

Comparación de Generadores Pseudoaleatorios mediante el Método de Monte Carlo

Integrantes:

- ARIAS, Eliana
- GUTIÉRREZ, Camilo

Fecha de entrega: 15 de junio 2025

Resumen

Este informe analiza y compara el desempeño de distintos generadores de números pseudoaleatorios aplicados a la estimación de una integral definida mediante el método de Monte Carlo. Se utilizan generadores clásicos como LCG y variantes más modernas como Xorshift y Xoshiro. Se evalúan su eficiencia, precisión, estabilidad y varianza obtenida en las simulaciones.

1. Descripción del problema de simulación

Se desea estimar la siguiente integral múltiple sobre el hipercubo $[0, 1]^d$:

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

Esta integral tiene una solución teórica conocida basada en el producto de funciones error:

$$I_d = \left(\frac{\sqrt{\pi} \cdot \text{erf}(1)}{2} \right)^d$$

Se utiliza el método de Monte Carlo para estimar su valor con diferentes generadores y distintas combinaciones de dimensión y cantidad de muestras.

2. Generadores estudiados

2.1 LCG (Linear Congruential Generator)

- **Fórmula:**

$$Y_{n+1} = (a \cdot Y_n) \bmod m$$

- **Parámetros usados:** $a = 16807$, $m = 2^{31} - 1$
- **Periodo:** $2^{31} - 2^{[1]}$
- **Ventajas:** Implementación sencilla, muy rápido.
- **Desventajas:** Correlaciones a largo plazo, no adecuado para aplicaciones criptográficas.

2.2 Xorshift (32, 64, 128 bits)

- Utiliza operaciones XOR y desplazamientos bit a bit^[2].
- **Xorshift32**
 - **Fórmula:**

$$y = y \oplus (y \ll a); \quad y = y \oplus (y \gg b); \quad y = y \oplus (y \ll c)$$

- **Parámetros usados:** $a = 13, b = 17, c = 5$
- **Periodo:** $2^{32} - 1$
- **Xorshift64**
 - **Fórmula:**

$$y = y \oplus (y \gg c); \quad y = y \oplus (y \ll b); \quad y = y \oplus (y \gg a)$$

- **Parámetros usados:** $a = 1, b = 13, c = 45$
- **Periodo:** $2^{64} - 1$
- **Xorshift128+**
 - **Fórmula:**

$$y = estado_0 + estado_1 \mod 2^{64}$$

- **Parámetros usados:** $a = 23, b = 18, c = 5$
- **Periodo:** $2^{128} - 1$
- **Ventajas:** Muy rápido, buena distribución estadística.
- **Desventajas:** No es criptográficamente seguro.

2.3 Xoshiro128++

- Uno de los generadores modernos más robustos; sin embargo, estamos utilizando una de las versiones menos precisas de la familia xoshiro^[3].
- **Fórmula:**

$$y = \text{rotl}(estado_0 + estado_3, r) + estado_0$$

- **Parámetros usados:** $a = 9, b = 11, r = 7$
- **Periodo:** $2^{128} - 1$
- **Ventajas:** Buena calidad, velocidad adecuada para simulaciones numéricas.
- **Desventajas:** Complejidad levemente mayor.

1. Por el Teorema 3.2. del apunte y por la verificación hecha en el [código](#)

[adjunto](#), podemos afirmar que el generador congruencial lineal tiene período máximo.

2. Las fórmulas y parámetros de los generadores xorshift32 y xorshift64 fueron elegidas de entre todas las posibles que garantizan un período máximo según [Marsaglia \(2003\)](#). Para el caso del generador xorshift128, nos basamos en la implementación de [Vigna \(2017\)](#) que garantiza período máximo.
 3. Nos guiamos de la implementación propuesta por [Blackman & Vigna\(2021\)](#) que garantiza período máximo.
-

3. Metodología

- **Lenguaje usado:** Python 3.12.3
- **Librerías:** numpy, matplotlib, math, time, ctypes, tempfile, subprocess, os, sympy, random, ipywidgets, mpl_toolkits, IPython
- **Simulación:**
Dado que el integrando es un producto de funciones que dependen **cada una de una sola variable** x_i , la integral se puede reescribir como un **producto de integrales unidimensionales**:

$$I_d = \int_{[0,1]^d} \prod_{i=1}^d e^{-x_i^2} dx = \prod_{i=1}^d \left(\int_0^1 e^{-x_i^2} dx_i \right)$$

Para estimar la integral, se hace uso del **método de Monte Carlo** que consiste en: generar N muestras $\mathbf{x}^{(k)} = (x_1^{(k)}, x_2^{(k)}, \dots, x_d^{(k)})$, para $k = 1, 2, \dots, N$ de forma uniforme, y evaluar en cada una de ellas la función:

$$f(\mathbf{x}) = \prod_{i=1}^d e^{-x_i^2}$$

Luego, la estimación de I_d es simplemente el promedio de estos valores:

$$\hat{I}_d = \frac{1}{N} \sum_{k=1}^N f(\mathbf{x}^{(k)})$$

En la implementación, esto se logra generando un valor aleatorio $x_i \sim \mathcal{U}(0, 1)$ para cada dimensión, y acumulando el producto $\prod_{i=1}^{i-1} e^{-x_i^2}$. Esta operación corresponde al siguiente fragmento de código:

```
def f(rng, d):
    prod = 1
    for _ in range(d):
        X = rng()
        prod *= exp(-X**2)
    return prod
```

Cada llamada a $f(rng, d)$ representa una muestra $\mathbf{x}^{(k)}$ y su evaluación del integrando.

Finalmente, para evaluar la eficiencia de la estimación, se repite este proceso con diferentes cantidades de muestras:

- **Muestras simuladas:**
 $N = 10^4, 10^5, 10^6$
- **Dimensiones simuladas:**
 $d = 2, 5, 10$

Y se calcula, para cada caso:

- El valor estimado \hat{I}_d
- El tiempo de ejecución
- La varianza muestral
- El error cuadrático medio respecto al valor exacto
- **Tests adicionales:** Eficiencia, Repetibilidad, Independencia y Uniformidad, Chi-cuadrado y Kolmogorov-Smirnov.

