)
- $l = 18$ (desplazamiento a la derecha final)
- $b =$ 0x9D2C5680 (Máscara para el paso con s)
- $c =$ 0xEFC60000 (Máscara para el paso con t)

En conclusión, primero se inicializa una única vez el arreglo mt y se realiza un *twist* sobre ese arreglo. Luego se extraen números de acuerdo al *indice*. El número que se extrae pasa por el proceso de *tempering*. Cuando el *indice* llega a 624 se vuelve a producir un *twist*.

El Mersenne Twister MT19937 es uno de los generadores de números pseudoaleatorios más utilizados debido a su alta calidad estadística y rendimiento. Algunas de las características más importantes son su período extremadamente largo $(2^{19937} - 1)$ lo que garantiza que la secuencia de números no se repita sino hasta después de generar una cantidad extremadamente grande de valores. También es muy eficiente computacionalmente en comparación con otros generadores de calidad similar ya que las operaciones que realiza son a nivel de bits y son muy rápidas a nivel de hardware.

Algunas desventajas de este generador son que tiene un estado interno grande, requiere 624 enteros de 32 bits, lo cual implica un uso de memoria mayor (aproximadamente 2.5 KB) en comparación con generadores más simples. También tiene el overhead que implica la inicialización antes de poder obtener el primer valor, lo cual puede ser un problema si se requiere reinicializar frecuentemente.

3. El problema

Se va a aproximar por simulaciones la siguiente integral:

$$\int_{[0,1]^5} \left(\sum_{i=1}^{5} x_i \right)^2 dx_1 ... dx_5$$

Pero primero vamos a calcular su valor real de forma teórica para luego compararlo con las simulaciones.

3.1. Resolución de la integral

Consideremos la integral múltiple:

$$\int_{[0,1]^5} (x_1 + x_2 + x_3 + x_4 + x_5)^2 \, dx_1 \, dx_2 \, dx_3 \, dx_4 \, dx_5$$

donde la integración se realiza sobre el hipercubo unitario $[0,1]^5$, es decir, cada variable x_i (con $i = 1, 2, 3, 4, 5$) varía independientemente de 0 a 1.

Primero expandimos $(x_1 + x_2 + x_3 + x_4 + x_5)^2$:

$$= x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 + 2x_1x_2 + 2x_1x_3 + 2x_1x_4 + 2x_1x_5$$

$$+2x_2x_3 + 2x_2x_4 + 2x_2x_5 + 2x_3x_4 + 2x_3x_5 + 2x_4x_5$$

Esto incluye 5 términos cuadrados (x_i^2) y 10 términos con pares ($2x_ix_j$ para $i < j$).

Vamos a integrar cada término por separado.

Para x_1^2:

$$\int_0^1 x_1^2 \, dx_1 = \left[\frac{x_1^3}{3} \right]_0^1 = \frac{1}{3}$$

Luego queda:

$$= \frac{1}{3} \cdot \int_0^1 dx_2 \cdot \int_0^1 dx_3 \cdot \int_0^1 dx_4 \cdot \int_0^1 dx_5$$

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 x_1^2 \, dx_2 \, dx_3 \, dx_4 \, dx_5 \, dx_1 = \frac{1}{3} \cdot 1 \cdot 1 \cdot 1 \cdot 1 = \frac{1}{3}$$

Entonces cada x_i^2 (con $i = 2, 3, 4, 5$) da el mismo resultado. Como son 5 términos de $\frac{1}{3}$ entonces:

$$\boxed{\text{Términos cuadrados} = 5 \cdot \frac{1}{3} = \frac{5}{3}}$$

Para $2x_1x_2$:

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 2x_1x_2 \, dx_3 \, dx_4 \, dx_5 \, dx_2 \, dx_1$$

$$= 2 \int_0^1 x_1 \, dx_1 \int_0^1 x_2 \, dx_2 \int_0^1 1 \, dx_3 \int_0^1 1 \, dx_4 \int_0^1 1 \, dx_5$$

Y

$$\int_0^1 x_1 \, dx_1 = \int_0^1 x_2 \, dx_2 = \left[\frac{x_2^2}{2} \right]_0^1 = \frac{1}{2}$$

Entonces:

$$2 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 = \frac{1}{2}$$

Integrando con respecto a x_1 primero:

$$\int_0^1 2x_1x_2 \, dx_1 = 2x_2 \int_0^1 x_1 \, dx_1 = 2x_2 \left[\frac{x_1^2}{2} \right]_0^1 = 2x_2 \cdot \frac{1}{2} = x_2$$

