TP 2.1 - GENERADORES PSEUDOALEATORIOS

Antonelli, Nicolás
Departamento de Ingeniería en Sistemas
Universidad Tecnológica Nacional - FR Rosario
Rosario, Zeballos 1341
niconelli2@gmail.com

Acciarri, Joshua
Departamento de Ingeniería en Sistemas
Universidad Tecnológica Nacional - FR Rosario
Rosario, Zeballos 1341
acciarrijoshua@gmail.com

15 de mayo de 2020

ABSTRACT

Los generadoes de números pseudoaleatorios tienen una importancia enorme en nuestro día a día, en especial para todo aquello relacionado con la seguridad informática. La gran pregunta es ¿Qué tan buenos son? ¿Hay alguno que destaque sobre otros? En este ensayo someteremos algunos generadores pseudoaleatorios a diversas pruebas y lo comprobaremos.

***K*eywords** Números Aleatorios · Números Pseudoaleatorios · Generadores · Tests · Artículo Científico

1. Introducción

Un número pseudoaleatorio es un número generado en un proceso que parece producir números al azar, pero no lo hace realmente. Las secuencias de números pseudo-aleatorios no muestran ningún patrón o regularidad aparente desde un punto de vista estadístico, a pesar de haber sido generadas por un algoritmo completamente determinista, en el que las mismas condiciones iniciales producen siempre el mismo resultado.

2. Marco Teórico

2.1. Verdaderos Números Aleatorios

Los números completamente aleatorios (no determinísticos) se basan en alguna fuente de aleatoreidad física que puede ser teóricamente impredecible (cuántica) o prácticamente impredecible (caótica). Algunos de los generadores verdaderos que conocemos son:

- RAND Corporation: en 1955 publicó una tabla de un millón de números aleatorios que fue ampliamente utilizada. Los números en la tabla se obtuvieron de una ruleta electrónica.
- random.org [1]: genera aleatoreidad a través de ruido atmosférico
- ERNIE: usa ruido térmico en transistores, el mismo se utiliza en la lotería de bonos de Reino Unido, por lo que podemos ver la confiablidad que presenta el mismo.

Estos métodos, si bien son generadores de verdaderos números aleatorios, suelen ser bastante costosos, tardados y no reproducibles.

Sin embargo, en la actualidad existen pequeños componentes de hardware[2] especializados que se colocan en la computadora para generar estos números de una forma más eficiente. Se los conoce como *Hardware random number generators (HRNG)*.

2.2. Números Pseudoaleatorios

Los números pseudoaleatorios se generan de manera secuencial con un algoritmo determinístico, podríamos describir un simple algoritmo de generación como:

-Función de inicialización: Recibe un número (la semilla) y pone al generador en su estado inicial.

-Función de transición: Transforma el estado del generador.

-Función de salidas: Transforma el estado para producir un número fijo de bits (0 ó 1).

Una sucesión de bits pseudoaleatorios se obtiene definiendo la semilla y llamando repetidamente la función de transición y la función de salidas. Esto implica, entre otras cosas, que una sucesión de números pseudoaletorios esta completamente determinada por la semilla. Dicho esto, buscaremos que una secuencia de números pseudoaleatorios no muestre ningún patrón o regularidad aparente desde un punto de vista estadístico y que además, dada una semilla inicial, se puedan generar muchos valores antes de repetir el ciclo.

2.3. Generadores Pseudoaleatorios

Un generador pseudoaleatorio de números (GPAN) es un algoritmo que produce una sucesión de números que es una muy buena aproximación a un conjunto aleatorio de números. La sucesión no es exactamente aleatoria en el sentido de que queda completamente determinada por un conjunto relativamente pequeño de valores iniciales, llamados el estado del GPAN. La mayoría de los algoritmos de generadores pseudoaleatorios producen sucesiones que poseen una distribución uniforme según varios tipos de pruebas[3]. Las clases más comunes de estos algoritmos son generadores lineales congruentes. Entre los desarrollos más recientes de algoritmos pseudoaleatorios se encuentran Blum Blum Shub, Fortuna, y el Mersenne twister (el cual hemos utilizado en el presente documento).

En la práctica, los resultados de muchos GPAN presentan artefactos matemáticos que hacen que los mismos fallen en pruebas de detección de parámetros estadísticos. Entre estos se incluyen:

- Períodos más cortos que lo esperado para algunos estados semilla. En este contexto dichos estados semilla pueden ser llamados 'débiles'
- Falta de uniformidad de la distribución
- Correlación de valores sucesivos
- Pobre distribución dimensional de la sucesión resultado
- Las distancias entre la ocurrencia de ciertos valores están distribuidas de manera distinta que la que corresponde a una sucesión aleatoria
- Algunas secuencias de bits son 'más aleatorias' que otras

Los defectos que son exhibidos por los GPAN van desde un rango de lo imperceptible hasta lo absolutamente obvio. El algoritmo de números aleatorios RANDU utilizado por décadas en grandes computadoras tipo mainframe poseía serias deficiencias, y como consecuencia mucho del trabajo de investigación producido en ese período es menos confiable de lo que podría haber sido.

El RANDU de IBM es ampliamente considerado como uno de los generadores de números aleatorios peor concebidos jamás diseñados, y fue descrito como "verdaderamente horrible"por Donald Knuth [4]. Falla rotundamente la *prueba espectral* [4] para dimensiones mayores que 2, y cada resultado entero es impar. Sin embargo, al menos ocho bits de orden inferior se descartan cuando se convierten en coma flotante de precisión simple (32 bits, mantisa de 24 bits).

La razón para elegir estos valores particulares es que con un tamaño de palabra entero de 32 bits, la aritmética de mod 2^{31} y $655392 = (2^{16} + 3)$ se pueden hacer rápidamente, usando características especiales de algunos equipos de computadora.

2.3.1. Middle-Square Method

Historia El método fue inventado por **John von Neumann** [5], y fue descrito en una conferencia en 1949.

Es correcto también decir *Middle-Squared Method*. En la charla de 1949, Von Neumann dijo: "Cualquiera que considere métodos aritméticos para producir dígitos aleatorios está, por supuesto, en un estado de pecado". Lo que quiso decir, explicó, fue que no había verdaderos "números aleatorios", solo medios para producirlos, y que "un procedimiento aritmético estricto" (como el método del cuadrado medio) "no es tal método". Sin embargo, encontró estos métodos cientos de veces más rápido que leer "al azar" números aleatorios de las tarjetas perforadas, lo que tenía importancia práctica para su trabajo ENIAC. Encontró que la "destrucción"de las secuencias del cuadrado medio era un factor a su favor, porque se podía detectar fácilmente: "siempre se teme la aparición de ciclos cortos no detectados" [5].

Nicholas Metropolis informó secuencias de 750,000 dígitos antes de la "destrucción" (se refiere a que después de una secuencia finita el método comienza a ser unitilizable, como comentaremos más abajo, degenera hasta terminar dando todos los resultados cero o a un pequeño ciclo limitado que se repite infinitamente) mediante el uso de números de 38 bits con el "Middle Squared Method" [4].

Primera teoría de la invención [6] El libro *The Broken Dice*, de **Ivar Ekeland**, ofrece una descripción extensa de cómo el método fue inventado por un fraile franciscano conocido solo como el hermano Edvin en algún momento entre 1240 y 1250. Supuestamente, el manuscrito ahora está perdido, pero **Jorge Luis Borges** le envió a Ekeland una copia que hizo en la Biblioteca del Vaticano.

Método El método de los cuadrados medios es un método propuesto en los años 40 por los matemáticos John von Neumann y Nicholas Metropolis. El método es sencillo y consta de los siguientes pasos:

1. Se inicia con una semilla de 4 dígitos
2. La semilla se eleva al cuadrado, produciendo un número de 8 dígitos (si el resultado tiene menos de 8 dígitos se añaden ceros al inicio
3. Los 4 números del centro (los de posición 3, 4, 5, 6 siendo las posiciones [1,8]) serán el siguiente número en la secuencia, y se devuelven como resultado

El inconveniente de éste método es que tiene una fuerte tendencia a degenerar a cero rápidamente y que además los números generados pueden repetirse cíclicamente después de una secuencia corta

2.3.2. Generador Lineal Congruencial

Un generador lineal congruencial (GLC)[7] es un algoritmo que permite obtener una secuencia de números pseudoaleatorios calculados con una función lineal definida a trozos discontinua. Es uno de los métodos más antiguos y conocidos para la generación de números pseudoaleatorios. La teoría que sustenta el proceso es relativamente fácil de entender, el algoritmo en si es de fácil implementación y su ejecución es rápida, especialmente cuando el hardware del ordenador puede soportar aritmética modular al truncar el bit de almacenamiento correspondiente. El generador está definido por la relación de recurrencia:

$$X_{n+1} = (\alpha X_n + c) \; mod \; m \tag{1}$$

Dónde:

- X: es la secuencia de valores pseudoaleatorios
- m: Modulo, siendo $m > 0$
- α: Multiplicador, siendo $0 \leq \alpha < m$
- c: Incremento, siendo $0 \leq c < m$
- X_0: Semilla inicial, siendo $0 \leq X_0 < m$

Siendo m, a, c y X_0 constantes enteras específicas para el generador. Si $c = 0$, el generador es llamado frecuentemente un generador congruencial multiplicativo *(GCM)*, en cambio si $c \neq 0$, el método es llamado un generador congruencial mixto *(GCM)*.