4. Resultados y Análisis

4.1 Semilla

Para obtener una semilla de buena calidad estadística, hacemos uso de la instrucción **RDSEED** la cual, dependiendo de la versión, genera un número **aleatorio** de 32 o 64 bits a partir del hardware. Suele ser bastante lenta ya que, además de recolectar entropía, realiza tests de autoverificación internos y puede incluso llegar a fallar si no hay suficiente entropía.

Como Python no permite directamente el uso de intrinsics, tuvimos que insertar código en C para poder utilizar la instrucción. Tal implementación se encuentra en el [código adjunto](#).

En lo que respecta a las simulaciones, utilizaremos las siguientes semillas obtenidas mediante la instrucción **RDSEED** :

Generador	Semilla(s)
CGL	4155788116
Xorshift32	3016030231
Xorshift64	17816317102425885533
Xorshift128+	8551759191851867145, 9037421324366481566
Xoshiro128++	608405726, 470204612, 4172113938, 3888846613

4.2 Simulación

Resultados obtenidos de las simulaciones para distintas combinaciones de dimensión y cantidad de muestras:

Generador	N	Estimación	Error	Varianza	ECM	Tiempo
LCG	10,000	0.557396	3.50e-04	4.68e-02	4.68e-02	0.01s
LCG	100,000	0.557629	1.17e-04	4.66e-02	4.66e-02	0.09s
LCG	1,000,000	0.557927	1.81e-04	4.67e-02	4.67e-02	0.82s
Xorshift32	10,000	0.560950	3.20e-03	4.67e-02	4.67e-02	0.01s
Xorshift32	100,000	0.557517	2.29e-04	4.67e-02	4.67e-02	0.11s
Xorshift32	1,000,000	0.558074	3.27e-04	4.67e-02	4.67e-02	1.08s
Xorshift64	10,000	0.562412	4.67e-03	4.67e-02	4.67e-02	0.01s
Xorshift64	100,000	0.557761	1.49e-05	4.68e-02	4.68e-02	0.14s
Xorshift64	1,000,000	0.557675	7.13e-05	4.67e-02	4.67e-02	1.28s
Xorshift128	10,000	0.557103	6.43e-04	4.75e-02	4.75e-02	0.02s
Xorshift128	100,000	0.558709	9.62e-04	4.66e-02	4.66e-02	0.16s
Xorshift128	1,000,000	0.558150	4.04e-04	4.67e-02	4.67e-02	1.41s
Xoshiro128	10,000	0.559855	2.11e-03	4.77e-02	4.77e-02	0.02s
Xoshiro128	100,000	0.559100	1.35e-03	4.69e-02	4.69e-02	0.17s
Xoshiro128	1,000,000	0.557454	2.92e-04	4.67e-02	4.67e-02	1.77s

Resultados para $d = 2$

Generador	N	Estimación	Error	Varianza	ECM	Tiempo
LCG	10,000	0.230003	2.32e-03	2.19e-02	2.19e-02	0.02s
LCG	100,000	0.231919	4.04e-04	2.24e-02	2.24e-02	0.16s
LCG	1,000,000	0.232385	6.27e-05	2.26e-02	2.26e-02	1.60s
Xorshift32	10,000	0.231253	1.07e-03	2.21e-02	2.21e-02	0.02s
Xorshift32	100,000	0.232458	1.36e-04	2.25e-02	2.25e-02	0.21s
Xorshift32	1,000,000	0.232196	1.27e-04	2.26e-02	2.26e-02	2.11s
Xorshift64	10,000	0.230702	1.62e-03	2.22e-02	2.22e-02	0.03s
Xorshift64	100,000	0.231745	5.78e-04	2.25e-02	2.25e-02	0.27s
Xorshift64	1,000,000	0.232465	1.43e-04	2.26e-02	2.26e-02	2.81s
Xorshift128	10,000	0.232701	3.78e-04	2.24e-02	2.24e-02	0.03s
Xorshift128	100,000	0.232442	1.19e-04	2.25e-02	2.25e-02	0.31s
Xorshift128	1,000,000	0.232196	1.27e-04	2.26e-02	2.26e-02	2.83s
Xoshiro128	10,000	0.232263	5.96e-05	2.26e-02	2.26e-02	0.04s
Xoshiro128	100,000	0.231895	4.27e-04	2.25e-02	2.25e-02	0.39s
Xoshiro128	1,000,000	0.232328	5.41e-06	2.26e-02	2.26e-02	3.90s

