UNIVERSIDAD AUSTRAL | INGENIERÍA

Maestría en Ciencia de Datos 2024/2025

Simulación y Optimización en Ciencia de Datos

Trabajo Práctico Integral

Profesores:

- DEL ROSSO, Rodrigo
- NUSKE, Ezequiel

Integrantes:

- CANCELAS, Martín
- FILIPUZZI, Juan Manuel
- GALLARDO, Ezequiel
- NICOLAU, Jorge

Introducción

Este Trabajo Práctico (TP) integra los contenidos vistos en las clases

- **Unidad I** – Generación de números pseudoaleatorios y Monte Carlo
- **Unidad II** – Bayes, cadenas de Markov y Metropolis–Hastings
- **Unidad III** – Simulación de eventos discretos (SED)
- **Unidad IV** – Procesos continuos (NHPP, CTMC, SDE)
- **Unidad V** – Reacciones químicas estocásticas: Gillespie SSA y Next Reaction Method

El objetivo es que el alumno implemente técnicas de simulación, compare métodos, valide sus resultados y presente visualizaciones claras.

Sobre el código fuente de este trabajo

Para facilitar la lectura y comprensión del trabajo, el código fuente completo de los graficos, tablas se encuentran disponibles en el siguiente repositorio de GitHub: https://github.com/georgsmeinung/tpi-simulacion/ en el Notebook Jupyter tpi-simulation-in-data-science.ipynb

Mientras que el el código base (originalmente en R, transformado a Python) se puede encontrar en el Anexo I de este documento.

Configuración

Para el presente trabajo se utilizan los siguientes parámetros globales y semillas, además los parámetros específicos del renderizado de gráficos. Para claridad estos últimos se encuentran en el Anexo de Código Fuente de Gráficos.

```
# CONFIGURACIÓN GENERAL DE LAS SIMULACIONES
# Parámetros (LCG estándar de C++ minstd_rand)
lgc_a, lgc_c, lgc_m = 48271, 0, 2**31 - 1
# Semilla
rnd_seed = 2371
```

Asimismo, se utilizan las siguientes librerías

```
import arviz as az
import heapq
import matplotlib.animation as animation
import matplotlib.font_manager as fm
import matplotlib.pyplot as plt
import numpy as np
import random
import scipy.stats as stats

from IPython.display import Image, display, Markdown
from matplotlib.collections import LineCollection, PolyCollection
from mpl_toolkits.mplot3d import Axes3D
```

Parte 1 - Generación de Números Pseudoaleatorios y Monte Carlo

Implementación de un Generador Congruencial Lineal (LCG)

La implementación de un Generador Congruencial Lineal (LCG) se fundamenta en un algoritmo iterativo y determinista regido por la relación de recurrencia

$$X_{n+1} = (aX_n + c) \mod m$$

mediante la cual se produce una secuencia de números pseudoaleatorios a partir de un valor inicial denominado semilla (X_0). El proceso requiere la definición de tres constantes enteras —el multiplicador (a), el incremento (c) y el módulo (m)— y opera calculando el siguiente valor de la serie al multiplicar el estado actual por a, sumar c y obtener el residuo de la división por m, resultando en una sucesión periódica que, si bien carece de aleatoriedad verdadera, es computacionalmente eficiente y totalmente reproducible si se mantienen los mismos parámetros iniciales.

Al usar operaciones aritméticas básicas, es extremadamente rápido computacionalmente. Como hay un número finito de resultados posibles (de 0 a $m-1$), la secuencia eventualmente se repetirá. A esto se le llama el "periodo". Por otra parte si se usa la misma semilla (X_0), se obtendrá exactamente la misma secuencia de números. Esto es útil para "debugging" o reproducir simulaciones científicas, pero malo para la seguridad. No es seguro criptográficamente: No se debe usar para contraseñas o claves de seguridad, ya que es relativamente fácil predecir los siguientes números analizando la secuencia previa.

```
# Implementación del LCG
def lcg_generator(seed, a, c, m, n):
    numbers = []
    x = seed
    for _ in range(n):
        x = (a * x + c) % m
        # Normalizar a [0, 1]
        numbers.append(x / m)
```

```
    return numbers
```

Con este algoritmo generamos una secuencia de números pseudoaleatorios uniformemente distribuidos en el intervalo [0, 1).

```
datos = lcg_generator(
    seed=rnd_seed, a=lgc_a, c=lgc_c, m=lgc_m, n=5000
)
```

Tabla 1 - Primeros 10 números generados

Índice	Valor
01	0.053295
02	0.611937
03	0.807027
04	0.010338
05	0.022969
06	0.754449
07	0.007679
08	0.677888
09	0.350518
10	0.868023

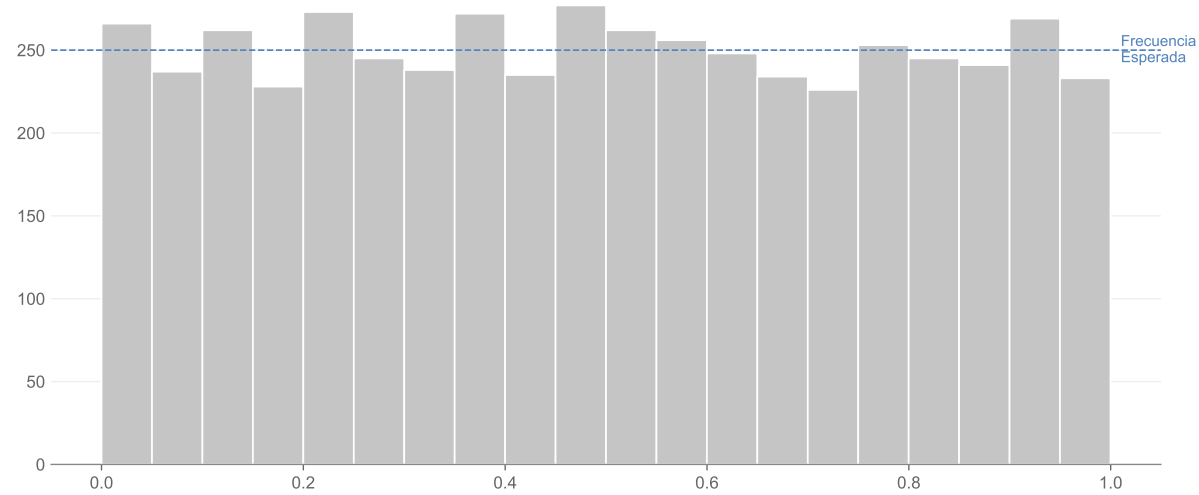
Para evaluar la calidad de los números pseudoaleatorios generados por uel LCG, es necesario verificar dos propiedades fundamentales: Uniformidad e Independencia. A continuación, se detalla la lógica de implementación y el código en Python para las tres herramientas solicitadas.

Histograma

El histograma visualiza la distribución de frecuencia de los números generados. Para un buen generador, esperamos una distribución uniforme (plana), lo que significa que cada intervalo del rango tiene aproximadamente la misma probabilidad de contener un número.

Figura 1 - Distribución de Uniformidad del Generador LCG

Muestra de 5000 números generados pseudoaleatoriamente



La presencia de alturas similares en todas las barras (“bins”) del histograma indica que el Generador Congruencial Lineal (LCG) está cumpliendo satisfactoriamente con la propiedad de uniformidad, lo que significa que cada sub-intervalo del rango tiene una probabilidad casi idéntica de contener un número generado. Esta distribución “plana” o rectangular sugiere que el algoritmo recorre el espacio muestral sin sesgos evidentes ni favoritismos hacia ciertos valores; sin embargo, es crucial notar que estas variaciones leves deben ser producto del azar natural (pequeñas fluctuaciones son esperadas y saludables), y aunque este patrón valida la equiprobabilidad, por sí solo no garantiza la independencia de los datos (ausencia de patrones secuenciales), por lo que un histograma visualmente equilibrado es una condición necesaria, pero no suficiente, para aprobar un generador.

Runs Test (Prueba de Rachas/Corridas)

Esta prueba verifica la independencia analizando la secuencia de oscilaciones. Una “racha” es una secuencia ininterrumpida de números crecientes o decrecientes.

Para la lógica de Implementación (con Corridas Arriba/Abajo) se toma la secuencia $u_1, u_2, \dots, u_n$. Se crea una nueva secuencia binaria basada en si el valor actual es mayor o menor que el anterior ($+$ si $u_i < u_{i+1}$, $-$ si $u_i > u_{i+1}$). Luego se cuenta el número total de rachas (cambios de $+$ a $-$ o viceversa). Se calcual el estadístico Z comparando el número de rachas obtenido con el esperado teóricamente usando la aproximación normal para muestras grandes ($N > 20$).

$$\mu_R = \frac{2N - 1}{3}$$

$$\sigma_R^2 = \frac{16N - 29}{90}$$

$$Z = \frac{R - \mu_R}{\sigma_R}$$

Si el valor absoluto de Z ($|Z|$) es mayor que el valor crítico (ej. 1.96 para un 95% de confianza), se rechaza la hipótesis de independencia.

Figura 2 - Prueba de Rachas (Runs Test)

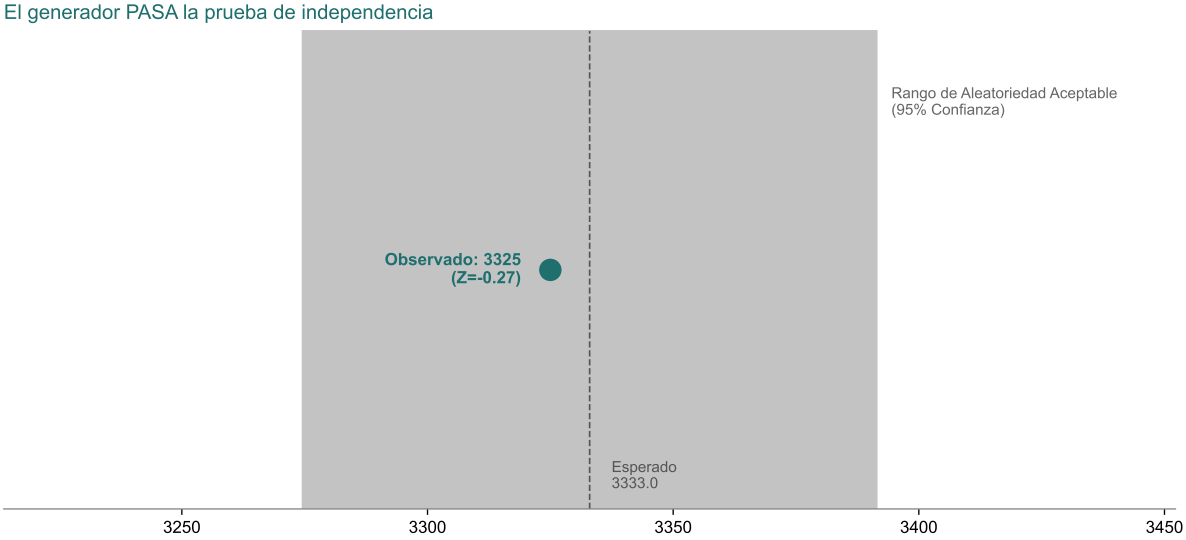


Tabla 2 - Prueba de Rachas

Métrica	Valor
Rachas Observadas	3325.00

Métrica	Valor
Rachas Esperadas (μ)	3333.00
Desviación Std (σ)	29.81
Z-Score	-0.2684

