

Trabajo Práctico Integral - Simulación en Ciencia de Datos

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

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
- NICOLAU, Jorge

Ver RESOLUCIÓN

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.

Cada apartado incluye consignas, código sugerido y entregables obligatorios.

TEMARIO

Parte 1 - Generación de Números Pseudoaleatorios y Monte Carlo

1.1 Implementación de un LCG

Implementar un Generador Congruencial Lineal (LCG):

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

Parámetros obligatorios

- Un set "bueno"
- Uno "malo" (ej., RANDU)

```
# Código base

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

print(lcg(10))
```

[0.20531828328967094, 0.6873863865621388, 0.7767890496179461, 0.4024707484059036,

Entregables

- Histogramas
- Runs Test
- Gráfico de triples (u_i, u_{i+1}, u_{i+2})

1.2 Estimación de π por Monte Carlo

Simular puntos en el cuadrado $[0, 1]^2$ y estimar:

$$\pi \approx 4 \cdot \text{mean}(x^2 + y^2 \leq 1)$$

```
# Código base

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(pi_est)
```

3.1496

Parte 2 - Metropolis-Hastings y Bayesian Inference

2.1 Posterior beta analítica

Sea una moneda con 10 lanzamientos y 7 caras.

Posterior = Beta (8, 4)

2.2 Implementación MH

```
# Código base
```

```
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 d
```

```
            # Equivalente en R: dbinom(7, 10, prob)
```

```
            num = binom.pmf(k=7, n=10, p=prop) * 1 # * 1 representa el Prio
```

```
            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:
            theta[t] = prop      # Aceptar la propuesta
        else:
            theta[t] = theta[t-1] # Rechazar y mantener el valor anterior

    return theta

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

```

Media Posterior Estimada: 0.6691

Diagnósticos obligatorios

- Traceplot
- Autocorrelación
- Histograma vs Beta(8,4)
- R-hat con 4 cadenas
- ESS

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

Simular durante 8 horas un sistema de colas con:

- Llegadas Poisson: $\lambda = 10$ / hora
- Servicio exponencial: $\lambda = 4$ / hora
- Opcional: c servidores

Se debe implementar LEF (Lista de Eventos Futuros).

```

# Codigo base

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 = [] # En Python usamos listas dinámicas en lugar de vectores c(

```

```

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), programamos ser
        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
        wait_times.append(t_dep) # Añadimos el tiempo a la lista

        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')

# Retornamos un diccionario (equivalente a la lista nombrada de R)
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(f"Últimos 5 tiempos de salida: {res['wait_times'][-5:]}")

```

Llegadas totales: 200

Salidas totales: 198

Últimos 5 tiempos de salida: [96.58641873631348, 97.65565634421469, 99.37625953476

Parte 4 — Modelos Continuos (Elegir uno)

4.1 Opción A: NHPP por Thinning

$$\lambda(t) = 5 + 3 \sin\left(\frac{2\pi t}{24}\right)$$

Simular 7 días (168 horas)

4.2 Opción B: CTMC

Matriz de tasas:

From	To	Rate
A	B	0.4
A	C	0.2
B	A	0.1
C	A	0.3

Simular 2000 saltos.

4.3 Opción C: SDE (GBM)

$$dS_t = 0.05S_t dt + 0.2S_t dW_t$$

Simular 1000 trayectorias

Parte 5 — Gillespie SSA o Next Reaction Method

Sistema:

- \rightarrow mRNA con tasa ($k_1 = 10$)
- $\text{mRNA} \rightarrow \emptyset$ con tasa ($k_2 = 1$)

```
# Codigo base
```

```
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 (más eficientes que vectores en Python)
    ts = [t]
    xs = [X]
```

```
    while t < Tmax:
```

```

# Cálculo de propensiones
a1 = k1
a2 = k2 * X
a0 = a1 + a2

# Seguridad: 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
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  0.187355  1
2  0.242329  2
3  0.258962  3
4  0.387466  2

```

Parte Integradora (Opcional)

Combinar dos modelos (ejemplo):

- NHPP \rightarrow pacientes
- M/M/1 \rightarrow atención
- Gillespie \rightarrow biomarcadores

Evaluar: % de pacientes cuyo biomarcador excede un umbral antes de ser atendidos.

Formato y entrega

- Informe PDF (generado desde este Quarto)
- Código R/Python
- Gráficos
- Discusión de supuestos
- Semillas fijas
- Comparaciones teóricas vs simuladas
- Fecha de entrega: 20 de diciembre de 2025

RESOLUCIÓN

Configuración

Para el presente trabajo se utilizan los siguientes parámetros globales y semilla

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

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.

```
import matplotlib.pyplot as plt
import numpy as np

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

    return numbers
```

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.

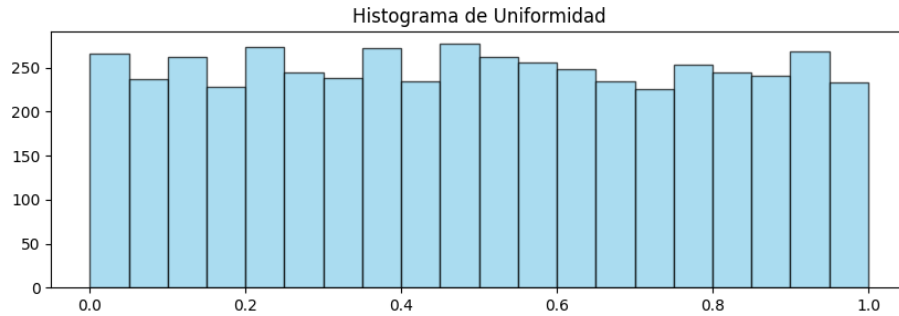
Histograma

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.

