Evaluación de métodos de generación de números aleatorios.

Heldert Villegas Jaramillo,[1]

[1] *Estudiante de Matemáticas de la Universidad Nacional de Colombia*

[*] *hvillegas@unal.edu.co*

Fabian Andres Ruiz Tortello,[2]

[2] *Estudiante de Ciencias de la Computación de la Universidad Nacional de Colombia*

[*] *fruiz@unal.edu.co*

Samuel Jacobo Garavito Segura,[3]

[3] *Estudiante de Matemáticas de la Universidad Nacional de Colombia*

[*] *sgaravito@unal.edu.co*

Abstract: El siguiente artículo documenta el estudio de la aleatoriedad de sucesiones provenientes de tres (3) generadores de números pseudo-aleatorios. Este análisis se realiza por medio de tres (3) test de aleatoriedad con el propósito de determinar la efectividad del método mediante su implementación haciendo uso de la herramienta de Google Colab con IPython .

Introducción

Seguridad, simulación y modelación son algunos de los campos cuyo funcionamiento se soporta sobre largas cadenas de números aleatorios. A día de hoy, los números aleatorios constituyen un papel fundamental en procesos criptográficos y de simulación, por ende la comprensión y estudio de su provenir es de vital importancia para determinar su efectividad sobre el campo de uso. Dicho estudio se realiza por medio de un Test de aleatoriedad, el cual decide si hay evidencia suficiente para afirmar que estos provienen de una distribución de probabilidad en específico o no [1]. Existen diferentes métodos de generación de números aleatorios, unos más populares que otros. En el siguiente artículo estudiaremos la naturaleza aleatoria de las sucesiones de números generadas por tres (3) métodos mediante tres (3) test de aleatoriedad.

Números aleatorios

Un **número aleatorio** busca ser *"un número cuyo valor en su primera aparición sea impredecible"*. Debido a su naturaleza *"impredecible"* se pretende emplearlos en distintas situaciones, como la generación de claves privadas y tokens en criptografía o en la generación de muestras provenientes de una distribución de probabilidad para la simulación de un fenómeno. Los métodos para generar este tipo de números se agrupan en tres grandes conjuntos [1]:

- **Humanos:** Este método se basa en que el ser humano idee un número aleatorio. Este es el método de generación más accesible, sin embargo es el menos efectivo y eficiente, pues se ha demostrado que las personas tenemos ciertos problemas (como sesgos) al momento de generar secuencias robustas y largas de números aleatoriamente.
- **Físicos:** Dentro de los métodos más antiguos y efectivos se encuentran los métodos físicos, los cuales, como su nombre lo indica, dependen de un factor físico para su generación, los típicos ejemplos son el tirado de una moneda al aire o el resultado de una ruleta americana. Su desventaja yace en su eficiencia, ya que generar largas cadenas de números aleatorios se requiere de un periodo de tiempo prolongado.

- **Computacionales:** Estos métodos son muy efectivos y los más eficientes a día de hoy. Se implementan algoritmos matemáticos computacionalmente, con el fin de generar una larga cadena de números aleatorios. Su desventaja está sujeta a su efectividad, puesto que aunque existen algoritmos de generación bastante robustos, estos al ser implementados matemáticamente son deterministas y, por tanto, susceptibles a predicciones si se conoce el patrón que los genera.

En este artículo estudiaremos los métodos pertenecientes al conjunto de los computacionales.

Números Pseudo-aleatorios

Las secuencias de números generados por los métodos computacionales, a pesar de ser *"aleatorios"* provienen de un proceso determinista, a diferencia de los físicos, los cuales son no deterministas, por ende los procesos mediante los cuales se generen estos números se denominaran **pseudo-aleatorios**. Irónicamente, estas secuencias construidas correctamente demuestran mejor aleatoriedad que los mismos **números aleatorios** provenientes de un método físico, pues una serie de transformaciones de los métodos pseudo-aleatorio pueden eliminar correlaciones estadísticas entre los mismos resultados [1].

Para ejemplificar esto de una mejor manera, el proceso físico de ruido eléctrico naturalmente sufre superposición de estructuras regulares como ondas o fenómenos periódicos (interferencias), algo que parece impredecible, sin embargo, a la vista de los test estadísticos, cuyo veredicto es el final, se determina que los resultados no demuestran aleatoriedad. Mientras que un método como el denominado **Mersenne twister**, de los más populares ideado por *Makoto Matsumoto y Takuji Nishimura*, generan números pseudo-aleatorios que si demuestran esta propiedad.

Generadores Pseudo-aleatorios (GNPA)

Primero que todo introducimos un par de definiciones que son fundamentales para describir matemáticamente el comportamiento de un *generador de numeros pseudo-aleatorios*.

Definición 1.0: Un generador de números pseudo-aleatorios *es una estructura* $G = (X, x_0, T, U, g)$*, donde* X *es un conjunto finito de estados,* $x_0 \in X$ *es el estado inicial (semilla), la aplicación* $T : X \longrightarrow X$ *es la función de transición,* U *es el conjunto finito de posibles observaciones y* $g : X \longrightarrow U$ *es la función de salida.* [2]

Se elige una semilla inicial cualquiera x_0, y se genera una sucesión de valores x_n por medio de una relación recursiva $x_n = T(x_{n-1})$. Cada uno de estos valores proporciona un número pseudo-aleatorio u_n definido a través de alguna relación $u_n = g(x_n)$. Esta sucesión de estados es periódica, puesto que X es finito, de manera que en algún momento ocurrirá que $x_j = x_i$ para algún $i < j$, $x_{j+k} = x_{i+k}$ y $u_{j+k} = u_{i+k}$ para todo $k \geq 0$, por ende se tiene la siguiente definición [2].