Resultados para $d = 5$

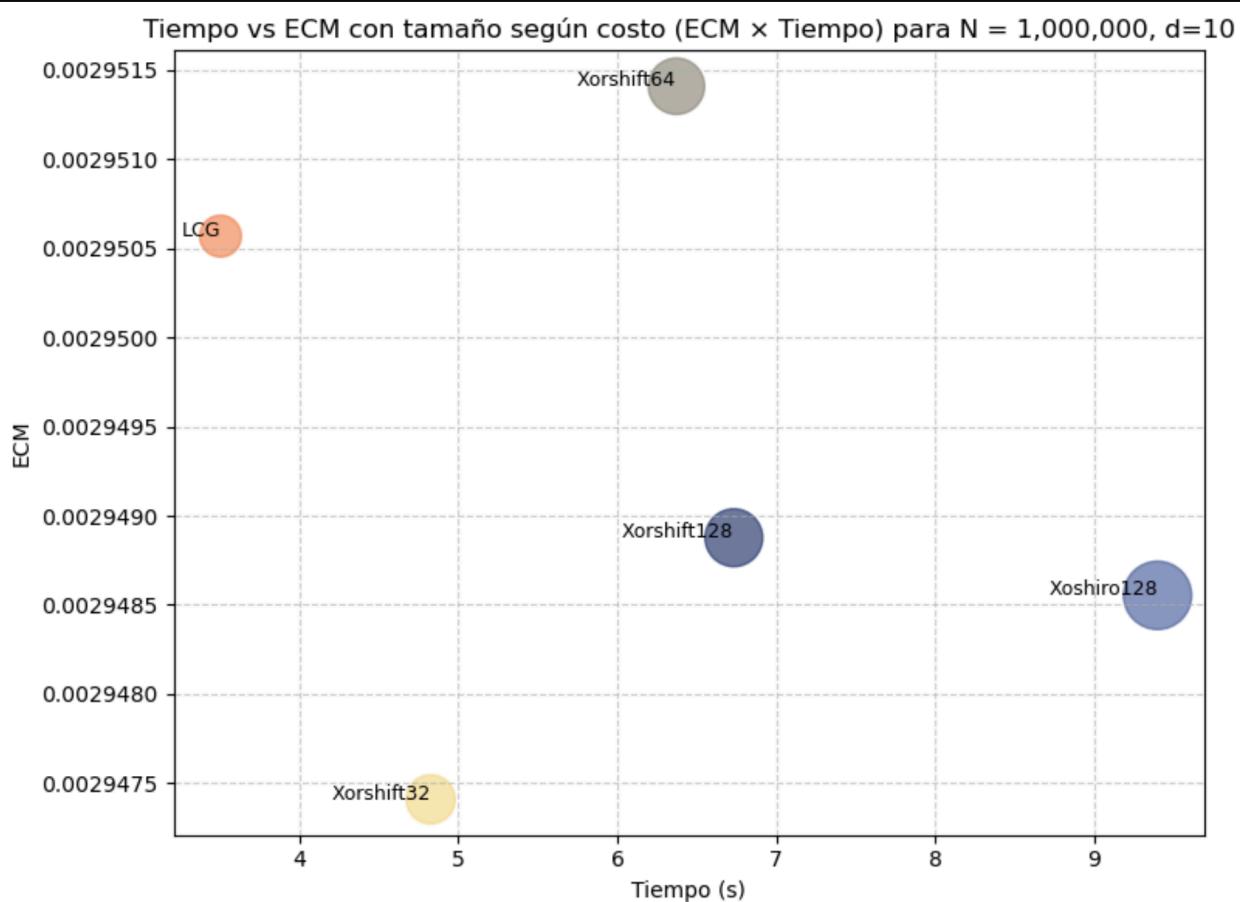
Generador	N	Estimación	Error	Varianza	ECM	Tiempo
LCG	10,000	0.054575	6.01e-04	2.96e-03	2.96e-03	0.03s
LCG	100,000	0.054014	4.05e-05	2.93e-03	2.93e-03	0.29s
LCG	1,000,000	0.053958	1.61e-05	2.95e-03	2.95e-03	2.83s
Xorshift32	10,000	0.054059	8.49e-05	3.02e-03	3.02e-03	0.04s
Xorshift32	100,000	0.054109	1.35e-04	2.97e-03	2.97e-03	0.38s
Xorshift32	1,000,000	0.054091	1.17e-04	2.96e-03	2.96e-03	3.78s
Xorshift64	10,000	0.054309	3.35e-04	3.08e-03	3.08e-03	0.05s
Xorshift64	100,000	0.053948	2.62e-05	2.93e-03	2.93e-03	0.50s
Xorshift64	1,000,000	0.053902	7.22e-05	2.94e-03	2.94e-03	5.00s
Xorshift128	10,000	0.053989	1.54e-05	2.88e-03	2.88e-03	0.05s
Xorshift128	100,000	0.053876	9.78e-05	2.91e-03	2.91e-03	0.52s
Xorshift128	1,000,000	0.053990	1.61e-05	2.94e-03	2.94e-03	5.29s
Xoshiro128	10,000	0.054081	1.07e-04	2.83e-03	2.83e-03	0.08s
Xoshiro128	100,000	0.053928	4.59e-05	2.94e-03	2.94e-03	0.77s
Xoshiro128	1,000,000	0.053998	2.45e-05	2.95e-03	2.95e-03	7.50s

Resultados para $d = 10$

4.3. Costos

Definimos el *costo* de un generador en función del producto entre el Error Cuadrático Medio y el Tiempo de Simulación.

Tomando el caso de estudio más grande ($N = 10^6$, $d = 10$), promediamos 100 simulaciones Monte Carlo para realizar siguiente gráfica:



Mientras más a la izquierda se encuentre una burbuja, más rápido es su generador asociado; y mientras más abajo se encuentre, más preciso es su generador asociado. Por lo tanto, el punto óptimo se encuentra en la esquina inferior izquierda.

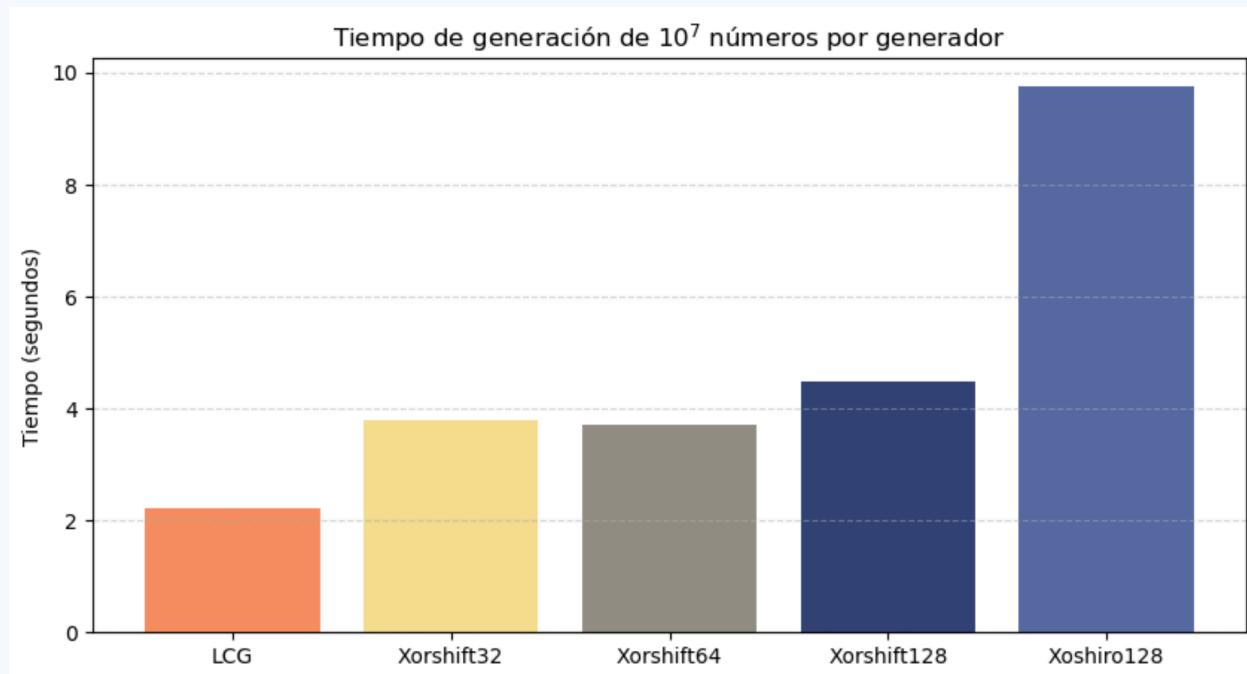
El tamaño de cada burbuja representa su costo, a mayor tamaño el generador asociado es más costoso. Por lo tanto, son preferibles las burbujas más pequeñas.