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

# --- A. Histograma ---
plt.figure(figsize=(10, 3))
plt.hist(datos, bins=20, color='skyblue', edgecolor='black', alpha=0.7)
```

```
plt.title('Histograma de Uniformidad')
plt.show()
```



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 calcula 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.

```
# --- B. Runs Test (Lógica) ---
runs = 1 # Empieza la primera racha
for i in range(len(datos) - 1):
    # Si la dirección cambia respecto al anterior, es nueva racha
    if i > 0:
        prev_diff = datos[i] - datos[i-1]
        curr_diff = datos[i+1] - datos[i]
        # Si el producto es negativo, hubo cambio de signo (dirección)
        if prev_diff * curr_diff < 0:
            runs += 1

expected_runs = (2 * len(datos) - 1) / 3
print(f"Rachas observadas: {runs}")
print(f"Rachas esperadas: {expected_runs:.2f}")
```

```
Rachas observadas: 3325
Rachas esperadas: 3333.00
```

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

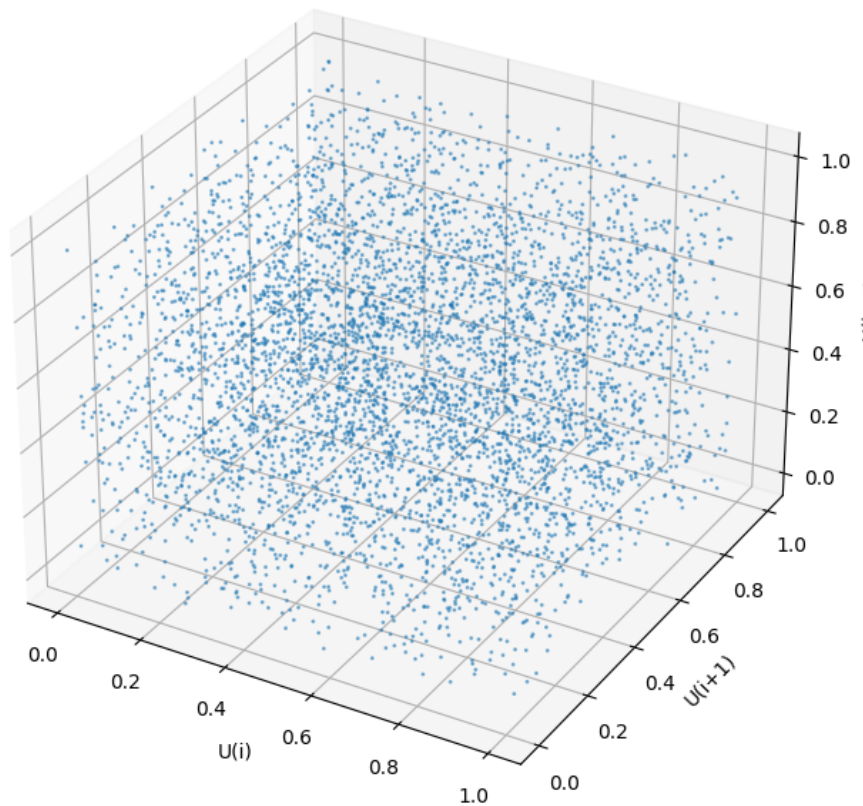
fundamental es que, si los números fueran verdaderamente independientes, los puntos deberían llenar el volumen de manera caótica y uniforme como una “nube de polvo”; sin embargo, debido a la naturaleza lineal de la fórmula $X_{n+1} = (aX_n + c)$, los LCGs deficientes exhiben una estructura cristalina 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.

```
# --- C. Gráfico de Triples (3D) ---
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

# Crear tripletas (x, y, z)
x_vals = datos[:-2]
y_vals = datos[1:-1]
z_vals = datos[2:]

ax.scatter(x_vals, y_vals, z_vals, s=1, alpha=0.5)
ax.set_title('Gráfico de Triples (Correlación Serial)')
ax.set_xlabel('U(i)'); ax.set_ylabel('U(i+1)'); ax.set_zlabel('U(i+2)')
plt.show()
```

Gráfico de Triples (Correlación Serial)



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.

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: $\pi \approx 4 \times \frac{\text{Aciertos}}{\text{Total de Puntos}}$.