Definición 2.0: El periodo de un generador *es el menor entero* $\alpha > 0$ *tal que para algún entero* $\beta \geq 0$ *se tiene que* $x_k = x_{\alpha+k}$ *para todo* $k \leq \beta$*. El periodo no puede superar el cardinal de* X*.* [2]

Un buen generador de números *pseudo-aleatorios* debe cumplir con las siguientes propiedades:

- **Distribución:** La sucesión de valores que proporcione debe provenir de una muestra aleatoria $U(0,1)$.
- **Reproducibilidad:** Los resultados deben ser reproducibles bajo las mismas condiciones iniciales *(semillas)*.
- **Periodicidad:** La sucesión de valores generados debe tener un ciclo no repetitivo tan largo como sea posible.
- **Rendimiento:** Debe ser eficiente y ocupar poca memoria.

Pero usualmente no se alcanzan todas simultáneamente. Según el contexto en el que se esté se necesitará priorizar una ante otra; y para saber si un determinado **GNPA** efectivamente tiene una de estas propiedades existen distintos test diseñados para evaluar cada una de las mismas. En este artículo nos enfocamos principalmente en métodos que evalúan la propiedad de **distribución**, y luego en uno que aprovecha la **periodicidad**.

Generador Merssene Twister

El algoritmo generador *GNPA* Merssene Twister es utilizado para la generación de números pseudo-aleatorios, desarrollado en 1997 por *Makoto Matsumoto y Takuji Nishimura* [12]. Actualmente, se emplea como estructura estándar de generación de números pseudo-aleatorios en programas y lenguajes de programación como Python, R, Ruby, entre otros.

El Mersenne twister (M.T.) es una variante del algoritmo TGFSR, *twisted generalized feedback shift register*, que es una mejora al generalized feedback shift register (GFSR). El M.T. es modificado para tomar un arreglo incompleto para admitir un número primo de Merssenne como periodo, por tanto, el nombre. Como resultado, este generador tiene un periodo de $2^{19937}-1$ [13] y está diseñado para generar una distribución $U(0,1)$ de números reales aleatorios, con especial atención a los bits más significativos [12].

El algoritmo M.T. genera una secuencia de vectores palabra, que son considerados números pseudo-aleatorios uniformes entre 0 y 2^w-1 con w el número de bits que se desean. Dividiendo por 2^w-1 se transforman las palabras a números reales en $[0,1]$.

El algoritmo está basado en la recurrencia (1) donde se tiene: un entero n, el cual es el grado de la recurrencia, un entero r *(escondido en la definición de* x_k^u*)*, $0 \leq r \leq w-1$, un entero m, $1 \leq m \leq n$, una matriz constante A con entradas en el cuerpo de dos elementos $\mathbb{F}_2$ y con $x_0, x_1, \cdots, x_{n-1}$ como semillas iniciales. Luego se genera x_n con $k=0$. Al variar k se generan $x_{n+1}, x_{n+2}, \ldots$. En la parte derecha de la recurrencia x_k^u significa *los mayores* $w-r$ *bits* de x_k y x_{k+1}^l *los menores* r *bits* dé x_{k+1}. Así, si $x=(x_{w-1}, x_{w-2}, \cdots, 0)$, entonces x^u es el vector de $w-r$ bits $(x_{w-1}, \ldots, x_r)$ y x^l es el vector de r bits $(x_{r-1}, \ldots, x_0)$. $(x_k^u | x_{k+1}^l)$ representa la concatenación de ambos. Finalmente, se suma x_{k+m} a este vector *($\oplus$ es la adición bit-wise en módulo 2)*, y luego se genera el siguiente vector x_{k+n} [12].

$$x_{k+n} \quad = \quad x_{k+m} \oplus (x_k^u | x_{k+1}^l) A \tag{1}$$

Generador Xorshift

En varias áreas de la computación, como por ejemplo la computación gráfica, se hace una cantidad inmensa de cálculos, por tanto, es de vital importancia hacerlos de la manera más eficiente posible. Con esto en mente, *George Marsaglia* [3] publica el método *Xorshift* que es de implementación sencilla con miras a ser utilizado en gran escala y en tiempo real.