Luego, integrando con respecto a x_2:

$$\int_0^1 x_2 \, dx_2 = \left[\frac{x_2^2}{2} \right]_0^1 = \frac{1}{2}$$

Hay 10 términos así, por lo que:

$$\boxed{\text{Términos de pares} = 10 \cdot \frac{1}{2} = 5}$$

La integral total entonces queda:

$$\int_{[0,1]^5} (x_1 + x_2 + x_3 + x_4 + x_5)^2 \, dx_1 \, dx_2 \, dx_3 \, dx_4 \, dx_5$$

$$= \frac{5}{3} + 10 = \frac{5}{3} + 5 = \boxed{\frac{20}{3} \approx 6{,}6667}$$

4. Metodología

Para probar los diferentes generadores aproximando esta integral usaremos Python y librerías como numpy, scipy y matplotlib para cualquier ayuda que necesitemos durante la implementación.

Luego de implementar los generadores, vemos que los tres tienen una distribución uniforme generando 10^8 números aleatorios para cada generador:

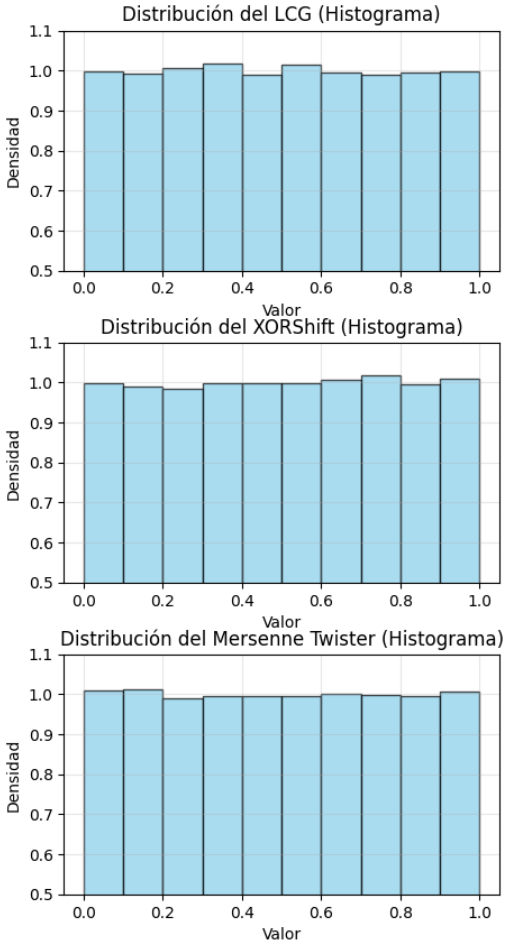

Figura 2: Densidad de los números generados con cada generador

Pero queremos ver qué diferencias tienen con respecto a la simulación de

la integral. Para esto vamos a hacer simulaciones de Monte Carlo.

Todos los generadores usan *yield* así que primero los inicializamos y luego cada vez que necesitemos un número aleatorio llamamos a $next(gen)$ donde gen es el generador que queremos usar.

Luego hacemos como mínimo 100 iteraciones, parando cuando el número de iteraciones llegue a un límite definido $Nsim$. Cada Monte Carlo va a calcular los siguientes valores para guardarlos al final de sus iteraciones:

- n = Número de simulaciones realizadas.
- $media$ = La media o el valor estimado de la integral.
- $varianza$ = La varianza actual de la muestra.
- $semiancho$ = El semiancho de un intervalo de confianza del 95 %.
- ecm = El error cuadrático medio de la simulación.

La media y la varianza se van calculando recursivamente a medida que avanza la simulación.
Aquí dejo un bloque con el código, pero si se quiere ver más de la implementación se puede chequear el notebook con esta.

```
    def montecarlo(f, Nsim, generator, data_storage):
    start_time = time.time()
    z = norm.ppf(1 - 0.025)  # Para un IC 95%
    media = f(generator)
    ecm = (media-20/3)**2
    SS, n = 0, 1
    while (n < 100  or n < Nsim):
        n += 1
        media_ant = media
        ecm += (media - 20/3)**2
        media = media_ant + (f(generator) - media_ant) / n
        SS = SS * (1 - 1 / (n - 1)) + n * (media - media_ant)
    ** 2

        #if (not (n % 250000) or n == 2):
            #printero(n, media, SS / (n - 1), z * sqrt(SS / n
   ))
    end_time = time.time()
    tiempo = end_time - start_time
```

```
18
19      varianza = SS / (n - 1)
20      semiancho = z * sqrt(SS / n)
21      data_storage.add_simulation(n, media, varianza, semiancho
        , tiempo, ecm/n)
22      return None
23
```