```
# Función Monte Carlo con Gráfico y Retorno de Valor
def estimar_y_graficar_pi(n_puntos):
    # Generamos los números necesarios (2 por punto)
    lista_completa = lcg_generator(rnd_seed, lgc_a, lgc_c, lgc_m, n_puntos * 2)
    iterador_numeros = iter(lista_completa)

    # Listas para guardar coordenadas (visualización)
    inside_x, inside_y = [], []
    outside_x, outside_y = [], []

    dentro_count = 0

    for _ in range(n_puntos):
        # Extraer par (x, y)
        x = next(iterador_numeros)
        y = next(iterador_numeros)

        # Clasificar
        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)

    # Estimación del valor
    pi_estimado = 4 * (dentro_count / n_puntos)
```

```

# --- GRAFICO CONTEO ---
plt.figure(figsize=(6, 6))

# Puntos dentro (Azul) y fuera (Rojo)
plt.scatter(inside_x, inside_y, color='dodgerblue', s=1, label='Dentro')
plt.scatter(outside_x, outside_y, color='salmon', s=1, label='Fuera')

# Arco del círculo (estético)
t = np.linspace(0, np.pi/2, 100)
plt.plot(np.cos(t), np.sin(t), 'k-', lw=2)

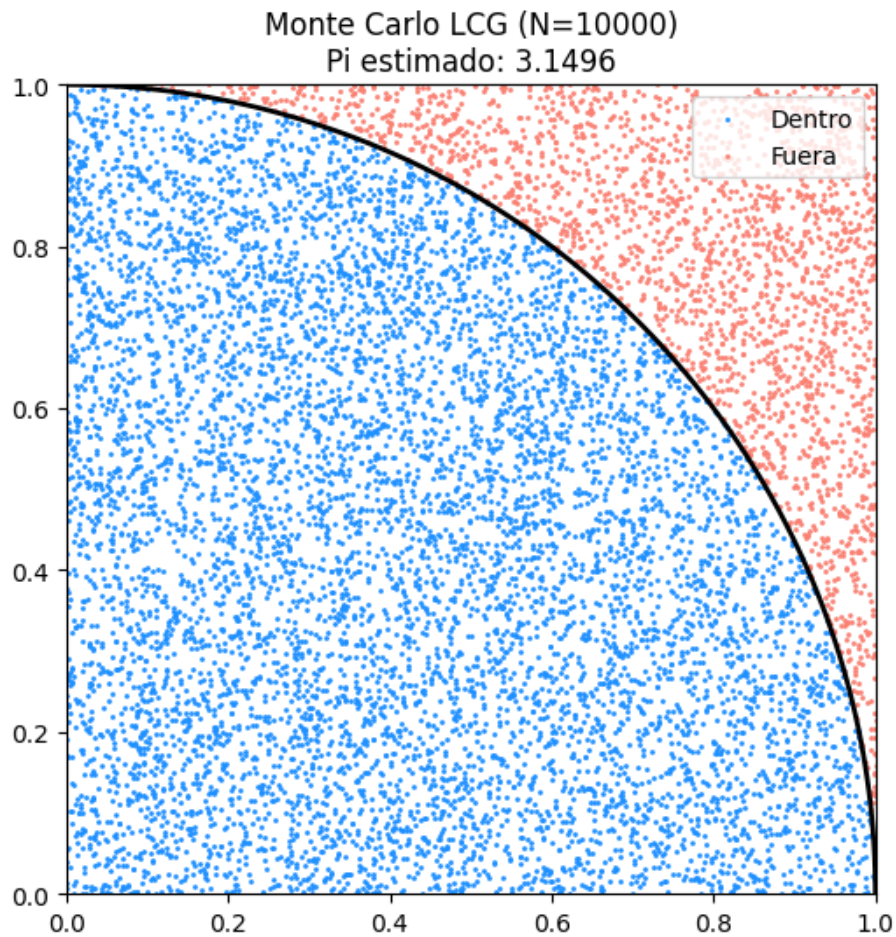
plt.xlim(0, 1); plt.ylim(0, 1)
plt.gca().set_aspect('equal')
plt.title(f"Monte Carlo LCG (N={n_puntos})\nPi estimado: {pi_estimado}")
plt.legend(loc="upper right")
plt.show()

# --- RETORNO ESTIMACIÓN ---
return pi_estimado

# Ejecución con diez mil puntos
n = 10000
pi_val = estimar_y_graficar_pi(n)

print(f"Puntos procesados: {n}")
print(f"Valor estimado de Pi: {pi_val}")
print(f"Diferencia real: {abs(pi_val - 3.14159265):.6f}")

```



Puntos procesados: 10000
Valor estimado de Pi: 3.1496
Diferencia real: 0.008007

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

En términos de Python:

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import arviz as az

# 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  $\theta^k * (1-\theta)^{(n-k)}$ 
    return (heads * np.log(theta)) + ((trials - heads) * np.log(1 - theta))

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 $\text{Beta}(8, 4)$ 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).

```

# Configuración del Problema
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 explícitamente para R-hat
n_samples = 5000      # Muestras por cadena
burnin = 1000         # Periodo de calentamiento (burn-in)
step_size = 0.1       # Desviación estándar de la propuesta (Proposal width)

# 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 para cada cadena
current_theta = np.random.uniform(0.1, 0.9)
current_log_post = log_posterior(current_theta, heads, trials)

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

    # 3. Ratio de Aceptación (en escala logarítmica para estabilidad)
    #  $r = p(\text{new})/p(\text{old})$ .  $\log(r) = \log(p(\text{new})) - \log(p(\text{old}))$ 
    log_ratio = proposal_log_post - current_log_post

    # 4. Decisión de aceptación
    if np.log(np.random.rand()) < 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)

# Ejecutar el sampler
chains = metropolis_hastings(n_samples, n_chains, n_heads, n_trials, step_size)

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

```

Iniciando muestreo con 4 cadenas...

Cadena 1 completada.

Cadena 2 completada.

Cadena 3 completada.

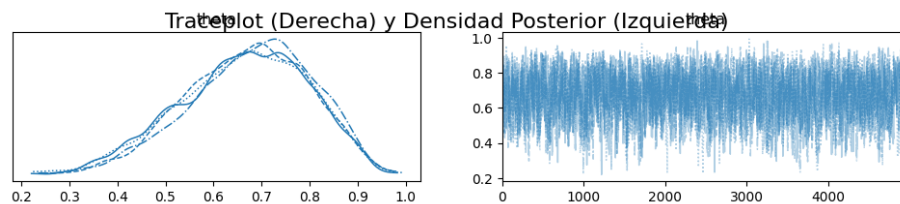
Cadena 4 completada.

Diagnósticos Obligatorios

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.