El período de un generador congruencial mixto general es como máximo su módulo, y para algunos valores específicos del multiplicador α este período se reduce considerablemente por debajo de este máximo. El generador congruencial mixto tendrá un período completo para todas las semillas si y sólo si:

1. m y el incremento c son primos entre si
2. $\alpha - 1$ es divisible entre todos los factores primos de m
3. $\alpha - 1$ es divisible entre 4 si m es divisible por 4

2.3.3. Mersenne-Twister

Historia El generador Mersenne-Twister se desarrolló en 1997 por Makoto Matsumoto y Takuji Nishimura [8], es el generador default en muchos programas, por ejemplo, en Python, Ruby, C++ estándar, Excel y Matlab.

Este generador tiene propiedades deseables como un periodo largo $(2^{19937} - 1)$ y el hecho que pasa muchas pruebas de aleatoriedad [8], incluidas las pruebas *Diehard*, y la mayoría (pero no todas) de las pruebas *TestU01*.

Método Para nosotros este algoritmo lo tomamos como *caja negra*, dado que es el que utiliza internamente *Python*[16]. Los detalles algorítmicos del mísmo, se encuentran detallados en la fuente principal que utilizamos sobre Mersenne-Twister [8]. Nuestra intención, como se verá en la *Metodología*, es utilizarlo como punto de comparación con el algoritmo de GCL que realizamos nosotros en código.

2.4. Tests de Aleatoriedad

En estadística y análisis preliminar de datos, las pruebas de aleatoriedad (o tests de aleatoriedad) [9] [10], son pruebas estadísticas usadas para decidir si una determinada muestra o conjuntos de datos responde a un patrón o puede considerarse aleatoria. El ojo humano no es muy bueno discriminando aleatoriedad y las escalas de las gráficas, es por ello que resulta conveniente hacer pruebas estadísticas para evaluar la calidad de los generadores de números pseudoaleatorios. Hay dos tipos de pruebas:

- **Empíricas:** evalúan estadísticas de sucesiones de números.
- **Teóricas:** se establecen las características de las sucesiones usando métodos de teoría de números con base en la regla de recurrencia que generó la sucesión.

2.4.1. Prueba χ^2 (Pearson)

Este contraste sirve para analizar la aleatoriedad de los datos en cuanto a frecuencias [11]. Se trata de contrastar la hipótesis de que los datos proceden de una distribución uniforme en el intervalo [0, 1); es decir $\sim$ U(0,1).

El test χ^2 está orientado a datos categóricos, por lo que para aplicarlo al caso de números aleatorios hemos de agrupar los datos en *categorías*. El punto de partida será una secuencia de n números $X1,...,X_n, X_i \in [0,1), i = 1,...,n$, cuya aleatoriedad queremos contrastar.

El proceso de análisis lo podemos desglosar en los siguientes pasos:

1. Dividir el intervalo $[0,1)$ en d partes iguales, que llamaremos *clases*
2. Contabilizar cuántos números caen en cada clase. A la cantidad de números que caen en una clase $i = 1,...,d$ la llamaremos frecuencia observada de la clase i, y la denotaremos por θ_i.

   Obsérvese que: $\sum_{i=0}^{d} \theta_i = n$.
3. Comparamos las frecuencias observadas con las frecuencias esperadas E_i, es decir, con la cantidad de números que esperaríamos encontrar en la clase i si la secuencia fuera realmente aleatoria.

La frecuencia esperada E_i se puede calcular como:

$$E_i = np_i = n\frac{1}{d} = \frac{n}{d}, \quad E_i > 5 \tag{2}$$

Dónde:

- n: es el tamaño de la secuencia (población)
- p_i: es la probabilidad de que un número elegido al azar entre 0 y 1 caiga en la clase i, es decir $p_i = \frac{1}{d}$
- d: cantidad total de clases

A la hora de implementar este test, es útil detectar de forma sencilla la clase en que cae cada número de la secuencia. Si llamamos C_i a la clase en la que cae el número X_i, se cumple que:

$$C_i = \lfloor d \times X_i \rfloor \tag{3}$$

Dónde: $\lfloor . \rfloor$ es la función parte entera de la operación

Una vez llegados a este punto, podemos medir el ajuste de los datos a una distribución U[0, 1) usando el estadístico:

$$\chi^2 = \sum_{i=0}^{d} \frac{(\theta_i - E_i)^2}{E_i} = \frac{d}{n} \sum_{i=0}^{d} \theta_i^2 - n \tag{4}$$

Que se distribuye asintóticamente según una distribución $\chi^2(d-1)$. Es decir, tendrá $d-1$ grados de libertad. Notar en la fórmula de χ^2 que la simplificación realizada para su cálculo no será demostrada en el alcance de este documento.

Podemos construir un contraste para la hipótesis nula: H_0: los datos proceden de U[0, 1), frente a H_1 : los datos no proceden de U[0, 1).

Fijando un nivel de significación α = 0.05, la *región crítica* es:

$$\{\chi^2 \leq \chi^2_{d-1,\ a/2}\} \cup \{\chi^2 \geq \chi^2_{d-1,\ 1-(a/2)}\} \implies \{\chi^2 \leq \chi^2_{d-1,\ 0{,}025}\} \cup \{\chi^2 \geq \chi^2_{d-1,\ 0{,}975}\} \tag{5}$$

Decimos que se cumple la hipótesis nula, es decir H_0: los datos proceden de U[0, 1), y por lo tanto la prueba está totalmente superada si: $\chi^2 \notin$ en la región crítica.

2.4.2. Kolmogorov-Smirnov Test

En estadística, la prueba de Kolmogórov-Smirnov[12] (también prueba K-S) es una prueba no paramétrica que determina la bondad de ajuste de dos distribuciones de probabilidad entre sí. Cuantifica una distancia entre la función de distribución empírica de la muestra y la función de distribución acumulativa de la distribución de referencia.

$$\textbf{Estadistico}: \quad F_n(x) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & \text{si } y_i \leq x_i \\ 0 & \text{caso contrario} \end{cases} \tag{6}$$

Para dos colas el estadístico viene dado por:

$$\begin{aligned} D_n^+ &= max(F_n(x) - F(x)) \\ D_n^- &= max(F(x) - F_n(x)) \end{aligned} \tag{7}$$

donde $F(x)$ es la distribución presentada como hipótesis, en este caso, de distribución $U(0;1)$

2.4.3. Prueba de Huecos o de Distancia

Esta prueba consiste en verificar el tamaño del hueco que hay entre ocurrencias sucesivas de un número, basado en las siguientes hipótesis[13]:

$$\begin{aligned} H_0: &\quad r_i \sim \text{Independiente} \\ H_1: &\quad r_i \sim \text{Dependiente} \end{aligned} \tag{8}$$

La prueba se inicia al definir un intervalo de prueba (α, β) donde $(\alpha, \beta) \in$ (0,1), posteriormente se construye una secuencia de unos y ceros de esta manera: se asigna un 1 si $r_i \in$ al intervalo (α, β), y un 0 si $\notin$ dicho intervalo.

Una vez realizado lo anterior se determina la frecuencia observada O_i contabilizando el número de ocurrencias de cada tamaño de hueco y su correspondiente frecuencia esperada E_i, de acuerdo con: $E_i = (h)(\beta - \alpha)(1 - (\beta - \alpha))^i$ donde h es el número total de huecos en la muestra. La frecuencia del último intervalo se puede calcular mediante la diferencia entre el total y la suma de las frecuencias esperadas de los intervalos anteriores.

Por último se calcula el estadístico de prueba con la ecuación de la Prueba de χ^2.

Si $(\chi^2)_{\text{table}} < (\chi^2)_{\text{calculated}}$, entonces la hipótesis nula H_0 es aceptada y los números generados son independientes.