El método consiste en efectuar sobre una o a varias semillas una sucesión de operaciones de bits, específicamente *xors* y *bitshifts*. La motivación viene dada, ya que este procedimiento se puede modelar matemáticamente mediante operaciones lineales por la matriz $I+L$ donde L es una matriz de la forma:

$$L \quad = \quad \begin{pmatrix} 0 & 0 & \ldots & 0 \\ 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 1 & 0 \end{pmatrix}$$

O una potencia de la misma [4], esta operación es llamada un *left bitshift*, el cual un computador efectúa de manera muy eficiente. De la misma manera, la suma por I es la que efectúa el *xor* sobre el vector. Para garantizar aleatoriedad, se realizan varias de estas operaciones sucesivamente. En resumidas cuentas, si $v^{(0)}$ el vector binario semilla, el método se desarrolla mediante el proceso iterativo $v^{(k+1)} = (I+L^a)(I+R^b)(I+L^c)v^{(k)}$, donde $a,b,c \in \mathbb{Z}$ y R es la matriz que modela los *right bitshifts*.

$$R = \begin{pmatrix} 0 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 \\ 0 & \dots & 0 & 0 \end{pmatrix}$$

La escogencia de a,b y c se hacen para garantizar la invertibilidad del producto, detalles al respecto pueden ser consultados en [3]. Para fines de implementación se utilizó la tripla $(13,17,5)$ que sugiere el autor para ser usada. La implementación del mismo puede ser vista en el archivo de Google Colab adjunto [18], de esta resalta que demanda pocos cálculos por iteración y que tiene que ser efectuado en cadenas de 32 bits. Cabe aclarar que se pueden generar cadenas de otros tamaños, solo hay que tener especial cuidado con la escogencia de a,b,c; por ejemplo, si buscamos generar números de 64 bits podríamos usar la tripleta (5,59,35), (12,11,47), entre otras [3].

Generador Itamaracá (Ita)

La aparición de este algoritmo es bastante reciente, fue ideado en 2021 por *Daniel Henrique Pereira*, un estudiante de la *Universidad Federal de Minas Gerais* de Brasil. Este ha adquirido popularidad por ser el primer algoritmo GNPA que tiene la función de valor absoluto en su base.

A diferencia de los anteriores, este emplea tres (3) semillas para la generación de la sucesión. Dependiendo de un $N \in \mathbb{N}$ dado genera una cantidad de números deseados donde el valor máximo deseable de estos se encuentra dentro del rango 1 y N. Dadas tres semillas S_1, S_2, S_3 estas deben cumplir que $0 < S_i < N$ para todo i $\in \{1,2,3\}$. El procedimiento se divide en dos (2) pasos, P_n (*n-esimo* proceso) y cálculo final.

1. P_n **(*n-esimo* proceso):** En este paso se toma el valor absoluto de la suma de las diferencias entre siguientes semillas:

$$\begin{aligned} P_n &= |(S_{n+2} - S_{n+1}) + (S_{n+1} - S_n)| \\ &= |S_{n+2} - S_n| \end{aligned}$$

2. **Cálculo final:** Realizado el paso previo, se escoge arbitrariamente un valor fijo y racional con expansión finita muy cercano o igual a 2 denotado X. Ya con esta información, se obtiene el próximo valor de la sucesión calculando el valor absoluto de la diferencia entre N y la parte entera del producto entre P_n y X, es decir:

$$I_n = |N - \lceil (P_n \cdot X) \rceil|$$

   donde I_n es el $n-esimo$ valor de la sucesión de Itamaracá. Finalmente, se establece la semilla $S_{n+3} = I_n$ de tal forma que se pueda continuar con la siguiente iteración de ser requerido. [5].

En aras de ilustrar de una mejor manera este algoritmo, se plantea el siguiente ejemplo.

Sea $n = 200, S_1 = 37, S_2 = 100, S_3 = 178$ y $X = 2$. Como puede observar, se cumplen todas las condiciones necesarias para el algoritmo *(Los valores se han escogido arbitrariamente)*. Ahora, se desea obtener los tres (3) primeros números aleatorios de esta sucesión, para ello, se procede a ejecutar el algoritmo. Cabe recordar que estos números se generaran dentro del intervalo $[1,200]$.

Primeramente, se calcula P_1 de la siguiente manera:

$$\begin{aligned} P_1 &= |S_3 - S_1| \\ &= |178 - 37| \\ &= 141 \end{aligned}$$

Ya obtenido el primer proceso se realiza el cálculo final. Así,

$$\begin{aligned} I_1 &= |200 - \lceil (P_1 \cdot X) \rceil| \\ &= |200 - \lceil (141 \cdot 2) \rceil| \\ &= 82 \end{aligned}$$

Se ha obtenido el primer número aleatorio. Para continuar con la siguiente iteración, se designa $S_4 = I_1 = 82$. Bajo este mismo *modus operandi*,

$$\begin{aligned} P_2 &= |S_4 - S_2| \\ &= 18 \end{aligned}$$

y con ello,

$$\begin{aligned} I_2 &= |200 - \lceil (P_2 \cdot X) \rceil| \\ &= |200 - \lceil (18 \cdot 2) \rceil| \\ &= 164 \end{aligned}$$

De este modo, se ha obtenido el segundo número y se asigna $S_5 = I_2 = 164$. Por ultimo:

$$\begin{aligned} P_3 &= |S_5 - S_3| \\ &= 14 \end{aligned}$$

por lo que,

$$\begin{aligned} I_3 &= |200 - \lceil (P_3 \cdot X) \rceil| \\ &= |200 - \lceil (14 \cdot 2) \rceil| \\ &= 172 \end{aligned}$$

De esta forma se han obtenido I_1, I_2, I_3 números aleatorios correspondientes a $82, 164$ y 172. En el caso de precisar n números aleatorios, debe realizar las n iteración con sus respectivos pasos. El algoritmo es bastante sencillo y fácil de implementar, pero es bastante efectivo.

