

# **Modelos y simulación - Trabajo especial**

FAMAF - UNC

**Juliana García - Santiago López Pereyra**

June 14, 2025

# 1 Resumen

El objetivo de este trabajo es explorar el rendimiento de distintos generadores de números aleatorios en la simulación de procesos complejos. Como caso de estudio, se toma un sistema de colas FIFO de un solo servidor donde los arribos siguen un proceso de Poisson no homogéneo y tiempos de atención exponenciales. Los generadores elegidos fueron:

- Generador congruencial lineal (GCL)
- XOR Shift
- PCG

## 2 Descripción teórica de los generadores

### 2.1 GCL

El Generador Congruencial Lineal (GCL) genera números aleatorios a través de una fórmula recurrente:

$$y_{n+1} = (ay_n + c) \mod m$$

El  $k$ -ésimo número generado se corresponde con  $y_k$  y el número inicial  $y_0$  es la semilla.

Una ventaja clara es su simplicidad. Esto lo hace eficiente y fácil de implementar. Sin embargo, si no se eligen cuidadosamente los parámetros  $a$ ,  $c$ ,  $m$ , el generador puede producir periodos cortos. Esto es indeseable, dado que una vez que se conoce el periodo del generador, se conoce exactamente qué valor generará en cada iteración.

Un teorema visto en el teórico garantiza que la longitud del período máximo es  $m$  si y solo si:

- $c$  y  $m$  son coprimos,
- $a - 1$  es divisible por todos los factores primos de  $m$ ,
- $a - 1$  es divisible por 4 si  $m$  es múltiplo de 4.

Asumiendo que la elección de  $a$ ,  $c$  y  $m$  satisface las condiciones del teorema, el GCL es un generador muy bueno.

Para garantizar que las variables generadas son uniformes, se normalizan los  $y_n$  dividiéndolos por  $m$ . Como  $m > y_k$  para todo  $k$ , esta normalización es perfectamente lógica.

## 2.2 XorSHIFT

XorSHIFT denota en realidad una familia de generadores basados en operaciones bit a bit (XOR y desplazamientos). Dichas operaciones se realizan sobre una variable de estado interna. Es decir que, al igual que GCL, tiene una noción de estado y recurrencia. En nuestra implementación, dado un estado  $y_k$ , las operaciones realizadas son:

```
x = yk      & 0xFFFFFFFF
x ^= (x << 13) & 0xFFFFFFFF
x ^= (x >> 17) & 0xFFFFFFFF
x ^= (x << 5)  & 0xFFFFFFFF
```

La operación  $\& 0xFFFFFFFF$  asegura que las operaciones se mantengan dentro del rango de 32 bits. Las tres operaciones siguientes desplazamientos, mezclando los bits de  $y_k$  y produciendo un nuevo número  $\widetilde{y_{k+1}}$  (en binario). Finalmente, la última operación normaliza  $\widetilde{y_{k+1}}$  haciendo

$$y_{k+1} = \frac{\widetilde{y_{k+1}}}{2^{32} - 1}$$

dando el nuevo valor generado, que ahora pasará a ser el estado.

Las constantes 13, 17 y 15 no son teóricas: se eligieron porque la experimentación empírica mostró que producen buenas propiedades estadísticas. El periodo de XorSHIFT depende del tamaño (en bits) del estado y en general es largo. Al operar en tan bajo nivel, es extremadamente eficiente.

## 2.3 PCG

El Generador Congruencial Permutado (PCG) es una mejora de los GCL tradicionales. Consiste en aplicar una permutación sobre los bits del output de un GCL. Aunque el algoritmo es simple, es más complejo que los otros y por lo tanto es más lento, pero según investigamos es más confiable.

Nosotros usamos la variante PCG-XSH-RR que es la más recomendada en general. En el algoritmo se mantiene una variable global entera de 64 bits, el estado, que va evolucionando con cada iteración. Entonces tomamos:

$$X_{n+1} = aX_n + c \mod 2^{64}$$

Donde  $X_n$  son los bits 27 a 58 del estado (32 bits). Luego se usan los bits 59 al 63 (5 bits) para rotar  $X_n$ .

$$\text{output} = \text{rot32}(\text{state} \gg 27, \text{state} \gg 59)$$

Y los bits restantes son usados para evolucionar el estado.