2.4.4. Prueba de Paridad

Esta es una prueba propia que utilizamos para verificar la calidad del generador de números pseudoaleatorios[10]. La probabilidad de que en una sucesión de longitud L obtengamos $\frac{L}{2}$ números pares tiende a 1 para un L convenientemente grande, por lo cual deberia cumplirse para cualquier generador que la frecuencia relativa de números pares sea aproximadamente 0.5.

Dado que la paridad es una propiedad exclusiva de los numeros enteros y teniendo en cuenta que los números siguen una distribución $U(0;1)$, lo que llevamos a cabo para poder realizar este test fue multiplicar cada número obtenido por 100 y luego redondearlo.

Ejemplo para una sucesión de longitud 5: [0.35714, 0.43414, 0.83131, 0.13613, 0.56841]. Luego nuestra nueva sucesión de prueba de paridad estaría conformada por: [36, 43, 83, 14, 57], lo que daría una frecuencia relativa de paridad de 0.4.

2.4.5. Otras Pruebas

Hay muchas otras pruebas [9] [10] que no fueron cubiertas en este ensayo, de las cuales nombraremos algunas que pueden llegar a ser interesantes al lector para investigar por su cuenta:

- Test de Series
- Test de Rachas
- Test de Autocorrelación
- Test de Póker
- Test de Subidas/Bajadas
- Conjunto de tests *Diehard* [14]
  - Birthday spacings
  - Overlapping permutations
  - Ranks of matrices
  - Monkey tests
  - Count the 1s
  - Parking lot test
  - Minimum distance test
  - Random spheres test
  - The squeeze test
  - Overlapping sums test
  - Runs test
  - The craps test
- Conjunto de tests *TestU01* [15]
  - Small Crush (consiste en 10 test)
  - Crush (consiste en 96 tests)
  - Big Crush (consiste en 160 tests)

3. Metodología

Para componer este trabajo práctico, se ha utilizado íntegramente el lenguaje **Python**[16], escrito el código en el IDE **Visual Studio Code**[17] con su correspondiente *Plugin*.

3.1. Librerías y Módulos de Python Utilizados

Numpy [18] Esta librería es muy completa. Primero, la utilización de *Arrays y Matrices* en vez de *Listas*, pues estas últimas son objetos, y con los arrays de Numpy se consigue una mayor eficiencia (es recomendable utilizar arrays y matrices de tamaño fijo, pues como no se almacena de forma contigua espacio extra con Numpy, tamaños variables destruyen esta eficiencia mayor pues aumentar el tamaño sería volver a construirlo). También tiene muchos métodos útiles para el manejo de arrays multidimensionales.

Si bien nosotros escribimos el código necesario para generar nuestros propios números pseudoaleatorios, Numpy Tiene un módulo llamado random (Numpy.Random) que se utilizó para comparar los que éstos se generan, con el nuestro. Este generador recibe el nombre de "Mersenne-Twister".

Pyplot [19] Este módulo de la librería **matplotlib** es ideal para graficar toda la información calculada de una forma simple: unas pocas configuraciones y un array o lista a graficar. Su sintaxis es similar a usar **Matlab**. Los gráficos que genera son muy personalizables y rápidos de generarse. También posee la posibilidad de hacer *Subplots*, es decir más de una gráfica en simultáneo, y plottearlas a todas juntas en una misma ventana. Se utilizó para todas las gráficas de la simulación.

Seaborn [20] Librería basada en Pyplot que extienede la naturaleza gráfica de ésta última y le agrega muchos tipos diferenetes de gráficos para estadística y computer science en general, así como más elementos de personalización y parametrización.

ScyPy [21] Esta librería posee muchísimas herramientas útiles para complementar cualquier trabajo estadístico computacional, nosotros hemos tomado en particular el objeto que nos trae todo lo necesario para el análisis de la distribución de χ^2, y cálculos con variables aleatorias que poseen esa distribución.

Pandas [22] Pandas es excelente para manejar correctamente grandes volúmenes de datos almacenados en formas de tablas, conocidos como "DataSets". En nuestro caso, lo usamos para una pequeña tabla para los resultados finales de los tests, y luego lo ploteamos haciendo uso de Pyplot.

3.2. Métodos de Resolución Aplicados

Inicialmente codificamos los diferentes generadores de números pseudoaleatorios. Entre ellos se encuentran el generador lineal congruencial (GLC) en sus dos versiones (multiplicativo y mixto) los cuales codificamos utilizando la fórmula que se encuentra en *Marco Teórico*, el generador que utiliza el método de los cuadrados medios que también explicamos como realizar (utilizando la semilla 5787), e hicimos uso del propio generador de Python (Mersenne-Twister). Para GLC: usamos valores de varios parámetros sacados de nuestra fuente principal para este generador[7].

Los parámetros de GLC que mencionamos, son los que utiliza el algoritmo de la biblioteca del lenguaje **C** de **GNU** (*glibc*), en donde: $m = 2^{32} - 1$, $a = 1103515245$, $c = 12345$; pero con una ligera variación: $c = 42$ para nuestro GCL Mixto, y por supuesto $c = 0$ para el GCL Multiplicativo.

Se han probado otros valores para m, a, c, pero los mejores resultados los hemos obtenido de esta forma, por lo que lo adoptamos como *representate de GCL*. En tanto para la semilla, tomamos un valor arbitrario: 876543210.

El próximo paso fue codificar las distintas pruebas (tests) y aplicarlas a las sucesiones generadas por los generadores mencionados. Para cada una de ellas explicitamos de distintas maneras si la prueba fue pasada o no, lo que más adelante se traduciría en una tabla. Por consola, mostramos más detalle de porque la secuencia superó o no cada test.

Los tests que realizamos, todos explicados ya en *Marco Teórico*, son: **Bondad de ajuste**, con el enfoque de χ^2 - Pearson, **Kolmogorov-Smirnov Test** (ambos de estos tests son de la categoría *Tests Frecuenciales*), **Prueba de Huecos o de Distancia**, **Prueba de Paridad**.

Para un mejor análisis de los datos realizamos gráficas de **dispersión**, **histogramas**, **tablas de tests** que nos ayudan a ver los datos de una manera más visual y entendible. Además de esto, capturamos la pantalla para recabar la información brindada por consola a través de las diferentes etapas del trabajo.

4. Resultados

4.1. Gráficas obtenidas

Nota: Mostraremos 3 ejemplos, en donde el parámetro a variar será la cantidad de números pseudoaleatorios a generar; concretamente 380, 1200 y 10000. En las gráficas **histogramas simultáneos** y **dispersiones simultáneas**, no se exhibirá el *Middle Square Method* junto a los otros generadores.

4.1.1. Generación de 380 números pseudoaleatorios

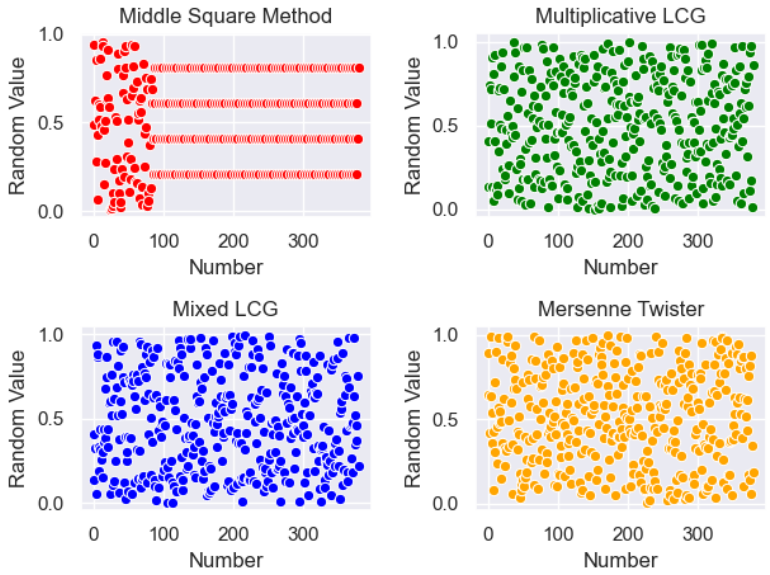

Figura 1: Gráfica de dispersiones (scatter) - Separadas - 380 números pseudoaleatorios