```
# A. Traceplot y Densidad (Histograma implícito en kde)
# Arviz genera automáticamente el Traceplot a la derecha y la densidad a la izquierda
az.plot_trace(idata)
plt.suptitle("Traceplot (Derecha) y Densidad Posterior (Izquierda)", fontsize=16)
plt.show()
```



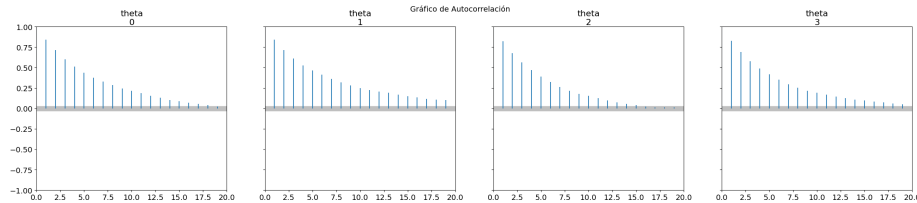
El gráfico confirma la convergencia exitosa y robusta del algoritmo Metropolis-Hastings, validando la calidad de la inferencia realizada. En el panel derecho (Traceplot), se observa una oscilación densa y constante alrededor de un eje central sin tendencias visibles —patrón “oruga peluda”—, lo cual demuestra 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.

```
# B. Autocorrelación
az.plot_autocorr(idata, max_lag=20)
plt.suptitle("Gráfico de Autocorrelación", fontsize=14)
plt.show()
```



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.

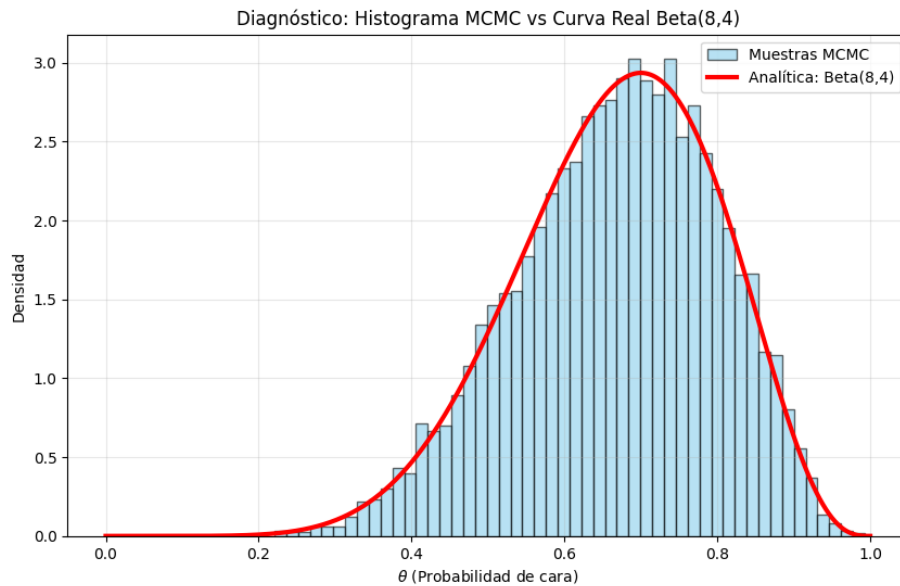
```
# C. Histograma vs Analítica Beta(8,4)
plt.figure(figsize=(10, 6))

# Aplanar las cadenas para el histograma
flat_samples = chains.flatten()

# 1. Histograma de muestras MCMC
plt.hist(flat_samples, bins=50, density=True, alpha=0.6, color='skyblue', label=
```

```
# 2. Curva Analítica Real Beta(8,4)
x = np.linspace(0, 1, 1000)
pdf_true = stats.beta.pdf(x, 8, 4) # Posterior analítica: 7 caras + 1, 3 cruces
plt.plot(x, pdf_true, 'r-', lw=3, label='Analítica: Beta(8,4)')

plt.title("Diagnóstico: Histograma MCMC vs Curva Real Beta(8,4)")
plt.xlabel(r"$\theta$ (Probabilidad de cara)")
plt.ylabel("Densidad")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```



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.

```
# D. & E. R-hat y ESS (Effective Sample Size)
summary = az.summary(idata, kind="diagnostics")

print("-" * 40)
print("DIAGNÓSTICOS NUMÉRICOS")
print("-" * 40)
print(summary)
print("-" * 40)

# Verificación explícita de valores
r_hat_val = summary.loc['theta', 'r_hat']
ess_val = summary.loc['theta', 'ess_bulk']

print(f"R-hat obtenido (Ideal < 1.01): {r_hat_val:.4f}")
print(f"ESS (Effective Sample Size): {ess_val:.1f}")
```

```
-----
DIAGNÓSTICOS NUMÉRICOS
-----
      mcse_mean  mcse_sd  ess_bulk  ess_tail  r_hat
theta         0.003    0.002   1691.0   1971.0    1.0
-----
R-hat obtenido (Ideal < 1.01): 1.0000
ESS (Effective Sample Size): 1691.0
```