Es importante señalar que de necesitar valores $U(0,1)$ simplemente hace falta dividir cada uno por su rango, por ejemplo para el caso anterior:

$$\begin{aligned} U_I &= \left\{ \frac{82}{200}, \frac{164}{200}, \frac{172}{200} \right\} \\ &= \{0.41, 0.82, 0.86\} \end{aligned}$$

Tests de Aleatoriedad

Test de Kolmogorov-Smirnov

En 1933,*Kolmogorov* demostró que dada una distribución de probabilidad $F_0(X)$, la sucesión $\{F_n(X)\}$ de *distribuciones empíricas de probabilidad* dadas por:

$$F_n(X) = \frac{1}{n} \sum_{i=1}^{n} 1_{(-\infty, X_i)}(X) \quad \textbf{(1)}$$

Donde $\{X_1,...,X_n\}$ son números aleatorios independientes tomados de la distribución F_0 y 1_I denota la función característica del intervalo correspondiente, convergen uniformemente a F_0 [7].

Basados en esta convergencia, este test pretende medir la bondad de ajuste de una colección de datos con una distribución de probabilidad base, la cual debe ser una distribución continua [8]. En particular, los resultados aquí presentados comparan las sucesiones de números aleatorios, con la distribución uniforme entre 0 y 1 ($U(0,1)$).

Para producir el P-valor de este test, lo primero que se hace es crear una distribución empírica de probabilidad como en la ecuación **(1)**, y luego se calcula el estadístico D dado por:

$$\begin{aligned} D_N &= \sup_{x\in\mathbb{R}} |F_e(x) - F_0(x)| \\ &= \sup_{x\in\mathbb{R}} |F_e(x) - U_{(0,1)}(x)| \end{aligned}$$

Se tiene además que $P(\sqrt{n}D_n \leq z) \to L(z)$ cuando $n \to \infty$ donde,

$$L(z) = 1 - 2\sum_{y=1}^{\infty}(-1)^{y-1}e^{-y^2z^2} \qquad [9]$$

Ahora, considerando la cantidad de números aleatorios que se hayan tomado, se comparan $\sqrt{n}D_N$ con K_α, donde K_α es tal que $L(K_\alpha) = 1-\alpha$ y α es el parámetro de tolerancia que se desee, para los códigos aquí mostrados, se utiliza un valor de 0.05. Si se tiene que $\sqrt{n}D_n \leq K_\alpha$, se concluye que la muestra si está distribuida según $U(0,1)$ [10]. Para el cálculo del p-valor está dado por $1 - L(\sqrt{n}D_n)$ y es calculado usando el método *kstest* de scipy [11].

Implementación

En el siguiente apartado se estudiara el comportamiento de los $p-valores$ para cada uno de los generadores por medio de una simulación con la finalidad de determinar su nivel de confiabilidad. Este proceso de llevo a cabo generando 1000 sucesiones de números aleatorios de longitud 2000 en el rango $[1,10000]$ a partir de semillas aleatorias que satisfacen las condiciones de cada método y $X = 2.1$ para (Ita). Para mas detalles, remitirse a la referencia [18].

Para el Generador de *Merssene Twister* se obtuvo lo siguiente:

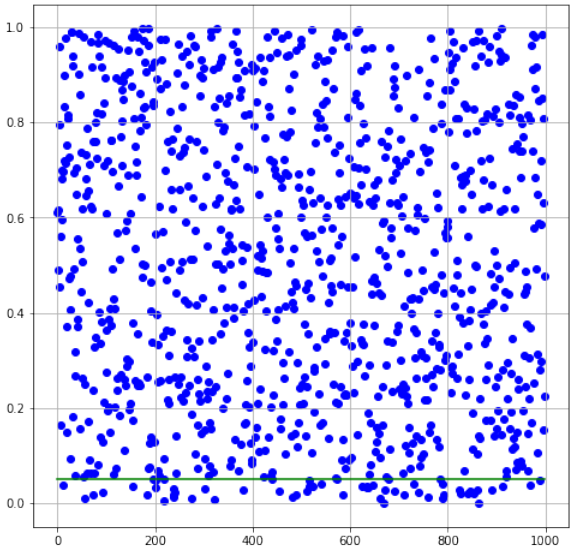

Figura 1. Simulación de *p valores* mediante el test para Merssene.

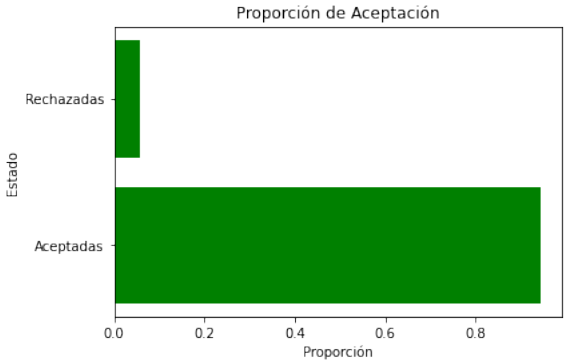

Figura 2. Gráfico de proporciones para Merssene.