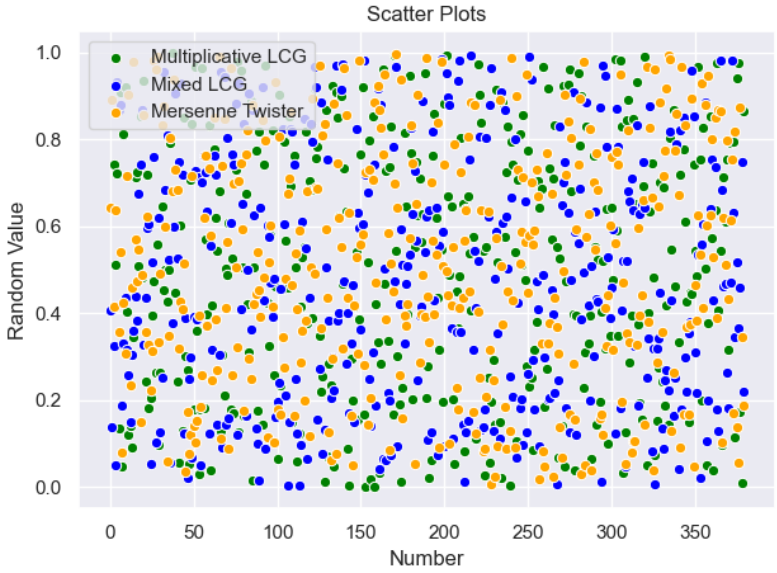

Figura 2: Gráfica de dispersiones (scatter) - Simultáneas - 380 números pseudoaleatorios

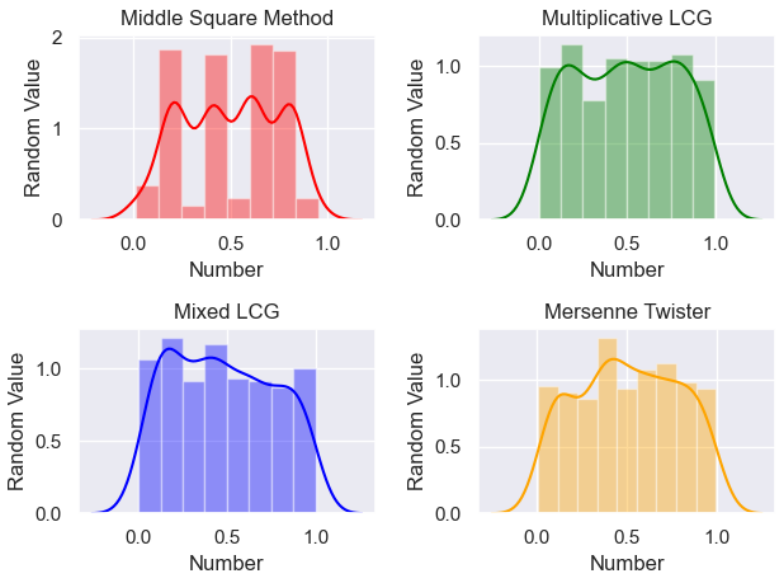

Figura 3: Histogramas - Separados - 380 números pseudoaleatorios

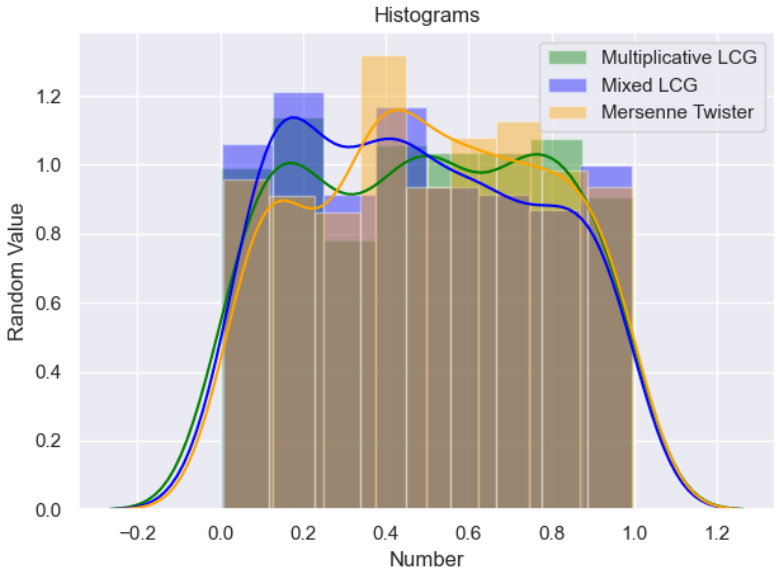

Figura 4: Histogramas - Simultáneos - 380 números pseudoaleatorios

	Middle Squared Method	Multiplicative LCG	Mixed LCG	Mersenne Twister
Pearson χ2	Rejected	Approved	Approved	Approved
Parity	Rejected	Approved	Approved	Approved
KS	Rejected	Approved	Approved	Approved
Gaps	Rejected	Approved	Approved	Approved

Figura 5: Tabla resúmen con los resultados de los tests - 380 números pseudoaleatorios

4.1.2. Generación de 1200 números pseudoaleatorios

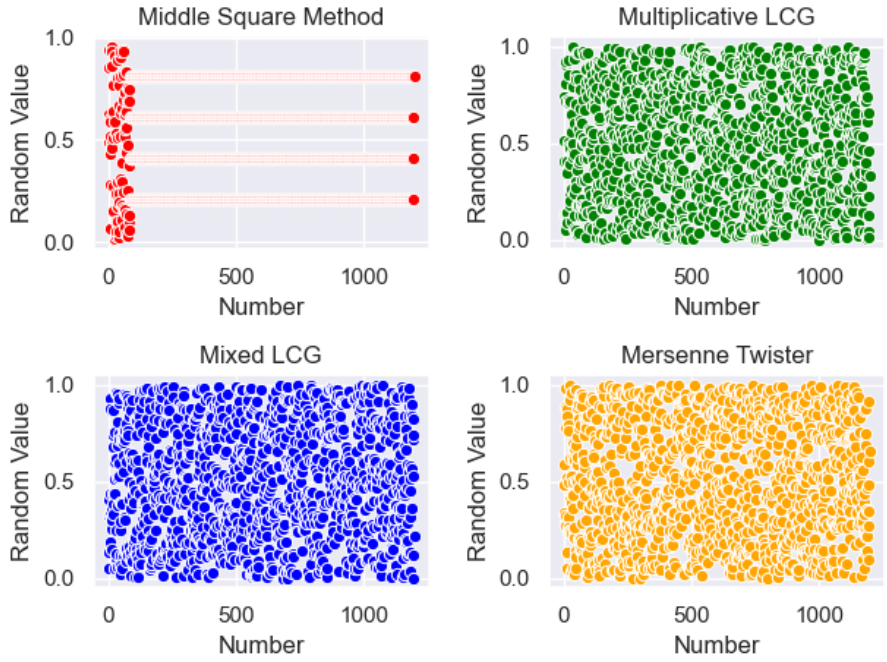

Figura 6: Gráfica de dispersiones (scatter) - Separadas - 1200 números pseudoaleatorios

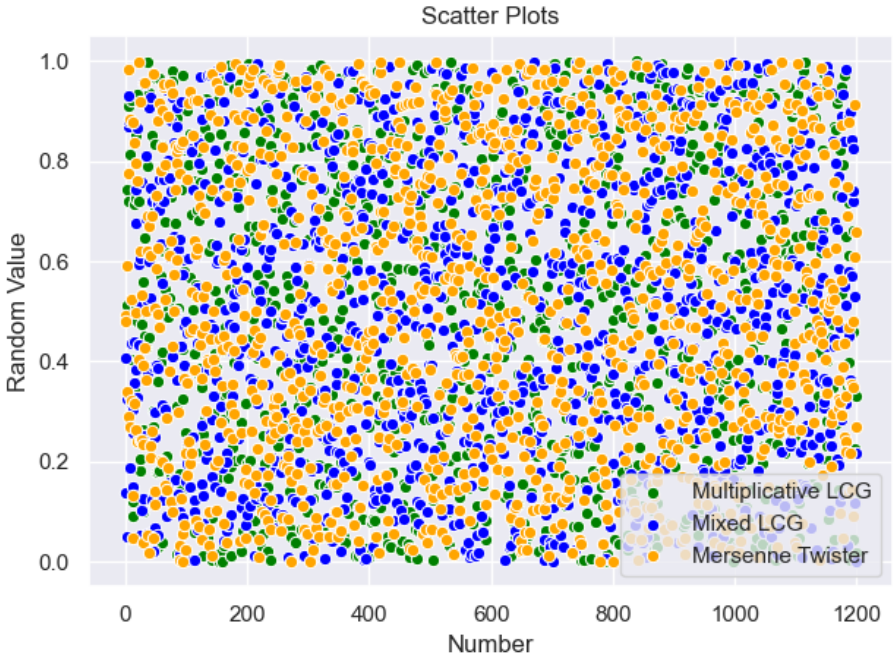

Figura 7: Gráfica de dispersiones (scatter) - Simultáneas - 1200 números pseudoaleatorios