Esto proporciona una distribución de salida más uniforme y pasa muchas más pruebas estadísticas que los GCL clásicos o incluso que XorSHIFT. Además, mantiene eficiencia computacional y es fácil de implementar.

El valor resultante se normaliza al intervalo  $[0, 1]$  del mismo modo que en XOR-Shift:

$$u = \frac{\text{output}}{2^{32} - 1}$$

### 3 Descripción del problema

Sea

$$\lambda(t) = 20 + 10 \cos\left(\frac{\pi t}{12}\right) \quad (1)$$

Se desea simular un sistema de colas de un solo servidor, donde los arribos siguen un proceso de Poisson no homogéneo con intensidad  $\lambda(t)$  y tiempos de atención exponenciales con media  $\mu = 35$  clientes/h. El sistema atiende por orden de llegada y no hay límite a la cantidad de elementos en una cola.

#### 3.1 Caracterizando propiedades simples del sistema

Como el coseno oscila en  $[-1, 1]$ ,  $\lambda(t)$  tiene máximo 30 y mínimo 10. Más aún,  $\pi\left(\frac{\pi t}{12}\right)$  completa un ciclo cuando  $\pi t/12 = 2\pi \iff t = 24$ . Se sigue que en  $t = 12$  alcanza su mínimo (mitad del ciclo recorrido).

Nos interesa caracterizar los períodos donde el servidor tendrá mayor y menor actividad. Los caracterizaremos como las regiones de  $t$  en que  $\lambda(t)$  está por encima y por debajo de su punto medio, respectivamente. No es difícil ver que

$\lambda(t) > 20 \iff \cos(\pi t/12) > 0$ . Pero el coseno es positivo si su argumento pertenece a  $[-\pi/2, \pi/2]$ . Por ende,

$$\lambda(t) > 20 \iff -\pi/2 + 2k\pi \leq \frac{\pi t}{12} \leq \pi/2 + 2k\pi \quad (2)$$

$$\iff -6 + 24k \leq t \leq 6 + 24k \quad (3)$$

Si restringimos  $t \in [0, 48]$ , esto vale si y solo si

$$t \in (0, 6) \cup (18, 30) \cup (42, 48) \quad (4)$$

El complemento de este conjunto sobre el universo  $[0, 48]$  nos da los periodos de menor actividad. El valor medio de llegadas en las 48 horas es:

$$\int_0^{48} \lambda(t) dt = \int_0^1 20 + 10 \cos\left(\frac{\pi t}{12}\right) dt = 960 \quad (5)$$

Esto implica que  $\frac{960}{48} = 20$  es el valor medio de llegadas por hora. Incluso en periodos de máxima actividad, la cantidad esperada de llegadas por hora es prácticamente la misma:

$$\frac{1}{6} \int_0^6 \lambda(t) dt = 21 \quad (6)$$

Como se atiende 35 personas por hora, esto significa que incluso en los periodos de mayor actividad se espera que el servidor atienda a todas las personas.

### 3.2 Método de simulación

La simulación consiste primero en generar el proceso no homogéneo que representa la llegada de clientes a la cola. Esto es logrado a través de la función

`poisson_no_homogeneo(T, generator, ...)`

que simula el proceso de Poisson no homogéneo usando un algoritmo generador de números aleatorios dado. La simulación del proceso de Poisson usa el método de adelgazamiento visto en clase. Una vez las llegadas (`arrivals`) se han simulado, la función

`simular_cola(arrivals, mu, generator)`

simula la atención en la cola de un servidor FIFO con los parámetros deseados (e.g. tiempo de atención exponencial con media  $\mu$  clientes por hora). La función `main` realiza la simulación con las funciones antedichas y organiza los resultados en una base de datos bien estructurada.

## 4 Metodología

El experimento se implementó en Python utilizando librerías para el análisis y el gráfico de los datos. La simulación y los generadores fueron implementados en archivos `.py`, pero ejecutados y analizados en una notebook de Jupyter titulada `Graficos.ipynb`. Los gráficos se realizaron utilizando `seaborn`, porque consideramos que produce gráficos más bonitos que `matplotlib` crudo.