RESULTADO: ALEATORIO *(No se rechaza H0 al 95%)*

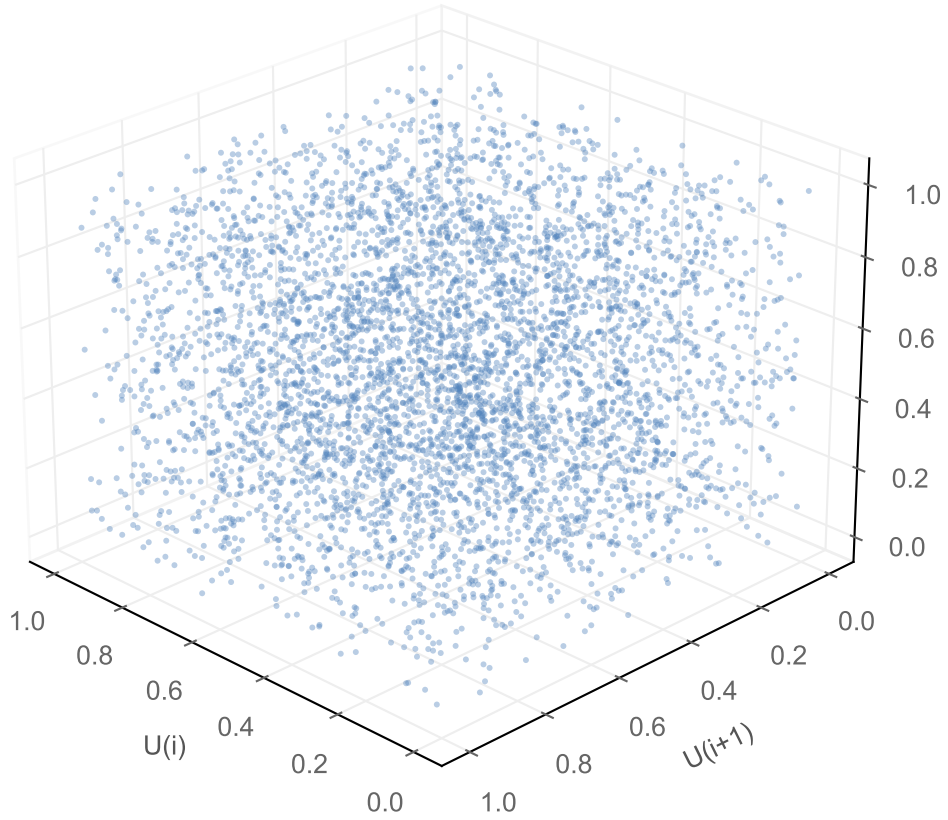
La cercanía entre las rachas observadas y las esperadas revela una discrepancia numérica mínima, lo cual sugiere que el Generador Congruencial Lineal (LCG) cumple satisfactoriamente con la propiedad de independencia estadística. Este resultado indica que la secuencia de números fluctúa (asciende y desciende) con una frecuencia consistente con el azar puro, descartando la presencia de patrones de dependencia serial; al no existir una desviación significativa (la diferencia es muy pequeña considerando el tamaño de la muestra), el valor estadístico Z resultante sería muy cercano a cero, lo que impide rechazar la hipótesis nula y permite concluir que los datos no están correlacionados entre sí, validando el motor como un generador robusto en términos de oscilación.

Gráfico de Triples (Spectral Test Visual)

Esta es una visualización en 3D para detectar correlaciones seriales. Los generadores LCG malos tienden a concentrar los puntos en planos discretos en lugar de llenar el espacio uniformemente (fenómeno conocido como planos de Marsaglia).

El gráfico de triples opera bajo la lógica de visualización del espacio de fases para detectar correlaciones seriales de largo alcance, mapeando tríadas consecutivas de la secuencia generada (u_n, u_{n+1}, u_{n+2}) como coordenadas espaciales (x, y, z) en un cubo tridimensional. La premisa fundamental es que, si los números fueran verdaderamente independientes, los puntos deberían llenar el volumen de manera caótica y uniforme como una “nube de polvo”; sin embargo, debido a la naturaleza lineal de la fórmula $X_{n+1} = (aX_n + c)$, los LCGs deficientes exhiben una estructura cristalina donde los puntos se alinean rígidamente en un número finito de planos paralelos (fenómeno conocido como planos de Marsaglia), revelando visualmente que la aleatoriedad es solo aparente y que existen dependencias matemáticas estrictas entre valores sucesivos.

Figura 3 - Correlación Serial: Gráfico de Triples



La observación de una nube de puntos dispersa y volumétrica que llena el cubo de manera homogénea se interpreta como una validación exitosa de la calidad espectral del generador, indicando que la correlación serial entre ternas consecutivas (u_n, u_{n+1}, u_{n+2}) es despreciable o inexistente para fines prácticos. Visualmente, esto contrasta con los generadores deficientes que agrupan los puntos en "rebanadas" o planos paralelos separados; por el contrario, una distribución uniforme implica que el algoritmo posee un periodo lo suficientemente largo y unos parámetros adecuados para "romper" la estructura reticular visible, garantizando que no existen dependencias geométricas fuertes y haciendo al generador apto para simulaciones Monte Carlo multidimensionales.

Estimación de π por Monte Carlo

La estimación del valor de π mediante el método de Monte Carlo utiliza el Generador Congruencial Lineal (LCG) para producir pares de coordenadas (x, y) uniformemente distribuidas en el intervalo $[0, 1)$, simulando el lanzamiento aleatorio de puntos sobre un cuadrado unitario que contiene un cuarto de círculo inscrito. El procedimiento se fundamenta en la geometría probabilística: dado que el área del cuarto de círculo es $\pi/4$ y el área del cuadrado es 1, la probabilidad de que un punto caiga dentro del círculo es exactamente $\pi/4$; por consiguiente, al verificar cuántos puntos cumplen la condición $x^2 + y^2 \leq 1$ y dividir esa cantidad por el número total de puntos generados, se obtiene una proporción que, multiplicada por 4, aproxima el valor de π, sirviendo esto a su vez como una prueba funcional de la calidad del LCG, ya que un generador sesgado o correlacionado (como el que forma líneas en el gráfico de triples) arrojará un valor de π incorrecto al no cubrir el área uniformemente.

Lógica de Implementación 1. Generación de Pares: Se utiliza el LCG para obtener dos números consecutivos normalizados (u_i, u_{i+1}) que actúan como coordenadas x e y. 2. Condición Geométrica: Se calcula la distancia al origen ($d = x^2 + y^2$). 3. Conteo: Si $d \leq 1$, el punto está "dentro" del círculo

(Acierto). 4. Cálculo Final: $\pi \approx 4 \times \frac{\text{Aciertos}}{\text{Total de Puntos}}$.
En términos de Pyhton:

```
def simular_pi_montecarlo(n_puntos):
    # Asumimos variables globales existen
    s, a, c, m = rnd_seed, lgc_a, lgc_c, lgc_m

    # Generamos el doble de puntos
    lista_completa = lcg_generator(
        s, a, c, m, n_puntos * 2
    )
    iterador_numeros = iter(lista_completa)

    inside_x, inside_y = [], []
    outside_x, outside_y = [], []
    dentro_count = 0

    for _ in range(n_puntos):
        x = next(iterador_numeros)
        y = next(iterador_numeros)

        if x**2 + y**2 <= 1.0:
            dentro_count += 1
            inside_x.append(x)
            inside_y.append(y)
        else:
            outside_x.append(x)
            outside_y.append(y)

    pi_estimado = 4 * (dentro_count / n_puntos)

    return (
        pi_estimado,
        inside_x, inside_y,
        outside_x, outside_y
    )
```

Así, utilizamos la implemntación de estimación de arriba en el siguiente código:

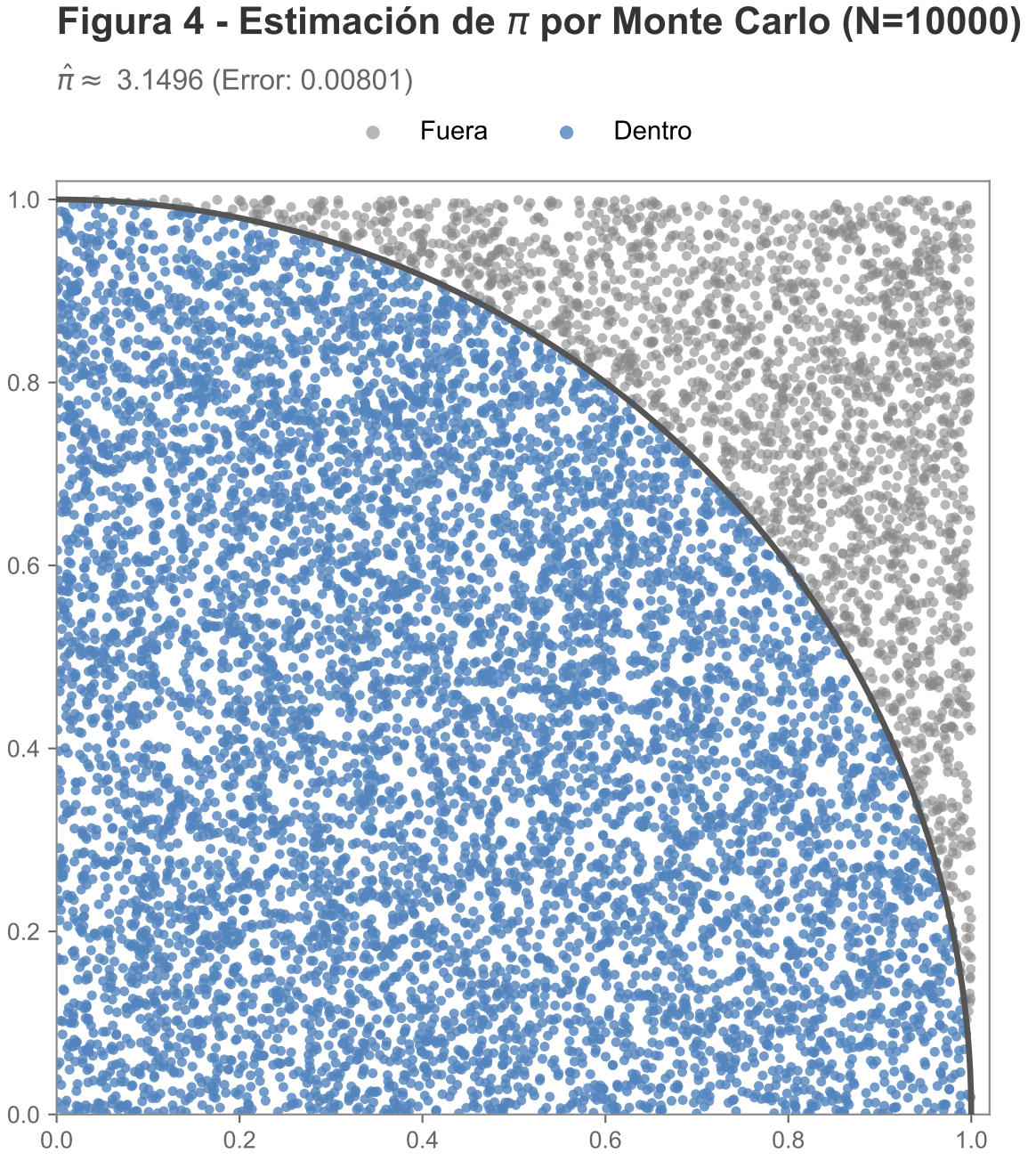
```
# Cantidad de puntos para la estimación
n = 10000

# Ejecutar cálculo
resultado = simular_pi_montecarlo(n)
(
    pi_est,
    in_x, in_y,
    out_x, out_y
) = resultado

# Calcular el error porcentual
```

```
error_pi = abs(np.pi - pi_est) / np.pi * 100
```

Tabla 3 - Resultado Estimación π

Métrica	Valor
Valor Final Estimado	3.1496
Error Porcentual	0.254882 %

Figura 4 - Estimación de π por Monte Carlo (N=10000)



Parte 2 - Metropolis–Hastings y Bayesian Inference

La relación entre el algoritmo de Metropolis-Hastings y la inferencia bayesiana es fundamentalmente instrumental, donde el primero actúa como la solución computacional para los desafíos analíticos plantea-

dos por el segundo. La inferencia bayesiana busca estimar la distribución posterior de parámetros desconocidos, denotada como $P(\theta|D)$, mediante la aplicación del Teorema de Bayes. No obstante, este proceso conlleva frecuentemente el cálculo de una constante de normalización —la evidencia marginal— que implica resolver integrales multidimensionales analíticamente intratables, lo cual impide la obtención directa de la distribución posterior en modelos complejos.