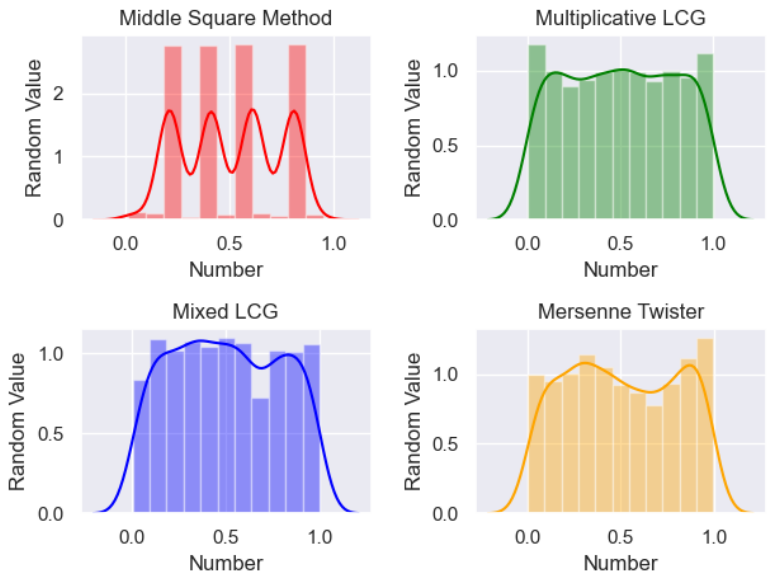

Figura 8: Histogramas - Separados - 1200 números pseudoaleatorios

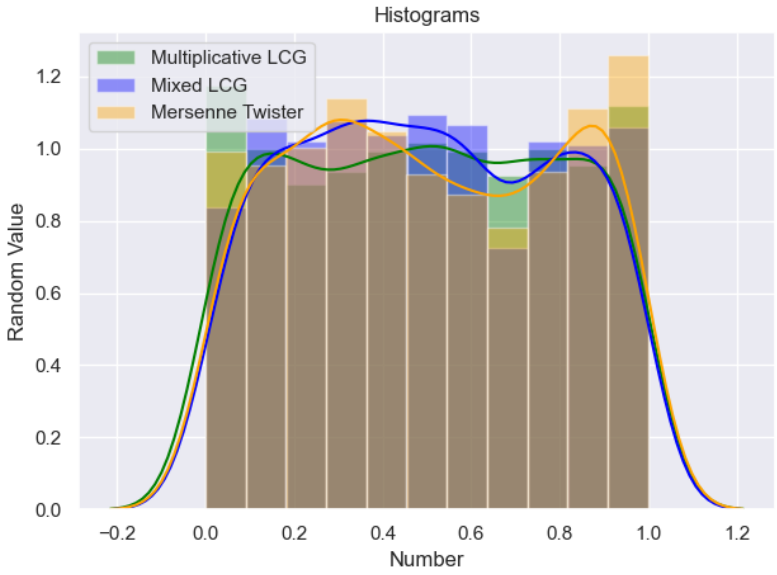

Figura 9: Histogramas - Simultáneos - 1200 números pseudoaleatorios

	Middle Squared Method	Multiplicative LCG	Mixed LCG	Mersenne Twister
Pearson χ2	Rejected	Approved	Approved	Approved
Parity	Rejected	Approved	Approved	Approved
KS	Rejected	Approved	Approved	Approved
Gaps	Rejected	Approved	Approved	Approved

Figura 10: Tabla resúmen con los resultados de los tests - 1200 números pseudoaleatorios

4.1.3. Generación de 10000 números pseudoaleatorios

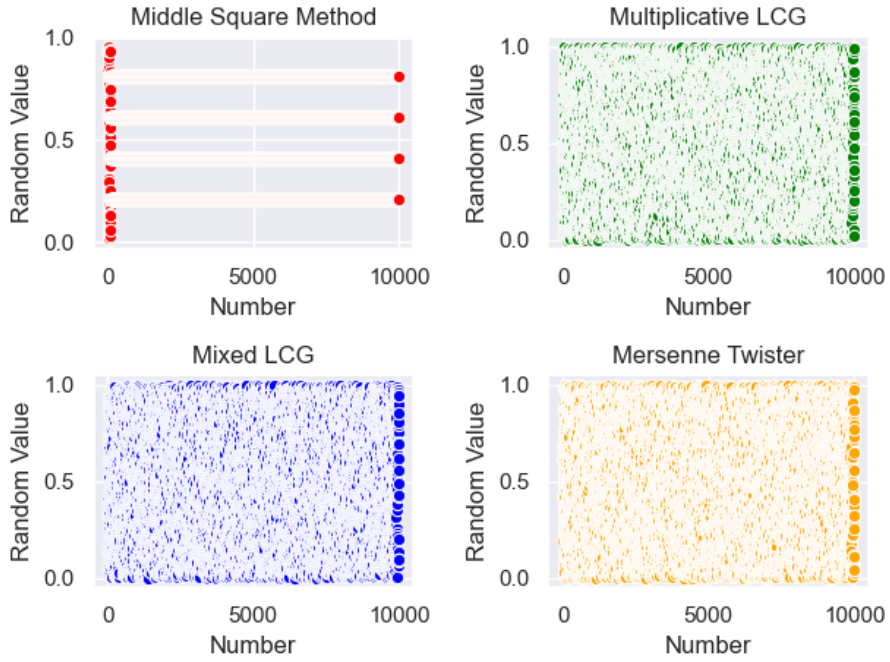

Figura 11: Gráfica de dispersiones (scatter) - Separadas - 10000 números pseudoaleatorios

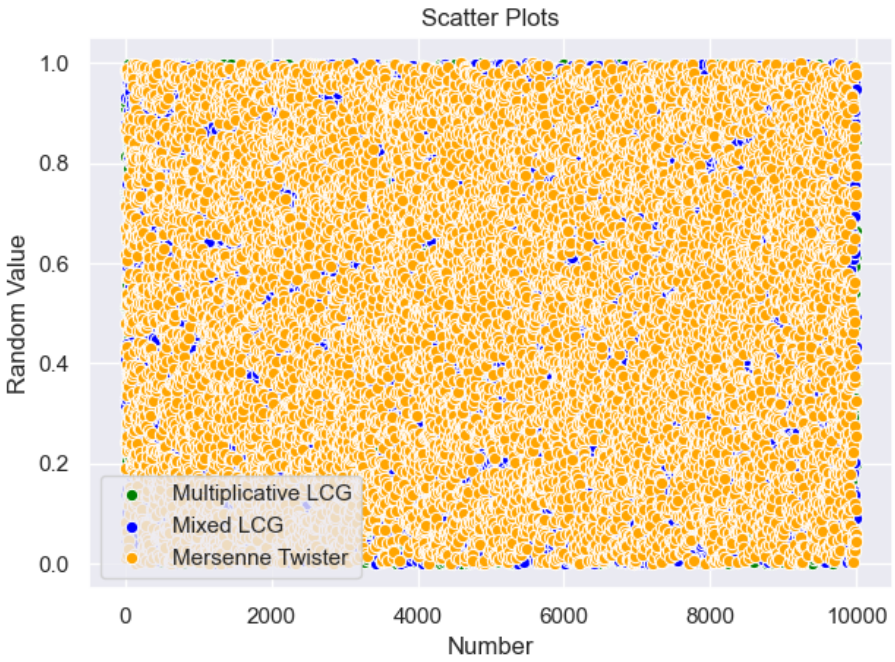

Figura 12: Gráfica de dispersiones (scatter) - Simultáneas - 10000 números pseudoaleatorios

