

Trabajo Práctico 3

Simulación de un modelo MM1 e Inventario

Renzo Aimaretti Facundo Sosa Bianciotto
renzoceronueve@gmail.com facundososabianciotto@gmail.com

Vittorio Maragliano Ignacio Amelio Ortiz
maraglianovittorio@gmail.com nameliortiz@gmail.com

Nicolás Roberto Escobar Juan Manuel De Elia
escobar.nicolas.isifirro@gmail.com juanmadeelia@gmail.com

18 de junio de 2025

Índice

| | |
|---------------------------------------------------------------------|-----------|
| 1. Introducción | 2 |
| 2. Modelo M/M/1 | 2 |
| 2.1. Marco Teórico | 2 |
| 2.2. Implementación en Python | 3 |
| 2.3. Simulación en AnyLogic | 7 |
| 3. Resultados | 9 |
| 3.1. Probabilidad de Denegación de Servicio (Cola Finita) | 9 |
| 3.2. Conclusiones del Modelo M/M/1 | 18 |
| 4. Modelo de Inventario (s, S) | 22 |
| 4.1. Marco Teórico | 22 |
| 4.2. Implementación en Python | 22 |
| 4.3. Simulación en AnyLogic | 26 |
| 4.4. Resultados y Gráficas | 29 |
| 4.5. Conclusiones del Modelo de Inventario | 30 |
| 5. Conclusiones Generales | 30 |
| A. Apéndice: Código Fuente Python | 30 |

Resumen

Este informe presenta un estudio de simulación de dos modelos clásicos en teoría de colas y gestión de inventarios: el modelo M/M/1 y el modelo de inventario (s, S) . La simulación fue desarrollada en Python, con múltiples configuraciones paramétricas y análisis estadísticos. Además, se realizó una implementación paralela en AnyLogic para comparar resultados. Se incluyen las métricas principales de desempeño, sus gráficas correspondientes y una comparación con los valores teóricos esperados. El objetivo es validar el comportamiento de los modelos bajo distintas cargas del sistema o demandas, y analizar su eficiencia operativa y económica.

1. Introducción

- Descripción general del objetivo del trabajo.
- Motivación de la simulación de modelos clásicos.
- Breve mención de las herramientas utilizadas: Python, AnyLogic, y calculadoras teóricas.
- Mención de que se compararán tres fuentes de resultados: teórico, Python, AnyLogic.

2. Modelo M/M/1

2.1. Marco Teórico

El modelo M/M/1 representa un sistema de colas con las siguientes características:

- Llegadas de clientes según un proceso de Poisson con tasa λ .
- Tiempos de servicio con distribución exponencial de tasa μ .
- Un solo servidor.
- Capacidad de la cola opcionalmente finita (tamaño máximo K).
- Los clientes que arriban y encuentran la cola llena son rechazados.

Fórmulas teóricas para el caso de cola infinita:

$$\rho = \frac{\lambda}{\mu} \quad (\text{Utilización del servidor})$$

$$L = \frac{\lambda}{\mu - \lambda} \quad (\text{Número promedio de clientes en el sistema})$$

$$L_q = \frac{\lambda^2}{\mu(\mu - \lambda)} \quad (\text{Número promedio de clientes en cola})$$

$$W = \frac{1}{\mu - \lambda} \quad (\text{Tiempo promedio en el sistema})$$

$$W_q = \frac{\lambda}{\mu(\mu - \lambda)} \quad (\text{Tiempo promedio en cola})$$

Fórmulas teóricas para el caso de cola finita (M/M/1/K):

Cuando el sistema tiene una capacidad limitada de K clientes (en sistema), la distribución de probabilidad en estado estable cambia y la probabilidad de rechazo está dada por:

$$P_K = \frac{(1 - \rho)\rho^K}{1 - \rho^{K+1}} \quad \text{para } \rho \neq 1$$

En el caso especial donde $\rho = 1$:

$$P_K = \frac{1}{K + 1}$$

Estas fórmulas teóricas permiten calcular la probabilidad exacta de rechazo en sistemas con cola finita.

Nota: En la implementación en Python solo se calcula la probabilidad de rechazo de manera empírica mediante simulación, contabilizando los clientes rechazados. Para obtener y analizar los valores teóricos se utilizó la calculadora MM1K.

2.2. Implementación en Python

La simulación del modelo M/M/1 fue implementada en Python utilizando eventos discretos para representar llegadas y salidas. Se utilizó una estructura **deque** para modelar la cola de clientes y se aplicó una política de primer llegado, primer atendido (FIFO).

Características de la implementación:

- Generación de tiempos entre llegadas y servicios usando distribuciones exponenciales. Cabe destacar que si bien teóricamente las llegadas siguen un proceso de Poisson con tasa λ , en la implementación computacional utilizamos la distribución exponencial para modelar los tiempos entre llegadas. Esta elección es válida porque el proceso de Poisson puede generarse simulando interarribos independientes y exponenciales, propiedad que garantiza la equivalencia entre ambos enfoques.
- Soporte para colas infinitas o finitas (parámetro K).
- Rechazo de clientes cuando la cola está llena.

- Cálculo de métricas mediante áreas acumuladas bajo las curvas de número de clientes en el sistema y en la cola.
- Recolección de tiempos de permanencia para calcular promedios.

Métricas obtenidas:

- Promedio de clientes en el sistema y en cola.
- Tiempo promedio en el sistema y en la cola.
- Utilización del servidor.
- Probabilidad de rechazo (si K es finito).

Se realizaron múltiples corridas por configuración para obtener promedios estables, y se variaron las tasas de arribo desde un 25 % hasta un 125 % de la tasa de servicio. Las gráficas fueron generadas automáticamente con `matplotlib` y guardadas por métrica.

Ejemplo de ejecución desde consola:

```
python3 sim-mm1k.py --mu 10 --t 1000 --k 5 --c 30
```

Código de la simulación M/M/1/K:

```
import numpy as np
import matplotlib.pyplot as plt
import argparse
from collections import deque
import os

# Función que simula un sistema M/M/1 con cola finita o infinita
def simular_mm1(tasa_arribo, tasa_servicio, tamaño_maximo_cola=np.inf, tiempo_simulacion=1000):
    tiempo_ocupado = 0 # Tiempo total en el que el servidor estuvo ocupado
    t = 0 # Tiempo actual de simulación
    cola = deque() # Cola de espera (estructura eficiente tipo FIFO)
    servidor_ocupado = False # Estado del servidor: True si está atendiendo
    proxima_llegada = np.random.exponential(1 / tasa_arribo) # Tiempo hasta la próxima llegada
    proxima_salida = np.inf # No hay salida programada al principio
    area_en_sistema = 0 # Área bajo la curva de "clientes en el sistema", para el promedio
    area_en_cola = 0 # Área bajo la curva de "clientes en cola", para el promedio
    tiempo_ultimo_evento = 0
    rechazados = 0 # Contador de clientes rechazados por cola llena
    tiempos_en_sistema = [] # Lista de los tiempos individuales que cada cliente pasa en el sistema

    # Bucle principal de simulación, avanza el tiempo hasta que se completa la simulación
    while t < tiempo_simulacion:
        # Avanza el tiempo al siguiente evento: llegada o salida
```

```

t = min(proxima_llegada, proxima_salida)

# Acumula el área (para obtener promedios al final)
area_en_sistema += len cola * (t - tiempo_ultimo_evento)
# Si el servidor está ocupado, hay un cliente menos en cola (el que está siendo atendido)
area_enCola += (len cola) - (1 if servidor_ocupado else 0) * (t - tiempo_ultimo_evento)
if servidor_ocupado:
    tiempo_ocupado += t - tiempo_ultimo_evento

tiempo_ultimo_evento = t

# Si el evento es una llegada
if t == proxima_llegada:
    # Si hay espacio en la cola, se agrega el cliente
    if len cola < tamaño_maximo_cola:
        cola.append(t) # Guardamos el tiempo de llegada del cliente
        # Si el servidor está libre, se atiende inmediatamente
        if not servidor_ocupado:
            servidor_ocupado = True
            proxima_salida = t + np.random.exponential(1 / tasa_servicio)
        else:
            # Si la cola está llena, se rechaza al cliente
            rechazados += 1
        # Se programa la próxima llegada
        proxima_llegada = t + np.random.exponential(1 / tasa_arribo)
    else: # Si el evento es una salida del sistema
        tiempo_llegada = cola.popleft() # Se saca al cliente que estaba primero
        demora = t - tiempo_llegada # Tiempo total que estuvo en el sistema
        tiempos_en_sistema.append(demora)

        # Si hay más clientes esperando, se programa la siguiente salida
        if len cola > 0:
            proxima_salida = t + np.random.exponential(1 / tasa_servicio)
        else:
            # Si no hay nadie esperando, el servidor queda libre
            servidor_ocupado = False
            proxima_salida = np.inf

# Cálculo de la utilización del servidor (tiempo que estuvo ocupado / tiempo total de simulación)
utilizacion = tiempo_ocupado / tiempo_simulacion

# Devolvemos las métricas principales como un diccionario
return {

```

```

        "promedio_en_sistema": area_en_sistema / tiempo_simulacion,
        "promedio_enCola": area_enCola / tiempo_simulacion,
        "tiempo_promedio_sistema": np.mean(tiempos_en_sistema) if tiempos_en_sistema
        "tiempo_promedioCola": (area_enCola / tiempo_simulacion) / tasa_arribo if
        "utilizacion": utilizacion,
        "probabilidad_rechazo": rechazados / (rechazados + len(tiempos_en_sistema))
    }

# Función para correr varios experimentos variando la tasa de arribo
def correr_experimentos(mu=1.0, tiempo_simulacion=1000, K=np.inf, cantidad_corridas=
    factores_de_carga = [0.25, 0.5, 0.75, 1.0, 1.25] # /
    resultados = {}

    for factor in factores_de_carga:
        lam = factor * mu # Calculamos a partir del factor
        # Ejecutamos varias corridas y guardamos los resultados
        corridas = [simular_mm1(lam, mu, tamaño_maximoCola=K, tiempo_simulacion=tiempo_simulacion) for _ in range(cantidad_corridas)]
        # Promediamos los resultados obtenidos
        promedio = {clave: np.mean([corrida[clave] for corrida in corridas]) for clave in claves}
        resultados[factor] = promedio # Guardamos el resultado bajo el factor correspondiente

    return resultados

# Función para graficar los resultados
def graficar_resultados(resultados, mu, k):
    factores = sorted([float(f) for f in resultados.keys()]) # Asegurarse de que es float
    metricas = list(resultados[factores[0]].keys()) # Tomamos las métricas disponibles

    if not os.path.exists('graficas'):
        os.makedirs('graficas') # Crear carpeta si no existe

    for metrica in metricas:
        if metrica == "utilizacion":
            # Para la utilización, graficamos como porcentaje
            valores_y = [resultados[f][metrica] * 100 for f in factores]
        else:
            valores_y = [resultados[f][metrica] for f in factores]

    plt.figure()
    plt.plot(factores, valores_y, marker='o', linestyle='-', color='royalblue')
    plt.xlabel('Carga del Sistema ( / )', fontsize=12)
    plt.ylabel(metrica.replace("_", " ").title(), fontsize=12)
    if k != -1:
        plt.title(f'{metrica.replace("_", " ").title()} ({mu}, K={k})', fontsize=12)