El algoritmo de Metropolis-Hastings, perteneciente a la familia de métodos de Monte Carlo vía Cadenas de Markov (MCMC), se distingue por su capacidad para generar muestras de una distribución de probabilidad objetivo sin necesidad de conocer su constante de normalización. El algoritmo opera construyendo una cadena de Markov que converge asintóticamente a la distribución deseada, requiriendo únicamente una función que sea proporcional a dicha densidad objetivo para evaluar los ratios de aceptación de las muestras propuestas.

En consecuencia, la conexión crítica reside en que la distribución posterior no normalizada en la estadística bayesiana es proporcional al producto de la función de verosimilitud y la distribución a priori ($Likelihood \times Prior$). Dado que Metropolis-Hastings puede operar bajo condiciones de proporcionalidad, este algoritmo permite muestrear la distribución posterior y realizar inferencias sobre los parámetros sin tener que calcular la integral de la evidencia marginal, haciendo viable el análisis bayesiano en escenarios de alta dimensionalidad donde las soluciones cerradas son imposibles.

Posterior beta analítica

Sea una moneda con 10 lanzamientos y 7 caras, para obtener una posterior Beta$(8, 4)$ a partir de 7 caras ($k = 7$) y 3 cruces ($n - k = 3$), debemos asumir un prior Uniforme o Beta$(1, 1)$. En la inferencia Bayesiana, demode que si se usa un Prior Beta y un Likelihood Binomial, el Posterior siempre es otra Beta:

- Likelihood (Verosimilitud): $P(X|\theta) \propto \theta^7(1-\theta)^3$
- Prior (A priori): $P(\theta) \sim$ Beta$(1, 1) \propto 1$
- Posterior (A posteriori): $P(\theta|X) \propto \theta^{7+1-1}(1-\theta)^{3+1-1} = \theta^7(1-\theta)^3 \rightarrow$ Beta$(8, 4)$

En términos de Python:

```python
# Funciones del modelo

def log_prior(theta):
    """
    Prior Beta(1,1) (Uniforme en [0, 1]).
    Retorna 0 si está en rango (log(1)), -inf si no.
    """
    if 0 <= theta <= 1:
        return 0.0
    return -np.inf

def log_likelihood(theta, heads, trials):
    """
    Log-Likelihood Binomial.
    """
    if theta < 0 or theta > 1:
        return -np.inf

    # Proporcional a theta^k * (1-theta)^(n-k)
    term_heads = heads * np.log(theta)
    term_tails = (trials - heads) * np.log(1 - theta)
```

```
    return term_heads + term_tails

def log_posterior(theta, heads, trials):
    lp = log_prior(theta)

    if not np.isfinite(lp):
        return -np.inf

    return lp + log_likelihood(theta, heads, trials)
```

Como el objetivo es obtener exactamente Beta$(8, 4)$ con 7 caras y 3 cruces, implícitamente se está asumiendo que no se aporta información previa (el 1 inicial de la Beta actúa como un “neutro” o punto de partida cero en términos de influencia).

Configuramos el problema en Python:

```
# CONFIGURACIÓN DEL PROBLEMA
# Establecer semilla
np.random.seed(rnd_seed)

# Datos observados: 10 lanzamientos, 7 caras
n_trials = 10
n_heads = 7

# Parámetros del algoritmo MCMC
n_chains = 4      # Solicitado para diagnóstico R-hat
n_samples = 5000  # Muestras por cadena
burnin = 1000     # Periodo de calentamiento
step_size = 0.1   # Desviación estándar (Proposal width)

# DEFINICIONES AUXILIARES
def log_posterior(theta, heads, trials):
    """
    Calcula el log-posterior (Likelihood Binomial + Prior Uniforme).
    Necesario para el algoritmo MH.
    """
    # Prior Uniforme [0, 1]
    if theta < 0 or theta > 1:
        return -np.inf

    # Log-Likelihood Binomial
    # P(D|theta) = theta^h * (1-theta)^(n-h)
    ll = (heads * np.log(theta) +
          (trials - heads) * np.log(1 - theta))
    return ll
```

Implmentamos el algoritmo de Metropolis-Hastings para muestrear la posterior:

```
# ALGORITMO METROPOLIS-HASTINGS
def metropolis_hastings(n_samples, n_chains, heads,
                        trials, step_size):
```

```
    chains = []
    print(f"Iniciando muestreo con {n_chains} cadenas...")

    for chain_idx in range(n_chains):
        samples = []

        # Punto de inicio aleatorio
        current_theta = np.random.uniform(0.1, 0.9)

        # Evaluar estado inicial
        current_log_post = log_posterior(
            current_theta, heads, trials
        )

        # Bucle de muestreo (+ burnin)
        for i in range(n_samples + burnin):
            # 1. Propuesta: Random Walk
            # (Normal centrada en theta actual)
            proposal = np.random.normal(current_theta, step_size)

            # 2. Log-Posterior de la propuesta
            proposal_log_post = log_posterior(
                proposal, heads, trials
            )

            # 3. Ratio de Aceptación (log scale)
            # log(r) = log(p(new)) - log(p(old))
            log_ratio = proposal_log_post - current_log_post

            # 4. Decisión de aceptación
            accept_threshold = np.log(np.random.rand())

            if accept_threshold < log_ratio:
                current_theta = proposal
                current_log_post = proposal_log_post

            # Guardar solo después del burn-in
            if i >= burnin:
                samples.append(current_theta)

        chains.append(samples)
        print(f"Cadena {chain_idx + 1} completada.")

    return np.array(chains)
```

Utilizamos la implementación para obtener muestras de la posterior:

```
# EJECUCIÓN
chains = metropolis_hastings(
    n_samples, n_chains, n_heads, n_trials, step_size
)
```

```
# Convertir a objeto InferenceData de Arviz
idata = az.from_dict(posterior={"theta": chains})
```

```
Iniciando muestreo con 4 cadenas...
Cadena 1 completada.
Cadena 2 completada.
Cadena 3 completada.
Cadena 4 completada.
```

Tabla 4 - Resumen ArviZ: Media Estimada $\theta \approx 0.6680$
Intervalo HDI (3% - 97%): [0.420, 0.897]

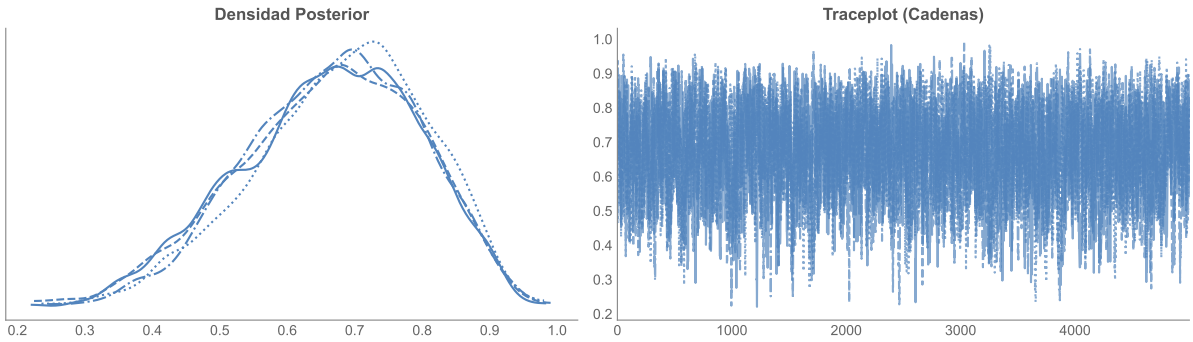
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
theta	0.668	0.131	0.42	0.897	0.003	0.002	1691.0	1971.0	1.0

Diagnósticos Obligtatorios

Traceplot

El Traceplot permite visualizar la evolución temporal de las cadenas de Markov, facilitando la verificación inmediata de la convergencia asintótica y la calidad del muestreo. A través de este gráfico se evalúa si el algoritmo ha alcanzado la estacionariedad (oscilando alrededor de un valor estable sin tendencias) y si existe una mezcla adecuada del espacio de parámetros (exploración eficiente), lo cual es indispensable para validar que las muestras obtenidas son representativas de la distribución posterior objetivo y descartar problemas como una alta autocorrelación o una configuración errónea del tamaño de paso.

Figura 5 - theta - θ

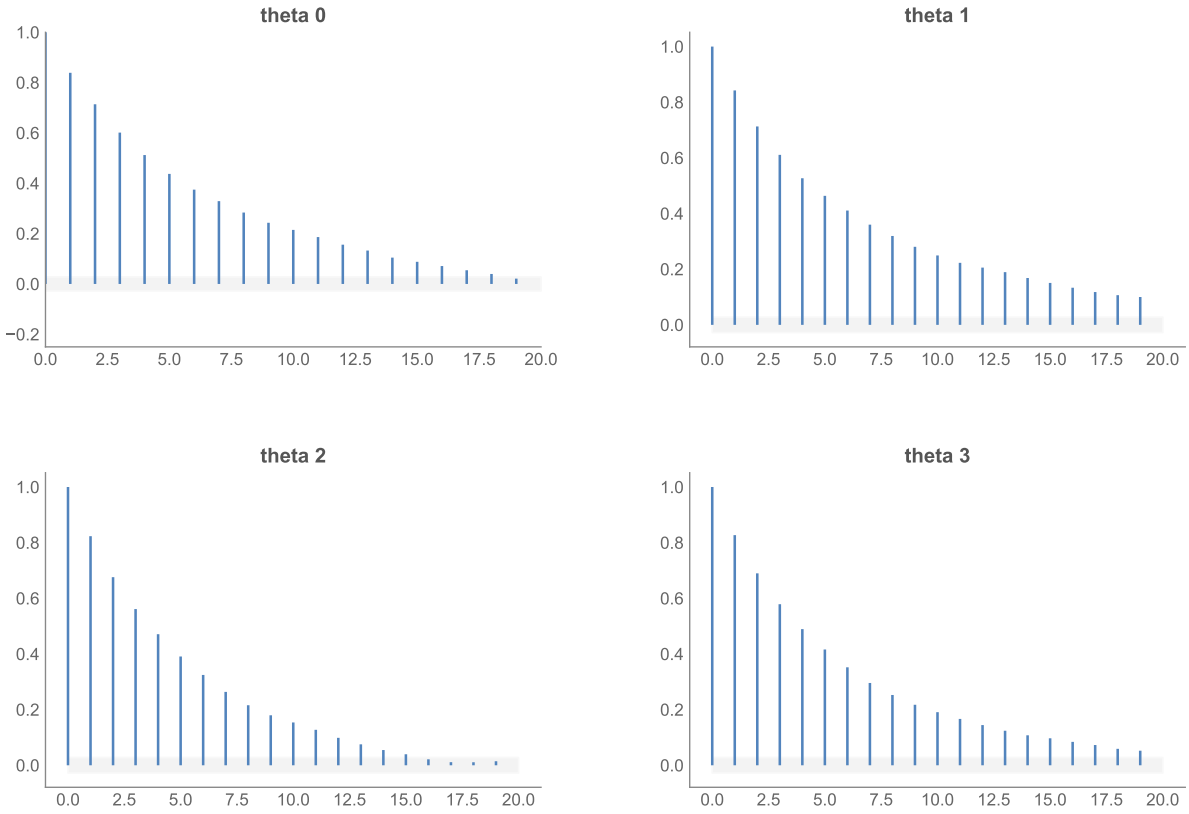


El gráfico confirma la convergencia exitosa y robusta del algoritmo Metropolis-Hastings, validando la calidad de la inferencia realizada. En el panel derecho (Traceplot), se observa una oscilación densa y constante alrededor de un eje central sin tendencias visibles —patrón “oruga peluda”—, lo cual demuestra que las cuatro cadenas han alcanzado la estacionariedad y exploran el espacio de parámetros con una mezcla eficiente y homogénea. Corroborando esto, el panel izquierdo (Densidad Posterior) exhibe una superposición casi perfecta de las distribuciones estimadas por cada cadena, indicando consistencia interna (bajo R-hat) y revelando una moda centrada aproximadamente en 0.7, valor que coincide con el máximo teórico esperado para la distribución Beta(8,4).