Listing 1: Implementación del Monte Carlo

Primero vamos a setear una lista de $Nsim$'s: [100, 1000, 10000, 100000]. Entonces para cada $Nsim$ vamos a estimar la integral usando Monte Carlo con $Nsim$ iteraciones para cada generador con una repetición de 100 veces por cada $Nsim$ y a eso le sacamos el promedio. Además, también vamos a tomar su tiempo de cómputo. Cada generador va a tener seteada una semilla para que el entorno sea más controlado y reproducible.

5. Resultados

5.1. 1 repetición por $Nsim$

Los resultados que obtuvimos fueron los siguientes:

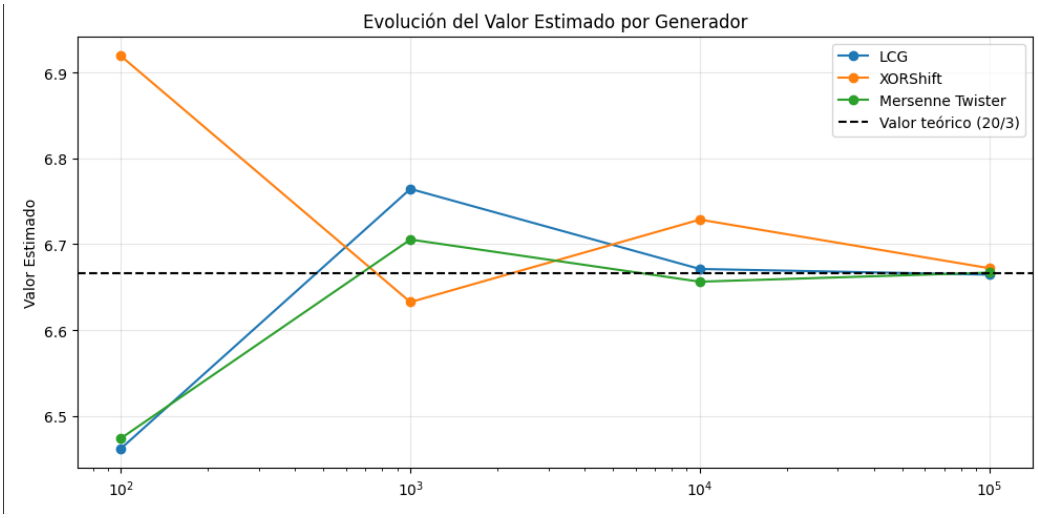