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 implementación en 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.

```
import random
import heapq
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from IPython.display import Image, display
```



```

# Implementación de la Simulación
class SimulacionMMC_Log:
    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_enCola = 0
        self.servidores_ocupados = 0
        self.lef = [] #Lista de Eventos Futuros

        # Estadísticas y Acumuladores
        self.total_llegadas = 0
        self.total_atendidos = 0
        self.areaCola = 0.0 # Para calcular Lq (Longitud promedio cola)
        self.area_ocupados = 0.0 # Para calcular utilización promedio
        self.tiempo_ultimo_evento = 0.0

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

    def actualizar_estadisticas(self, tiempo_actual):
        """Calcula áreas bajo la curva antes de cambiar el estado."""
        delta_t = tiempo_actual - self.tiempo_ultimo_evento
        self.areaCola += self.num_enCola * delta_t
        self.area_ocupados += self.servidores_ocupados * delta_t
        self.tiempo_ultimo_evento = tiempo_actual

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

        print(f"--- Iniciando Simulación M/M/{self.n_servidores} por {self.tiempo_max}")

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

            if tiempo_evento > self.tiempo_max:
                # Se Actualizan estadísticas hasta el tiempo final exacto antes
                self.actualizar_estadisticas(self.tiempo_max)
                self.reloj = self.tiempo_max

```

```

        break

        # 1. Actualizar estadísticas (acumular áreas)
        self.actualizar_estadisticas(tiempo_evento)

        # 2. Avanzar reloj
        self.reloj = tiempo_evento

        # 3. Procesar evento
        if tipo_evento == 0: # LLEGADA
            self.procesar_llegada()
        else: # SALIDA
            self.procesar_salida()

        # 4. Guardar foto para la animación
        self.historia.append((self.reloj, self.num_en_cola, self.servidores_

# AL FINALIZAR EL BUCLE: MOSTRAR RESULTADOS
self.reporte_final()

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

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

def reporte_final(self):
    print("\n" + "="*40)
    print("      RESULTADOS DE LA SIMULACIÓN")
    print("="*40)

```

```

print(f"Tiempo simulado: {self.reloj:.2f} horas")
print(f"Total llegadas: {self.total_llegadas}")
print(f"Total atendidos: {self.total_atendidos}")
print("-" * 40)

# Cálculos de promedios
# Lq = Promedio de clientes en cola
lq = self.areaCola / self.reloj

# Rho (Utilización promedio por servidor)
# Promedio de servidores ocupados / Total servidores
promedio_ocupados = self.area_ocupados / self.reloj
utilizacion_global = promedio_ocupados / self.n_servidores

# Wq (Tiempo promedio en cola) usando la fórmula de Little: Lq = lambda
# Usamos la tasa real de llegadas (total_llegadas / tiempo) por si varió
lambda_real = self.total_llegadas / self.reloj
wq = lq / lambda_real if lambda_real > 0 else 0

print(f"Longitud promedio cola (Lq): {lq:.4f} clientes")
print(f"Tiempo promedio en cola (Wq): {wq*60:.2f} minutos")
print(f"Servidores ocupados (promedio): {promedio_ocupados:.2f}")
print(f"Utilización del sistema (Rho): {utilizacion_global*100:.2f}%")
print("="*40 + "\n")

```

A continuación se utiliza la clase de Simulación M/M/c implementada. Si $n_{servidores}=1$ la cola crece infinitamente, esto produce un cuello de botella

```

# Configurar la simulación (8 horas, lambda=10, mu=4, c=3)
mi_simulacion = SimulacionMMC_Log(
    tiempo_max=8.0,
    tasa_llegada=10.0,
    tasa_servicio=4.0,
    n_servidores=3
)

# Correr la simulación matemática
mi_simulacion.correr()

```

--- Iniciando Simulación M/M/3 por 8.0 horas ---

```

=====
                RESULTADOS DE LA SIMULACIÓN
=====
Tiempo simulado: 8.00 horas

```

Total llegadas: 80

Total atendidos: 79

Longitud promedio cola (Lq): 1.5918 clientes
Tiempo promedio en cola (Wq): 9.55 minutos
Servidores ocupados (promedio): 2.16
Utilización del sistema (Rho): 71.85%
=====

Función para visualizar la simulación con GIF animado

def generar_gif_simulacion(simulacion, filename="simulacion_mmc.gif"):

"""

Toma un objeto SimulacionMMC_Log ya ejecutado y genera un GIF.