Para el Generador de $Xorshift$ se obtuvo lo siguiente:

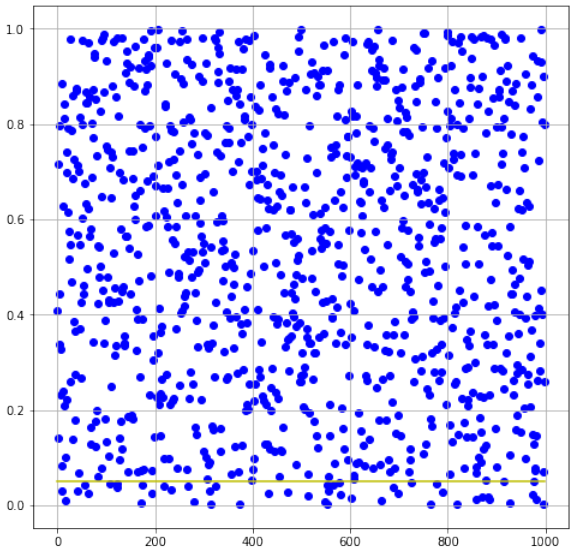

Figura 3. Simulación de *p valores* mediante el test para Xorshift.

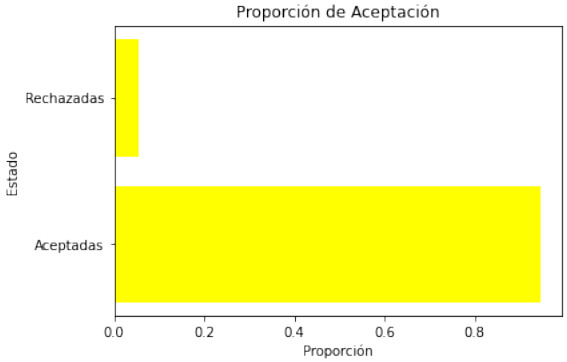

Figura 4. Gráfico de proporciones para Xorshift.

Para el Generador de *Ita* se dio lo siguiente.

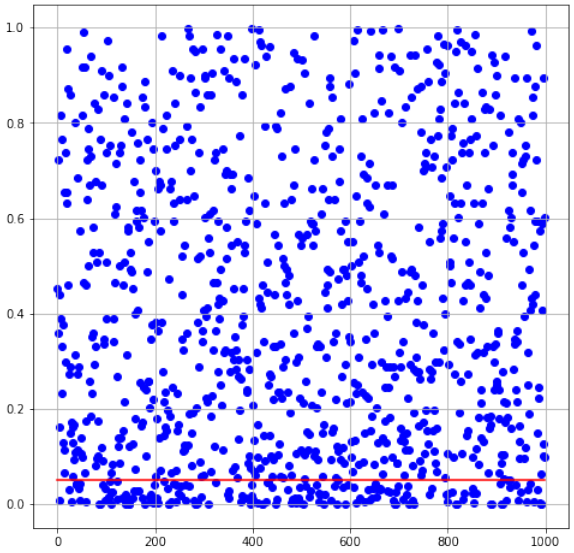

Figura 5. Simulación de *p valores* mediante el test para Ita.

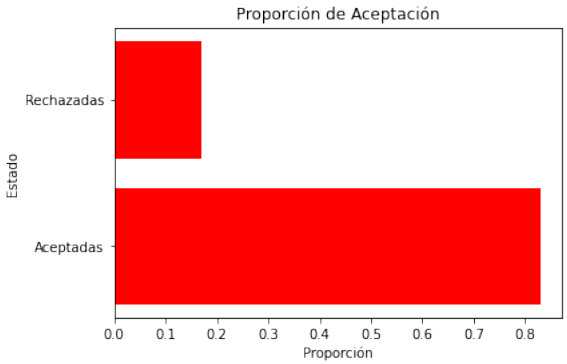

Figura 6. Gráfico de proporciones para Ita.

Test de la χ^2

Al igual que el test de *Kolmogorov-Smirnov*, este es un test no paramétrico el cual sirve para someter a prueba para decidir si una sucesión de números fue extraída de una distribución dada, que en nuestro caso es la $U(0,1)$. Con miras en este objetivo lo que se hace es comparar las frecuencias de los números con las frecuencias que se obtendrían de la distribución deseada [14]. Este proceso se lleva a cabo en n pasos:

1. Se toma el arreglo de n números aleatorios del intervalo $[0,1)$ y se hace el conteo de la cantidad en el intervalo $\left[\frac{k}{100}, \frac{k+1}{100}\right)$ para cada $k \in \{0,99\}$, a estas frecuencias las llamaremos f_k.*(Acá se utiliza estos intervalos, pero se podría usar cualquier partición de* $[0,1)$*).*

2. Se considera ahora un arreglo de frecuencias esperadas e_k y se calula el siguiente estadístico

$$\chi^2 = \sum_{i=0}^{99} \frac{(f_k - e_k)^2}{e_k} \quad [17]$$

A este estadístico se le denomina un χ^2, y se usa en distintos métodos. En este caso, $e_k = \frac{longituddelvector}{100}$ dado que pretendemos comparar los datos con la distribución uniforme.

3. Por último, el p-valor será tal que $Pr(x \geq \chi^2) = p$ donde la probabilidad es la de la distribución χ^2 [15].

Para determinar si la sucesión de números es aleatorio lo que hacemos es tomar comparar el p-valor con 0.05, si está por debajo **SI** se considera aleatoria y viceversa. Para facilitarnos el cálculo de p-valores existen tablas previamente establecidas que relacionan el estadístico con el mismo como la que se puede encontrar en [16].