5. Tests

Se realizaron distintos tests para analizar propiedades estadísticas fundamentales de los generadores implementados

5.1 Eficiencia y Repetibilidad

Se midió el tiempo de generación de 10^7 números por generador.



El LCG resultó ser el más rápido de todos debido a la sencillez de su fórmula. Los generadores de tipo xorshift tienen demoras bastante parecidas ya que cada uno genera números pseudoaleatorios de maneras similares; notar la influencia que tienen la operación xor y la suma extra en la fórmula del xorshift128+. Por su parte, el generador xoshiro128++, cuya fórmula es la más compleja, resultó ser el más lento para generar valores como era de esperarse.

Además, se probó que todos los generadores son deterministas. Al restaurar el estado inicial (semilla), producen exactamente la misma secuencia de números. Se validó esto superponiendo dos secuencias generadas desde la misma semilla, obteniendo coincidencias perfectas en cada caso.

El test de repetibilidad evalúa si un generador de números pseudoaleatorios produce la misma secuencia cuando se inicializa con el mismo estado o semilla. Este comportamiento es fundamental para asegurar la reproducibilidad en simulaciones y experimentos numéricos.

En esta prueba, la función `test_repetibilidad` se utilizó para comprobar si, dado el mismo estado inicial, cada generador produce secuencias idénticas en dos ejecuciones consecutivas.

Para cada generador:

1. Se restauró el estado inicial con la semilla correspondiente.
2. Se generó una secuencia de números pseudoaleatorios.
3. Se restauró nuevamente el mismo estado inicial.
4. Se generó una segunda secuencia de números.
5. Se compararon ambas secuencias para verificar igualdad.

Si ambas secuencias coinciden exactamente, el generador pasa el test de repetibilidad.

Generador	Eficiencia	Repetibilidad
LCG	Muy alta	✓
Xorshift32	Alta	✓
Xorshift64	Alta	✓
Xorshift128+	Media-Alta	✓
Xoshiro128++	Baja	✓

Todos los generadores evaluados demostraron un comportamiento correcto en términos de repetibilidad, generando secuencias idénticas cuando se inicializan con la misma semilla o estado. Esto confirma que la implementación respeta el requisito fundamental de reproducibilidad.

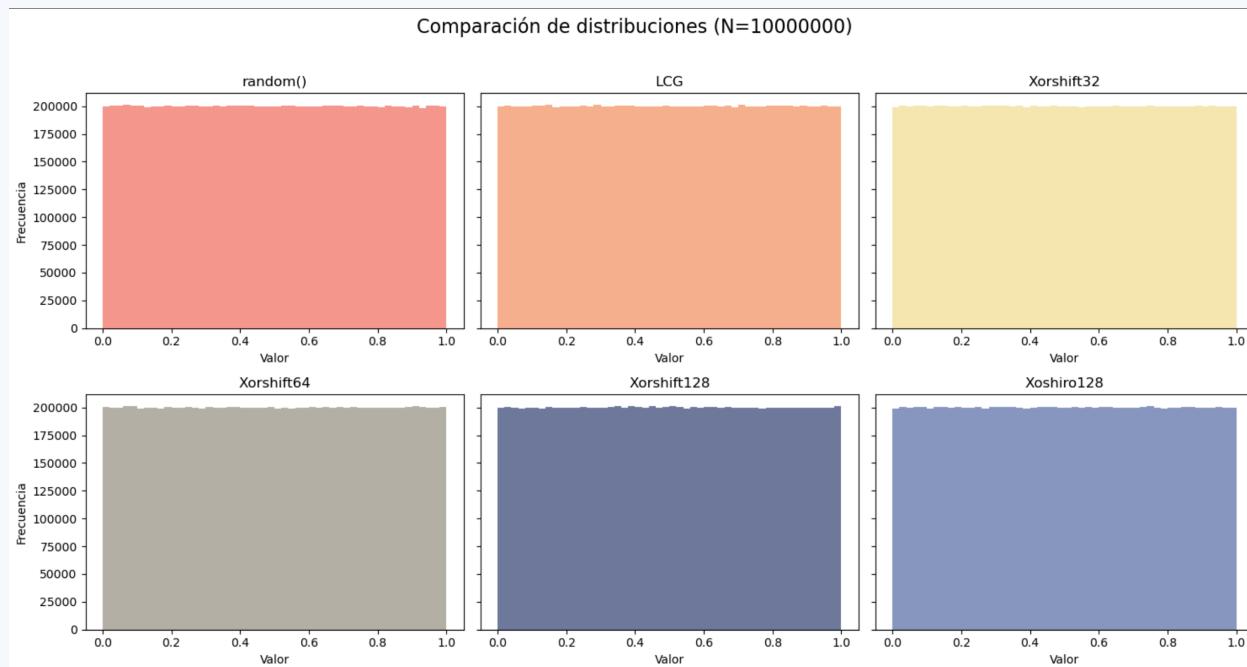
Este resultado es esperado, dado que los generadores de números pseudoaleatorios deben garantizar la misma secuencia para un mismo estado inicial, lo cual es clave para validaciones y pruebas en simulaciones numéricas y experimentos de Monte Carlo.

5.2 Evaluación Visual de Independencia y Uniformidad

El objetivo de las siguientes pruebas es detectar correlaciones no deseadas o estructuras geométricas (como alineamientos o concentración en planos) que indicarían problemas con el generador.