"""

historial = simulacion.historia

tiempo_total = simulacion.tiempo_max

servidores_total = simulacion.n_servidores

Configuración de la figura

fig, ax = plt.subplots(figsize=(10, 4))

ax.set_xlim(-1, servidores_total + 2)

ax.set_ylim(-1, 3)

ax.axis('off') # Quitar ejes para limpieza visual

titulo = ax.text(0.5, 1.05, f"Simulación M/M/{servidores_total}",
transform=ax.transAxes, ha='center', fontsize=14, weight='b')

Elementos estáticos

rects = []

for i **in** range(servidores_total):

Servidores (cuadrados)

r = plt.Rectangle((i + 1, 0), 0.8, 1, color='green', alpha=0.7)

ax.add_patch(r)

ax.text(i + 1.4, 0.5, f"S{i+1}", ha='center', color='black')

rects.append(r)

Elementos dinámicos

txt_reloj = ax.text(0.05, 0.9, '', transform=ax.transAxes, fontsize=12)

txt_info = ax.text(0.05, 0.8, '', transform=ax.transAxes, fontsize=10)

cola_scatter = ax.scatter([], [], c='blue', s=80, edgecolors='black', label=)

Función auxiliar para buscar estado en el tiempo t

def get_state_at_time(t):

Busca el último estado registrado antes o en el tiempo t

estado_actual = (0, 0, 0)

for estado **in** historial:

```

        if estado[0] > t:
            break
        estado_actual = estado
    return estado_actual

# Configuración de frames
fps = 15
segundos_gif = 8 # Cuánto quieres que dure el GIF en tiempo real
total_frames = fps * segundos_gif

print(f"Generando animación ({total_frames} frames)... esto puede tardar unos segundos")

def update(frame):
    # Mapear frame actual al tiempo de simulación (0 a 8 horas)
    t_sim = (frame / total_frames) * tiempo_total

    # Obtener estado
    _, nCola, nOcupados = get_state_at_time(t_sim)

    # Actualizar textos
    txt_reloj.set_text(f"Hora Simulación: {t_sim:.2f} h")
    txt_info.set_text(f"En Cola: {nCola} | Ocupados: {nOcupados}/{servidores}")

    # Actualizar color servidores
    for i, rect in enumerate(rects):
        if i < nOcupados:
            rect.set_color('#FF0000') # Rojo (Ocupado)
        else:
            rect.set_color('#00FF00') # Verde (Libre)

    # Actualizar puntos de la cola
    xCola, yCola = [], []
    # Limitamos visualmente a 30 personas para que no explote la gráfica
    visualCola = min(nCola, 30)
    for k in range(visualCola):
        xCola.append(0.5 - (k * 0.15)) # Se apilan hacia la izquierda
        yCola.append(0.5 + np.sin(k*1.5)*0.1) # Ondulación para efecto visual

    cola_scatter.set_offsets(np.c_[xCola, yCola])

    return rects + [cola_scatter, txt_reloj, txt_info]

anim = animation.FuncAnimation(fig, update, frames=total_frames, interval=1000)

# Guardar GIF

```

```
anim.save(filename, writer='pillow', fps=fps)
plt.close(fig) # Cerrar para no mostrar el plot estático
print(f"GIF guardado como: {filename}")

# Devolver imagen para Jupyter
return Image(filename=filename)
```

Para visualizar en cámara rápida (timelapse) lo simulado, generamos una imagen GIF animada en loop. Cuando los servidores están ocupados se muestran en rosa, y cuando están disponibles en verde. Si hay clientes esperando, se muestra la cola con puntos azules.

```
# Generar y mostrar el GIF
# Esto devolverá la animación directamente en el notebook
# En el PDF sólo se visualiza el primer fotograma estático
gif_animado = generar_gif_simulacion(mi_simulacion, "resultado_colas.gif")
display(gif_animado)
```

Generando animación (120 frames)... esto puede tardar unos segundos.
GIF guardado como: resultado_colas.gif

<IPython.core.display.Image object>

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 $P = \frac{\lambda(t)}{\lambda_{max}}$.

```
import numpy as np
import matplotlib.pyplot as plt

# 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

# Generación de reporte de la simulación
def generar_reporte_texto(events, total_candidates, t_max, dias):
    """
    Imprime un resumen estadístico de la simulación en la consola.
    """
    num_events = len(events)
    # Calcular tiempos entre llegadas (Inter-arrival times)
    iat = np.diff(events)

    print("="*60)
    print(f" REPORTE DE SIMULACIÓN DE EVENTOS (NHPP) - {dias} DÍAS")
    print("="*60)

    print(f"\n--- MÉTRICAS GENERALES ---")
    print(f"Tiempo total simulado : {t_max} horas")
    print(f"Eventos generados      : {num_events}")
    print(f"Candidatos totales      : {total_candidates}")
    print(f"Eficiencia del Thinning: {num_events/total_candidates:.2%} (Eventos)")
    print(f"Promedio global         : {num_events/t_max:.2f} eventos/hora")

    if len(iat) > 0:
        print(f"\n--- ESTADÍSTICAS DE TIEMPOS ENTRE LLEGADAS (IAT) ---")
        print(f"IAT Promedio           : {np.mean(iat):.4f} horas ({np.mean(iat)*")
        print(f"IAT Mínimo             : {np.min(iat):.6f} horas")
        print(f"IAT Máximo             : {np.max(iat):.4f} horas")
        print(f"Desviación Estándar    : {np.std(iat):.4f}")

    print(f"\n--- DESGLOSE DIARIO ---")
    print(f"{'Día':<10} | {'Rango Horario':<20} | {'Eventos':<10} | {'Promedio (")
    print("-" * 65)

    for d in range(dias):
        start_t = d * 24