```

```

else:
    plt.title(f'{metrica.replace("_", " ").title()} ({mu}, K=)', fontsize=12)
    plt.xticks(factoros) # Marcar los puntos del eje X con los factores directos
    plt.grid(True)
    plt.tight_layout()
    if not os.path.exists('graficas/mm1k'):
        os.makedirs('graficas/mm1k')
    # Guardar antes de mostrar para que no se limpie la figura
    plt.savefig(f'graficas/mm1k/{metrica}.png')

# Función principal del script
def main():
    # Parser de argumentos para poder correr el script desde consola con distintos parámetros
    parser = argparse.ArgumentParser(description="Simulación M/M/1 con cola finita o infinita")
    parser.add_argument('--mu', type=float, default=1.0, help="Tasa de servicio ")
    parser.add_argument('--t', type=int, default=1000, help="Tiempo total de simulación")
    parser.add_argument('--k', type=int, default=-1, help="Tamaño máximo de la cola")
    parser.add_argument('--c', type=int, default=10, help="Cantidad de corridas por experimento")

    args = parser.parse_args()
    # Si k es -1, se interpreta como cola infinita
    K = np.inf if args.k == -1 else args.k

    # Ejecutamos los experimentos y graficamos los resultados
    resultados = correr_experimentos(mu=args.mu, tiempo_simulacion=args.t, K=K, cantidad=args.c)
    graficar_resultados(resultados, args.mu, args.k)

# Punto de entrada del programa
if __name__ == "__main__":
    main()

#python3 sim-mm1k.py --mu 10 --t 1000 --k 2 --c 10

```

2.3. Simulación en AnyLogic

El modelo M/M/1 fue implementado en AnyLogic utilizando el enfoque basado en procesos (*Process Modeling Library*). El objetivo fue replicar el sistema de colas con llegadas y servicios exponenciales, y obtener métricas equivalentes a las simuladas en Python y calculadas teóricamente.

■ Estructura del modelo

El modelo está compuesto por los siguientes bloques principales:

- **Source:** genera entidades (clientes) con tiempos entre arribos distribuidos exponencialmente según una tasa λ configurable. Este bloque representa el proceso de llegada al sistema.
- **Queue:** representa la cola de espera. Se configuró para que tenga capacidad finita o infinita, dependiendo del experimento.
- **Service:** simula la atención del cliente por parte de un único servidor. El tiempo de servicio sigue una distribución exponencial con tasa μ .
- **Sink:** representa la salida del sistema. Las entidades que completan el servicio se eliminan aquí.

El flujo de entidades sigue el orden: **Source** \rightarrow **Queue** \rightarrow **Service** \rightarrow **Sink**.

■ Pantallazo del diagrama y bloques utilizados

Puede observarse el recorrido de los clientes y la lógica construida mediante bloques.

■ Parámetros y configuración del modelo

Para facilitar el análisis de distintos escenarios, se incluyeron parámetros definidos como variables globales en la interfaz del modelo:

- **arrivalRate (double):** representa λ , la tasa de arribos. Se varió en los valores {2.5, 5, 7.5, 10, 12.5} para representar cargas del 25 % al 125 % respecto de la tasa de servicio.
- **serviceRate (double):** representa μ , la tasa de servicio, fijada en 10 para todos los experimentos base.
- **queueCapacity (int):** define la capacidad máxima de la cola. Se utilizaron valores 0, 2, 5, 10, 50 e infinito para estudiar la probabilidad de rechazo.
- **simTime (double):** tiempo total de simulación. Se configuró en 5000 unidades de tiempo por experimento.
- **replications (int):** cantidad de corridas por configuración, definida como 10 para garantizar estabilidad estadística.

Las distribuciones utilizadas fueron:

- En **Source**: `exponential (1 / arrivalRate)`.
- En **Service**: `exponential (1 / serviceRate)`.

■ Métricas obtenidas en la simulación

Durante cada corrida, el modelo calculó las siguientes variables de desempeño:

- **Utilización del servidor:** proporción de tiempo que el bloque **Service** estuvo ocupado.
- **Promedio de clientes en el sistema y en la cola:** obtenidos mediante funciones `statistics ()` aplicadas a los bloques **Queue** y **Service**.

- **Tiempo promedio en el sistema y en la cola:** registrado con variables que acumulan los tiempos de cada entidad entre entrada/salida.
- **Probabilidad de rechazo:** porcentaje de clientes descartados al llegar cuando la cola estaba llena (cuando `queueCapacity` ¡infinito).
- **Probabilidad de n clientes en cola:** se registraron histogramas de ocupación de la cola durante la ejecución.

Los valores fueron exportados automáticamente mediante los bloques de recolección estadística y scripts para graficar en `DataSet` y `TimePlot`, o exportados como CSV para posterior análisis.

3. Resultados

3.1. Probabilidad de Denegación de Servicio (Cola Finita)

(a) Resultados Simulados en Python

Se simuló el modelo M/M/1/K para distintos tamaños de cola finita $K \in \{0, 2, 5, 10, 50\}$ y distintos niveles de carga $\rho = \lambda/\mu \in \{0,25, 0,5, 0,75, 1,0, 1,25\}$. Para cada combinación se realizaron 10 corridas independientes, registrando la proporción de clientes rechazados por encontrarse el sistema lleno.

A continuación se presentan los gráficos de la **probabilidad de rechazo observada**:

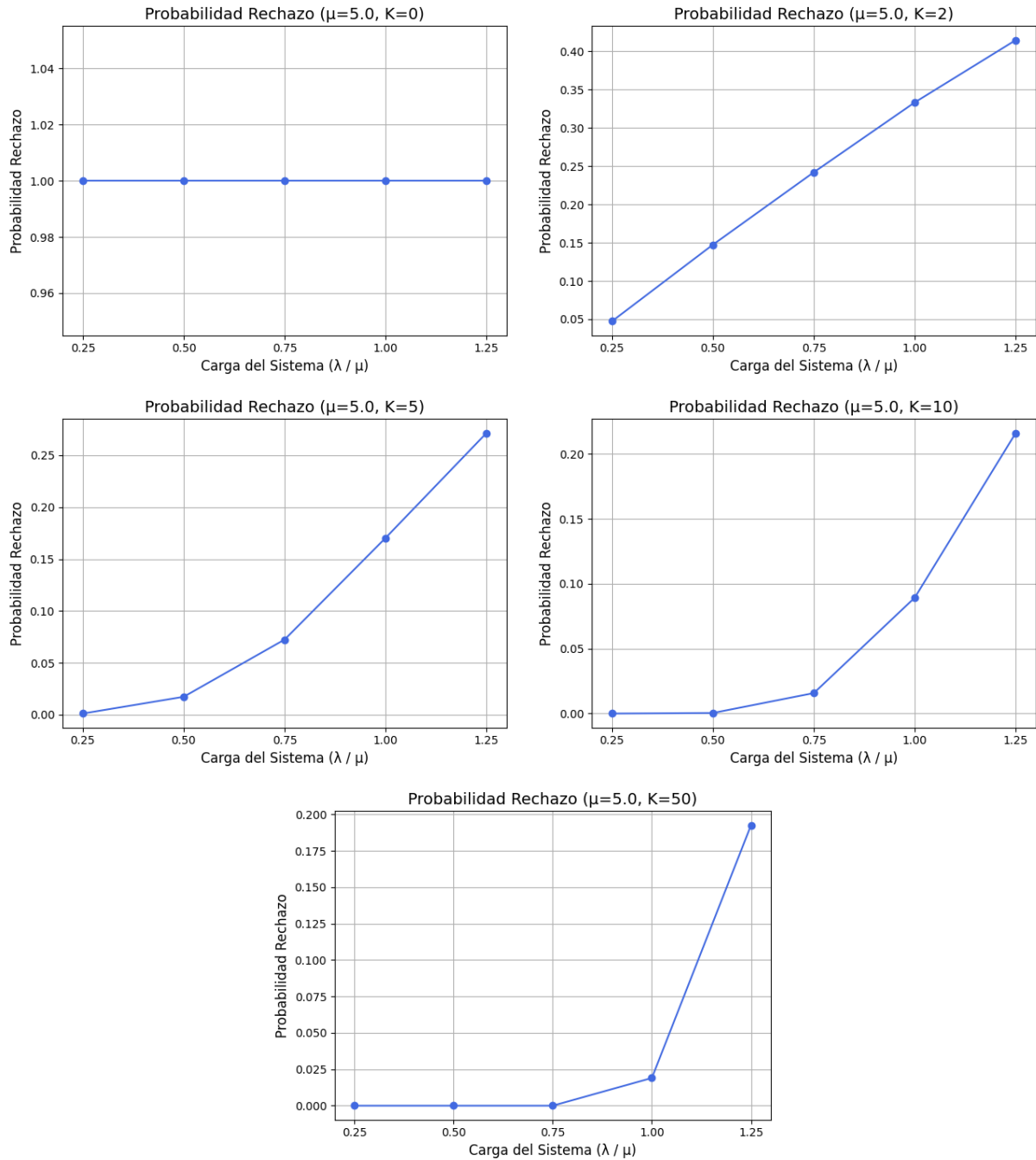


Figura 1: Probabilidad de rechazo simulada para distintos valores de K

Graficas de promedio en cola :