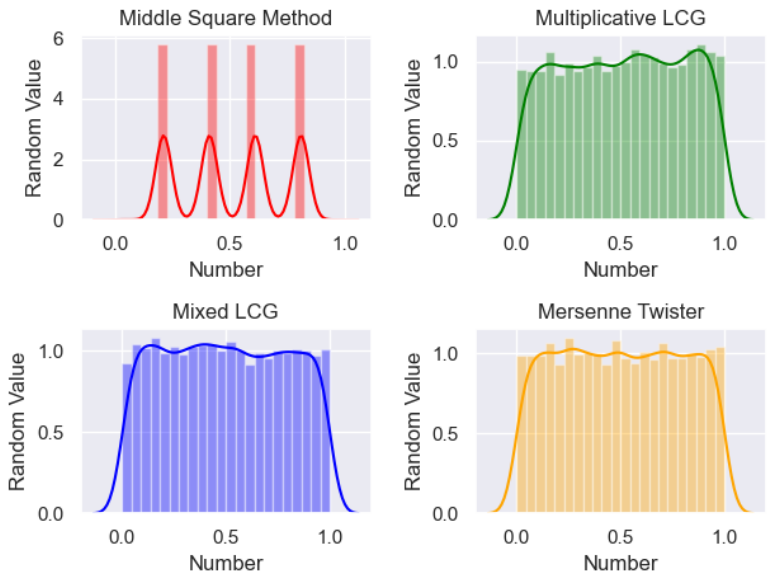

Figura 13: Histogramas - Separados - 10000 números pseudoaleatorios

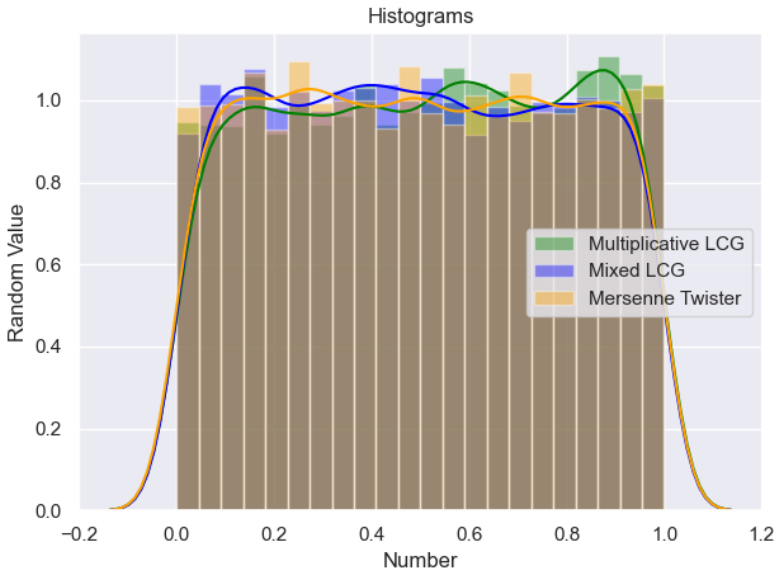

Figura 14: Histogramas - Simultáneos - 10000 números pseudoaleatorios

	Middle Squared Method	Multiplicative LCG	Mixed LCG	Mersenne Twister
Pearson $\chi 2$	Rejected	Approved	Approved	Rejected
Parity	Rejected	Approved	Approved	Approved
KS	Rejected	Rejected	Approved	Approved
Gaps	Rejected	Rejected	Approved	Approved

Figura 15: Tabla resúmen con los resultados de los tests - 10000 números pseudoaleatorios

4.2. Análisis e Interpretación

Para series de números pseudoaleatorios menores a 300 números, consideramos que no es un tamaño suficientemente grande para llegar a un resultado válido, pues los resultados de los tests en estos tamaños son demasiado variables.

Lo primero a destacar es que el **Middle Squared Method**, en cualquier tamaño razonable de secuencias, no superó ninguno de los tests realmente. Esto demuestra que este método tiene una calidad muy mala de series de números pseudoaleatorios generados.

Es por esta razón que, después de observar resultados de tests, y las figuras que toma en el scatter plot y el histograma, decidimos dejarlo de lado en los gráficos que exhiben los otros 3 generadores en simultáneo.

El comportamiento de los **GCL** es variado, da resultados bastante malos con ciertos parámetros a, m, c (ejemplo: Randu), pero resultados notablemente buenos con algunas combinaciones de parámetros, en especial el que hemos definido en la *metodología* (el del lenguaje C de GNU). Las gráficas exhibidas han sido con los parámetros de este último, como explicamos también en *metodología*.

Hay un detalle que no pasa desapercibido en GCL multiplicativo ($c = 0$) en la generación de 10000 números pseudoaleatorios o más: No siempre supera el Komolgorov-Smirnov Test ni el Gaps Test.

Con respecto al generador propio de Python, el **Mersenne-Twister**, lo hemos utilizado sin modificarle nada y los resultados que da no son tan prometedores como esperábamos. Nos encontramos que frente a una secuencia de 10mil o más hay veces en las que no supera el $\chi^2 -$ Pearson Test, si bien los otros tests que hemos codificado si los supera siempre.

Puede observarse que para todos los generadores excepto el middle squared method, que en los scatter plot tanto separados como simultáneos, que conforme crece el número de pseudoaleatorios a generar, el comportamiento de los valores de los números es cada vez más homogéneo y que no hay patrones visibles.

En tanto con los histogramas pasa lo mismo que con los scatter, para los otros 3 generadores se observa que al crecer el tamaño de población son cada vez más uniformes pues en cada intervalo de $\frac{9}{10}$, se aproxima el valor cada vez más a 1.0, en cada respectivo histograma. Esto se aprecia más notoriamente en la *curva de densidad* que pasa sobre cada histograma y como tiende ésta a ser cada vez más horizontal.

5. Conclusiones

El **Middle Squared Method** realmente no es confiable ni sirve para ser utilizado en un estudio de esadística computacional o Computer Science, así como para aplicaciones específicas en seguridad informática entre otras cosas.

El comportamiento de **GCL Mixto** vs **GCL Multiplicativo** con todos los restantes parámetros idénticos y que el mixto suela superar algunas pruebas que el multiplicativo no, es una prueba de que agregarle un factor de *c* diferente a cero influye positivamente en los resultados, por más que sea éste muy pequeño, sin embargo se requerirían más tipos de pruebas diferentes para concluir esto con mayor determinación.

Los resultados del **Mersenne-Twister** que mencionamos no cumplir nuestras espectativas no son una prueba definitiva de ser inferior a GCL, pues solo falla esporádicamente en un único test; sin embargo, esto puede deberse a imperfecciones nuestras de optimización, se requeriría realizar diferentes pruebas comparando GCL vs Mersenne-Twister en cada una para dilucidar correctamente y llegar a una conclusión más fiable y completa para ver en que aspecto es mejor cada uno.

Por otro lado, no encontramos diferencias notorias relevantes para decir que se debería utilizar más un GCL que un Mersenne-Twister, ni viceversa.

Con la información que disponemos, concluímos que ambos pueden formar parte de un estudio científico que requiera generación de números pseudoaleatorios, así como para su aplicación en sistemas que requieren estos números para seguridad informática entre otras cosas, pues no tenemos pruebas contundentes que nos hagan dejar de recomendarlos.

6. Discusión

Se podría replicar este estudio utilizando otros tipos diferentes de tests que no hayamos codificado, como los de *diehard* y aquellos que hemos nombrado en *Marco Teórico*, pues sería muy útil para comparar mano a mano GCL y Mersenne-Twister. Ésto puede llevarse a cabo de una forma menos costosa si se utilizan métodos de Tests que ya estén implementados en librerías de *Python* como *ScyPy*.

Sería útil también considerar otros gráficos como *bitmaps* o quizá *heatmaps* para encontrar diferencias visuales en los mismos que complementen a los Tests.

También sería de interés tener gráficas y parámetros de rendimiento de implementaciones de GCL específicas y descubrir cuál es realmente la combinación más óptima, teniendo como comparación el nuesto de C junto a otros como el GCL de Java, de C++, de Pascal, etc...

7. Anexo: Salidas de pantalla

Se exhibe el ejemplo de generación de 380 números pseudoaleatorios

```
PSEUDORANDOM GENERATORS
How many pseudorandoms numbers do you want to analyze?: 380
```

Figura 16: Consola: Output inicial

```
Middle Square Method
[0.4893 0.9414 0.6233 0.8502 0.284  0.0656 0.4303 0.5158 0.6049 0.5904
 0.8572 0.4791 0.9536 0.9352 0.4599 0.1508 0.274  0.5076 0.7657 0.6296
 0.6396 0.9088 0.5917 0.0108 0.0116 0.0134 0.0179 0.032  0.1024 0.0485
 0.2352 0.5319 0.2917 0.5088 0.8877 0.8011 0.1761 0.1011 0.0221 0.0488
 0.2381 0.6691 0.7694 0.1976 0.9045 0.812  0.9344 0.3103 0.6286 0.5137
 0.3887 0.1087 0.1815 0.2942 0.6553 0.9418 0.6987 0.8181 0.9287 0.2483
 0.1652 0.7291 0.1586 0.5153 0.5534 0.6251 0.075  0.5625 0.6406 0.0368
 0.1354 0.8333 0.4388 0.2545 0.477  0.7529 0.6858 0.0321 0.103  0.0609
 0.3708 0.7492 0.13   0.69   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81
 0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21
 0.41   0.81   0.61   0.21   0.41   0.81   0.61   0.21   0.41   0.81  ]
```