5.2.1 Distribución Individual

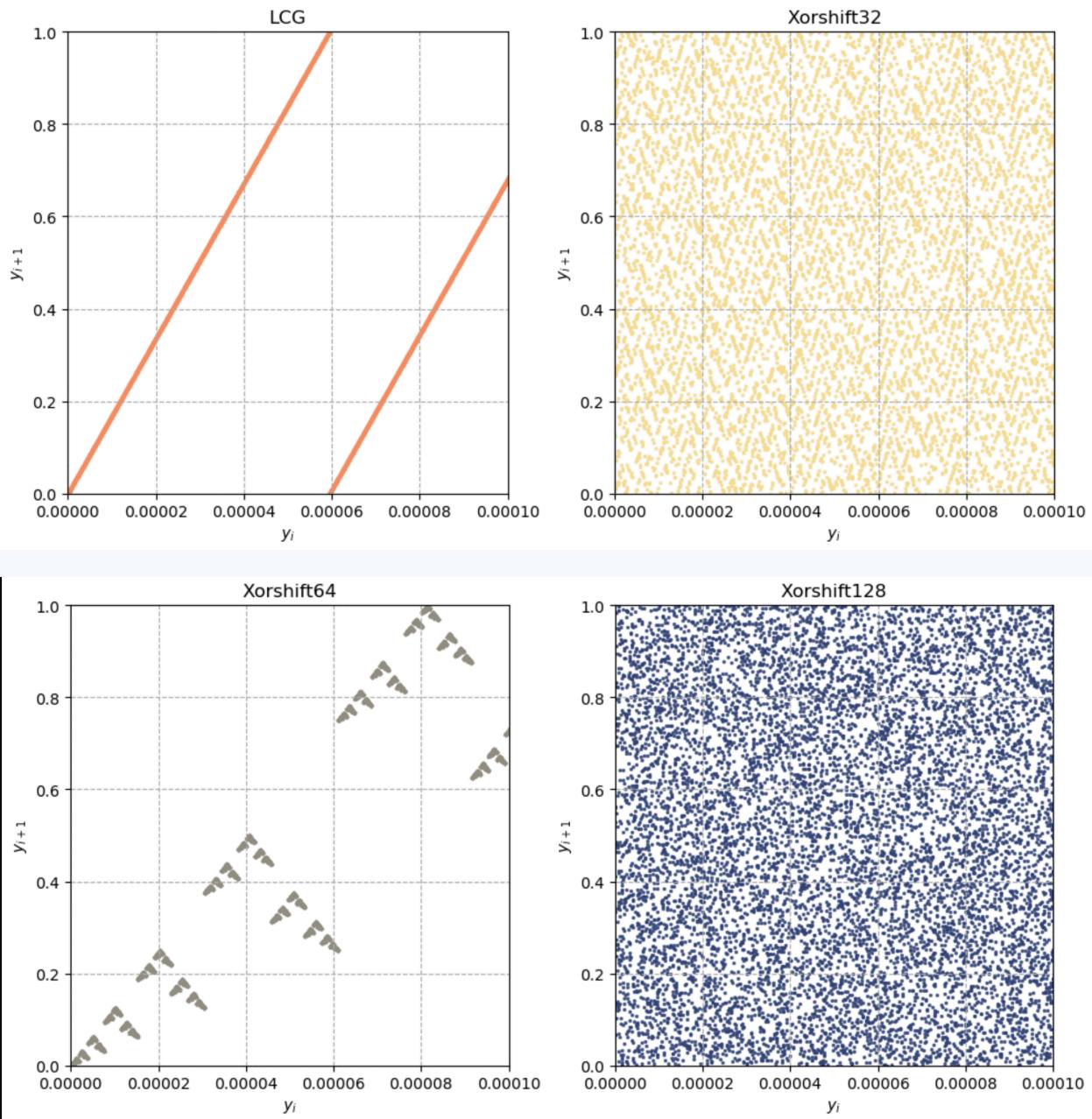
Se generaron 10 millones de muestras para cada generador normalizado en el intervalo [0,1). También se utilizó como referencia el generador de números aleatorios en el intervalo [0,1) de la librería random de Python con una semilla fija de 42, el cual se implementa internamente con el algoritmo [Mersenne Twister MT19937](#) y está ampliamente probado que tiene una alta calidad estadística.

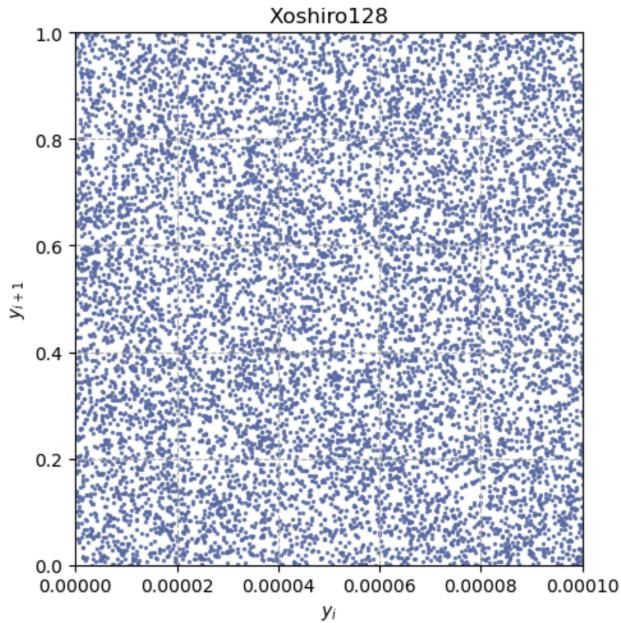


Como se observa en los histogramas, para un N grande, la distribución se aproxima a una uniforme en el intervalo (0,1), lo cual es lo que esperaríamos de un buen generador.

5.2.2 Distribución de a Pares

Se generaron 100 millones de pares del tipo (y_i, y_{i+1}) para cada generador normalizado en el intervalo $[0,1]$. Se requirió un valor elevado de pares para poder observar patrones en regiones reducidas del cuadrado unitario.