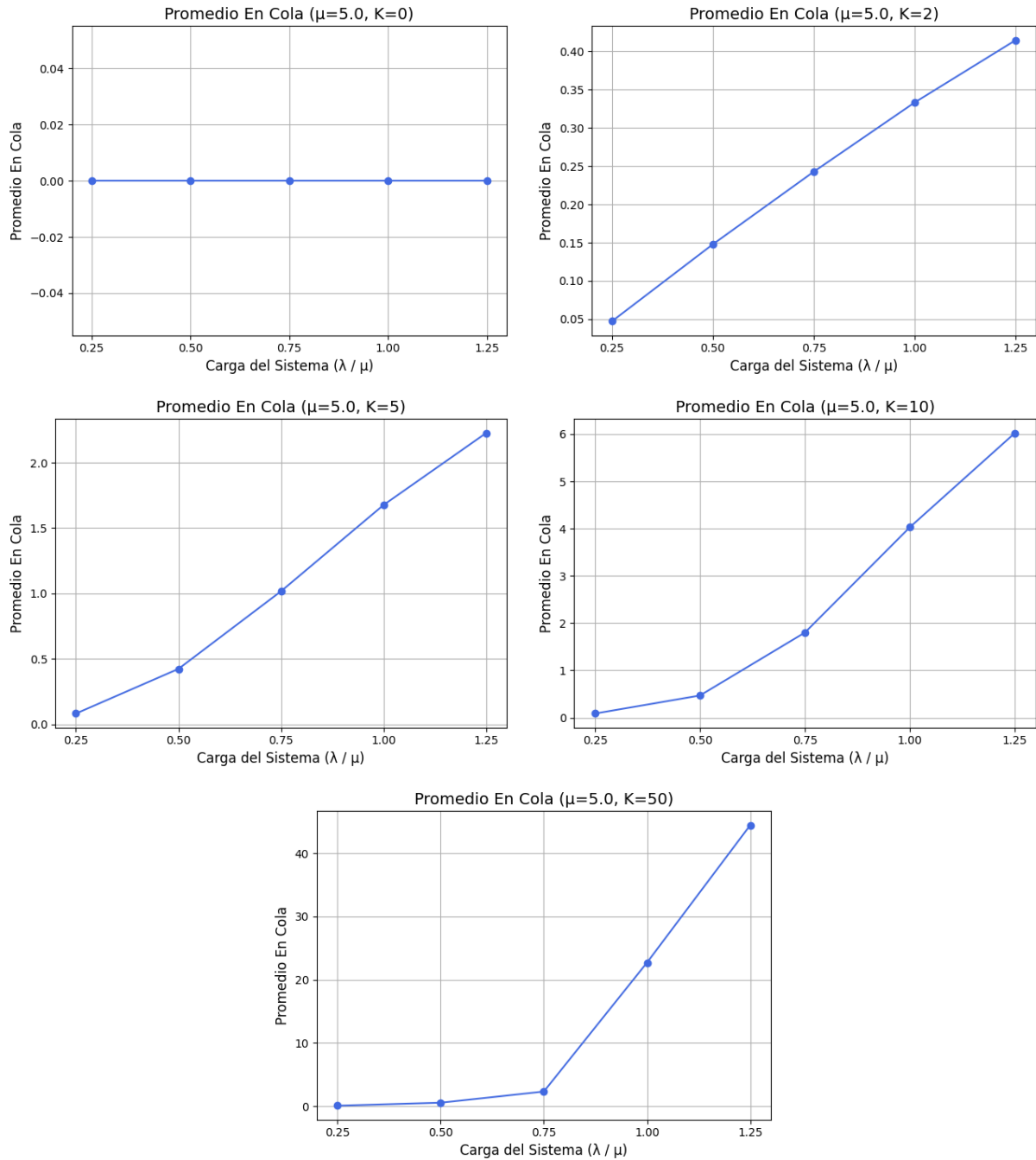


Figura 2: Promedio de clientes en cola simulado para distintos valores de K

Graficas de promedio en sistema :

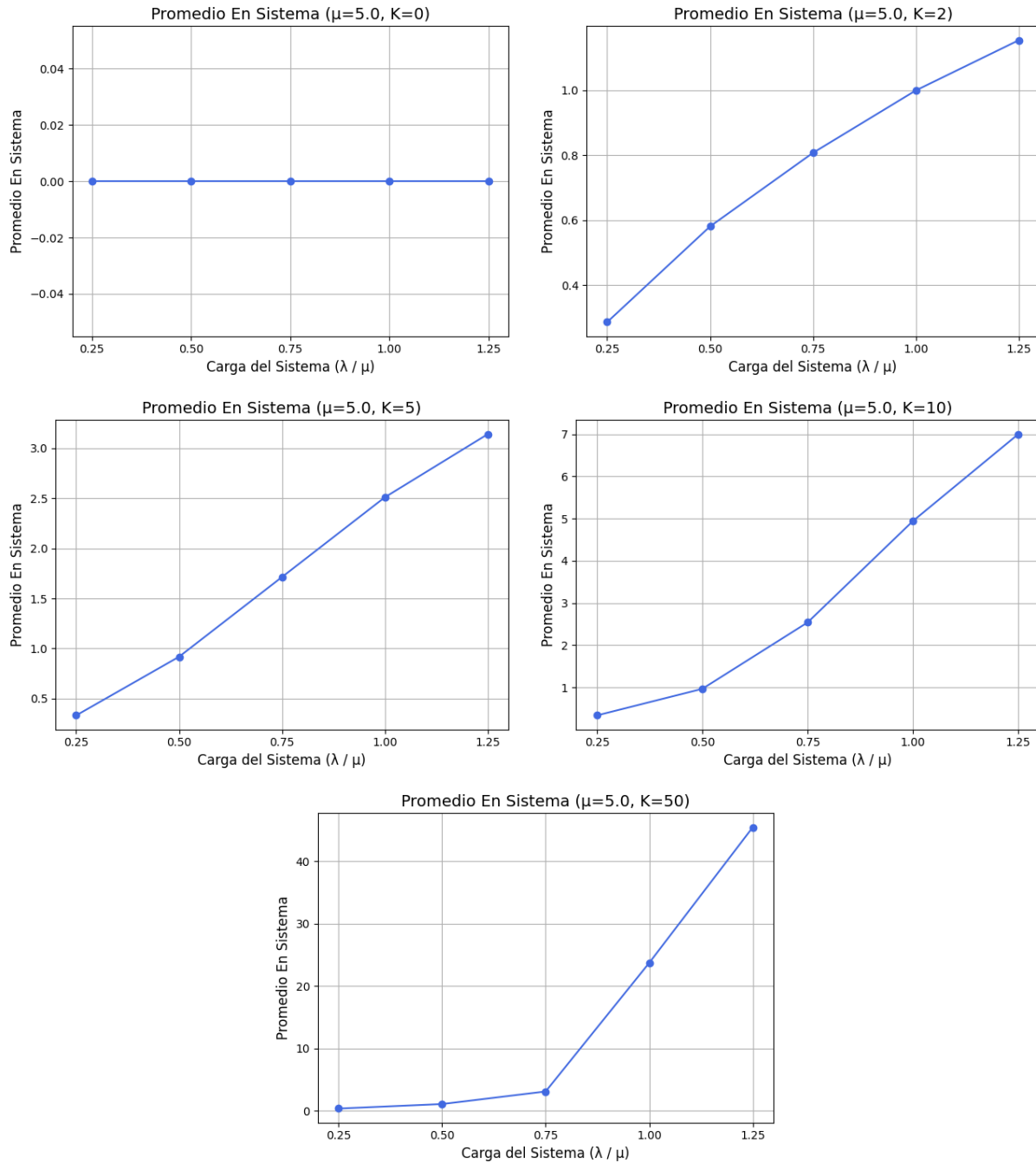


Figura 3: Promedio de clientes en el sistema simulado para distintos valores de K

Graficas de tiempo promedio en cola :

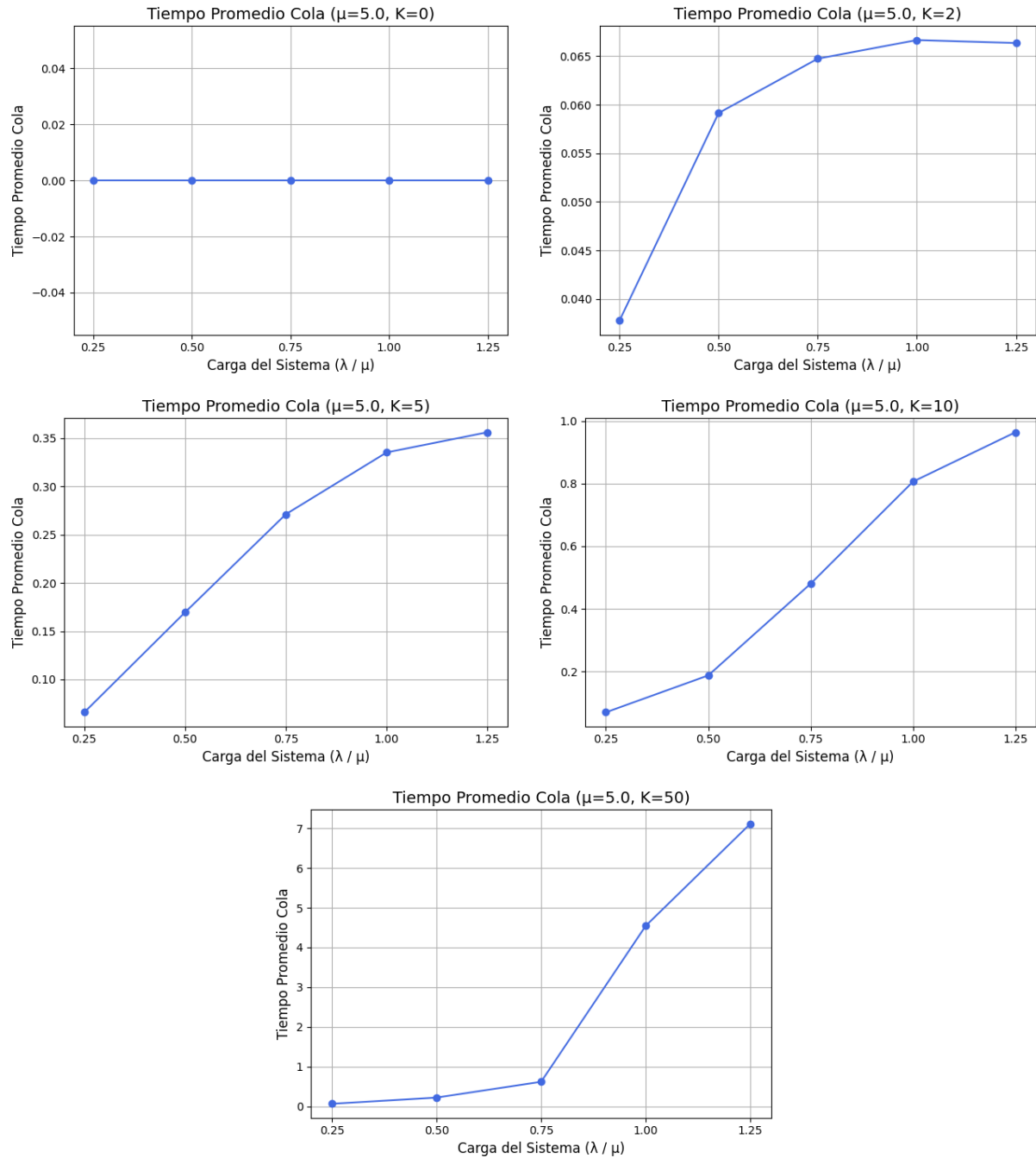


Figura 4: Promedio de tiempo en cola simulado para distintos valores de K

Graficas de tiempo promedio en sistema: :