```



```

end_t = (d + 1) * 24
# Filtrar eventos que caen en este día
day_events = events[(events >= start_t) & (events < end_t)]
count = len(day_events)
avg = count / 24
print(f"Día {d+1:<6} | {start_t:03d}h - {end_t:03d}h | {count:<10}

print("="*60)

```

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)

# Generar Reporte de Texto
generar_reporte_texto(events, candidates, HORAS_TOTALES, DIAS)

```

```

=====
REPORTE DE SIMULACIÓN DE EVENTOS (NHPP) - 7 DÍAS
=====

--- MÉTRICAS GENERALES ---
Tiempo total simulado : 168 horas
Eventos generados      : 3294
Candidatos totales     : 5776
Eficiencia del Thinning: 57.03% (Eventos aceptados / Candidatos)
Promedio global        : 19.61 eventos/hora

--- ESTADÍSTICAS DE TIEMPOS ENTRE LLEGADAS (IAT) ---
IAT Promedio          : 0.0510 horas (3.06 minutos)
IAT Mínimo             : 0.000020 horas
IAT Máximo             : 1.1459 horas
Desviación Estándar    : 0.0724

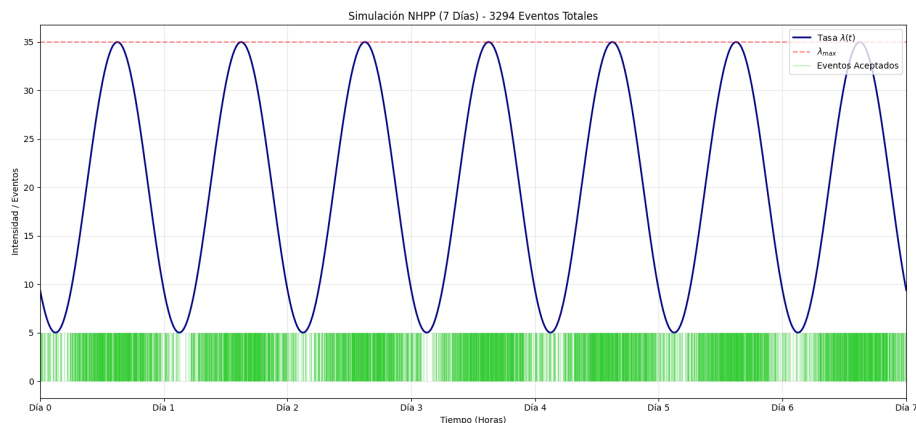
--- DESGLOSE DIARIO ---
Día      | Rango Horario      | Eventos | Promedio (ev/h)
-----
-----
Día 1    | 000h - 024h        | 487     | 20.29
Día 2    | 024h - 048h        | 454     | 18.92

```

Día 3	048h - 072h	442	18.42
Día 4	072h - 096h	479	19.96
Día 5	096h - 120h	472	19.67
Día 6	120h - 144h	469	19.54
Día 7	144h - 168h	491	20.46

Se visualizan los resultados de la simulación:

```
# Visualización
plt.figure(figsize=(15, 7))
t_values = np.linspace(0, HORAS_TOTALES, 500)
lambda_values = [intensity_function(t) for t in t_values]
plt.plot(t_values, lambda_values, color='navy', lw=2, label='Tasa  $\lambda(t)$ ')
plt.hlines(LAMBDA_MAX, 0, HORAS_TOTALES, colors='red', linestyle='--', alpha=0.3)
plt.vlines(x=events, ymin=0, ymax=5, colors='limegreen', alpha=0.4, linewidth=1)
plt.title(f'Simulación NHPP (7 Días) - {len(events)} Eventos Totales')
plt.xlabel('Tiempo (Horas)')
plt.ylabel('Intensidad / Eventos')
plt.xlim(0, HORAS_TOTALES)
dias_ticks = np.arange(0, HORAS_TOTALES + 1, 24)
plt.xticks(dias_ticks, [f'Día {i}' for i in range(len(dias_ticks))])
plt.grid(True, alpha=0.3)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



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.

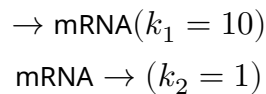
Parte 5 - Gillespie SSA o Next Reaction Method

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 ($P_{\text{ntesis}} = a_1/a_0$ y $P_{\text{degradacin}} = a_2/a_0$). 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:



Basado en esta descripción tenemos dos reacciones: 1. Producción (Transcripción): $\emptyset \xrightarrow{k_1} \text{mRNA}$. Tasa constante: $k_1 = 10$. Esta reacción incrementa el conteo de ARNm en 1. 3. Degradación: $\text{mRNA} \xrightarrow{k_2} \emptyset$. 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 ($k_2 \cdot \text{mRNA}$), el balance se logra matemáticamente cuando $k_1 = k_2 \cdot \text{mRNA}$; 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.

```
import numpy as np
import matplotlib.pyplot as plt

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

    # 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: Probabilidad de degradación (proporcional al número de moléculas)
        a2 = k2 * mRNA

        a_sum = a1 + a2

        # Si a_sum es 0, el sistema se detiene (no hay reacciones posibles)
        if a_sum == 0:
            break

        # 2. Determinar el tiempo hasta la 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

# Función para Generar el Informe
def generar_informe_texto(tiempos, conteos, k1, k2):
    # Convertir a arrays de numpy para facilitar cálculos estadísticos
    arr_conteos = np.array(conteos)

    # Cálculos estadísticos sobre la simulación
    media_simulada = np.mean(arr_conteos)
    desviacion_std = np.std(arr_conteos)
    varianza_sim = np.var(arr_conteos)
    min_val = np.min(arr_conteos)
    max_val = np.max(arr_conteos)
    total_eventos = len(tiempos) - 1
    tiempo_final = tiempos[-1]

    # Valores Teóricos
    media_teorica = k1 / k2
    # En un proceso de Poisson ideal, la varianza es igual a la media
    varianza_teorica = media_teorica

    # Cálculo del error relativo
    error_relativo = abs(media_simulada - media_teorica) / media_teorica * 100

    # Construcción del reporte de texto (f-string)
    informe = f"""
=====