Autocorrelación

El análisis de la autocorrelación permite cuantificar la dependencia serial inherente entre las muestras generadas por la cadena de Markov, dado que los algoritmos MCMC producen, por definición, observaciones correlacionadas en lugar de independientes. Este diagnóstico es crítico para evaluar la eficiencia del muestreo (mixing), ya que una persistencia alta de la correlación a través de múltiples retardos (lags) indica una exploración lenta del espacio de parámetros y reduce el tamaño de muestra efectivo (ESS), lo que alertaría sobre la necesidad de incrementar el número de iteraciones o ajustar el tamaño de paso para garantizar que la inferencia estadística sobre la posterior sea fiable y precisa.

Figura 6 - Autocorrelación

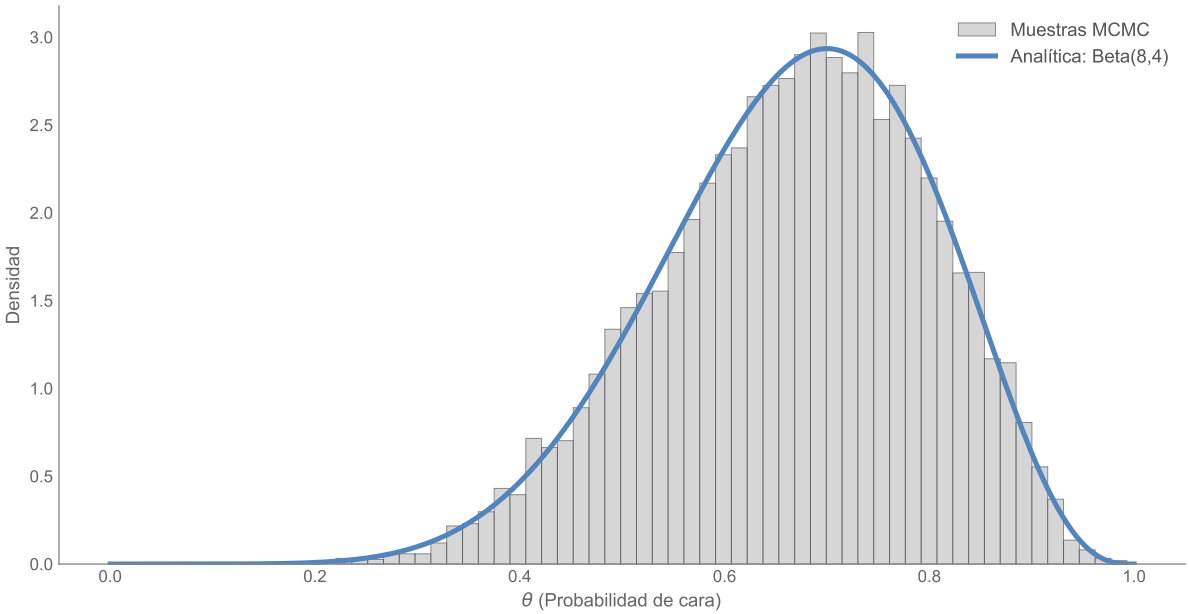


Los gráficos de autocorrelación revelan una persistente dependencia serial entre las muestras sucesivas, evidenciada por un decaimiento lento de las barras que mantienen valores significativos incluso después de 15 a 20 retardos (*lags*). Este comportamiento, consistente en las cuatro cadenas analizadas, indica una eficiencia de mezcla (*mixing*) moderada y una exploración del espacio de parámetros con cierta “viscosidad”, lo cual reduce el Tamaño de Muestra Efectivo (ESS) e implica que, para fines de inferencia estadística robusta, la cantidad de información independiente real es considerablemente menor al número total de iteraciones computadas.

Histograma vs Analítica Beta(8,4)

La superposición del histograma de frecuencias empíricas frente a la curva analítica de la distribución Beta es una prueba definitiva de validación, aprovechando que el modelo Beta-Binomial posee una solución cerrada conocida que sirve como “verdad fundamental” (*ground truth*). Este diagnóstico visual permite confirmar rigurosamente que el algoritmo Metropolis-Hastings está muestreando fielmente de la distribución objetivo y no de una aproximación errónea, verificando así que la lógica computacional, la función de verosimilitud y el mecanismo de aceptación/rechazo han sido implementados con exactitud matemática y que el muestreador recupera la geometría correcta de la posterior sin sesgos.

Figura 7 - Validación: MCMC vs Analítica



El gráfico evidencia una correspondencia altamente satisfactoria entre la distribución empírica generada por el algoritmo (histograma azul) y la solución analítica teórica (curva roja), demostrando la exactitud de la implementación. La alineación precisa de las frecuencias de las barras con el perfil de la función de densidad Beta(8,4) confirma que el muestreador ha logrado capturar fielmente la estructura probabilística de la posterior, reproduciendo correctamente tanto la ubicación de la moda en torno a 0.7 como la dispersión asociada, lo cual valida que las muestras obtenidas son estadísticamente representativas de la distribución objetivo y legitima el uso de la simulación para la inferencia paramétrica.

R-hat con 4 cadenas y ESS (Effective Sample Size)

Las métricas R-hat y ESS (Tamaño de Muestra Efectivo) proporcionan una validación cuantitativa y objetiva de la fiabilidad y precisión de la simulación, superando las limitaciones de la inspección visual. El estadístico R-hat resulta indispensable al utilizar múltiples cadenas para verificar matemáticamente la convergencia global, asegurando a través de la comparación de varianzas que el algoritmo no ha quedado atrapado en óptimos locales; simultáneamente, el ESS es crítico para cuantificar el volumen real de información independiente generada descontando la autocorrelación, garantizando así que la estimación de la posterior posea un error estándar de Monte Carlo lo suficientemente reducido para ser estadísticamente válida.

Tabla 5 - Diagnóstico Numérico (MCMC)

Parámetro analizado: `theta`

Métrica	Valor	Criterio	Estado
R-hat (Gelman-Rubin)	`1.0000`	< 1.01	ESTABLE
ESS (Bulk)	1691	> 400	SUFICIENTE

Los diagnósticos numéricos validan la robustez técnica de la simulación, destacando un estadístico R-hat de 1.0 que confirma la convergencia perfecta de las cadenas y su indistinguibilidad estadística. Por su parte, el Tamaño de Muestra Efectivo (ESS) de 1691.0, aunque inferior al total de iteraciones debido a la correlación serial, resulta suficiente para garantizar estimaciones estables de la tendencia

central, lo cual, sumado a un error estándar de Monte Carlo (mcse) marginal de 0.003, asegura que la precisión de la inferencia sobre la posterior es elevada y que el error introducido por el método de muestreo es despreciable.

Parte 3 - Simulación de Eventos Discretos (M/M/1 o M/M/c)

Para implementar esta simulación, el motor central debe basarse en una **Lista de Eventos Futuros (LEF)** ordenada cronológicamente y una variable de reloj (T_{now}). El tiempo no avanza de forma continua, sino que "salta" discretamente al instante del evento más próximo en la lista. Inicialmente, se genera la primera llegada aleatoria (usando la tasa $\lambda = 10$) y se inserta en la LEF; el ciclo de simulación consiste en extraer repetidamente el evento de menor tiempo de la lista, actualizar el reloj a ese instante y ejecutar la lógica de cambio de estado correspondiente hasta que T_{now} supere las 8 horas.

La lógica de estado maneja dos eventos principales: Llegada y Salida. Al procesar una Llegada, se programa inmediatamente la siguiente llegada futura y se evalúan los servidores: si hay alguno libre (de los c disponibles), se ocupa y se calcula una duración de servicio (con tasa $\mu = 4$) para insertar un evento de Salida en la LEF; si todos están ocupados, se incrementa el contador de la cola. Por otro lado, al procesar una Salida, se libera el servidor, pero si la cola no está vacía, se decrementa inmediatamente para ingresar al siguiente cliente al servicio, generando su respectivo evento de finalización futuro.

Simulación 8 horas de un sistemas de colas

Esta es una implementaciónen Python de una simulación de eventos discretos (DES) utilizando una Lista de Eventos Futuros (LEF) implementada con una cola de prioridad (heapq). El código está diseñado para soportar tanto M/M/1 (1 servidor) como M/M/c (múltiples servidores), pero además tenemos estos parámetros:

- λ (Tasa de llegadas): 10 clientes/hora.
- μ (Tasa de servicio): 4 clientes/hora.
- c (Servidores): Variable.

Es importante resaltar que si usamos $c = 1$ (M/M/1), el sistema será inestable porque la tasa de llegada (10) es mayor que la capacidad de servicio (4). La cola crecerá infinitamente. Para un sistema estable, necesitamos $c \geq 3$ (capacidad 12 > 10). Por esto se ha configurado el código por defecto con NUM_SERVIDORES = 3, pero puede cambiarse a 1 para observar cómo se satura.