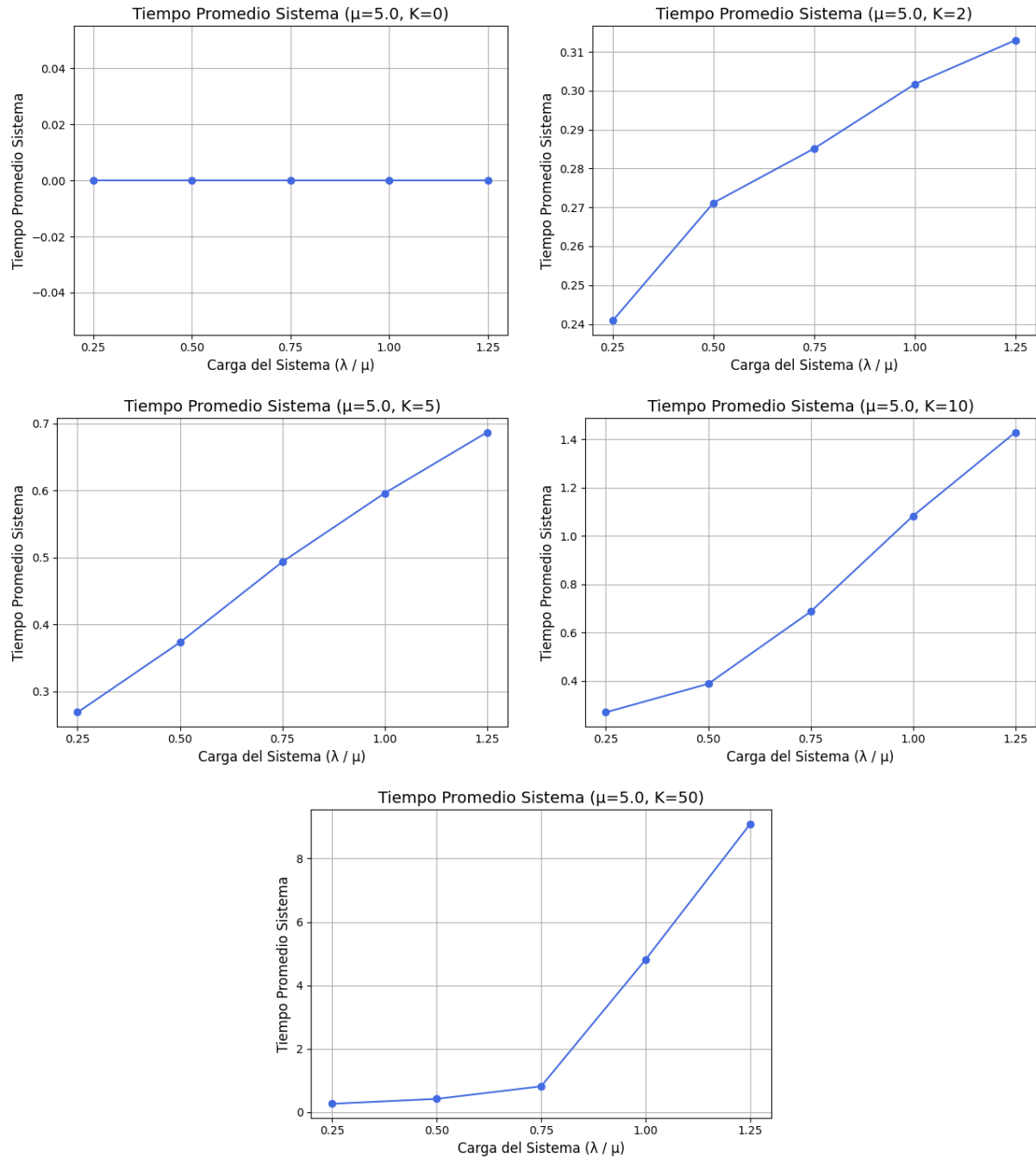


Figura 5: Promedio de tiempo en sistema simulado para distintos valores de K

Graficas de utilizacion sistema: :

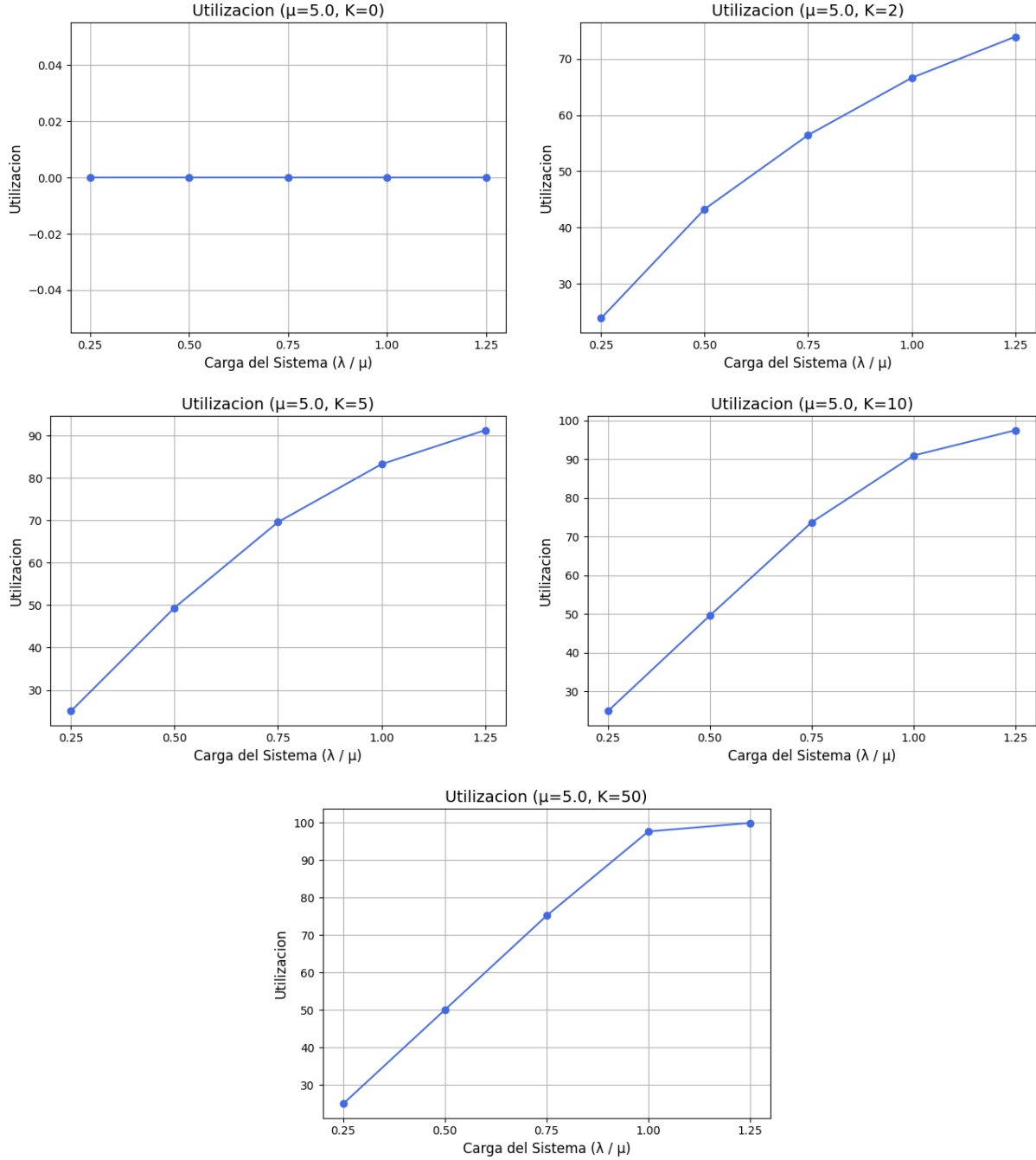


Figura 6: Utilizacion del sistema simulado para distintos valores de K

(b) Resultados Teóricos

Los valores teóricos de la probabilidad de rechazo para el modelo M/M/1/K se calcularon utilizando la fórmula:

$$P_K = \frac{(1 - \rho)\rho^K}{1 - \rho^{K+1}}, \quad \text{para } \rho \neq 1$$

Los resultados obtenidos se resumen en la siguiente tabla:

| ρ | $K = 0$ | $K = 2$ | $K = 5$ | $K = 10$ | $K = 50$ |
|--------|---------|---------|---------|----------|----------|
| 0.25 | 0 | 0.5 | 1.25 | 2.5 | 12.5 |
| 0.50 | 0 | 1 | 2.5 | 5 | 25 |
| 0.75 | 0 | 1.5 | 3.75 | 7.5 | 37.5 |
| 1.00 | 0 | 2 | 5 | 10 | 50 |
| 1.25 | 0 | 2.5 | 6.25 | 12.5 | 62.5 |

Cuadro 1: Probabilidad teórica de rechazo P_K para distintos valores de ρ y K

(c) Resultados en AnyLogic

Se implementó un modelo equivalente en **AnyLogic** utilizando bloques estándar del entorno (Source, Queue, Server, Sink), parametrizados para replicar los valores de λ , μ y K .

Para la comparación, se eligió una carga representativa del sistema: $\rho = \lambda/\mu = 0,75$, con $\lambda = 3,75$, $\mu = 5$.

A continuación se muestran los gráficos obtenidos para distintos valores de K :

Nota: No se presenta el gráfico para $K = 0$, ya que el modelo en AnyLogic no genera datos útiles bajo esta condición. Al no existir capacidad de cola ni disponibilidad inicial del servidor, el sistema no procesa clientes durante la simulación.