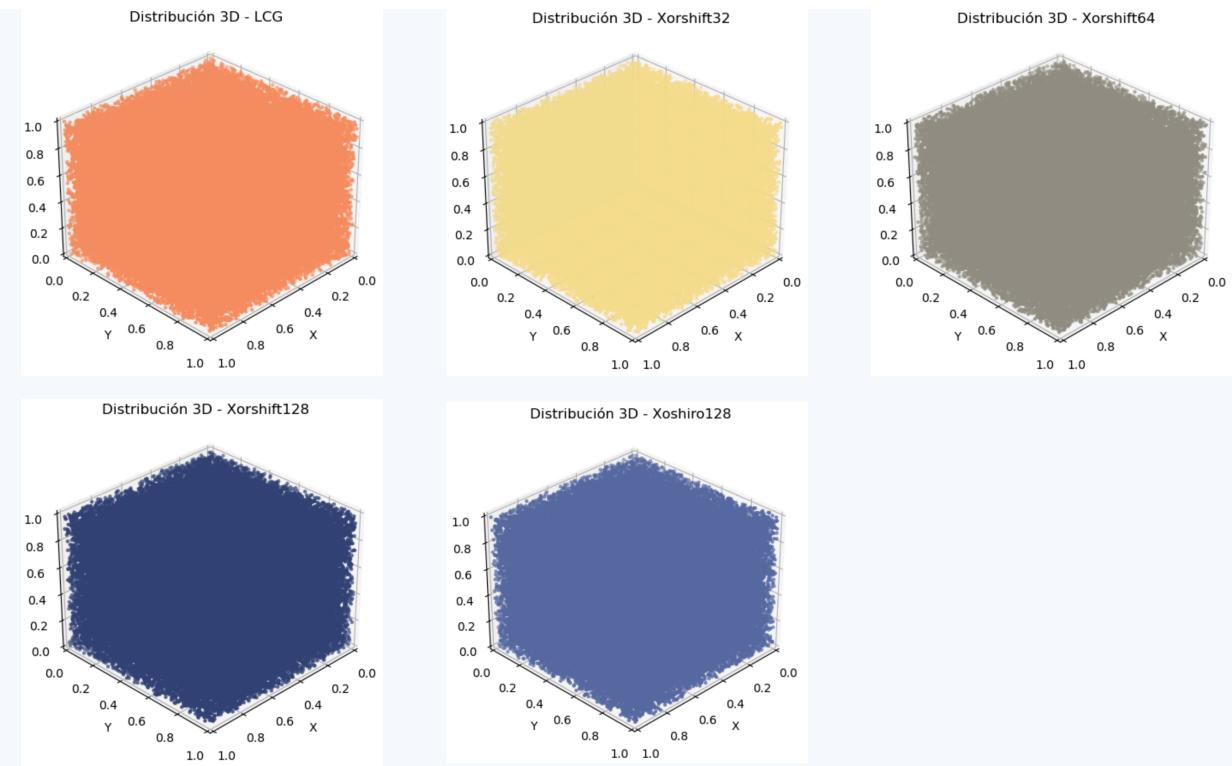




De lo anterior, podemos ver como los generadores de menor periodo son más propensos a mostrar patrones, lo cual indica una aleatoriedad inferior que aquellos con periodo $2^{128} - 1$ como `xorshift128+` o `xoshiro128++` que no presentan patrones observables incluso con regiones de largo 10^{-4} . Resulta interesante ver como los puntos generados por `xorshift32` están mejor distribuidos que aquellos generados por `xorshift64`; atribuimos este fenómeno al hecho de que ambos comparten fórmulas similares con la diferencia de que tienen distintos tamaños de estado, lo que implica que para números mayores a 2^{32} no alcanza solamente con desplazamientos y operaciones xor para lograr una buena distribución.

5.2.3 Distribución en el Cubo Unitario

Se generaron 1 millón de triples del tipo (y_i, y_{i+1}, y_{i+2}) para cada generador normalizado en el intervalo $[0,1]$. La motivación detrás de las gráficas era detectar algún patrón visible que pudiera revelar sesgos o correlaciones no deseadas en la secuencia de números generados.



Por lo visto en las gráficas, todos los generadores logran estimar adecuadamente el volumen del cubo unitario. Además, existe la posibilidad de interactuar con estas visualizaciones en el [código](#) pudiendo modificar el punto de vista y la cantidad de puntos generados.

5.3 Tests de Bondad de Ajuste Chi-cuadrado

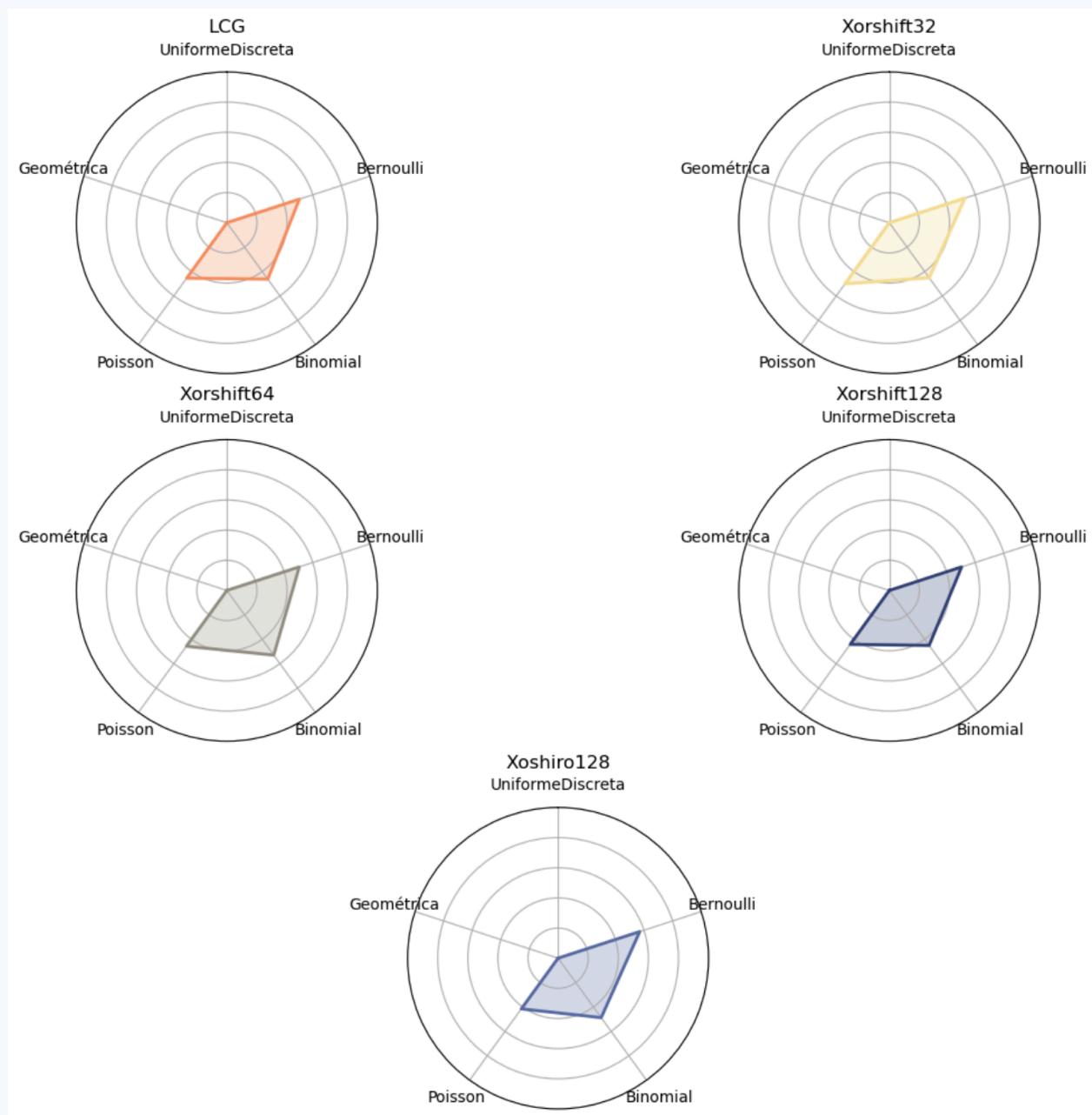
La idea de esta batería de tests fue ver el nivel de confianza alcanzado cuando se simula una muestra proveniente de una distribución discreta con cada uno de los generadores.