En pimera instancia, definimos las clases y funciones necesarias para la simulación:

```
class SimulacionMMC_Core:
    def __init__(self, tiempo_max, tasa_llegada,
                 tasa_servicio, n_servidores):
        self.tiempo_max = tiempo_max
        self.tasa_llegada = tasa_llegada
        self.tasa_servicio = tasa_servicio
        self.n_servidores = n_servidores

        # Estado del sistema
        self.reloj = 0.0
        self.num_en_cola = 0
        self.servidores_ocupados = 0
        self.lef = []  # Lista de Eventos Futuros

        # Estadísticas
        self.total_llegadas = 0
```

```
        self.total_atendidos = 0
        self.area_cola = 0.0
        self.area_ocupados = 0.0
        self.tiempo_ultimo_evento = 0.0

        # Historial
        self.historia = [(0.0, 0, 0)]

    def actualizar_estadisticas(self, tiempo_actual):
        delta_t = tiempo_actual - self.tiempo_ultimo_evento
        self.area_cola += self.num_en_cola * delta_t
        self.area_ocupados += self.servidores_ocupados * delta_t
        self.tiempo_ultimo_evento = tiempo_actual

    def procesar_llegada(self):
        self.total_llegadas += 1
        prox = self.reloj + random.expovariate(self.tasa_llegada)
        heapq.heappush(self.lef, (prox, 0))

        if self.servidores_ocupados < self.n_servidores:
            self.servidores_ocupados += 1
            duracion = random.expovariate(self.tasa_servicio)
            t_salida = self.reloj + duracion
            heapq.heappush(self.lef, (t_salida, 1))
        else:
            self.num_en_cola += 1

    def procesar_salida(self):
        self.total_atendidos += 1
        if self.num_en_cola > 0:
            self.num_en_cola -= 1
            duracion = random.expovariate(self.tasa_servicio)
            t_salida = self.reloj + duracion
            heapq.heappush(self.lef, (t_salida, 1))
        else:
            self.servidores_ocupados -= 1

    def correr(self):
        # Programar primera llegada
        t_primera = random.expovariate(self.tasa_llegada)
        heapq.heappush(self.lef, (t_primera, 0))

        while self.reloj < self.tiempo_max and self.lef:
            tiempo_evento, tipo_evento = heapq.heappop(self.lef)

            if tiempo_evento > self.tiempo_max:
                self.actualizar_estadisticas(self.tiempo_max)
                self.reloj = self.tiempo_max
                break
```

```
            self.actualizar_estadisticas(tiempo_evento)
            self.reloj = tiempo_evento

            if tipo_evento == 0:
                self.procesar_llegada()
            else:
                self.procesar_salida()

            self.historia.append((
                self.reloj,
                self.num_en_cola,
                self.servidores_ocupados
            ))

        # Cálculos finales
        lq = self.area_cola / self.reloj
        prom_ocupados = self.area_ocupados / self.reloj
        utilizacion = prom_ocupados / self.n_servidores
        lambda_real = self.total_llegadas / self.reloj
        wq = lq / lambda_real if lambda_real > 0 else 0

        # Retornar diccionario de resultados
        return {
            "tiempo_simulado": self.reloj,
            "total_llegadas": self.total_llegadas,
            "total_atendidos": self.total_atendidos,
            "lq": lq,
            "wq": wq,
            "utilizacion": utilizacion,
            "prom_ocupados": prom_ocupados,
            "historia": self.historia,
            "capacidad": self.n_servidores
        }
```

A continuación se utiliza la clase de Simulación M/M/c implementada. Si `n_servidores=1` la cola crece infinitamente, esto ese produce un cuello de botella

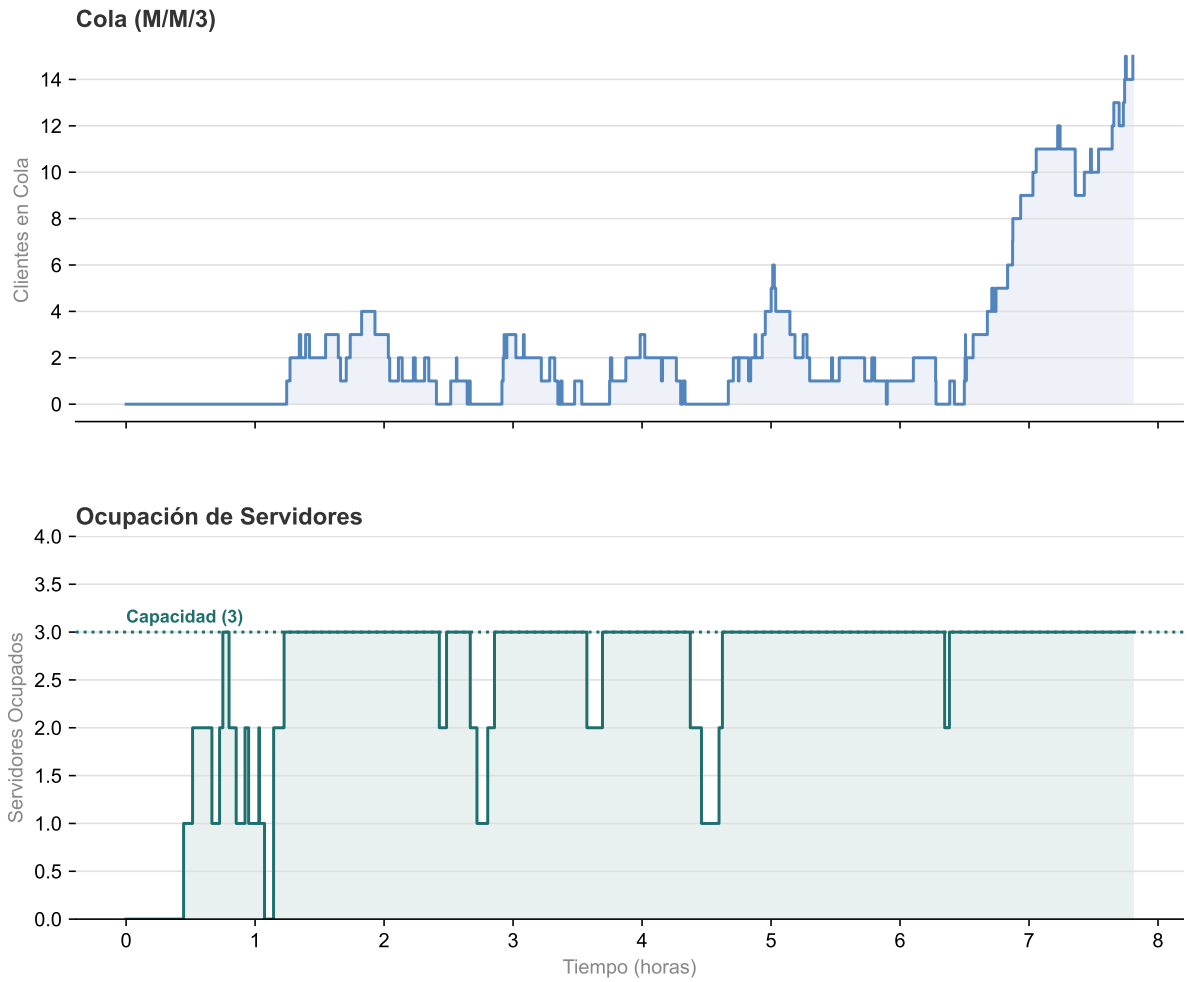
```
# Configurar la simulación (8 horas, lambda=10, mu=4, c=3)
random.seed(rnd_seed)

sim = SimulacionMMC_Core(
    tiempo_max=8,
    tasa_llegada=10,  # Lambda
    tasa_servicio=4,  # Mu
    n_servidores=3    # c
)

datos = sim.correr()
```

Tabla 6 - Resultados de la Simulación M/M/3

Métrica	Valor	Descripción
Tiempo Simulado	`8.00 h`	Duración total
Llegadas Totales	`84`	Clientes que entraron
Atendidos	`66`	Clientes completados
Lq (Cola prom)	`2.7528`	Longitud media de cola
Wq (Tiempo cola)	`15.73 min`	Tiempo medio espera
Utilización (ρ)	`85.85%`	SATURADO

Para visualizar los resultados de la simulación de eventos discretos se usan gráficos de paso (step plots) dad que el estado del sistema (número en cola en espera o servidores ocupados) permanece constante entre eventos y va cambiando con cada arribo.

Figura 8 - Dinámica del Sistema M/M/c



Parte 4 - Modelos Continuos

Para implementar un modelo de simulación de eventos basado en un **Proceso de Poisson No Homogéneo (NHPP)** mediante el **método de Thinning** (o adelgazamiento de Lewis-Shedler), el primer paso consiste en definir una tasa mayorante constante λ^*, la cual debe ser igual o superior al valor máximo que alcanza la función de intensidad variable $\lambda(t)$ durante todo el periodo de simulación. El motor de simulación avanza generando una secuencia de tiempos de arribo “candidatos” utilizando esta tasa máxima constante, creando efectivamente un proceso de Poisson homogéneo que “sobremuestrea” la línea de tiempo con más eventos de los necesarios.

El segundo paso es el proceso de filtrado estocástico que da nombre al método. Para cada evento candidato generado en el instante t, se evalúa si se conserva o se descarta mediante una prueba de aceptación-rechazo: se genera un número aleatorio uniforme $u \in [0, 1]$ y se compara con el ratio $\lambda(t)/\lambda^*$. Si u es menor o igual a esta proporción, el evento se acepta y se procesa; de lo contrario, se ignora (se "adelgaza" la secuencia). De esta forma, la probabilidad de aceptar un evento es proporcional a la intensidad real en ese momento, resultando en una distribución de eventos que se ajusta fielmente a la curva de la tasa variable $\lambda(t)$.

Implementación de NHPPP por Thinning

Así para esta simulación se diseña una función de intensidad $\lambda(t)$ que es realista para una simulación de 7 días: simula ciclos diarios (como el tráfico de una web o clientes en una tienda) con picos durante el día y valles durante la noche. El algoritmo de Thinning (o Aceptación-Rechazo) funciona en tres pasos:

1. Dominancia: Se define una tasa constante λ_{max} que sea mayor o igual a la tasa real $\lambda(t)$ en todo momento.
2. Generación: Se generan "candidatos" a eventos usando un Proceso de Poisson Homogéneo con la tasa máxima λ_{max}.
3. Filtrado (Thinning): Se acepta cada candidato como un evento real con probabilidad $P = \frac{\lambda(t)}{\lambda_{max}}$.

```
# Función de Intensidad
def intensity_function(t):
    """
    Define la tasa de llegada lambda(t).
    Ciclo diario (24h) con picos al mediodía.
    """
    cycle = 15 * np.sin(2 * np.pi * (t - 9) / 24)
    rate = 20 + cycle
    return max(0, rate)

# Función de Filtro
def simulate_nhpp_thinning(t_max, lambda_upper_bound):
    """
    Simulación NHPP mediante Thinning.
    Retorna: (lista de eventos, número total de intentos)
    """
    t = 0
    events = []
    candidates_count = 0

    while t < t_max:
        # 1. Generar candidato (Poisson Homogéneo)
        u1 = np.random.uniform(0, 1)
        w = -np.log(u1) / lambda_upper_bound
        t = t + w

        if t >= t_max:
            break

        candidates_count += 1
```

```
        # 2. Probabilidad de aceptación
        prob_acceptance = intensity_function(t) / lambda_upper_bound

        # 3. Test de aceptación (Thinning)
        u2 = np.random.uniform(0, 1)
        if u2 <= prob_acceptance:
            events.append(t)

    return np.array(events), candidates_count
```

A continuación estas funciones se usan en una simulación de 7 días o 168 horas:

```
# Parámetros
DIAS = 7
HORAS_TOTALES = DIAS * 24
LAMBDA_MAX = 35

# Ejecución
events, candidates = simulate_nhpp_thinning(HORAS_TOTALES, LAMBDA_MAX)
```

Tabla 7 - Reporte de Simulación (NHPP - 7 Días)

1. Métricas Generales

Métrica	Valor
Tiempo Total	168 horas
Eventos Generados	3294
Candidatos	5776
Eficiencia Thinning	57.03%
Promedio Global	19.61 ev/h

2. Tiempos entre llegadas (IAT)

Estadístico	Valor
Promedio	0.0510 h (3.1 min)
Mínimo	0.000020 h
Máximo	1.1459 h
Desv. Std	0.0724