Implementación

En el siguiente apartado se estudiará el comportamiento de los $p-valores$ para cada uno de los generadores por medio de una simulación con la finalidad de determinar su nivel de confiabilidad. Este proceso se llevó a cabo generando 1000 sucesiones de números aleatorios de longitud 2000 en el rango $[1, 10000]$ a partir de semillas aleatorias que satisfacen las condiciones de cada método y $X = 2.1$ para (Ita). Para más detalles, remitirse a la referencia [18].

Para el Generador de *Merssene Twister* se obtuvo lo siguiente:

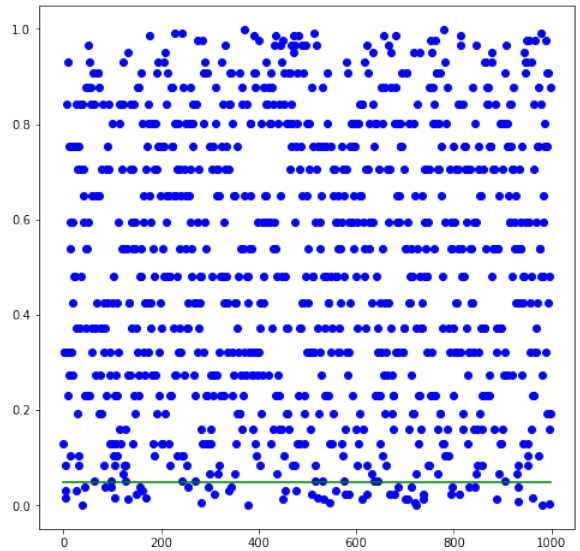

Figura 7. Simulación de *p valores* mediante el test para Merssene.

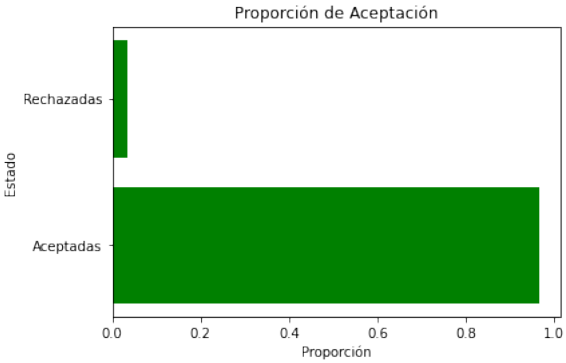

Figura 8. Gráfico de proporciones para Merssene.

Para el Generador de $Xorshift$ se obtuvo lo siguiente:

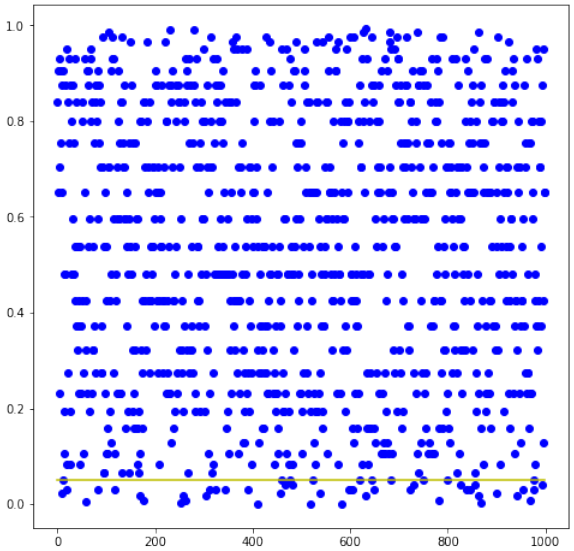

Figura 9. Simulación de *p valores* mediante el test para Xorshift.

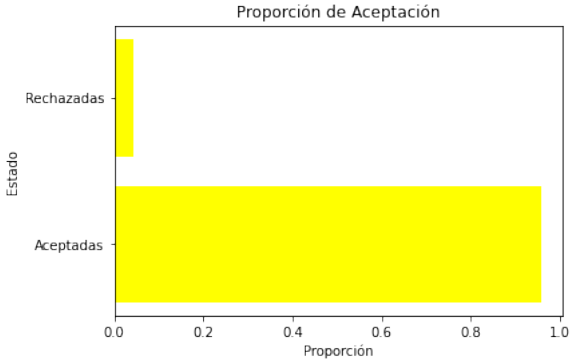

Figura 10. Gráfico de proporciones para $Xorshift$.

Para el Generador de *Ita* se dio lo siguiente.

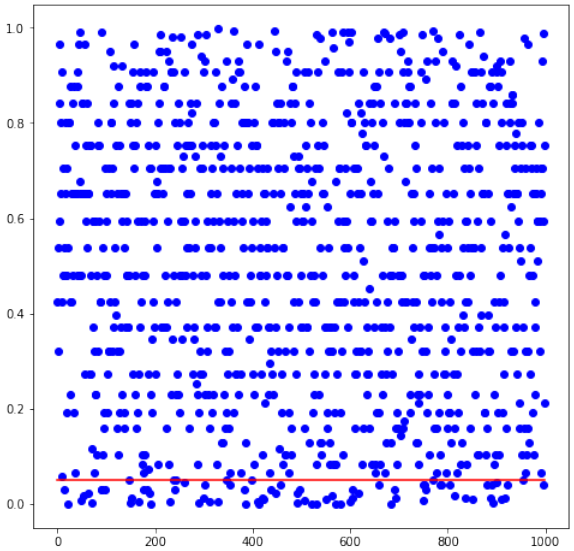

Figura 11. Simulación de *p valores* mediante el test para Ita.