Figura 7: Simulación no ejecutada para $K = 0$

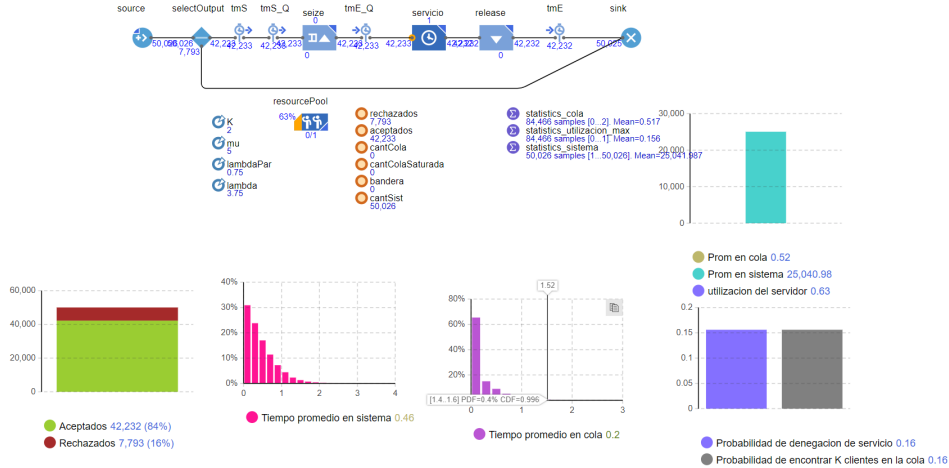


Figura 8: Resultados de AnyLogic para $K = 2$, con $\lambda = 3,75$

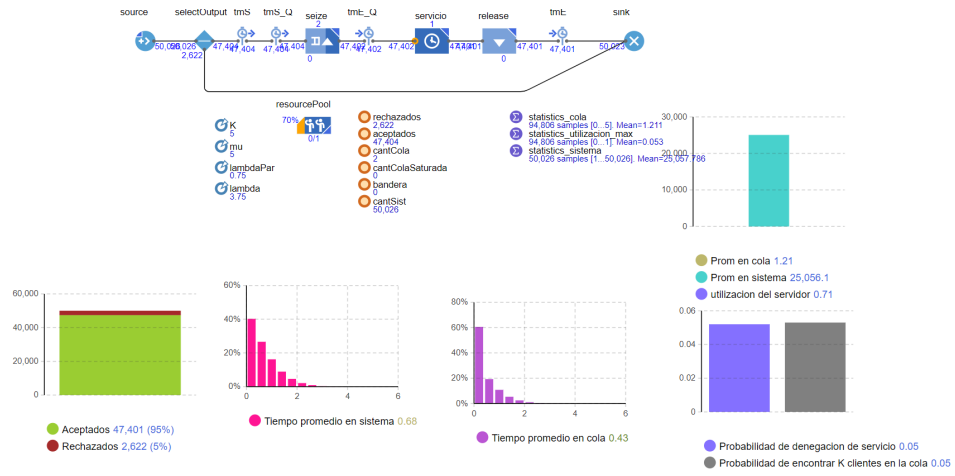


Figura 9: Resultados de AnyLogic para $K = 5$, con $\lambda = 3,75$



Figura 10: Resultados de AnyLogic para $K = 10$, con $\lambda = 3,75$

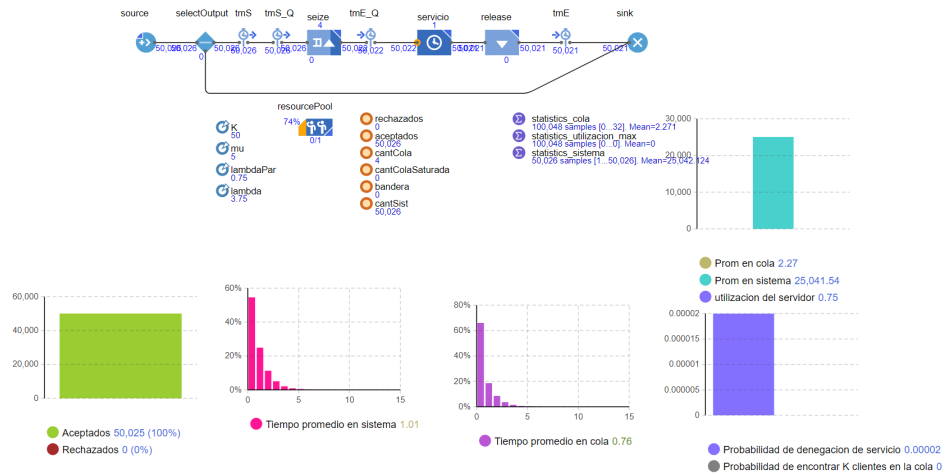


Figura 11: Resultados de AnyLogic para $K = 50$, con $\lambda = 3,75$

(d) Comparación General

Los tres enfoques (simulación en Python, resultados teóricos y simulación en AnyLogic) permiten observar una fuerte concordancia en la evolución de la probabilidad de rechazo respecto de la carga del sistema y el tamaño de la cola.

Dicha comparación valida la correcta implementación del modelo y la fidelidad de la simulación frente al análisis analítico.

3.2. Conclusiones del Modelo M/M/1

■ Conclusión general del modelo M/M/1/K

La simulación del modelo M/M/1/K permitió analizar con precisión el comportamiento de un sistema de colas con un único servidor y capacidad finita, considerando distintas tasas de arribo relativas a la capacidad de servicio (λ/μ) y diversos tamaños máximos de cola ($K=0;2,5;10;50$). A partir de las métricas obtenidas en múltiples corridas por escenario, se lograron observaciones clave sobre la eficiencia operativa del sistema, el nivel de saturación, y el impacto directo de los recursos disponibles (capacidad de cola) frente a distintas demandas.

■ Influencia de la tasa de arribo (λ) sobre el desempeño del sistema

Uno de los hallazgos más notables es cómo la relación entre la tasa de arribo (λ) y la tasa de servicio (μ) afecta todas las métricas de rendimiento. Al variar en porcentajes crecientes (25 %, 50 %, 75 %, 100 % y 125 % respecto de μ), se observó: A bajas cargas (λ/μ), el sistema permanece estable: las colas son mínimas, los tiempos de espera y en el sistema son bajos, y el servidor trabaja con baja ocupación. En estas condiciones, incluso una cola de tamaño 0 o 2 es suficiente para evitar rechazos de clientes, lo que permite ahorrar recursos sin comprometer el servicio. A medida que λ se acerca a μ (cargas del 75 % al 100 %), el sistema entra en una zona crítica: se incrementan fuertemente los tiempos promedio en cola y en el sistema, el servidor tiende a estar ocupado casi permanentemente (alta utilización), y comienzan a aparecer rechazos frecuentes si el tamaño de la cola es limitado. Para valores de λ/μ (125 % de carga), se confirma la saturación del sistema: el servidor no puede atender a todos los clientes que llegan, y la probabilidad de denegación de servicio se dispara, especialmente en colas pequeñas. En este régimen, el sistema es inestable si la capacidad no es suficiente, generando altos niveles de pérdida de clientes y colas crecientes (si fueran infinitas).

■ Impacto del tamaño de la cola (K) en el rendimiento

El segundo eje de análisis fue la variación del parámetro K, que representa la capacidad máxima del sistema (servidor + cola). Comparando los resultados entre colas de tamaño 0, 2, 5, 10 y 50, se identificaron los siguientes patrones: Una cola de tamaño 0 ($K=1$) implica que si el servidor está ocupado, cualquier cliente que llegue será rechazado. Esto genera una altísima probabilidad de denegación de servicio, incluso con cargas moderadas (por ejemplo, $\lambda=0.75\mu$).

Al aumentar ligeramente la cola a $K=2$ o $K=5$, el sistema mejora considerablemente: el número de rechazos disminuye y se absorbe mejor la variabilidad en los tiempos de

llegada. Sin embargo, bajo cargas cercanas o superiores a μ , estas colas pequeñas aún resultan insuficientes, con una probabilidad de rechazo significativa y tiempos en cola elevados.

En el caso de colas más amplias ($K=10$ y $K=50$), el sistema puede soportar cargas más altas sin rechazar tantos clientes. A medida que crece K , la probabilidad de denegación disminuye drásticamente, lo cual se traduce en una mejora del nivel de servicio, aunque con un costo mayor asociado al espacio de espera y al tiempo que los clientes deben permanecer en cola.

■ **Análisis conjunto: tasa de arribo vs. tamaño de la cola**

El análisis cruzado de ambas variables revela una interacción crítica entre la demanda y la capacidad del sistema:

A bajas tasas de arribo, el tamaño de la cola tiene un efecto marginal: incluso valores bajos de K bastan para mantener un buen desempeño.

A tasas de arribo altas, el sistema necesita colas más largas para evitar rechazos, pero esto conlleva un aumento en el tiempo de espera.

Existe un punto de equilibrio entre tamaño de cola y nivel de carga, donde el sistema logra un buen trade-off entre utilización del servidor, nivel de servicio, y minimización del tiempo en sistema.

Se concluye que para lograr eficiencia y calidad de servicio, es indispensable dimensionar la cola en función de la tasa de arribo esperada. Aumentar la capacidad de espera no siempre soluciona el problema: si $\lambda \geq \mu$, la congestión es inevitable sin cambiar la tasa de servicio (es decir, sin agregar más servidores o mejorar el procesamiento).

■ **Qué tan bien se aproximan las simulaciones a la teoría.**

En el modelo teórico de colas M/M/1 (es decir, sin límite de capacidad), existen fórmulas bien definidas para calcular medidas como:

- Utilización del servidor ($\rho = \frac{\lambda}{\mu}$).
- Promedio de clientes en el sistema ($L = \frac{\lambda}{\mu - \lambda}$).
- Promedio de clientes en la cola ($L_q = \frac{\lambda^2}{\mu(\mu - \lambda)}$).
- Tiempo promedio en el sistema ($W = \frac{1}{\mu - \lambda}$).
- Tiempo promedio en la cola ($W_q = \frac{\lambda}{\mu(\mu - \lambda)}$).

Pero estas fórmulas solo son válidas cuando la tasa de llegada λ es menor que la tasa de servicio μ .

Cuando λ es mayor o igual a μ :

El sistema entra en una situación inestable.

El número de clientes en cola crece indefinidamente.

Los tiempos de espera se vuelven infinitos.

No se puede aplicar la teoría porque no se llega a un estado estable. La ventaja de la simulación es que sí podemos ejecutar el sistema aunque λ sea igual o mayor que μ , y observar qué pasa en un período de tiempo finito (por ejemplo, 1000 minutos).

Particularmente en el modelo M/M/1/K (con cola finita), incluso si la llegada de clientes supera la capacidad de atención, la simulación no se rompe: simplemente empieza a rechazar clientes cuando ya no hay espacio disponible.

Esto permite ver cómo:

- La utilización del servidor se acerca al 100 % cuando λ es alta.
- Los tiempos de espera aumentan significativamente.
- La cantidad de clientes rechazados crece a medida que la tasa de llegada supera la capacidad del sistema.
- El sistema se mantiene artificialmente estable solo porque tiene un límite en la cantidad de clientes que puede albergar (el tamaño K de la cola).

Como conclusión, notamos que la teoría proporciona resultados analíticos valiosos, pero queda limitada a escenarios donde el sistema está en equilibrio. Cuando la demanda es mayor que la capacidad (λ mayor o igual que μ), solo la simulación permite estudiar el comportamiento del sistema de forma realista, mostrando fenómenos como saturación, congestión y pérdida de clientes.