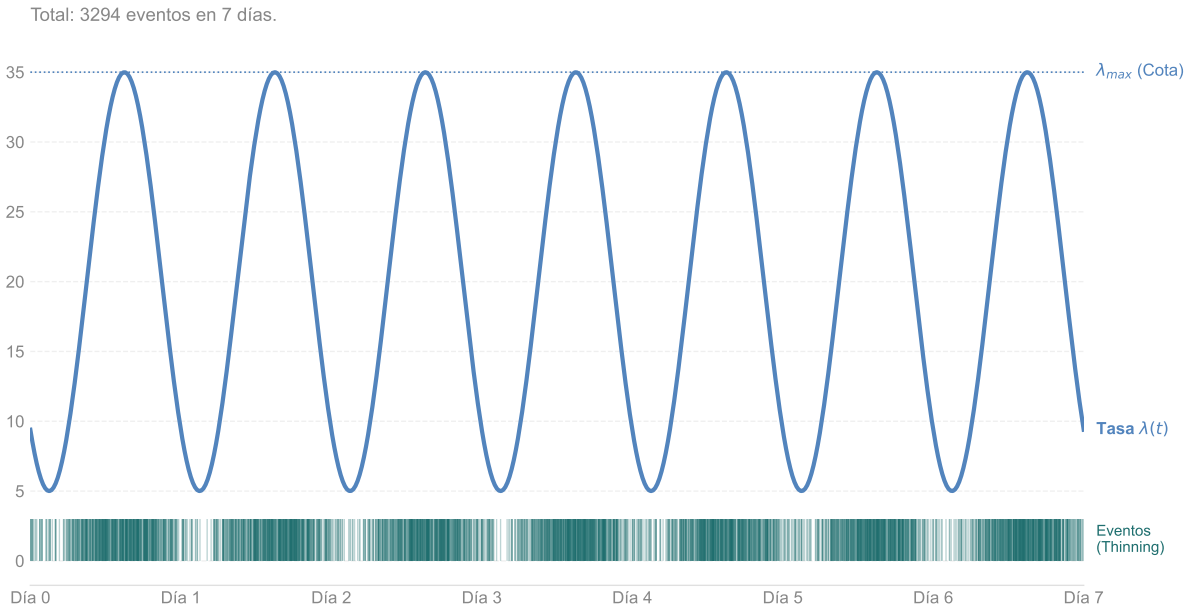
3. Desglose Diario

Día	Rango	Eventos	Promedio (ev/h)
1	000 - 024 h	487	20.29
2	024 - 048 h	454	18.92
3	048 - 072 h	442	18.42
4	072 - 096 h	479	19.96
5	096 - 120 h	472	19.67
6	120 - 144 h	469	19.54

Día	Rango	Eventos	Promedio (ev/h)
7	144 - 168 h	491	20.46

Se visualizan los resultados de la simulación:

Figura 9 - Simulación NHPP: Ciclos Diarios y Eventos Estocásticos



La visualización ilustra la evolución temporal de un Proceso de Poisson No Homogéneo (NHPP) durante un periodo de 168 horas, equivalente a siete días completos. El componente principal es la curva azul oscuro que representa la función de intensidad $\lambda(t)$, la cual exhibe un comportamiento perfectamente cíclico y sinusoidal, oscilando entre una tasa base cercana a 5 y un pico máximo de 35 eventos por hora. Sobre esta curva se proyecta una línea roja discontinua que marca la cota superior λ_{max} (establecida en 35), la cual actúa como el techo de referencia necesario para la generación de eventos candidatos dentro del algoritmo de *Thinning*.

En la franja inferior, las líneas verticales verdes denotan los instantes exactos de los eventos finalmente aceptados por el modelo. Se observa una correlación directa entre la densidad de estas líneas y la magnitud de la función de intensidad: la concentración de eventos se vuelve densa y compacta coincidiendo con los picos de la onda sinusoidal, mientras que se dispersa notablemente durante los valles. Esta distribución visual confirma la eficacia del método de aceptación-rechazo, demostrando que la frecuencia de ocurrencia de los eventos se modula dinámicamente en función de la tasa variable $\lambda(t)$ en cada instante del tiempo.

Parte 5 - Gillespie SSA o Next Reaction Method

Para implementar el **algoritmo de Gillespie (SSA) en este sistema**, primero se deben calcular en cada iteración las propensiones (a_v) que determinan la probabilidad instantánea de cada canal de reacción. Para la síntesis (orden cero), la propensión es constante, $a_1 = k_1 = 10$; para la degradación (primer orden), la propensión depende del estado actual de la población, $a_2 = k_2 \times [\text{mRNA}]$. Se calcula la propensión total $a_0 = a_1 + a_2$ y se genera el tiempo hasta el próximo evento (τ) muestreando una distribución exponencial con media $1/a_0$ (usualmente $-\ln(u_1)/a_0$), lo que define cuánto tiempo transcurre en el sistema antes de que la configuración molecular cambie.

Una vez determinado el "cuándo", se decide el "qué" seleccionando una de las dos reacciones con probabilidad proporcional a su magnitud relativa ($P_{sntesis} = a_1/a_0$ y $P_{degradacin} = a_2/a_0$). Dependiendo de la reacción elegida, se actualiza el contador de moléculas de mRNA incrementándolo o

decrementándolo en una unidad, se avanza el tiempo de simulación ($t \leftarrow t + \tau$) y, crucialmente, se recalculan las propensiones para el siguiente paso, dado que a_2 cambiará cada vez que varíe la cantidad de mRNA, capturando así las fluctuaciones estocásticas intrínsecas del sistema.

Algoritmo de Gillespie (Método Directo)

Esta es una implementación del Algoritmo de Gillespie (Método Directo) en Python para el sistema de nacimiento y muerte (producción y degradación de ARNm) planteado:

$$\rightarrow \text{mRNA}(k_1 = 10)$$

$$\text{mRNA} \rightarrow (k_2 = 1)$$

Basado en esta descripción tenemos dos reacciones: 1. Producción (Transcripción): $\emptyset \xrightarrow{k_1}$ mRNA. Tasa constante: $k_1 = 10$. Esta reacción incrementa el conteo de ARNm en 1. 3. Degradación: mRNA $\xrightarrow{k_2} \emptyset$. Tasa dependiente de la cantidad actual: $k_2 = 1$. Esta reacción disminuye el conteo de ARNm en 1.

Cabe destacar que el estado estacionario promedio esperado es $k_1/k_2 = 10$ moléculas. El estado estacionario promedio de 10 moléculas se fundamenta en el principio de equilibrio dinámico, el cual se alcanza cuando la velocidad de entrada (producción) iguala a la velocidad de salida (degradación) del sistema. Dado que la producción ocurre a una tasa constante k_1 y la degradación es proporcional a la cantidad de moléculas presentes ($k_2 \cdot$ mRNA), el balance se logra matemáticamente cuando $k_1 = k_2 \cdot$ mRNA; al despejar la concentración de equilibrio, se obtiene el cociente k_1/k_2, lo que resulta en un valor promedio de 10 moléculas para los parámetros dados.

```python
# Funcion de Simulacion Gillespie SSA
def gillespie_ssa(k1, k2, t_max):
    # Inicialización
    t = 0.0
    # Condición inicial (puedes cambiarla)
    mRNA = 0

    # Listas para guardar el historial (para graficar)
    time_points = [t]
    mRNA_counts = [mRNA]

    while t < t_max:
        # 1. Calcular las propensiones (propensities)
        # a1: Probabilidad de producción (constante)
        a1 = k1
        # a2: Prob. degradación (proporcional a moléculas)
        a2 = k2 * mRNA

        a_sum = a1 + a2

        # Si a_sum es 0, el sistema para (sin reacciones)
        if a_sum == 0:
            break

        # 2. Determinar tiempo hasta próxima reacción (tau)
        # Se extrae de una distribución exponencial
        r1 = np.random.rand()
```

```
        tau = (1.0 / a_sum) * np.log(1.0 / r1)

        # 3. Determinar qué reacción ocurre
        r2 = np.random.rand()

        if r2 < (a1 / a_sum):
            # Ocurre Reacción 1: Producción
            mRNA += 1
        else:
            # Ocurre Reacción 2: Degradación
            mRNA -= 1

        # 4. Actualizar tiempo y guardar estado
        t += tau
        time_points.append(t)
        mRNA_counts.append(mRNA)

    return time_points, mRNA_counts
```

A continuación se usa la simulación para el sistema planteado:

```
# Parámetros
# Dado que se simulan ARNm, los procesos de transcripción y
# degradación suelen medirse en minutos o horas.
k1 = 10.0
k2 = 1.0
t_max = 1000.0

# Simular
tiempos, conteos = gillespie_ssa(k1, k2, t_max)
```

Tabla 8 - Resultados Simulación Estocástica (Gillespie SSA)

1. Parámetros del Sistema

Parámetro	Valor
Producción (k_1)	10.0
Degradación (k_2)	1.0
Tiempo Total	1000.00 u.t.
Eventos Totales	20185

2. Análisis de Trayectoria (Estadística)

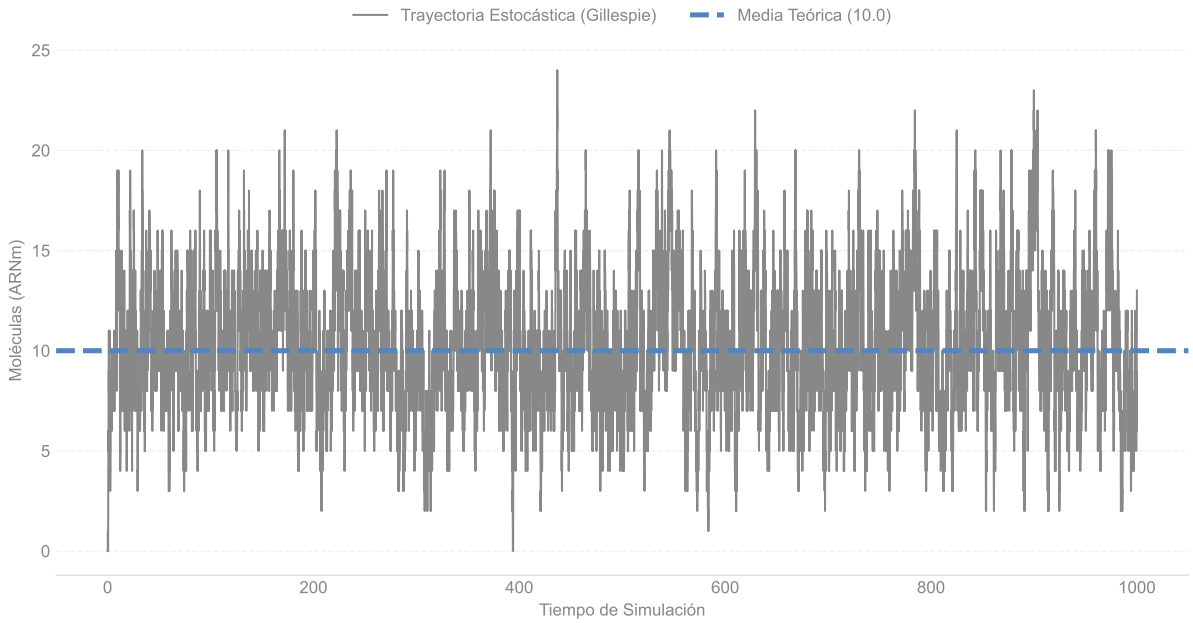
Métrica	Valor Observado	Rango
Media (μ)	10.7573	-
Desv. Std (σ)	3.3082	-
Varianza (σ^2)	10.9444	-
Fluctuación	-	[0 - 24]

3. Validación Teórica (vs Poisson)

Concepto	Esperado	Observado	Error
Estado Estacionario	10.0000	10.7573	NOT OK 7.57%
Varianza	10.0000	10.9444	-

Se visualizan los resultados de la simulación:

Figura 10 - Simulación Gillespie: Producción y Degradación de ARNm



La visualización ilustra la naturaleza estocástica del sistema de producción y degradación de ARNm, donde la trayectoria azul exhibe fluctuaciones continuas y aleatorias inherentes al ruido intrínseco del proceso biológico. Se observa que, tras superar la condición inicial nula, el número de moléculas oscila dinámicamente alrededor del valor de equilibrio teórico señalado por la línea discontinua roja (10 moléculas), demostrando que, si bien el promedio temporal del sistema converge al modelo determinista, el estado instantáneo varía constantemente dentro de un rango de dispersión característico (aproximadamente entre 0 y 23 moléculas).