Figura 3: Valor estimado por generador

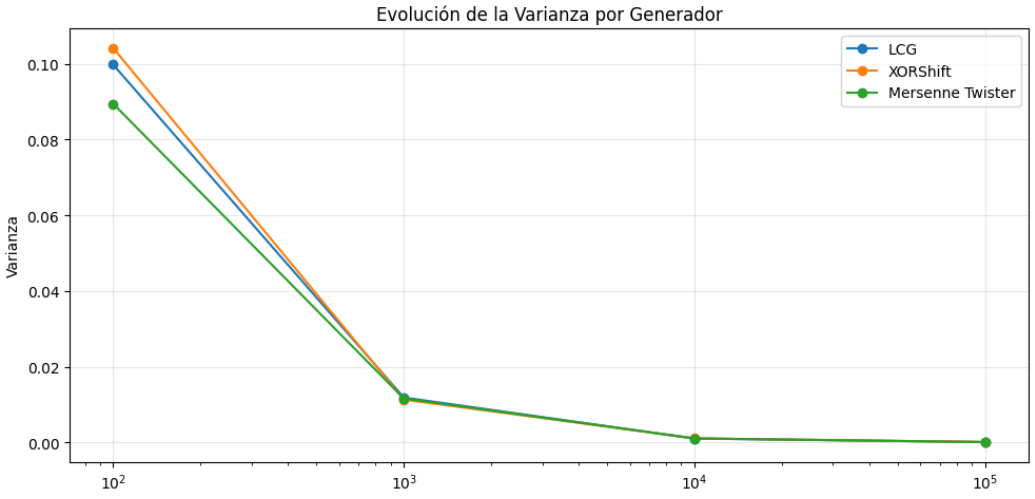

Figura 4: Evolución de la varianza por generador

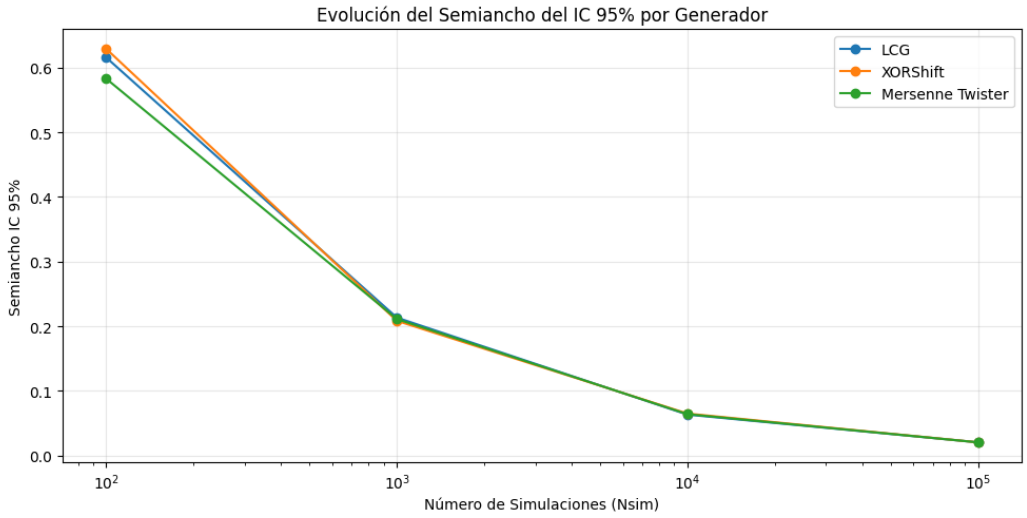

Figura 5: Evolución del Semiancho del IC 95 % por generador

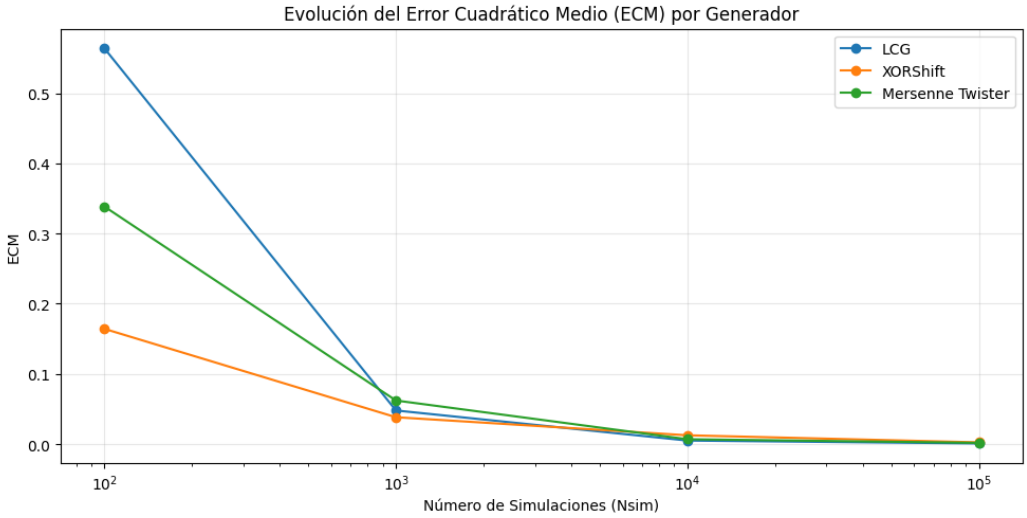

Figura 6: Evolución del ECM por generador

Luego, para cada generador:

Nsim	**Media**	**Varianza**	**Semiancho IC 95 %**	**ECM**	**Tiempo (s)**
100	6.4622	0.0997	0.6159	0.5638	0.0015
1000	6.7644	0.0119	0.2134	0.0479	0.0029
10000	6.6712	0.0010	0.0633	0.0050	0.0274
100000	6.6646	0.0001	0.0204	0.0009	0.3052