```

INFORME TÉCNICO: SIMULACIÓN ESTOCÁSTICA (GILLESPIE SSA)

1. PARÁMETROS DEL SISTEMA

Reacción 1 (Producción) k_1 : {k1}
Reacción 2 (Degradación) k_2 : {k2}
Tiempo total simulado : {tiempo_final:.2f} u.t.
Total de eventos ocurridos : {total_eventos}

2. ANÁLISIS ESTADÍSTICO DE LA TRAYECTORIA

Media poblacional observada: {media_simulada:.4f} moléculas
Desviación estándar : {desviacion_std:.4f}
Varianza observada : {varianza_sim:.4f}
Rango de fluctuación : Mín [{min_val}] - Máx [{max_val}]

3. VALIDACIÓN CON MODELO TEÓRICO

Estado Estacionario Esp. : {media_teorica:.4f} moléculas
Varianza Esperada (Poisson): {varianza_teorica:.4f}
Error Relativo (Media) : {error_relativo:.2f}%
"""

return informe

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)

# Generar e imprimir informe
reporte = generar_informe_texto(tiempos, conteos, k1, k2)
print(reporte)
```

INFORME TÉCNICO: SIMULACIÓN ESTOCÁSTICA (GILLESPIE SSA)

1. PARÁMETROS DEL SISTEMA

-
Reacción 1 (Producción) k_1 : 10.0
Reacción 2 (Degradación) k_2 : 1.0
Tiempo total simulado : 1000.00 u.t.
Total de eventos ocurridos : 20185

2. ANÁLISIS ESTADÍSTICO DE LA TRAYECTORIA

-
Media poblacional observada: 10.7573 moléculas
Desviación estándar : 3.3082
Varianza observada : 10.9444
Rango de fluctuación : Mín [0] - Máx [24]

3. VALIDACIÓN CON MODELO TEÓRICO

-
Estado Estacionario Esp. : 10.0000 moléculas
Varianza Esperada (Poisson): 10.0000
Error Relativo (Media) : 7.57%

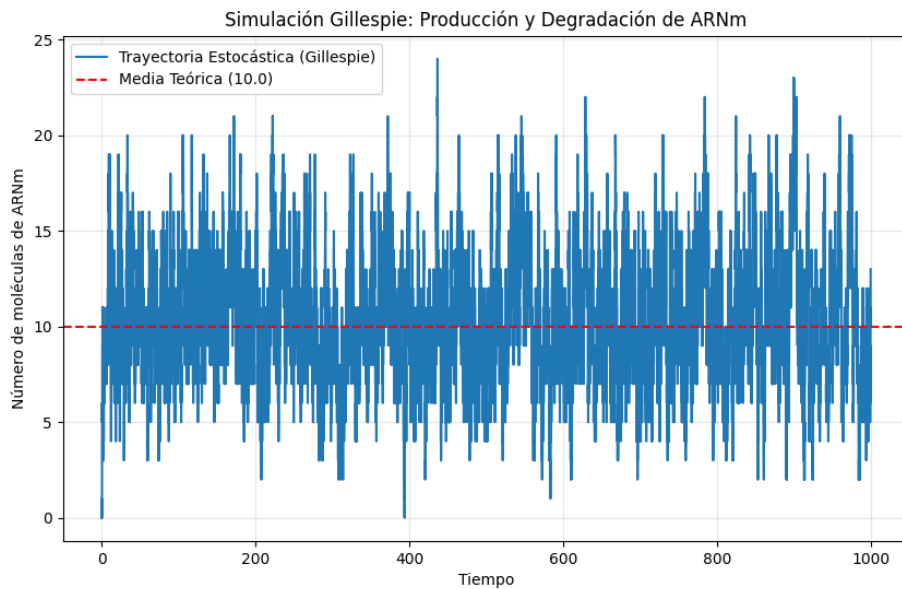
Se visualizan los resultados de la simulación:

```
# Visualización
plt.figure(figsize=(10, 6))

# Graficar la trayectoria estocástica (step plot es ideal para saltos discretos)
plt.step(tiempos, conteos, where='post', label='Trayectoria Estocástica (Gillespie)')

# Graficar la media teórica ( $k_1 / k_2$ )
media_teorica = k1 / k2
plt.axhline(y=media_teorica, color='r', linestyle='--', label=f'Media Teórica ({k1} / {k2})')

plt.title('Simulación Gillespie: Producción y Degradación de ARNm')
plt.xlabel('Tiempo')
plt.ylabel('Número de moléculas de ARNm')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



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

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 ($\mu \rightarrow 0$), provocando que las operaciones de aterrizaje y despegue se aborren. 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.