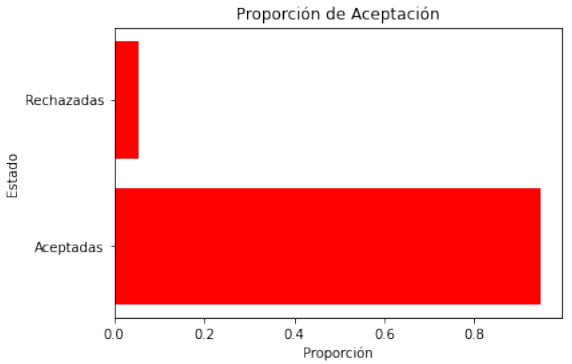

Figura 12.
Gráfico de proporciones para Ita.

Binary Matrix Rank Test

El *Binary Matrix Rank Test*, al contrario de los métodos previamente vistos que están diseñados para comparar los números con la distribución $U(0,1)$, está diseñado para aprovecharse de las posibles relaciones lineales entre los elementos de una cadena. En el artículo [6] *Bowman* expone que en una muestra de vectores binarios perfectamente aleatorios de dimensión k, rara vez forman estos conjuntos linealmente independientes. Cuando hay una relación lineal, esta resistencia a la independencia se rompe, causando que se pueda relacionar y predecir la secuencia, y por ende, no se tendría aleatoriedad.

Este test evalúa una cadena de ε de binarios en 5 pasos [1]:

1. Se toma la cadena $\varepsilon = \varepsilon_1, \varepsilon_2, ..., \varepsilon_n$ y se divide en $N = \left\lfloor \frac{n}{32*32} \right\rfloor$ sub-cadenas, luego con cada cadena se forman matrices R_i de 32×32 llenando por filas.

   Nota: *En este paso se podría estar descartando bits si* $\frac{n}{32*32}$ *no es un número entero.*

2. Se determina el rango de cada una de las R_i.
3. Se declaran F_M y F_{M-1} las cantidades de matrices que tienen rango 32 y 31 respectivamente; no se considera una variable para las de rango menor o igual que 30, dado que estas suelen ser pocas.
4. Se calcula la siguiente cantidad:

$$\chi^2(obs) = \frac{(F_M - 0.2888N)^2}{0.2888N} + \frac{(F_{M-1} - 0.5776N)^2}{0.5776N} + \frac{(N - F_M - F_{M-1} - 0.1336N)^2}{0.1336N}$$

5. Se calcula el *p-valor* con la expresión $P - valor = e^{-\chi^2(obs)/2}$

El test concluye que la secuencia de números **NO** es aleatoria si el *p-valor* está por debajo de 0.01. En caso contrario, se considera que la secuencia de números **SI** es aleatoria.

Implementación

En el siguiente apartado se muestran los resultados de implementar este test para el GNPA Xorshift dado que nos parece de interés como este test que había mostrado un buen comportamiento con respecto a los otros test falla catastróficamente al ser sometido a este. Específicamente estudiamos el comportamiento de los $p - values$ por medio de una simulación con la finalidad de determinar su nivel de confiabilidad. Este proceso se llevó a cabo generando 1000 sucesiones de números aleatorios de longitud 2000 a partir de semillas aleatorias. Para más detalles, remitirse a la referencia [18].

Para el Generador de $Xorshift$ se obtuvo lo siguiente:

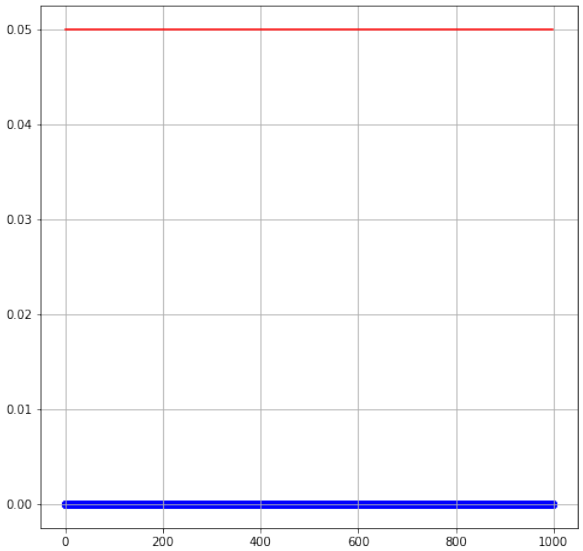

Figura 15. Simulación de *Binary Matrix.*

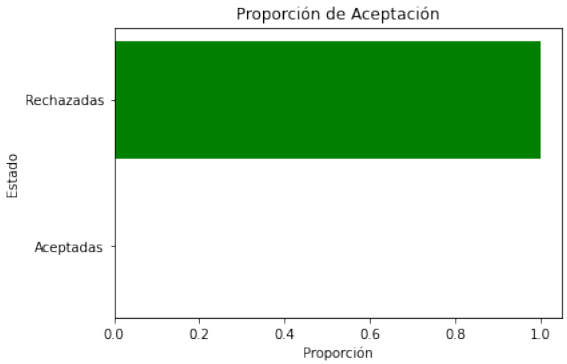

Figura 16. Gráfico de proporciones para Xorshift.