Parte Integradora - Simulación de Tráfico Aéreo

Se propone un modelo de simulación híbrida de tráfico aéreo donde el flujo de entrada de aeronaves se gestiona mediante un Proceso de Poisson No Homogéneo (NHPP) utilizando el método de Thinning. Este componente permite modelar fielmente las curvas de demanda operativa del aeropuerto, generando llegadas estocásticas que respetan los picos horarios (horas punta) y los valles nocturnos. Estas entidades (aviones) ingresan a un sistema de colas modelado bajo la lógica M/M/1, donde la pista de aterrizaje actúa como el servidor único con tiempos de servicio exponenciales, gestionando tanto la cola de espera en el aire (holding pattern) como en tierra (taxiway).

De forma paralela y asíncrona, se ejecuta un motor basado en el Algoritmo de Gillespie (SSA) para simular la dinámica meteorológica. En este contexto, los estados del clima (ej. “Despejado”, “Viento Cruzado”, “Tormenta Eléctrica”) se tratan como “especies químicas” discretas que transicionan estocásticamente en función de tasas de cambio históricas (propensidades). El algoritmo de Gillespie determina los intervalos exactos de tiempo en los que el sistema permanece en cada estado climático, generando una línea de tiempo de condiciones ambientales independiente del tráfico aéreo pero estadísticamente exacta.

La integración de ambos modelos se realiza mediante una modulación dinámica de la tasa de servicio (μ). El estado vigente dictado por el módulo Gillespie actúa como una variable de control sobre el modelo M/M/1: cuando Gillespie transiciona a un estado de "Tormenta", la tasa de servicio de la pista se reduce drásticamente o se anula ($\mu \to 0$), provocando que las operaciones de aterrizaje y despegue se aborten. Esto fuerza al sistema de colas a entrar en un régimen de saturación o desvío de entidades, permitiendo analizar no solo la congestión por volumen de tráfico, sino la resiliencia del aeropuerto ante ventanas de inoperatividad estocástica y su capacidad de recuperación (recovery rate) una vez restablecidas las condiciones favorables.

El pipeline de ejecución se orquesta mediante un reloj maestro que sincroniza dos generadores de eventos concurrentes: un módulo de tráfico que inyecta entidades (aeronaves) en la línea de tiempo usando NHPP por Thinning para replicar la demanda horaria, y un motor Gillespie independiente que actualiza asincrónicamente el estado global del clima (variables de entorno). Dentro del bucle principal de procesamiento (LEF), la lógica del servidor M/M/1 consulta en tiempo real el estado vigente dictado por Gillespie antes de intentar atender a una aeronave; si el motor climático indica un evento adverso (ej. "Tormenta"), se activa una interrupción que anula o penaliza drásticamente la tasa de servicio (μ), forzando la acumulación de entidades en la cola hasta que una nueva transición estocástica del clima restablezca la operatividad, permitiendo así medir el impacto sistémico de la interrupción.

En primer lugar configuramos los parámetros globales de la simulación:

```
# CONFIGURACIÓN Y PARÁMETROS
SIM_DURATION = 24.0  # Horas
SEED = rnd_seed

# --- Parámetros de Tráfico (NHPP) ---
# Tasa máxima de llegadas (aviones/hora) para Thinning
LAMBDA_MAX = 20.0
# Tasa de servicio de la pista (aviones/hora) en buen clima
MU_OPERATIVO = 25.0

# --- Parámetros Climáticos (Gillespie) ---
# Estado 0: Despejado, Estado 1: Tormenta
# Frecuencia de aparición de tormentas
TASA_DESPEJADO_A_TORMENTA = 0.15
# Velocidad de recuperación (duración tormenta)
TASA_TORMENTA_A_DESPEJADO = 0.8

# --- Tipos de Eventos ---
EVT_ARRIVAL = 0
EVT_DEPARTURE = 1
EVT_WEATHER = 2

# --- Estados del Clima ---
CLIMA_CLEAR = 0
CLIMA_STORM = 1

np.random.seed(SEED)
random.seed(SEED)
```

A continuación se implementa en un clase la simulación híbrida con los dos motores concurrentes (NHPP + Gillespie SSA), estructurando la lógica de eventos y la interacción entre ambos modelos:

```python
class HybridAirportSim:
    def __init__(self, t_max):
        self.t_max = t_max
        self.clock = 0.0

        # Estado del Sistema
        self.queue_count = 0        # En cola (aire + tierra)
        self.server_busy = False    # Estado de la pista
        self.weather_state = CLIMA_CLEAR

        # Gestión de Eventos
        self.events = []  # Heapq
        self.current_departure_token = None

        # Estadísticas
        self.stats = {
            'arrivals': 0, 'completed': 0,
            'aborted_ops': 0, 'weather_changes': 0
        }

        # Historiales
        self.history_occupancy = [(0.0, 0)]
        self.history_weather = [(0.0, CLIMA_CLEAR)]
        self.history_aborts = []

    def schedule_event(self, time, event_type, token=None):
        if time <= self.t_max:
            heapq.heappush(self.events, (time, event_type, token))

    # --- MÓDULO NHPP (Tráfico) ---
    def get_arrival_rate(self, t):
        # Base de 5 + oscilación
        cycle = np.sin((t - 6) * np.pi / 12) ** 2
        return 5 + 15 * cycle

    def schedule_next_arrival_nhpp(self):
        t_curr = self.clock
        while True:
            u1 = random.random()
            dt = -np.log(u1) / LAMBDA_MAX
            t_curr += dt

            if t_curr > self.t_max: return

            # Thinning (Aceptación/Rechazo)
            lambda_t = self.get_arrival_rate(t_curr)
            if random.random() <= (lambda_t / LAMBDA_MAX):
                self.schedule_event(t_curr, EVT_ARRIVAL)
                break
```

```
    # --- MÓDULO CLIMÁTICO (Gillespie) ---
    def schedule_next_weather_change(self):
        if self.weather_state == CLIMA_CLEAR:
            a0 = TASA_DESPEJADO_A_TORMENTA
        else:
            a0 = TASA_TORMENTA_A_DESPEJADO

        if a0 > 0:
            r = random.random()
            tau = (1.0 / a0) * np.log(1.0 / r)
            self.schedule_event(self.clock + tau, EVT_WEATHER)

    # --- CONTROL DE EVENTOS ---
    def handle_arrival(self):
        self.stats['arrivals'] += 1
        self.schedule_next_arrival_nhpp()

        cond_free = not self.server_busy
        cond_weather = (self.weather_state == CLIMA_CLEAR)

        if cond_free and cond_weather:
            self.server_busy = True
            self.schedule_departure()
        else:
            self.queue_count += 1

    def schedule_departure(self):
        s_time = random.expovariate(MU_OPERATIVO)
        token = random.randint(0, 1000000000)
        self.current_departure_token = token
        self.schedule_event(self.clock + s_time, EVT_DEPARTURE, token)

    def handle_departure(self, token):
        if token != self.current_departure_token: return

        self.stats['completed'] += 1
        self.server_busy = False
        self.current_departure_token = None

        if self.queue_count > 0 and self.weather_state == CLIMA_CLEAR:
            self.queue_count -= 1
            self.server_busy = True
            self.schedule_departure()

    def handle_weather_change(self):
        self.stats['weather_changes'] += 1
        self.weather_state = 1 - self.weather_state

        if self.weather_state == CLIMA_STORM:
            if self.server_busy:
```

```
                # Abortar operación
                self.stats['aborted_ops'] += 1
                self.server_busy = False
                self.queue_count += 1
                self.current_departure_token = None
                self.history_aborts.append((self.clock, self.queue_count + 1))

        elif self.weather_state == CLIMA_CLEAR:
            if self.queue_count > 0 and not self.server_busy:
                self.queue_count -= 1
                self.server_busy = True
                self.schedule_departure()

        self.schedule_next_weather_change()

    def run(self):
        self.schedule_next_arrival_nhpp()
        self.schedule_next_weather_change()

        while self.events:
            time, type, token = heapq.heappop(self.events)
            self.clock = time

            # Registro histórico
            busy_int = 1 if self.server_busy else 0
            self.history_occupancy.append((self.clock, self.queue_count + busy_int))
            self.history_weather.append((self.clock, self.weather_state))

            if type == EVT_ARRIVAL: self.handle_arrival()
            elif type == EVT_DEPARTURE: self.handle_departure(token)
            elif type == EVT_WEATHER: self.handle_weather_change()

        # Generar curva de referencia para el gráfico
        t_ref = np.linspace(0, self.t_max, 200)
        l_ref = [self.get_arrival_rate(t) for t in t_ref]

        return {
            "t_max": self.t_max,
            "stats": self.stats,
            "hist_occupancy": self.history_occupancy,
            "hist_weather": self.history_weather,
            "hist_aborts": self.history_aborts,
            "ref_curve_t": t_ref,
            "ref_curve_y": l_ref
        }
```

Y ejecutamos la simulación para un periodo de 24 horas de trafico aéreo:

```
# EJECUCIÓN DE SIMULACIÓN COMPLETA
simulacion = HybridAirportSim(t_max=SIM_DURATION)
resultados = simulacion.run()
```

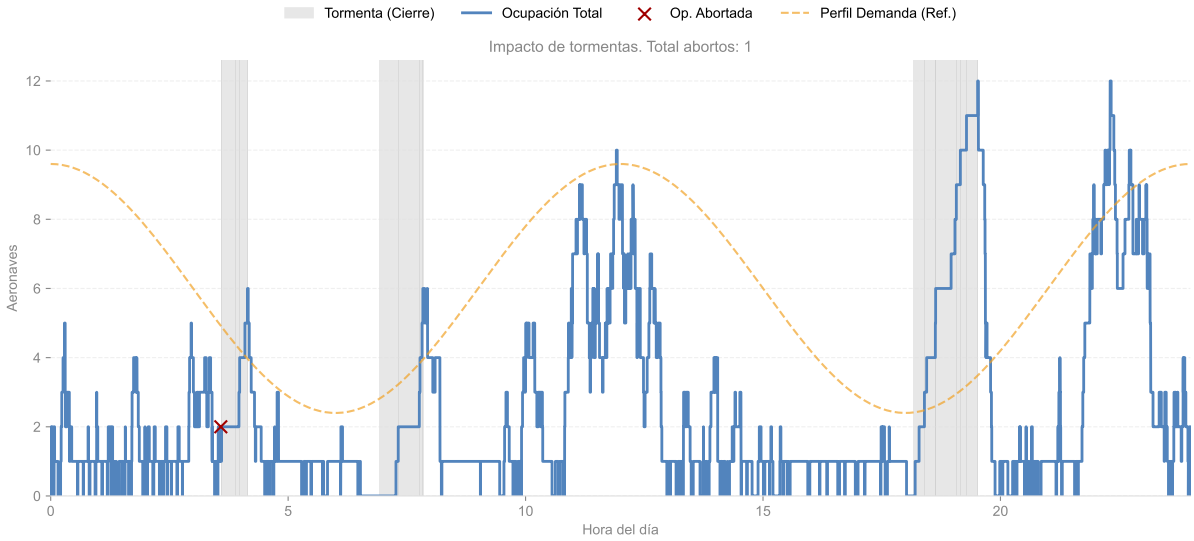
Tabla 9 - Reporte de Simulación Híbrida

Métrica Operativa	Valor
Duración	24.0 h
Llegadas Totales	304
Aterrizajes Exitosos	303
Operaciones Abortadas	1

Meteorología

Métrica Climática	Valor
Cambios de Clima	6
Tiempo en Tormenta	2.84 h (11.8%)

Figura 11 - Dinámica Aeroportuaria: Tráfico (NHPP) vs Clima (Gillespie)



Análisis de Resultados de la Simulación Híbrida y Discusión

La interpretación de los datos obtenidos a partir de la simulación híbrida, que integra un proceso de Poisson no homogéneo (NHPP) para la demanda y un motor estocástico de Gillespie para las condiciones meteorológicas, revela dinámicas operativas críticas en la gestión del tráfico aéreo.