Para cada distribución, el nivel de confianza está dado por la estimación del p-valor obtenido en una cantidad de simulaciones del test de hipótesis chi-cuadrado de Pearson bajo la hipótesis nula de que la muestra generada proviene de tal distribución. Se consideraron las siguientes distribuciones de las cuales se tienen implementaciones en el apunte de la materia:

- Uniforme Discreta {1,10}
- Bernoulli (0.5)
- Binomial (3, 0.5)
- Poisson(4)

- Geométrica(0.5)

En esta versión, se repitió 100 veces el test completo por distribución y se reporta el promedio de los 100 p-valores obtenidos, lo cual permite una estimación más estable del nivel de confianza asociado a cada generador. Para cada simulación, la muestra generada para calcular el estadístico t_i se obtuvo en base al generador de números pseudoaleatorios de la librería `random` de Python, el cual consideramos adecuado debido a sus [buenas propiedades estadísticas](#).



En cada caso, se generaron muestras de tamaño 800 y se realizaron 1250 simulaciones

5.4 Tests de Bondad de Ajuste Kolmogorov-Smirnov

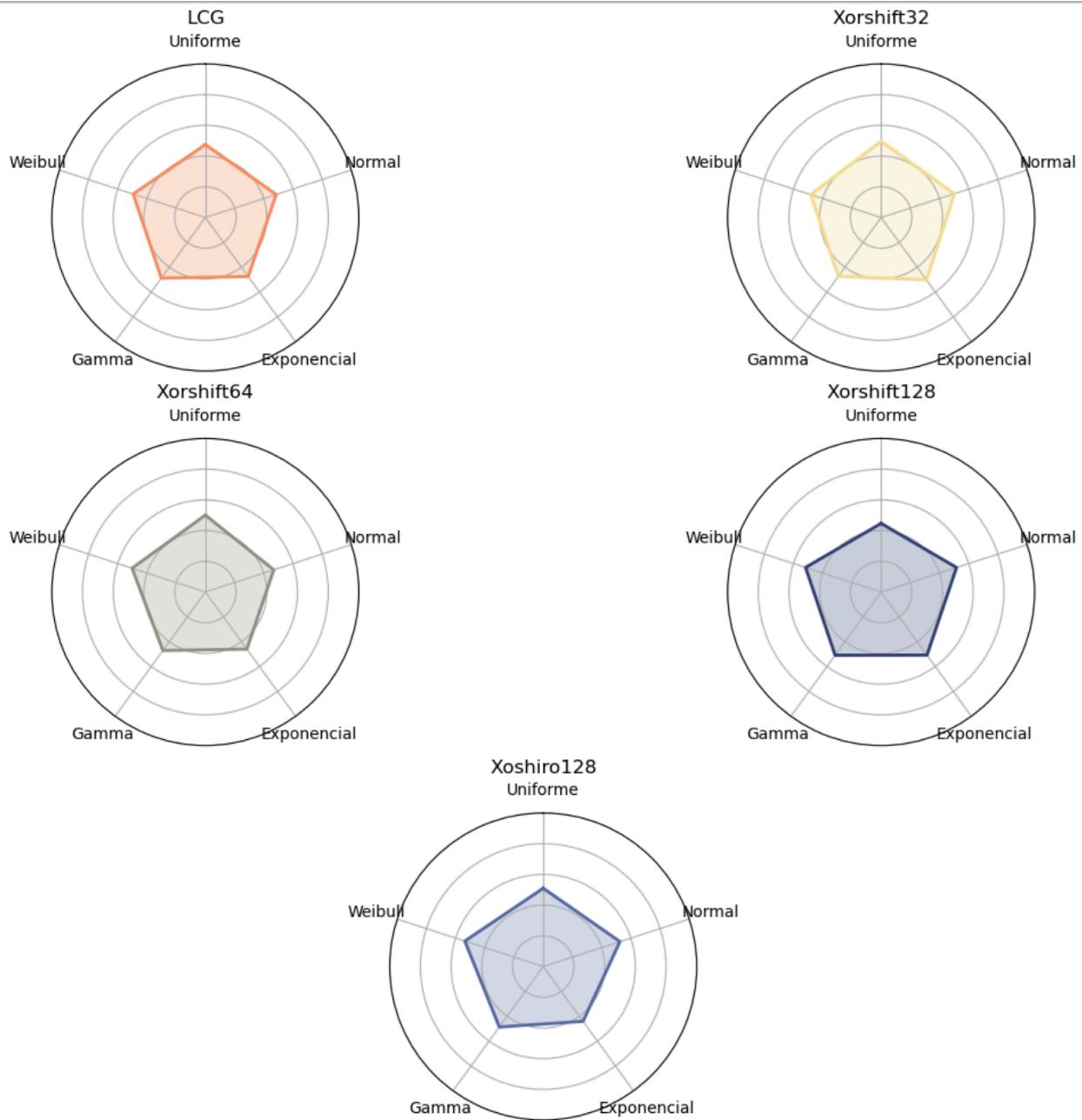
Similar a la batería de tests anterior, la motivación fue medir el nivel de confianza alcanzado cuando se simula una muestra proveniente de una distribución continua con cada uno de los generadores.