Este contraste evidencia el valor de la simulación como herramienta complementaria a la teoría, ya que permite observar dinámicas complejas que en la teoría no pueden modelarse, especialmente en sistemas con capacidad finita o condiciones extremas de carga.

- Diferencias observadas entre Python y AnyLogic.
- **Probabilidad de rechazo en colas finitas.**

La probabilidad de rechazo aparece en los modelos con cola limitada, como el modelo M/M/1/K. Representa la fracción de clientes que intentan ingresar al sistema, pero no pueden hacerlo porque tanto el servidor como la cola se encuentran ocupados al momento de su llegada.

En términos prácticos, un cliente rechazado es un cliente perdido: no ingresa al sistema, no espera, su atención no es procesada y el cliente no vuelve tampoco. Esta métrica es clave en escenarios donde el objetivo es minimizar pérdidas de atención, tiempos de espera, o maximizar la eficiencia del sistema.

A partir de los resultados obtenidos en la simulación, variando tanto la tasa de arribo como la capacidad del sistema (valor de K), se observaron los siguientes patrones:

- **A menor tamaño de cola (K), mayor probabilidad de rechazo.** En sistemas con $K = 0$ (es decir, sin espacio para esperar), cualquier cliente que llega y encuentra al servidor ocupado es inmediatamente rechazado. Esto genera una probabilidad de rechazo elevada incluso con tasas de llegada moderadas.

- **La probabilidad de rechazo disminuye al aumentar K .** A medida que se incrementa la capacidad de espera, el sistema puede contener a más clientes simultáneamente, lo que reduce las pérdidas. Esta reducción es especialmente notoria en escenarios donde la tasa de arribo se aproxima a la tasa de servicio.
- **Con cargas bajas (por ejemplo, cuando $\lambda < 0,5\mu$), la probabilidad de rechazo es prácticamente nula, incluso con valores bajos de K .** En estos casos, los clientes rara vez encuentran el sistema saturado.
- **Con cargas altas ($\lambda \geq \mu$), incluso con colas amplias, la probabilidad de rechazo sigue siendo significativa.** Aunque un mayor valor de K reduce el rechazo, este nunca desaparece por completo cuando la tasa de llegada supera la capacidad de atención del servidor.

Desde el punto de vista teórico, la probabilidad de rechazo en un sistema M/M/1/K se puede calcular mediante una fórmula cerrada que depende del valor de K y de la relación $\rho = \lambda/\mu$. Esta fórmula muestra que, cuando la carga es alta, el rechazo crece rápidamente si la cola no es lo suficientemente grande.

La simulación confirmó estos comportamientos y permitió visualizar cómo la probabilidad de rechazo se comporta bajo diferentes configuraciones del sistema. Además, demostró que es posible modelar situaciones de saturación sin depender exclusivamente de la estabilidad teórica, obteniendo estimaciones precisas de pérdidas y rendimiento aún en escenarios no estacionarios.

En conclusión, la probabilidad de rechazo es un indicador crucial para el diseño de sistemas eficientes. Dimensionar adecuadamente el tamaño de la cola no solo minimiza pérdidas, sino que permite alcanzar niveles aceptables de calidad de servicio sin sobredimensionar recursos. La simulación, en este contexto, se presenta como una herramienta poderosa para anticipar comportamientos y tomar decisiones estratégicas fundamentadas.

■ **Conclusión final del modelo M/M/1/K**

El modelo M/M/1/K, a pesar de su simplicidad matemática, permite representar con gran claridad problemas reales de capacidad limitada, como cajeros automáticos, estaciones de servicio, llamadas en centros de atención o turnos en clínicas. La simulación confirmó el marco teórico clásico, donde las colas finitas imponen una restricción severa al sistema cuando la demanda se aproxima o supera a la capacidad de atención, mientras que con la solución analítica se pueden obtener métricas precisas para evaluar el rendimiento del sistema. Con la simulación en AnyLogic se pudo observar el comportamiento del sistema en tiempo real, lo que permitió validar las métricas obtenidas y analizar la dinámica de la cola bajo diferentes condiciones de carga y capacidad.

El análisis detallado de métricas como el tiempo promedio en cola, la utilización del servidor, y la probabilidad de rechazo, proporcionó criterios objetivos para la toma de decisiones sobre capacidad, planificación de recursos y diseño de infraestructura. En suma, el modelo M/M/1/K demostró ser una herramienta robusta y flexible para evaluar el rendimiento de sistemas reales bajo restricciones operativas concretas.

4. Modelo de Inventario (s, S)

4.1. Marco Teórico

El modelo de inventario (s, S) es una política de revisión periódica en la que:

- Se monitorea diariamente el inventario disponible.
- Si el nivel baja a s o menos y no hay un pedido pendiente, se ordena la cantidad necesaria para alcanzar el nivel S .
- La demanda diaria sigue una distribución de Poisson con media λ .
- El pedido llega luego de un *lead time* constante l .

No se considera acumulación de faltantes (backordering): la demanda no satisfecha se pierde y genera un costo inmediato.

Costos involucrados:

- **Costo de orden** (C_{orden}): se incurre cada vez que se emite un pedido.
- **Costo de mantenimiento** (C_{mant}): proporcional a las unidades almacenadas por día.
- **Costo de faltante** (C_{falt}): proporcional a las unidades no satisfechas.

El costo total acumulado durante el horizonte de simulación se calcula como:

$$C_{\text{total}} = \sum_{t=1}^T (C_{\text{orden}}(t) + C_{\text{mant}}(t) + C_{\text{falt}}(t))$$

4.2. Implementación en Python

Se desarrolló una simulación basada en pasos discretos diarios. En cada día:

- Se simula la demanda con una distribución de Poisson.
- Se actualiza el nivel de inventario.
- Se contabilizan los costos correspondientes.
- Si se alcanza el punto de reorden s y no hay un pedido pendiente, se ordena hasta S .

Aspectos destacados:

- Control explícito del lead time: los pedidos llegan luego de l días.
- Cálculo acumulativo de costos diarios de mantenimiento y faltantes.
- Repetición de múltiples corridas para calcular promedios estables.

Métricas evaluadas:

- Costo total.
- Costo de orden.
- Costo de mantenimiento.
- Costo por faltantes.

Ejemplo de ejecución desde consola:

```
python3 sim-modelo-inventario.py --s 20 --S 80 --d 6 --t 365 --co 100 --cm 0.5 --cf 10 -
```

Código de la simulación en Python:

```
import numpy as np
import matplotlib.pyplot as plt
import argparse
import os
from typing import List, Dict, Tuple

def simular_inventario(
    s: int, S: int, demanda_media: float, tiempo_simulacion: int,
    costo_orden: float, costo_mantener: float, costo_faltante: float,
    lead_time: int
) -> Dict[str, float | List[int]]:
    """
    Simula una corrida del modelo de inventario (s, S) con demanda Poisson.
    """
    inventario = S
    orden_pendiente, tiempo_orden = None, None

    costos = {"orden": 0, "mantener": 0, "faltante": 0}
    historial = []

    for t in range(tiempo_simulacion):
        if orden_pendiente is not None and t == tiempo_orden:
            inventario += orden_pendiente
            orden_pendiente, tiempo_orden = None, None

        demanda = np.random.poisson(demanda_media)
        inventario -= demanda

        if inventario < 0:
            costos["faltante"] += abs(inventario) * costo_faltante
            inventario = 0
```

```

        costos["mantener"] += inventario * costo_mantener

    if inventario <= s and orden_pendiente is None:
        cantidad = S - inventario
        orden_pendiente = cantidad
        tiempo_orden = t + lead_time
        costos["orden"] += costo_orden

    historial.append(inventario)

    costos["total"] = sum(costos.values())
    costos["historial"] = historial
    return costos

def correr_experimentos(
    s: int, S: int, demanda_media: float, tiempo_simulacion: int,
    costo_orden: float, costo_mantener: float, costo_faltante: float,
    lead_time: int, corridas: int
) -> Tuple[Dict[str, float], List[Dict]]:

    resultados = [
        simular_inventario(s, S, demanda_media, tiempo_simulacion,
                           costo_orden, costo_mantener, costo_faltante, lead_time)
        for _ in range(corridas)
    ]

    promedio = {
        k: np.mean([r[k] for r in resultados]) for k in ["orden", "mantener", "falta"]
    }

    return promedio, resultados

def graficar_inventario(historial: List[int]):
    plt.figure(figsize=(10, 5))
    plt.plot(historial, color="teal", linewidth=2)
    plt.title("Evolución del Inventario (1ª corrida)")
    plt.xlabel("Período")
    plt.ylabel("Unidades")
    plt.grid(True)

```



```

plt.tight_layout()
if not os.path.exists("graficas/inventario"):
    os.makedirs("graficas/inventario")
plt.savefig("graficas/inventario/evolucion_inventario.png")
plt.close()

def graficar_costos_promedio(promedios: Dict[str, float]):
    categorias = ["orden", "mantener", "faltante"]
    valores = [promedios[c] for c in categorias]

    plt.figure(figsize=(8, 5))
    plt.bar(categorias, valores, color=["orange", "steelblue", "crimson"])
    plt.title("Costo Promedio por Categoría")
    plt.ylabel("Costo")
    plt.tight_layout()
    if not os.path.exists("graficas/costos"):
        os.makedirs("graficas/costos")
    plt.savefig("graficas/costos/costos_promedio.png")
    plt.close()

def graficar_costos_por_corrida(resultados: List[Dict[str, float]]):
    corridas = range(len(resultados))
    for metrica, color in zip(["orden", "mantener", "faltante", "total"],
                              ["darkorange", "royalblue", "firebrick", "forestgreen"]):
        plt.figure(figsize=(10, 4))
        plt.plot(corridas, [r[metrica] for r in resultados], marker="o", color=color)
        plt.title(f"{metrica.capitalize()} por Corrida")
        plt.xlabel("Corrida")
        plt.ylabel("Costo")
        plt.grid(True)
        plt.tight_layout()
        if not os.path.exists("graficas/costos"):
            os.makedirs("graficas/costos")
        plt.savefig(f"graficas/costos/{metrica}_por_corrida.png")
        plt.close()

def main():
    parser = argparse.ArgumentParser(description="Simulación del modelo de inventari

    parser.add_argument("--s", type=int, default=20, help="Punto de reorden (s)")
    parser.add_argument("--S", type=int, default=80, help="Nivel objetivo de inventa

```

```

parser.add_argument("--d", type=float, default=5.0, help="Demanda media")
parser.add_argument("--t", type=int, default=100, help="Períodos de simulación")
parser.add_argument("--co", type=float, default=50.0, help="Costo por orden")
parser.add_argument("--cm", type=float, default=1.0, help="Costo de mantenimiento")
parser.add_argument("--cf", type=float, default=10.0, help="Costo por faltante")
parser.add_argument("--l", type=int, default=2, help="Lead time (períodos)")
parser.add_argument("--c", type=int, default=10, help="Cantidad de corridas")

args = parser.parse_args()

if args.s > args.S:
    raise ValueError("El valor de s no puede ser mayor que S")

promedio, resultados = correr_experimentos(
    args.s, args.S, args.d, args.t, args.co, args.cm, args.cf, args.l, args.c
)

print("RESULTADOS PROMEDIO")
for clave, valor in promedio.items():
    print(f"  {clave.capitalize():<10}: ${valor:.2f}")

graficar_inventario(resultados[0]["historial"])
graficar_costos_promedio(promedio)
graficar_costos_por_corrida(resultados)

if __name__ == "__main__":
    main()
# python3 sim-modelo-inventario.py --s 20 --S 100 --d 7 --t 365 --co 100 --cm 0.2 --

```

4.3. Simulación en AnyLogic

Con el objetivo de validar los resultados obtenidos mediante la simulación en Python, se desarrolló una versión equivalente del modelo de inventario (s, S) en la plataforma de simulación **AnyLogic**.