1. Caracterización de Eventos Meteorológicos Estocásticos

El algoritmo de Gillespie generó tres eventos discretos de interrupción meteorológica (indicados como franjas grises verticales en el gráfico generado por la simulación) a lo largo del período simulado de 24 horas, lo que subraya la naturaleza aleatoria del modelo climático implementado:

- **Evento 1 (03:30 - 04:30 h):** Una interrupción de corta duración durante un período de baja demanda operacional.
- **Evento 2 (07:00 - 08:00 h):** Una interrupción estratégica inmediatamente anterior al primer pico matutino de tráfico aéreo.
- **Evento 3 (18:00 - 19:30 h):** El evento de mayor duración y criticidad del ciclo diario.

2. Interacción Crítica entre Demanda y Capacidad: El Escenario de "Tormenta Perfecta"

El análisis del gráfico generado identifica una interacción sinérgica adversa alrededor de las 19:00 h, que resultó en una saturación momentánea del sistema:

- **Coincidencia Temporal:** El tercer evento meteorológico coincidió exactamente con el pico máximo de la tasa de llegada (λ) definida por el perfil NHPP (línea amarilla punteada).
- **Saturación del Sistema:** La conjunción de una capacidad de servicio anulada ($\mu = 0$) debido a las condiciones climáticas y una tasa de llegada máxima provocó un incremento vertical y abrupto en la ocupación del sistema (línea azul). El sistema alcanzó su máximo histórico diario, registrando un total de 12 aeronaves en cola.
- **Vulnerabilidad Operacional:** Este resultado demuestra una vulnerabilidad inherente del aeropuerto: la capacidad ociosa durante períodos normales no mitiga el riesgo de congestión exponencial instantánea cuando una interrupción de servicio ocurre durante un pico de demanda.

3. Dinámica de Operaciones Abortadas (Go-Around)

Se identificó un caso específico de operación abortada, señalado con una "X" roja al inicio del primer evento de tormenta (aprox. 03:30 h). - **Mecanismo de Interrupción:** En el instante en que el motor de Gillespie cambió el estado del sistema a "Tormenta", una aeronave se encontraba en fase de aterrizaje (servidor ocupado). El protocolo del sistema forzó un go-around, cancelando el servicio y reintroduciendo la aeronave en la cola de espera, lo cual se corrobora visualmente por el incremento escalonado inmediato en la ocupación (línea azul).

4. Resiliencia y Tasas de Recuperación del Sistema

Las pendientes descendentes de la línea de ocupación inmediatamente después de la finalización de los eventos climáticos (específicamente tras las 19:30 h) proporcionan métricas clave sobre la resiliencia del sistema: - **Rápida Disminución de Cola:** El sistema exhibe una alta tasa de recuperación. Una vez restablecidas las condiciones operativas normales, la cola decreció rápidamente de 12 a 2 aeronaves en aproximadamente una hora. - **Adecuación de la Tasa de Servicio:** Este comportamiento indica que la tasa de servicio nominal (μ = 25 operaciones/hora) está adecuadamente dimensionada para absorber los retrasos acumulados, siempre y cuando las ventanas de interrupción climática no excedan una duración crítica.

Conclusión General:

La simulación validó la utilidad de la integración híbrida de los dos paradigmas de simulación estudiados. El tráfico aéreo siguió el perfil de horas de alta demanda (NHPP), pero fue severamente distorsionado por las inyecciones de caos del motor climático (Gillespie), generando escenarios de congestión realistas que un modelo estático de teoría de colas no habría podido evidenciar.

Uso de Inteligencia Artificial

En este trabajo se utilizó inteligencia artificial (Gemini) para:

- Transformar código base de R a Python
- Formatear gráficos
- Formatear código para que no ocupe más de 70 caracteres (restricción de renderizado de Quarto)
- Revisar estilo de la redacción

Anexo I - Código base en Python

```
# Código base
# Linear Congruential Generator (LGC)

def lcg(n, seed=123, a=1103515245, c=12345, m=2**31):
    x = seed
    seq = []
    for _ in range(n):
        x = (a * x + c) % m
        seq.append(x/m)
    return seq

alea = lcg(10)
for i in range(len(alea)):
    print(alea[i])
```

```
0.20531828328967094
0.6873863865621388
0.7767890496179461
0.4024707484059036
0.5324797946959734
0.10148253152146935
0.6351402262225747
0.34936570515856147
0.7226534709334373
0.027218241710215807
```

```
# Código base
# Estimación de Pi por Monte Carlo

import numpy as np

N = 100000
u1 = np.random.uniform(0, 1, N)
u2 = np.random.uniform(0, 1, N)
pi_est = 4 * np.mean(u1**2 + u2**2 <= 1)

print(f"Valor estimado de pi: {pi_est}")
```

```
Valor estimado de pi: 3.14836
```

```
# Código base
# Implementación Metropolis-Hastings

import numpy as np
from scipy.stats import binom

def mh(n_iter=5000, start=0.5, prop_sd=0.1):
    # Inicializar la cadena (array lleno de ceros)
    theta = np.zeros(n_iter)
    theta[0] = start
```

```
    # Bucle desde el segundo elemento (índice 1) hasta el final
    for t in range(1, n_iter):
        # Propuesta: Distribución normal centrada en el theta anterior
        # Equivalente en R: rnorm(1, theta[t-1], prop_sd)
        prop = np.random.normal(loc=theta[t-1], scale=prop_sd)

        # Verificación de límites (Prior Uniforme Implícito [0,1])
        # Si la propuesta se sale del rango [0, 1], se rechaza inmediatamente
        if prop < 0 or prop > 1:
            theta[t] = theta[t-1]
        else:
            # Calcular la Verosimilitud (Likelihood) usando la función
            # de masa de probabilidad
            # Equivalente en R: dbinom(7, 10, prob)
            # * 1 representa el Prior (Uniforme)
            num = binom.pmf(k=7, n=10, p=prop) * 1
            den = binom.pmf(k=7, n=10, p=theta[t-1]) * 1

            # Probabilidad de aceptación (alpha)
            ratio = num / den
            alpha = min(1, ratio)

            # Paso de Aceptación o Rechazo
            # Generamos un número aleatorio uniforme
            # y lo comparamos con alpha
            if np.random.uniform() < alpha:
                # Aceptar la propuesta
                theta[t] = prop
            else:
                # Rechazar y mantener el valor anterior
                theta[t] = theta[t-1]

    return theta

chain = mh()
print(f"Media Posterior Estimada: {np.mean(chain):.4f}")
```

```
Media Posterior Estimada: 0.6778
```

```
# Codigo base
# Simulación de Eventos Discretos (M/M/1 o M/M/c)

import numpy as np

def simulate_mm1(lam, mu, T_max):
    # Nota: 'lambda' es una palabra reservada en Python, usamos 'lam'
    t = 0.0
    n = 0

    # R usa la tasa (rate) para rexp: rexp(1, lambda)
```

```
    # Numpy usa la escala (scale = 1/rate): exponential(1/lam)
    t_arr = np.random.exponential(scale=1/lam)
    t_dep = float('inf')  # Infinito en Python

    arrivals = 0
    departures = 0
    wait_times = []

    while t < T_max:
        if t_arr < t_dep:
            # --- Evento de Llegada ---
            t = t_arr
            n += 1
            arrivals += 1

            # Si el servidor estaba libre (n=1 tras la llegada)
            # se programa servicio
            if n == 1:
                t_dep = t + np.random.exponential(scale=1/mu)

            # Programar la siguiente llegada
            t_arr = t + np.random.exponential(scale=1/lam)

        else:
            # Evento de Salida
            t = t_dep
            n -= 1
            departures += 1
            # Se agrega el tiempo a la lista
            wait_times.append(t_dep)

            if n > 0:
                # Si quedan clientes, programar la siguiente salida
                t_dep = t + np.random.exponential(scale=1/mu)
            else:
                # Si no hay nadie, la próxima salida es "infinita"
                t_dep = float('inf')

    # Se devuelve un diccionario con los tiempos
    return {
        "arrivals": arrivals,
        "departures": departures,
        "wait_times": wait_times
    }

# Lambda = 2 clientes/min, Mu = 3 clientes/min, Tiempo = 100 min
res = simulate_mm1(lam=2, mu=3, T_max=100)

print(f"Llegadas totales: {res['arrivals']}")
print(f"Salidas totales: {res['departures']}")
```

```
print("Últimos 5 tiempos de salida:")

for i in range(len(res['wait_times'])-5,len(res['wait_times'])):
    print(res['wait_times'][i])
```

```
Llegadas totales: 215
Salidas totales: 215
Últimos 5 tiempos de salida:
98.43347389754513
98.50944958825472
99.40263056754019
99.71489988562641
100.03483812961011
```

```
# Codigo base
# Implementación Gillespie SSA

import numpy as np
import pandas as pd

def gillespie(Tmax, k1=10, k2=1):
    t = 0.0
    X = 0

    # Listas para almacenar el historial
    ts = [t]
    xs = [X]

    while t < Tmax:
        # Cálculo de propensiones
        a1 = k1
        a2 = k2 * X
        a0 = a1 + a2

        # Si la propensión total es 0, el sistema se detiene
        if a0 == 0:
            break

        # 1. Generar tiempo hasta el próximo evento (tau)
        # R: -log(runif(1)) / a0
        # Esto es matemáticamente generar una variable
        # aleatoria Exponencial con tasa a0
        tau = np.random.exponential(scale=1/a0)

        # 2. Determinar qué evento ocurre
        r2 = np.random.uniform()

        if r2 * a0 < a1:
            # Evento 1: Nacimiento / Producción
            X += 1
```

```
        else:
            # Evento 2: Muerte / Degradación
            X = max(0, X - 1)

        # Actualizar tiempo y almacenar estado
        t += tau
        ts.append(t)
        xs.append(X)

    # Devolvemos un DataFrame de Pandas
    return pd.DataFrame({'time': ts, 'X': xs})

df_result = gillespie(Tmax=50)

# Mostrar las primeras 5 filas
print(df_result.head())
```

```
       time  X
0  0.000000  0
1  0.031461  1
2  0.052792  2
3  0.214520  3
4  0.226669  4
```