Figura 17: Consola: Secuencia de pseudoaleatorios de Middle Squared Method

```
Multiplicative LCG
[4.08172240e-01 1.38364018e-01 7.42375327e-01 5.12699178e-01
 7.21182150e-01 1.35310260e-01 9.09504759e-01 4.84421924e-02
 8.12656228e-01 9.21066224e-01 4.04863382e-01 1.20619397e-01
 1.40074952e-01 9.06173094e-02 3.46968880e-01 7.14147540e-01
 7.25832871e-01 5.07842451e-01 8.52336974e-01 4.81412102e-01
 9.34800364e-01 2.43252416e-01 7.20041789e-01 1.81037950e-01
 2.05165718e-01 3.94788720e-01 4.74818800e-01 2.64529005e-01
 5.59368617e-01 9.87665044e-02 2.64607400e-01 6.47017373e-01
 4.52472417e-01 4.99194858e-01 4.28660206e-01 5.00382507e-01
 7.91758330e-01 9.98382935e-01 9.37221355e-01 4.48402269e-01
 2.43250244e-01 5.68076498e-02 6.02335681e-01 1.29430853e-01
 8.52083067e-01 3.03086640e-01 2.69539245e-01 5.40869045e-02
 7.06270921e-01 8.35229743e-01 1.79627255e-01 9.66741624e-01
 3.61067925e-01 2.95610191e-02 1.80165554e-01 9.65606562e-01
 1.82868622e-01 5.71712505e-01 3.98266532e-01 8.31883700e-01
 3.96916061e-02 4.18109564e-01 7.77379739e-01 6.20781442e-01
 5.45130635e-01 6.80323981e-01 1.30955450e-01 3.05573722e-01
 2.11287490e-01 5.03122038e-01 4.88751137e-01 6.08284781e-01
 1.77906192e-01 5.27733726e-01 1.70127885e-01 7.29893119e-01
 9.80407076e-01 1.27627000e-01 9.60017528e-01 5.05573895e-01
 1.63272440e-01 6.76759637e-01 4.20943049e-01 3.10872564e-01
 8.87815212e-01 1.64426016e-02 5.07298272e-01 7.74715451e-01
 1.77653375e-01 8.19668631e-01 1.40655942e-01 1.43508708e-01
 7.43440806e-01 9.71826665e-01 4.24673322e-01 3.25247590e-01
 5.78514845e-01 8.21637807e-01 2.57998059e-01 4.61499397e-01
 6.66882005e-02 9.02263441e-01 2.35242495e-01 4.57071412e-01
 7.94245350e-01 5.68446658e-01 4.59342936e-01 8.54547963e-01
 6.92706384e-01 1.49172384e-01 7.61886184e-01 5.04911690e-01
 6.03510794e-02 1.23704238e-01 4.16958982e-01 5.28797204e-01
 5.54230266e-01 8.18958123e-01 3.76941166e-01 7.04011401e-01
 6.83484778e-01 3.87993127e-01 9.17857518e-01 7.57848280e-01
 7.84483364e-01 3.48712634e-01 2.20353101e-01 2.55414733e-01
```

Figura 18: Consola: Secuencia de los primeros pseudoaleatorios de GCL Multiplicativo

```
Mixed LCG
[0.40817224 0.13836404 0.3246772  0.05004082 0.9311612  0.86818674
 0.87977677 0.18805159 0.33160136 0.4375424  0.31500034 0.25752101
 0.14884636 0.30389157 0.35886282 0.43861633 0.38398485 0.67676878
 0.7441088  0.86677434 0.328584   0.43478881 0.59548417 0.60510515
 0.05327122 0.5173012  0.75915526 0.1039194  0.95811348 0.61976814
 0.12574909 0.25131421 0.95548643 0.68526408 0.81099918 0.52238917
 0.0571237  0.37842129 0.60014283 0.90605324 0.52689456 0.74754955
 0.92820324 0.71910484 0.12354069 0.38999995 0.02172047 0.24827661
 0.29804554 0.06887179 0.39185024 0.72905903 0.3869884  0.71305874
 0.05198417 0.70209483 0.7640217  0.72427697 0.63569634 0.12385857
 0.55496718 0.31495096 0.7642264  0.72023026 0.14374138 0.30537819
 0.95442985 0.0902929  0.94274132 0.14352364 0.80317553 0.15770663
 0.60769216 0.76632801 0.74204993 0.88246442 0.12141923 0.92885166
 0.41013577 0.65070746 0.83654331 0.16546548 0.50132652 0.90683215
 0.36360513 0.86936084 0.62450606 0.1356274  0.10039342 0.01636813
 0.64344204 0.91949803 0.43122485 0.08785024 0.60259769 0.57482334
 0.4711248  0.39235926 0.48045978 0.16031942 0.19753847 0.24098923
 0.4560214  0.82496752 0.60656635 0.21100492 0.0030267  0.14893785
 0.82581566 0.81955872 0.7428957  0.24873156 0.12347111 0.00367205
 0.09881443 0.61210773 0.84641701 0.54883069 0.06606763 0.43679726
 0.83630013 0.7965194  0.96751737 0.92033595 0.15580637 0.27251562
 0.88490438 0.07617874 0.13277586 0.50524166 0.75511843 0.38236422
 0.20437488 0.29849163 0.22247342 0.97078434 0.38320898 0.40186039
 0.91562582 0.97207961 0.47315779 0.28335514 0.40034951 0.16959716
 0.33140254 0.46488838 0.15033811 0.82301618 0.63169848 0.59714887
 0.36404231 0.60423297 0.98245469 0.71905087 0.67839641 0.24988707
 0.1398488  0.30953331 0.74259613 0.18673455 0.91786292 0.04248087
 0.88098125 0.06643533 0.50275013 0.06152832 0.21608189 0.20074653
 0.07499294 0.78333834 0.47385585 0.96739896 0.09560733 0.42180937
 0.59319932 0.51110622 0.33157076 0.46788924 0.50920889 0.53577251
 0.82744463 0.62982782 0.7283317  0.56639502 0.24648168 0.39403115
 0.41556354 0.88876673 0.54132665 0.62277955 0.10873993 0.63873978
 0.12613069 0.82001894 0.42934851 0.64353128 0.07375955 0.55485363
```

Figura 19: Consola: Secuencia de los primeros pseudoaleatorios de GCL Mixto

```
Python's own generator (Mersenne Twister)
[0.64297699 0.89122888 0.41484449 0.6373223  0.98644882 0.35599186
 0.54069866 0.07852149 0.4253838  0.30639879 0.71591738 0.90302808
 0.23502266 0.45921159 0.98034361 0.56972061 0.47493791 0.5471366
 0.14876323 0.48795177 0.85529389 0.35621343 0.62254659 0.32940584
 0.22217    0.31321054 0.98112505 0.96206339 0.98951343 0.33424134
 0.49027634 0.83364183 0.87780084 0.59888073 0.0585495  0.34956616
 0.80442881 0.68015024 0.9393069  0.73057802 0.68314235 0.31948284
 0.42684083 0.40999371 0.51406484 0.03665659 0.17445159 0.07703935
 0.11989635 0.71757186 0.14157632 0.10214716 0.15202316 0.39564741
 0.18468755 0.2574181  0.62121261 0.76396286 0.37188861 0.73318633
 0.57884862 0.24521104 0.85105362 0.38510689 0.27068367 0.99059041
 0.11095479 0.7393693  0.84744194 0.41192677 0.56410361 0.0879407
 0.96460357 0.25852246 0.69831266 0.50476665 0.32193194 0.7059201
 0.74652179 0.59513888 0.17665286 0.76586286 0.82236433 0.29595148
 0.24912286 0.88119156 0.42097721 0.44372011 0.54152475 0.45340482
 0.39329551 0.84617332 0.48314493 0.79944159 0.11388365 0.51651634
 0.60265903 0.41433117 0.10218092 0.93123901 0.17831075 0.80801268
 0.16814352 0.34518123 0.27578682 0.48044153 0.67503764 0.7110344
 0.86173978 0.46441032 0.16360099 0.53853858 0.40896563 0.59027078
 0.28678527 0.36091385 0.19818676 0.38461577 0.68183518 0.11907789
 0.50662602 0.89325951 0.49125497 0.68936256 0.68742043 0.96951668
 0.36494221 0.80873009 0.83641302 0.59337321 0.43716758 0.12798953
 0.06187746 0.29676073 0.88156264 0.07744129 0.74529695 0.61938747
 0.5538185  0.31829596 0.96433236 0.8559769  0.43466382 0.63305414
 0.16561554 0.04971628 0.41763629 0.52811811 0.30885073 0.98204835
 0.55969505 0.5821001  0.28133695 0.72514623 0.14346045 0.7317427
 0.20401472 0.70896277 0.88652993 0.47185835 0.92459608 0.86688763
 0.82612835 0.76470357 0.9513287  0.38719892 0.58496399 0.08622591
 0.51501116 0.45186647 0.69938082 0.99732054 0.64233221 0.25888716
 0.93932475 0.15832527 0.39556766 0.37077648 0.42160955 0.57268994
 0.83249439 0.74691694 0.05285411 0.65131535 0.90379278 0.28827318
 0.39878535 0.48968446 0.39263039 0.98653193 0.43810307 0.33257113
 0.70254461 0.56781859 0.39788709 0.53394653 0.83848134 0.09023634
 0.98005871 0.50792333 0.74297391 0.49754406 0.44320796 0.10072626
 0.41922624 0.55294981 0.7890719  0.22974976 0.17870702 0.41375449
```