Para cada distribución, el nivel de confianza está dado por la estimación del p-valor obtenido en una cantidad de simulaciones del test de hipótesis Kolmogorov-Smirnov bajo la hipótesis nula de que la muestra generada proviene de tal distribución. Se consideraron las siguientes distribuciones de las cuales se tienen implementaciones en el apunte de la materia o bien fueron programadas en algún ejercicio:

- Uniforme (continua) (1,10)
- Normal Estándar (0,1)
- Exponencial (4)
- Gamma (4,2)
- Weibull (4,2)

En esta versión, también se repitió 100 veces el test completo por distribución y se reporta el promedio de los 100 p-valores obtenidos para poder tener una estimación más estable. A diferencia del test anterior, decidimos tener un mayor tamaño de muestra que de simulaciones para ver si esa elección de parámetros tenía correlaciones con los resultados; sin embargo, no parece ser el caso.

Para cada simulación, la muestra generada para calcular el estadístico d_i se obtuvo en base al generador de números pseudoaleatorios de la librería `numpy.random` de Python ([PCG64](#)) y está probado que tiene una alta calidad estadística.



En cada caso, se generaron muestras de tamaño 5000 y se realizaron 200 simulaciones

6. Conclusiones

De los dos puntos anteriores, se concluye que:

- Generalmente, a medida que la dimensión y el tamaño de la muestra aumentan, las estimaciones tienden a ser más precisas, ya que el error absoluto disminuye, mientras que la varianza se mantiene relativamente estable. Esto sucede porque al generar una mayor cantidad de valores, se puede observar que las distribuciones de los generadores se aproximan mejor a una distribución uniforme, especialmente cuando el tamaño de la muestra es lo suficientemente grande. También se puede apreciar en las tablas, como la varianza y el error cuadrático medio son idénticos, lo cual refuerza el hecho de que la media muestral es un estimador insesgado de la integral.
- Para los 5 generadores estudiados, el análisis de costos nos dice que: si es más prioritaria la velocidad que la calidad estadística, generadores como el `congruencial lineal` o `xorshift32` son las mejores opciones; si por otro lado, interesa más la calidad de los valores generados, siempre conviene utilizar `xorshift128+` antes que `xoshiro128++` ya que el primero puede llegar a conseguir casi la misma precisión y en un menor tiempo; por último, se concluye que `xorshift64` no tiene muy buenas aplicaciones prácticas ya que los dos primeros generadores consiguen resultados más precisos en menor cantidad de tiempo.
- Todos los generadores son repetibles y su velocidad computacional está fuertemente ligada a la complejidad de su fórmula. En cuanto a probabilidad se refiere, no utilizamos ninguna función o librería específica de Python para la implementación de los generadores, ni tampoco tenemos dependencia de alguna arquitectura en específico, por lo que nuestras implementaciones de los generadores de números pseudoaleatorios son fácilmente replicables en cualquier otro lenguaje que soporte operaciones lógicas como xor, and, or, etc. Incluso, a diferencia de Python, la gran mayoría de lenguajes tienen límites estrictos para el tamaño de los números enteros, lo cual haría que no fueran necesarias las máscaras y se podría incluso mejorar la eficiencia.
- El período de un generador no influye tanto en la distribución, i.e. períodos grandes no garantizan alta aleatoriedad. Sin embargo, los generadores con los mayores períodos demuestran ser más efectivos para criptografía, ya que se hace computacionalmente inviable el poder

detectar patrones.

- Al momento de simular muestras de variables aleatorias discretas, todos los generadores tuvieron estadísticos muy altos para la distribución geométrica y uniforme que hicieron que el p-valor tendiese a 0. Confiamos en que las implementaciones, tanto de los generadores como de las distribuciones están bien realizadas, por lo que le atribuimos el bajo grado de confianza a la sensibilidad del test y/o a problemas en la discretización o agrupamiento de las clases. Para las demás distribuciones, no hay evidencias significativas de que un generador sea mejor que otro para alguna distribución en concreto; sin embargo, se llegan a alcanzar diferencias del 10% más de confianza para las distribuciones Binomial y de Poisson entre algunos generadores
- En cuanto a la simulaciones de muestras de variables aleatorias continuas, si bien también se pueden llegar a presentar diferencias del 10% de confianza entre generadores para la distribución exponencial, sobre todo si comparamos contra `xoshiro128++` que parece ser el que peor la simula. Atribuimos las similitudes de las estimaciones de los p-valores para el resto de distribuciones a la menor sensibilidad (respecto al test chi-cuadrado) que presenta el test de Kolmogorov-Smirnov.

7. Código Fuente

El código completo de simulación y generación de gráficos se incluye en el archivo `Trabajo_Especial_D.ipynb` adjunto.

Bibliografía

1. Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*. [\[Link\]](#)
2. Vigna, S. (2017). Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*. [\[Link\]](#)

3. Blackman, D., & Vigna, S. (2021). Scrambled Linear Pseudorandom Number Generators. ACM Transactions on Mathematical Software. [\[Link\]](#)
4. Intel Corporation. Intel® Digital Random Number Generator (DRNG) Software Implementation Guide. [\[Link\]](#)
5. Matsumoto, M., & Nishimura, T. (1998). ACM Transactions on Modeling and Computer Simulation. [\[Link\]](#)
6. O'Neill, M. E. (2014). PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Harvey Mudd College Technical Report. [\[Link\]](#)