## 5 Tests de aleatoriedad

Dimos una implementación propia del test de Kolmogorov-Smirnov. La misma fue utilizada en muestras de  $N_s = 1000$  elementos generadas por cada generador. En cada prueba, se simularon  $N_{\text{sim}} = 1000$  uniformes utilizando el módulo `random` de Python para dar con un  $p$ -valor simulado.

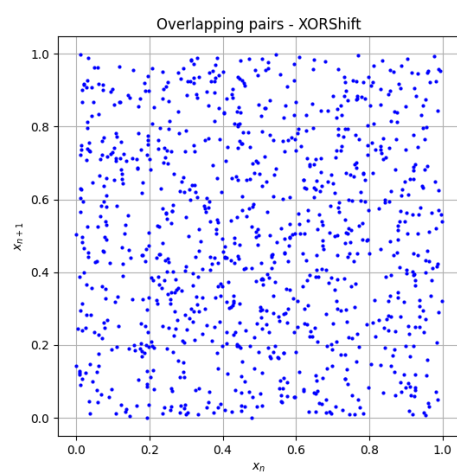
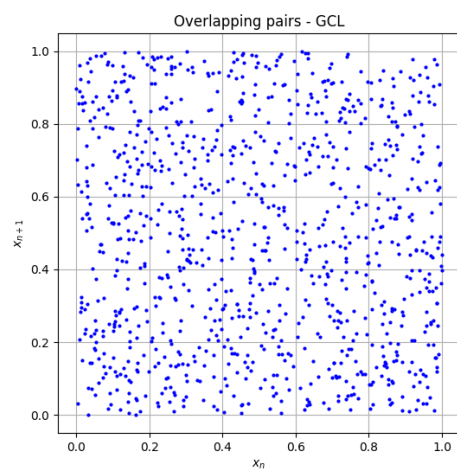
También se utilizaron plots de pares independientes. Con cada método, se generaron  $N = 1000$  valores  $x_1, \dots, x_N$  y se graficaron los puntos

$$(x_1, x_2), (x_2, x_3), \dots, (x_{N-1}, x_N), (x_N, x_1)$$

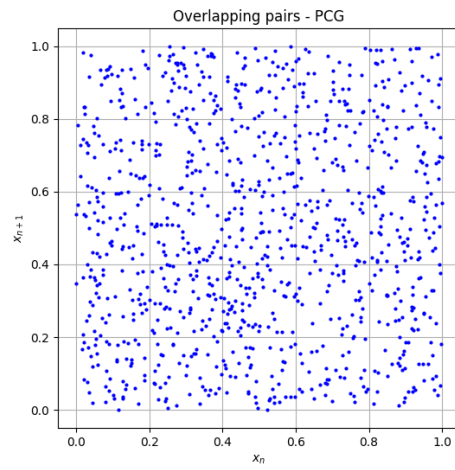
Se observó si dichos puntos rellenaban el cuadrado unitario con la dispersión adecuada y sin mistrar patrones o sesgos.

## 6 Resultados

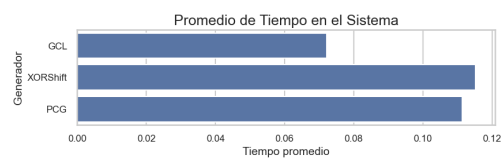
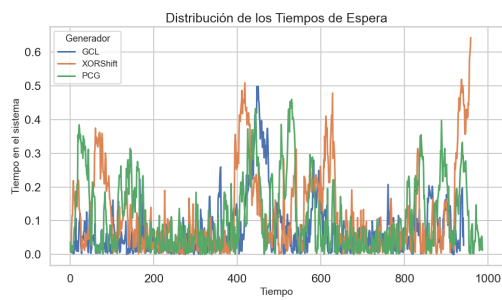
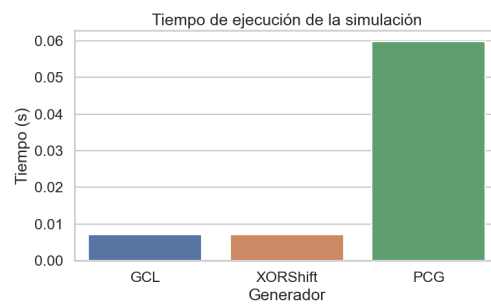
### 6.1 Tests de aleatoriedad