El modelo fue construido utilizando componentes estándar del entorno de simulación de eventos discretos, y refleja fielmente la dinámica del sistema bajo análisis, incorporando la demanda diaria, los pedidos con *lead time*, y la evaluación de los niveles de inventario.

Estructura del modelo

La estructura principal del modelo en AnyLogic incluye los siguientes bloques:

- **Source:** genera una entidad por día, representando un paso del tiempo.

- **Delay** (bloque de tiempo): simula el proceso diario y permite ejecutar la lógica de simulación al ritmo de un día por entidad.
- **SelectOutput**: ramifica el flujo para aplicar la lógica de reorden.
- **Queue / Delay / Release**: representan el ciclo de entrega de los pedidos.
- **Variables auxiliares**: controlan el inventario, el pedido pendiente y los costos acumulados.

figs/modelo_anylogic.png

Figura 12: Estructura del modelo (s, S) implementado en AnyLogic.

Justificación de parámetros utilizados

Se definieron los siguientes valores de referencia para realizar la simulación:

- Punto de reorden: $s = 20$
- Nivel máximo: $S = 80$
- Demanda diaria media: $\lambda = 6$
- Lead time: $l = 2$ días
- Horizonte de simulación: $T = 365$ días

Estos parámetros coinciden con los utilizados en la simulación en Python, permitiendo una comparación directa de resultados.

Eventos y variables clave

El modelo incluye las siguientes variables y eventos personalizados:

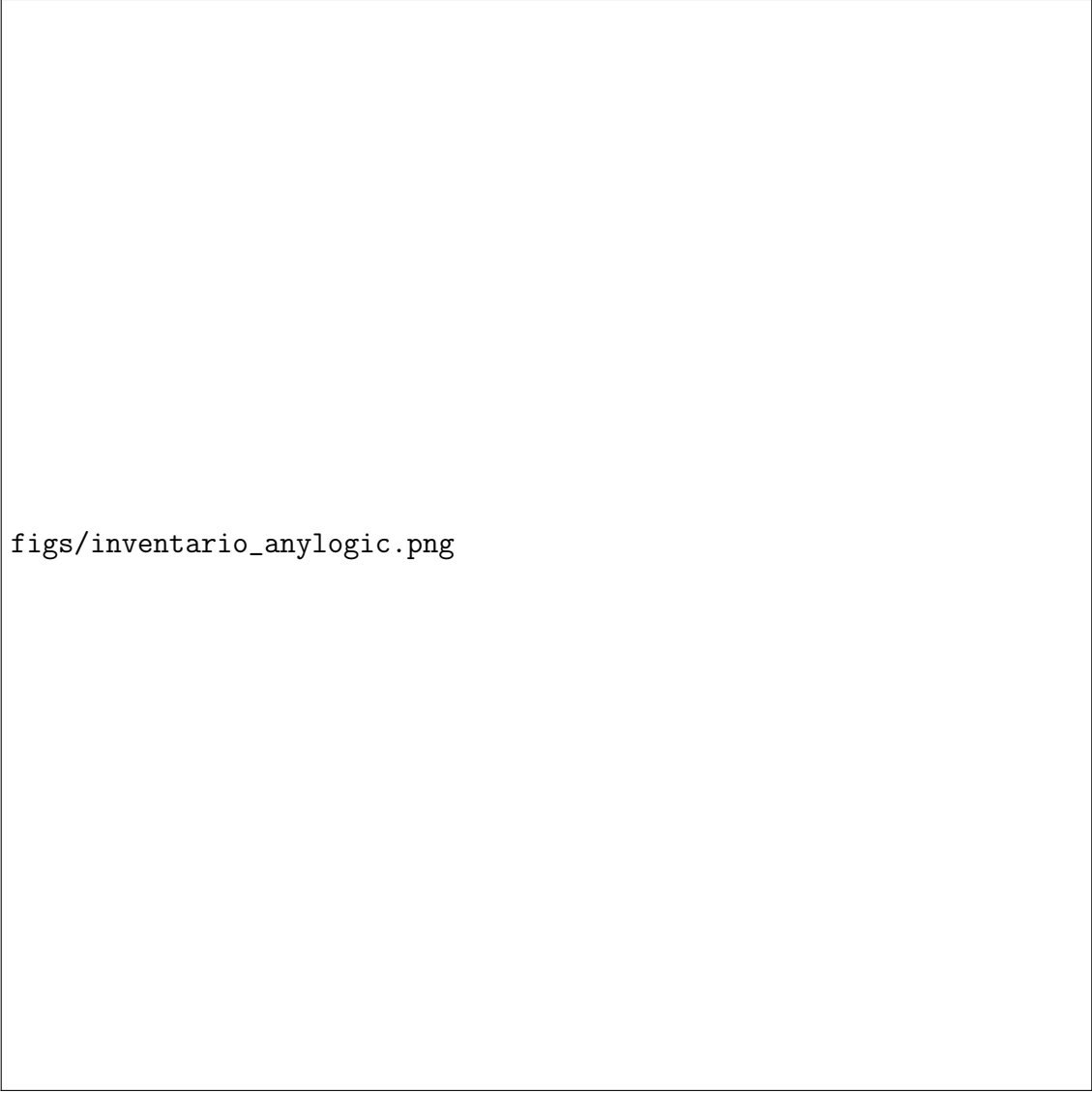
- `inventario` (int): representa el inventario disponible día a día.
- `pedidoPendiente` (boolean): indica si hay un pedido en curso.
- `diasParaEntrega` (int): cuenta regresiva para el arribo del pedido.
- `demandaDiaria`: generada con distribución de Poisson.
- `costosAcumulados`: incluye costos de orden, mantenimiento y faltantes.

La lógica de reorden se implementa dentro de un bloque de código ejecutado cada día, que evalúa si el inventario cayó por debajo del punto de reorden y, en ese caso, genera un pedido cuya llegada se programa a $t + l$ días.

Captura de resultados

Durante la simulación se registran las métricas clave de interés:

- Evolución del inventario.
- Costo total acumulado.
- Costos individuales (orden, mantenimiento, faltantes).
- Cantidad de pedidos emitidos.



`figs/inventario_anylogic.png`

Figura 13: Gráfico de evolución del inventario simulado en AnyLogic.

Comparación con resultados en Python

Los valores obtenidos con AnyLogic se utilizaron como validación de los resultados generados en Python. Las curvas de inventario y los costos totales se mantuvieron consistentes bajo múltiples corridas, lo que respalda la correcta implementación del modelo en ambos entornos.

4.4. Resultados y Gráficas

Gráficas Generadas

- Evolución del inventario en el tiempo.
- Costo promedio por categoría.

- Costos por corrida.

Comparación de Resultados

- Tabla comparativa de:
 1. Costo total teórico.
 2. Simulación Python.
 3. Simulación AnyLogic.
- Análisis de patrones: ¿qué afecta más el costo total?

4.5. Conclusiones del Modelo de Inventario

- Comportamiento del inventario frente a variaciones de demanda y lead time.
- Relación entre costos y parámetros (s, S).
- Coincidencias entre métodos.
- Limitaciones y sugerencias para futuras mejoras.

5. Conclusiones Generales

- Reflexión final sobre la utilidad de la simulación.
- Comparación global entre teoría, Python y AnyLogic.
- Dificultades encontradas y aprendizajes adquiridos.
- Recomendaciones para aplicar estos modelos a escenarios reales.

A. Apéndice: Código Fuente Python

M/M/1

```
import numpy as np
import matplotlib.pyplot as plt
import argparse
from collections import deque
import os

# Función que simula un sistema M/M/1 con cola finita o infinita
def simular_mm1(tasa_arribo, tasa_servicio, tamaño_maximo_cola=np.inf,
               tiempo_simulacion=1000):
    tiempo_ocupado = 0 # Tiempo total en el que el servidor estuvo
                        ocupado
```

```

t = 0 # Tiempo actual de simulaci n
cola = deque() # Cola de espera (estructura eficiente tipo FIFO)
servidor_ocupado = False # Estado del servidor: True si est
    atendiendo
proxima_llegada = np.random.exponential(1 / tasa_arribo) # Tiempo
    hasta la pr xima llegada (distribuci n exponencial)
proxima_salida = np.inf # No hay salida programada al principio
area_en_sistema = 0 # rea bajo la curva de "clientes en el sistema
    ", para el promedio
area_en_cola = 0 # rea bajo la curva de "clientes en cola", para el
    promedio
tiempo_ultimo_evento = 0
rechazados = 0 # Contador de clientes rechazados por cola llena
tiempos_en_sistema = [] # Lista de los tiempos individuales que cada
    cliente pas en el sistema

# Bucle principal de simulaci n, avanza el tiempo hasta que se
    completa la simulaci n
while t < tiempo_simulacion:
    # Avanza el tiempo al siguiente evento: llegada o salida
    t = min(proxima_llegada, proxima_salida)

    # Acumula el rea (para obtener promedios al final)
    area_en_sistema += len(cola) * (t - tiempo_ultimo_evento)
    # Si el servidor est ocupado, hay un cliente menos en cola (el
        que est siendo atendido)
    area_en_cola += (len(cola) - (1 if servidor_ocupado else 0)) * (t
        - tiempo_ultimo_evento)
    if servidor_ocupado:
        tiempo_ocupado += t - tiempo_ultimo_evento

    tiempo_ultimo_evento = t

    # Si el evento es una llegada
    if t == proxima_llegada:
        # Si hay espacio en la cola, se agrega el cliente
        if len(cola) < tama o_maximo_cola:
            cola.append(t) # Guardamos el tiempo de llegada del
                cliente
            # Si el servidor est libre, se atiende inmediatamente
            if not servidor_ocupado:
                servidor_ocupado = True
                proxima_salida = t + np.random.exponential(1 /
                    tasa_servicio)
            else:
                # Si la cola est llena, se rechaza al cliente
                rechazados += 1
            # Se programa la pr xima llegada
            proxima_llegada = t + np.random.exponential(1 / tasa_arribo)

        else: # Si el evento es una salida del sistema
            tiempo_llegada = cola.popleft() # Se saca al cliente que
                estaba primero en la cola