Conclusiones

Desde otro ángulo, después de las observaciones, se ha visto que el calificativo del método *"menos favorable"* se le atribuye al Generador de Ita, pues aunque genera un elevado número de cadenas que superan el test, a

comparación de los otros, sus proporciones resultan ser menores. El método resulta ser el más actual, razón por la cual no cuenta con muchos estudios e investigaciones sobre su algoritmo.

Por otro lado, cabe resaltar el peculiar comportamiento de la simulación para el test de χ^2 donde los $p-valores$ se estabilizaban sobre rectas horizontales equidistantes para la mayor parte de las cadenas aceptadas. Sin embargo, para cadenas cercanas y por debajo a la recta crítica estipulada $y = 0.05$, se presenta el mismo comportamiento de diáspora visto en el test de *Kolmogorov*, aparentemente se evidencia un comportamiento aleatorio en el mismo rechazo.

También se resalta el cómo según el objetivo con el que se implementen los números aleatorios se puede utilizar un método de generación en vez de otro; y la escogencia es importante, pues de tomar un método no adecuado podríamos causar malos funcionamientos o fallas en los sistemas. Por ejemplo, si se emplea Xorshift en un contexto de criptografía, un agente externo podría romper el código identificando la relación lineal entre los números generados y con ello encontrar las matrices correspondientes a cada paso de la iteración.

Asimismo, vale la pena resaltar que a pesar de que el generador Merssene es el más antiguo de los 3 se comporta como es esperado a la par de los otros e incluso en algunos casos tiene mejores resultados. Esto puede ser debido a esto que es utilizado como estándar en varios lenguajes de programación y tiene un algoritmo sólido y soportado por la amplia comunidad científica.

También, cabe señalar la gran versatilidad del Shell de *IPython* permitiendo integrar ámbitos estadístico, matemático e informático en un mismo ambiente de manera sencilla y potente.

Por último, se resalta que después de realizar las respectivas simulaciones, se pudo observar que los tres (3) métodos proporcionan secuencias de números aleatorios que satisfacen el criterio de distribución uniforme, comportándose, de acuerdo a lo esperado, claramente unos mejores que otros. También se observa como no hay una sola concepción de lo que significa *"aleatorio"*, pues cada uno de estos métodos de evaluación tiene capacidades y alcances distintos, su finalidad se enfoca sobre distintos aspectos como la uniformidad, la homogeneidad de distribución o la independencia lineal en la cadena generada.

References

1. Rukhin A.et.al.(2008)A Statistical test suite for random and pseudorandom number generators for cryptograpic applications. National Institute of Standarts and Technology. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf
2. Pulido, M. (2008). Generación de Números Aleatorios. Universidad de Murcia. https://webs.um.es/mpulido/miwiki/lib/exe/fetch.php?id=amio&cache=cache&media=wiki:simt1b.pdf
3. Marsaglia G.(2003) Xorshift RNGs. The Florida State University
4. Brent R.P. (2007) Some long-period random number generators using shifts and xors. ANZIAM Journal
5. Pereira, D. (2021). ITAMARACÁ: A NOVEL SIMPLE WAY TO GENERATE PSEUDO-RANDOM NUMBERS. Cambridge. https://www.cambridge.org/engage/api-gateway/coe/assets/orp/resource/item/61b410fadcbea24f839f0235/original/itamaraca-a-novel-simple-way-to-generate-pseudo-random-numbers.pdf.
6. Bowman K.et.ad.(2005) Linear independence in a random binary vector model. Clemson University.
7. Kolmogorov A. (1933) Sulla determinazione empirica di una legge di distribuzione. G. Ist. Ital. Attuari.
8. Feller W.(1948) On the Kolmogorov-Smirnov limit theorems for empirical distributions. Ann. Math. Statist.
9. Smirnov N. (1939) On the estimation of discrepancy between empirical curves of distribution for two different samples. Bulletin Mathématique de l'Université de Moscou, Vol. 2.
10. Stephanie Glen.(2022) Kolmogorov-Smirnov Goodness of Fit Test. From StatisticsHowTo.com: Elementary Statistics for the rest of us! https://www.statisticshowto.com/kolmogorov-smirnov-test/
11. The SciPy community (2022) scipy.stats.kstest. The API Reference. From: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_1samp.html#scipy.stats.ks_1samp
12. Matsumoto Makoto , Nishimura Takuji (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.
13. Jagannatam Archana , Tran Duyen .Mersenne Twister A Pseudo-Random Number Generator
14. Richard L.(2015) Concepts and Applications of Inferential Statistics. Online Statistic Textbook.
15. Blanco L.(2004) Probabilidad. Universidad Nacional de Colombia.
16. Pearson, K. (1900). X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 50(302), 157–175.

17. Sölpük Turhan, N (2020). Karl Pearson's chi-square tests. Educational Research and Reviews
18. Garavito S. Ruiz F. Villegas H.(2022) Colab. UNAL