Figura 20: Consola: Secuencia de los primeros pseudoaleatorios de Mersenne-Twister

```
TESTS: Middle Square Method

------------UNIFORM TEST-----------
Null hypothesis REJECTION, this list doesn't correspond to an uniform U(0,1) distribution
This is because 1940.0 is on the region {χ2 ≤ 22.105627} ∪ {χ2 ≥ 55.667973}

------------PARITY TEST------------
Total length of the array: 380
Odds Absolute Frequency: 344
Odds Relative Frequency: 0.9052631578947369
It seems that this is not a good generator, or may be more iterations are needed

------------KOMOLGOROV SMIRNOV TEST------------
is 0.15115789473684205  <  0.06976652794179049  ?
Null hypothesis REJECTION, the list of values doesn't correspond to an uniform U(0,1) distribution

------------GAPS TEST------------
is 182.89228723404258  <  11.070497693516355  ?
Null hypothesis REJECTION, these numbers are not independent according to the GAPS TEST

-----------------------------------
```

Figura 21: Consola: Tests Middle Squared Method

```
TESTS: Multiplicative Linear Congruential Generator

------------UNIFORM TEST-----------
Null hypothesis ACCEPTATION, this list indeed correspond to an uniform U(0,1) distribution
This is because 32.8 is NOT on the region {χ2 ≤ 22.105627} ∪ {χ2 ≥ 55.667973}

------------PARITY TEST------------
Total length of the array: 380
Odds Absolute Frequency: 193
Odds Relative Frequency: 0.5078947368421053
It seems to be a good generator

------------KOMOLGOROV SMIRNOV TEST------------
is 0.030673231465908418  <  0.06976652794179049  ?
Null hypothesis ACCEPTATION, the list of values does correspond to an uniform U(0,1) distribution

------------GAPS TEST------------
is 10.127520161290324  <  12.59158724374398  ?
Null hypothesis ACCEPTATION, these numbers are independent according to the GAPS TEST

-----------------------------------
```

Figura 22: Consola: Tests GCL Multiplicativo

```
TESTS: Mixed Linear Congruential Generator

------------UNIFORM TEST-----------
Null hypothesis ACCEPTATION, this list indeed correspond to an uniform U(0,1) distribution
This is because 38.4 is NOT on the region {χ2 ≤ 22.105627} ∪ {χ2 ≥ 55.667973}

------------PARITY TEST------------
Total length of the array: 380
Odds Absolute Frequency: 181
Odds Relative Frequency: 0.4763157894736842
It seems to be a good generator

------------KOMOLGOROV SMIRNOV TEST------------
is 0.05049327002174986  <  0.06976652794179049  ?
Null hypothesis ACCEPTATION, the list of values does correspond to an uniform U(0,1) distribution

------------GAPS TEST------------
is 10.466340174129353  <  11.070497693516355  ?
Null hypothesis ACCEPTATION, these numbers are independent according to the GAPS TEST

-----------------------------------
```

Figura 23: Consola: Tests GCL Mixto

```
TESTS: Python's own generator (Mersenne Twister)

------------UNIFORM TEST------------
Null hypothesis ACCEPTATION, this list indeed correspond to an uniform U(0,1) distribution
This is because 42.0 is NOT on the region {χ2 ≤ 22.105627} ∪ {χ2 ≥ 55.667973}

------------PARITY TEST------------
Total length of the array: 380
Odds Absolute Frequency: 197
Odds Relative Frequency: 0.5184210526315789
It seems to be a good generator

------------KOMOLGOROV SMIRNOV TEST------------
is 0.05381989075430621  <  0.06976652794179049  ?
Null hypothesis ACCEPTATION, the list of values does correspond to an uniform U(0,1) distribution

------------GAPS TEST------------
is 7.738471283783784  <  12.59158724374398  ?
Null hypothesis ACCEPTATION, these numbers are independent according to the GAPS TEST

-----------------------------------
```

Figura 24: Consola: Tests Mersenne-Twister

```
Tests Results' Summaring

           Middle Squared Method Multiplicative LCG Mixed LCG Mersenne Twister
Pearson χ2              Rejected           Approved  Approved         Approved
Parity                  Rejected           Approved  Approved         Approved
KS                      Rejected           Approved  Approved         Approved
Gaps                    Rejected           Approved  Approved         Approved

graph_380numbers_tests_results_summaring guardado correctamente
graph_380numbers_scatter_separated guardado correctamente
graph_380numbers_scatter_simultaneous guardado correctamente
graph_380numbers_histograms_separated guardado correctamente
graph_380numbers_histograms_simultaneous guardado correctamente
```

Figura 25: Consola: Output final

Referencias

[1] Random.org
https://www.random.org/analysis/

[2] Wikipedia EN. Hardware random number generator.
https://en.wikipedia.org/wiki/Hardware_random_number_generator

[3] Github IO. Numeros Pseudoaleatorios.
https://tereom.github.io/est-computacional-2018/numeros-pseudoaleatorios.html

[4] Knuth hablando sobre RANDU. Art of Computer Programming, Volume 2: Seminumerical Algorithms - Donald E. Knuth – 4 de noviembre 1997 - ISBN 0-201-03822-6 (Versión original 1981)

[5] Von Neumann & Middle Squared Method. Various techniques used in connection with random digits, John von Neumann, 1951

[6] Edvin & J. L. Borges: Teoría de Invención. The Broken Dice, and Other Mathematical Tales of Chance, Ivar Ekeland, 15 de Junio 1996, ISBN 978-0-226-19992-4

[7] Wikipedia ES. Generador lineal congruencial.
https://es.wikipedia.org/wiki/Generador_lineal_congruencial

[8] Wikipedia EN. Mersenne-Twister.
https://en.wikipedia.org/wiki/Mersenne_Twister

[9] UTN, Apunte Cátedra Simulación.
http://hemaruce.angelfire.com/Unidad_III.pdf

[10] Pruebas de aleatoriedad.
https://es.wikipedia.org/wiki/Pruebas_de_aleatoriedad

[11] Info importante sobre pruebas de aleatoriedad. Estadística Computacional, Antonio Salmeon Cerdán y María Morales Giraldo, 2001

[12] Prueba de Kolmogorov-Smirnov.
https://es.wikipedia.org/wiki/Prueba_de_Kolmogorov-Smirnov

[13] Prueba de huecos.
https://www.coursehero.com/file/p7ae8to/Prueba-de-huecos-Esta-prueba-consiste-en-verificar-el-tama%C3%B1o-del-hueco-que-hay/

[14] Wikipedia EN. Diehard Tests: explanation of the twelve tests.
https://en.wikipedia.org/wiki/Diehard_tests

[15] Wikipedia EN. Testu01: explanation of this tests' library.
https://en.wikipedia.org/wiki/TestU01

[16] Python Doc. Python 3.8.2 | Documentación Oficial.
https://docs.python.org/3/

[17] Microsoft (VSC). Visual Studio Code Official Webpage
https://code.visualstudio.com/

[18] Numpy Doc. Numpy 1.18 | Documentación Oficial.
https://numpy.org/doc/1.18/

[19] Pyplot Doc. Pyplot 3.1.1 | Documentación Oficial.
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.html

[20] Seaborn Doc. Seaborn 0.10.1 | Documentación Oficial.
https://seaborn.pydata.org/

[21] ScyPy Doc. ScyPy Library | Documentación Oficial.
https://www.scipy.org/scipylib/index.html

[22] Pandas Doc. Pandas 1.0.3 | Documentación Oficial.
https://pandas.pydata.org/