```

```

        demora = t - tiempo_llegada # Tiempo total que estuvo en el
            sistema
        tiempos_en_sistema.append(demora)

        # Si hay m s clientes esperando, se programa la siguiente
        salida
        if len(cola) > 0:
            proxima_salida = t + np.random.exponential(1 /
                tasa_servicio)
        else:
            # Si no hay nadie esperando, el servidor queda libre
            servidor_ocupado = False
            proxima_salida = np.inf

    # C lculo de la utilizaci n del servidor (tiempo que estuvo ocupado
    / tiempo total)
    utilizacion = tiempo_ocupado / tiempo_simulacion

    # Devolvemos las m tricas principales como un diccionario
    return {
        "promedio_en_sistema": area_en_sistema / tiempo_simulacion,
        "promedio_en_cola": area_en_cola / tiempo_simulacion,
        "tiempo_promedio_sistema": np.mean(tiempos_en_sistema) if
            tiempos_en_sistema else 0,
        "tiempo_promedio_cola": (area_en_cola / tiempo_simulacion) /
            tasa_arribo if tasa_arribo > 0 else 0,
        "utilizacion": utilizacion,
        "probabilidad_rechazo": rechazados / (rechazados + len(
            tiempos_en_sistema)) if (rechazados + len(tiempos_en_sistema))
            > 0 else 0
    }

}

# Funci n para correr varios experimentos variando la tasa de arribo
def correr_experimentos(mu=1.0, tiempo_simulacion=1000, K=np.inf,
    cantidad_corridas=50):
    factores_de_carga = [0.25, 0.5, 0.75, 1.0, 1.25] # /
    resultados = {}

    for factor in factores_de_carga:
        lam = factor * mu # Calculamos a partir del factor
        # Ejecutamos varias corridas y guardamos los resultados
        corridas = [simular_mm1(lam, mu, tama o_maximo_cola=K,
            tiempo_simulacion=tiempo_simulacion) for _ in range(
                cantidad_corridas)]
        # Promediamos los resultados obtenidos
        promedio = {clave: np.mean([corrida[clave] for corrida in corridas
            ]) for clave in corridas[0]}
        resultados[factor] = promedio # Guardamos el resultado bajo el
            factor correspondiente

    return resultados

# Funci n para graficar los resultados

```



```

def graficar_resultados(resultados,mu,k):
    factores = sorted([float(f) for f in resultados.keys()]) # Asegurarse
    de que est n ordenados como floats
    metricas = list(resultados[factores[0]].keys()) # Tomamos las
    m tricas disponibles

    if not os.path.exists('graficas'):
        os.makedirs('graficas') # Crear carpeta si no existe

    for metrica in metricas:
        if metrica == "utilizacion":
            # Para la utilizaci n, graficamos como porcentaje
            valores_y = [resultados[f][metrica] * 100 for f in factores]

        else:
            valores_y = [resultados[f][metrica] for f in factores]

        plt.figure()
        plt.plot(factores, valores_y, marker='o', linestyle='--', color='
royalblue')
        plt.xlabel('Carga del Sistema ( _/ _ )', fontsize=12)
        plt.ylabel(metrica.replace("_", " ").title(), fontsize=12)
        if k != -1:
            plt.title(f'{metrica.replace("_", " ").title()}_( ={mu},_K={k
})', fontsize=14)
        else:
            plt.title(f'{metrica.replace("_", " ").title()}_( ={mu},_K=
)', fontsize=14)
        plt.xticks(factores) # Marcar los puntos del eje X con los
factores directamente
        plt.grid(True)
        plt.tight_layout()
        if not os.path.exists('graficas/mm1k'):
            os.makedirs('graficas/mm1k')
        # Guardar antes de mostrar para que no se limpie la figura
        plt.savefig(f'graficas/mm1k/{metrica}.png')

# Funci n principal del script
def main():
    # Parser de argumentos para poder correr el script desde consola con
    distintos par metros
    parser = argparse.ArgumentParser(description="Simulaci n M/M/1 con _
cola finita o _
infinita")
    parser.add_argument('--mu', type=float, default=1.0, help="Tasa de _
servicio _
")
    parser.add_argument('--t', type=int, default=1000, help="Tiempo total _
de simulaci n")
    parser.add_argument('--k', type=int, default=-1, help="Tama o m ximo
de la cola (-1 para infinito)")
    parser.add_argument('--c', type=int, default=10, help="Cantidad de _
corridas por experimento")

```

```

args = parser.parse_args()
# Si k es -1, se interpreta como cola infinita
K = np.inf if args.k == -1 else args.k

# Ejecutamos los experimentos y graficamos los resultados
resultados = correr_experimentos(mu=args.mu, tiempo_simulacion=args.t,
    K=K, cantidad_corridas=args.c)
graficar_resultados(resultados, args.mu, args.k)

# Punto de entrada del programa
if __name__ == "__main__":
    main()
#python3 sim-mm1k.py --mu 10 --t 1000 --k 2 --c 10

```

Inventario

```

import numpy as np
import matplotlib.pyplot as plt
import argparse
import os
from typing import List, Dict, Tuple

def simular_inventario(
    s: int, S: int, demanda_media: float, tiempo_simulacion: int,
    costo_orden: float, costo_mantener: float, costo_faltante: float,
    lead_time: int
) -> Dict[str, float | List[int]]:
    """
    Simula una corrida del modelo de inventario (s, S) con demanda Poisson
    """
    inventario = S
    orden_pendiente, tiempo_orden = None, None

    costos = {"orden": 0, "mantener": 0, "faltante": 0}
    historial = []

    for t in range(tiempo_simulacion):
        if orden_pendiente is not None and t == tiempo_orden:
            inventario += orden_pendiente
            orden_pendiente, tiempo_orden = None, None

        demanda = np.random.poisson(demanda_media)
        inventario -= demanda

        if inventario < 0:
            costos["faltante"] += abs(inventario) * costo_faltante
            inventario = 0

        costos["mantener"] += inventario * costo_mantener

```

```

        if inventario <= s and orden_pendiente is None:
            cantidad = S - inventario
            orden_pendiente = cantidad
            tiempo_orden = t + lead_time
            costos["orden"] += costo_orden

        historial.append(inventario)

    costos["total"] = sum(costos.values())
    costos["historial"] = historial
    return costos

def correr_experimentos(
    s: int, S: int, demanda_media: float, tiempo_simulacion: int,
    costo_orden: float, costo_mantener: float, costo_faltante: float,
    lead_time: int, corridas: int
) -> Tuple[Dict[str, float], List[Dict]]:

    resultados = [
        simular_inventario(s, S, demanda_media, tiempo_simulacion,
                           costo_orden, costo_mantener, costo_faltante,
                           lead_time)
        for _ in range(corridas)
    ]

    promedio = {
        k: np.mean([r[k] for r in resultados]) for k in ["orden", "mantener", "faltante", "total"]
    }

    return promedio, resultados

def graficar_inventario(historial: List[int]):
    plt.figure(figsize=(10, 5))
    plt.plot(historial, color="teal", linewidth=2)
    plt.title("Evoluci n del Inventario (1 corrida)")
    plt.xlabel("Per odo")
    plt.ylabel("Unidades")
    plt.grid(True)
    plt.tight_layout()
    if not os.path.exists("graficas/inventario"):
        os.makedirs("graficas/inventario")
    plt.savefig("graficas/inventario/evolucion_inventario.png")
    plt.close()

def graficar_costos_promedio(promedios: Dict[str, float]):

```

```

categorias = ["orden", "mantener", "faltante"]
valores = [promedios[c] for c in categorias]

plt.figure(figsize=(8, 5))
plt.bar(categorias, valores, color=["orange", "steelblue", "crimson"])
plt.title("Costo_Promedio_por_Categoría")
plt.ylabel("Costo")
plt.tight_layout()
if not os.path.exists("graficas/costos"):
    os.makedirs("graficas/costos")
plt.savefig("graficas/costos/costos_promedio.png")
plt.close()

def graficar_costos_por_corrida(resultados: List[Dict[str, float]]):
    corridas = range(len(resultados))
    for metrica, color in zip(["orden", "mantener", "faltante", "total"],
                              ["darkorange", "royalblue", "firebrick", "forestgreen"]):
        plt.figure(figsize=(10, 4))
        plt.plot(corridas, [r[metrica] for r in resultados], marker="o",
                  color=color)
        plt.title(f"{metrica.capitalize()}_por_Corrida")
        plt.xlabel("Corrida")
        plt.ylabel("Costo")
        plt.grid(True)
        plt.tight_layout()
        if not os.path.exists("graficas/costos"):
            os.makedirs("graficas/costos")
        plt.savefig(f"graficas/costos/{metrica}_por_corrida.png")
        plt.close()

def main():
    parser = argparse.ArgumentParser(description="Simulación del modelo de inventario(s, S)")

    parser.add_argument("--s", type=int, default=20, help="Punto de reorden(s)")
    parser.add_argument("--S", type=int, default=80, help="Nivel objetivo de inventario(S)")
    parser.add_argument("--d", type=float, default=5.0, help="Demanda media")
    parser.add_argument("--t", type=int, default=100, help="Períodos de simulación")
    parser.add_argument("--co", type=float, default=50.0, help="Costo por orden")
    parser.add_argument("--cm", type=float, default=1.0, help="Costo de mantenimiento")
    parser.add_argument("--cf", type=float, default=10.0, help="Costo por faltante")
    parser.add_argument("--l", type=int, default=2, help="Lead time(períodos)")

```

```

parser.add_argument("--c", type=int, default=10, help="Cantidad de
corridas")

args = parser.parse_args()

if args.s > args.S:
    raise ValueError("El valor de s no puede ser mayor que S")

promedio, resultados = correr_experimentos(
    args.s, args.S, args.d, args.t, args.co, args.cm, args.cf, args.l,
    args.c
)

print("RESULTADOS PROMEDIO")
for clave, valor in promedio.items():
    print(f"{clave.capitalize():<10}: ${valor:.2f}")

graficar_inventario(resultados[0]["historial"])
graficar_costos_promedio(promedio)
graficar_costos_por_corrida(resultados)

if __name__ == "__main__":
    main()
# python3 sim-modelo-inventario.py --s 20 --S 100 --d 7 --t 365 --co 100
--cm 0.2 --cf 5 --l 3 --c 30

```