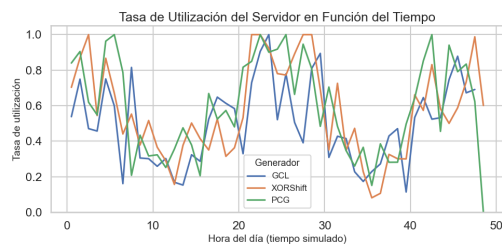
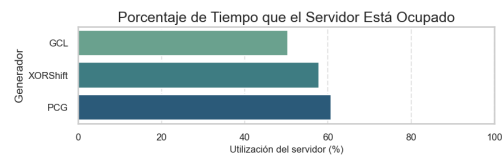
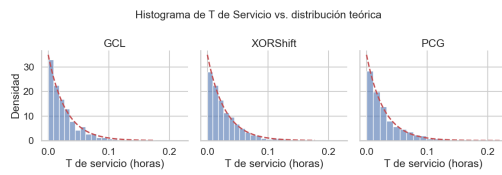
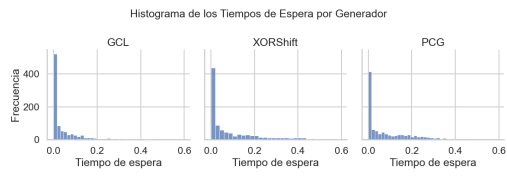
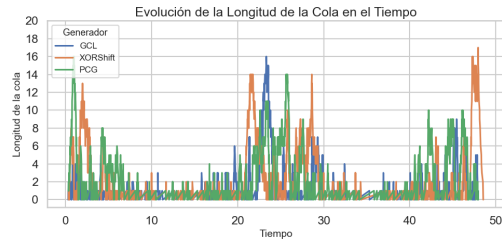






## 6.2 Simulación





## 7 Conclusiones

Tal como esperábamos por lo observado en la sección 3.1, en cada simulación fueron atendidos la mayor parte de los clientes. Además en los gráficos de la evolución de la cola en el tiempo y de la tasa de utilización del servidor en el tiempo se puede ver claramente como varían los gráficos con las horas del día, que va de acuerdo a la distribución de la llegada de clientes que buscábamos simular.

Encontramos diferencias poco significativas en el efecto del generador elegido sobre los resultados de la simulación. Sin embargo, mientras que las simulaciones bajo GCL y XORShift tomaron aproximadamente la misma cantidad de tiempo (0.005668 y 0.004248, respectivamente), la simulación bajo PCG fue significativamente más lenta (0.015764) que era lo que esperábamos.

Algunos datos como el tiempo de ejecución pueden variar en el archivo .ipynb por haberlo ejecutado luego de generar los gráficos del informe.

Finalmente consideramos importante señalar que los resultados fueron obtenidos con una semilla única para todos los generadores. Hicimos algunas pocas pruebas manuales comparando el comportamiento con distintas semillas y los resultados fueron significativamente diferentes sobre algunas variables como la distribución del tiempo de espera, el promedio del tiempo en el sistema y el porcentaje de tiempo en que el servidor está ocupado. Además el p-valor de los generadores oscilaba entre 7.5 y 0.1 con distintas semillas e incluso llegamos a obtener 0.001 en XORShift, que es suficiente para rechazarlo como generador. Un valor que se mantuvo relativamente constante fue el tiempo de ejecución de la simulación.

Recomendamos para trabajos posteriores correr la simulación con varias semillas distintas y comparar las medias y las desviaciones de las distintas variables para poder sacar mejores conclusiones sobre los generadores y analizar si estos son sensibles al cambio de la semilla.

## 8 Código Fuente

El código del proyecto se puede encontrar en el archivo .zip adjunto en el aula junto con el informe o en el siguiente [link](#) de nuestro repositorio