Cuadro 1: Resultados para el generador LCG

Nsim	**Media**	**Varianza**	**Semiancho IC 95 %**	**ECM**	**Tiempo (s)**
100	6.9191	0.1041	0.6292	0.1642	0.0007
1000	6.6326	0.0113	0.2084	0.0383	0.0037
10000	6.7287	0.0011	0.0649	0.0126	0.0397
100000	6.6721	0.0001	0.0203	0.0025	0.3535

Cuadro 2: Resultados para el generador XORshift

En la *Figura* 3 podemos observar que el generador *Mersenne Twister* converge más rápidamente al valor teórico a medida que aumenta el número de simulaciones. $XORshift$ tiene un comportamiento más inestable pero se

Nsim	Media	Varianza	Semiancho IC 95 %	ECM	Tiempo (s)
100	6.4741	0.0894	0.5831	0.3385	0.0011
1000	6,7054	0.0116	0.2111	0.0621	0.0088
10000	6.6564	0.0010	0.0640	0.0070	0.0780
100000	6.6669	0.0001	0.0203	0.0018	0.7937

Cuadro 3: Resultados para el generador Mersenne Twister

termina aproximando al valor teórico. LCG es más estable pero requiere más simulaciones que $Mersenne\ Twister$ para aproximarse al valor teórico.

En las $Figuras\ 4\ y\ 5$, todos los generadores disminuyen su varianza y semiancho con el aumento de simulaciones. $Mersenne\ Twister$ muestra menor varianza y semiancho en todos los niveles de simulación, lo que indica mayor precisión y menor dispersión en sus estimaciones.

En la $Figura\ 6$ vemos que en las primeras simulaciones LCG tiene un ECM muy alto, lo que indica una desviación alta del valor teórico. $Mersenne\ Twister$ está mejor en este aspecto pero el mejor es $XORshift$. A partir de las 1000 simulaciones todos los generadores mejoran notablemente.

En cuanto a la velocidad, tanto LCG como $XORshift$ tienen velocidades similares mientras que $Mersenne\ Twister$ es notablemente más lento. Esto se debe a las operaciones que tiene que realizar tanto para inicializar como para actualizar su estado interno.

5.2. 100 repeticiones por $Nsim$

Ahora si hacemos 100 repeticiones por cada $Nsim$ y vemos cual es el promedio entre esas 100:

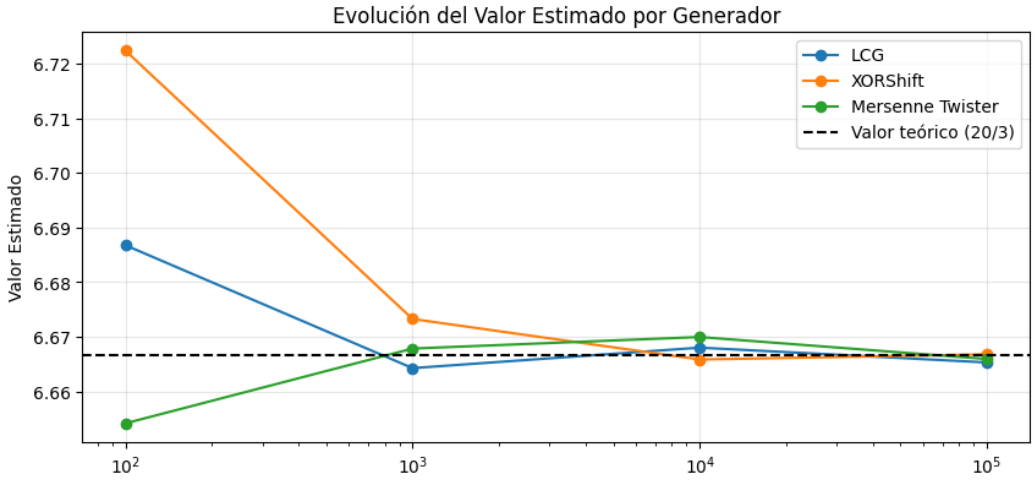

Figura 7: Evolución del valor estimado con 100 repeticiones de cada $Nsim$

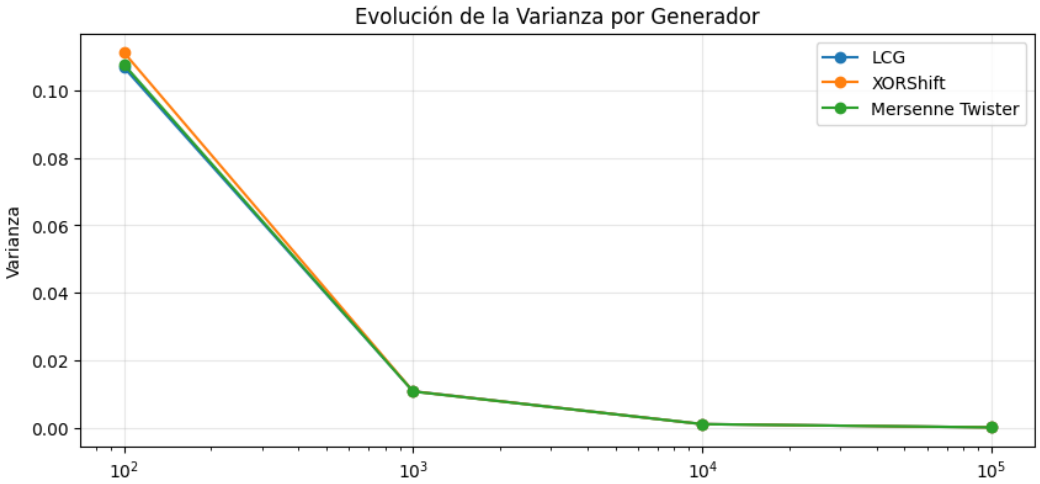

Figura 8: Evolución de la varianza con 100 repeticiones de cada $Nsim$

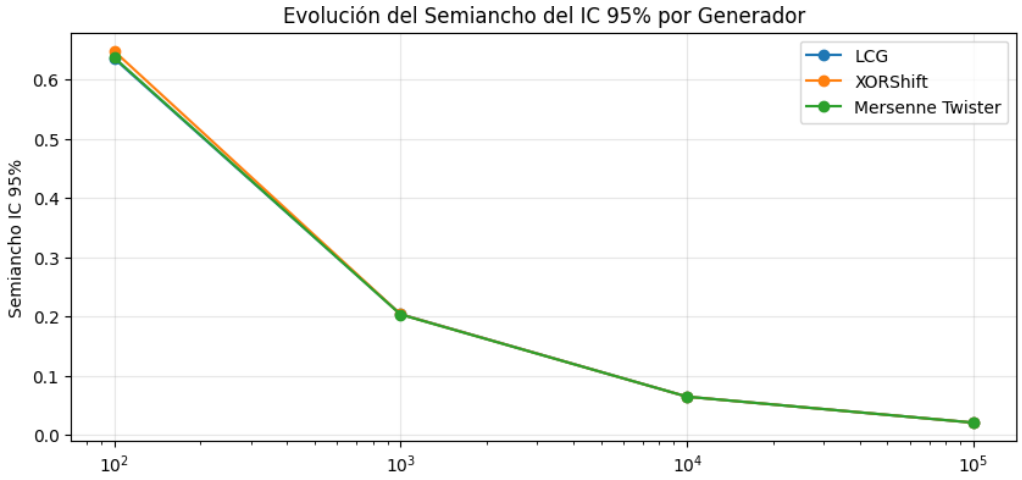

Figura 9: Evolución del semiancho del intervalo de confianza de 95 % con 100 repeticiones de cada $Nsim$

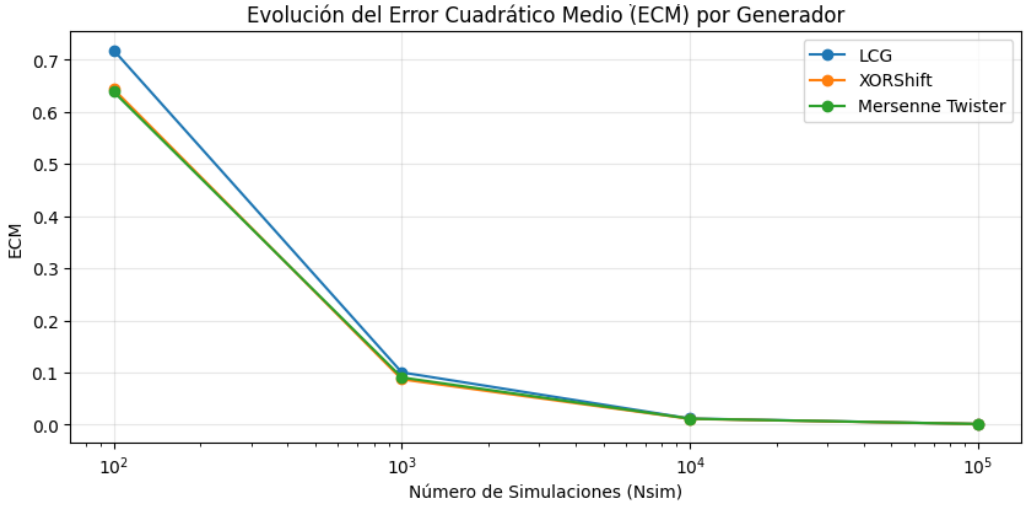

Figura 10: Evolución del ECM con 100 repeticiones de cada $Nsim$

Nsim	Media	Varianza	Semiancho IC 95 %	ECM	Tiempo (s)
100	6.686752	0.10682083	0.635675	0.718448	0.000765
1000	6.664248	0.01081140	0.203635	0.100463	0.004086
10000	6.667999	0.00107393	0.064225	0.011912	0.028391
100000	6.665285	0.00010719	0.020292	0.001378	0.284186

Cuadro 4: Resultados para el generador LCG

Nsim	Media	Varianza	Semiancho IC 95 %	ECM	Tiempo (s)
100	6.722395	0.11126286	0.648222	0.643773	0.000873
1000	6.673240	0.01078705	0.203400	0.087169	0.005401
10000	6.665825	0.00107322	0.064204	0.011336	0.038981
100000	6.666768	0.00010725	0.020298	0.001436	0.393723

Cuadro 5: Resultados para el generador XORShift

Nsim	Media	Varianza	Semiancho IC 95 %	ECM	Tiempo (s)
100	6.654133	0.10772641	0.637286	0.638760	0.001631
1000	6.667831	0.01074718	0.203026	0.090681	0.011161
10000	6.669946	0.00107315	0.064201	0.011641	0.086819
100000	6.665912	0.00010724	0.020297	0.001513	0.838074

Cuadro 6: Resultados para el generador Mersenne Twister

El MT y XOR están muy cerca en términos de precisión pero el primero tarda más y tiene ligeramente menos varianza.

6. Conclusiones

En este trabajo se compararon tres generadores de números pseudoaleatorios, *LCG*, *XORShift* y *Mersenne Twister*, mediante la estimación de una integral múltiple utilizando el método de Monte Carlo. Se evaluaron métricas clave como la media estimada, varianza, semiancho del intervalo de confianza, error cuadrático medio (ECM) y tiempo de ejecución.

Los resultados muestran que, si bien los tres generadores convergen hacia el valor teórico a medida que aumenta el número de simulaciones, existen diferencias que pueden determinar qué generador usar en función del contexto y a las prioridades del experimento.

LCG es el generador más rápido pero es menos preciso en simulaciones más pequeñas; sin embargo, se vuelve más preciso a medida que aumenta el número de simulaciones.

XORShift logró un buen equilibrio entre velocidad y precisión. Su ECM fue bajo desde etapas tempranas, aunque mostró una leve inestabilidad al aumentar el número de simulaciones. Es una opción adecuada cuando se requiere rapidez sin comprometer demasiado la calidad de los resultados.

Mersenne Twister ofreció la mayor estabilidad y precisión a lo largo de las simulaciones, con una rápida convergencia hacia el valor teórico y un comportamiento suave en el ECM. Aunque su tiempo de ejecución fue mayor al de los otros generadores, esto se ve compensado por la calidad de los resultados, como una menor varianza en la estimación.

En conclusión, si el objetivo es minimizar el tiempo de cómputo, LCG es preferible; si se busca un balance entre eficiencia y exactitud, XORShift es una buena alternativa; y si se prioriza la calidad estadística de las simulaciones, Mersenne Twister es la opción más adecuada. Especialmente hablando en los resultados de 100 repeticiones, el *Mersenne Twister* tiene ligeramente menor varianza que el *XORShift*.

También hay que notar que aunque se hicieron muchas simulaciones, no se llegó ni cerca del límite del período del *Mersenne Twister*. Por lo que si lo que se busca es simulaciones masivas, el *Mersenne Twister* es el mejor a cambio de tardar un poco más.