Simulación de Montecarlo

Investigación Operativa, Universidad de San Andrés

Si encuentran algún error en el documento o hay alguna duda, mandenmé un mail a rodriguezf@udesa.edu.ar y lo revisamos.

1. ¿Qué es?

La simulación de Montecarlo es una técnica que usa números aleatorios para resolver problemas. Básicamente, en lugar de resolver algo de forma analítica (que a veces es imposible), generamos miles de escenarios aleatorios y vemos qué pasa en promedio. El proceso es simple:

- Identificamos qué variables son aleatorias en nuestro problema
- Generamos miles de valores aleatorios para cada variable
- Ejecutamos nuestro modelo para cada combinación de valores
- Analizamos los resultados estadísticamente

Por ejemplo, si queremos saber cuánto inventario pedir pero la demanda es incierta, simulamos 10000 escenarios con demandas aleatorias y vemos cuál es la cantidad que nos da mejor resultado en promedio. La clave está en que cada escenario es una "realización" de lo que podría pasar. Si simulamos 10000 veces, tenemos 10000 versiones diferentes de la realidad. Al final, promediamos todos los resultados y obtenemos una estimación de lo que esperamos que pase.

¿Por qué funciona? Porque si generamos suficientes escenarios aleatorios, la ley de los grandes números nos dice que el promedio de nuestros resultados se va a acercar al valor esperado real del problema.

2. Distribuciones que usamos

Las más comunes son:

- **Uniforme:** $U(0, 1)$ - Genera valores equiprobables entre 0 y 1.

- **Normal:** $N(\mu, \sigma^2)$ - Para cosas como demanda, tiempos, etc. Es la más común porque muchas variables en la vida real siguen esta distribución (o se aproximan).
- **Exponencial:** $Exp(\lambda)$ - Para tiempos entre llegadas. Si los eventos llegan de forma aleatoria e independiente, los tiempos entre llegadas son exponenciales.
- **Poisson:** $Pois(\lambda)$ - Para contar eventos (clientes que llegan). Si los tiempos entre llegadas son exponenciales, entonces el número de llegadas en un período fijo es Poisson.
- **Binomial:** $Bin(n, p)$ - Para éxitos/fracasos. Cada ensayo es independiente y tiene la misma probabilidad de éxito.
- **Binomial Negativa:** $NB(r, p)$ - Para contar fallas antes de obtener r éxitos. Útil para modelar tiempos hasta el fallo o número de intentos hasta el éxito.

¿Cómo elegimos qué distribución usar? Depende del problema:

- Si no sabemos nada, empezamos con normal
- Si es tiempo entre eventos, exponencial
- Si es conteo de eventos, Poisson
- Si es sí/no, binomial
- Si es número de fallas antes del éxito, binomial negativa

3. ¿Cómo sabemos que está bien?

Hay que verificar que los resultados se estabilicen. Si simulamos muy pocas veces, el resultado puede ser muy variable. Si simulamos muchas veces, el resultado se estabiliza.

La idea es hacer una "media móvil": calculamos el promedio de los primeros 1000 resultados, luego de los primeros 2000, 3000, etc. Si el promedio

se estabiliza (no cambia mucho), entonces sabemos que tenemos suficientes simulaciones.

```python
def verificar_convergencia(resultados, ventana=1000):
    medias_moviles = []
    for i in range(ventana, len(resultados)):
        media_movil = np.mean(resultados[i-ventana:i])
        medias_moviles.append(media_movil)

    plt.plot(range(ventana, len(resultados)), medias_moviles)
    plt.xlabel('Numero de iteraciones')
    plt.ylabel('Media movil')
    plt.title('Convergencia de la Simulacion')
    plt.grid(True, alpha=0.3)
    plt.show()

    return medias_moviles
```

4. Análisis de sensibilidad

Una vez que encontramos la solución óptima de nuestro problema, es importante entender qué tan sensible es esta solución a cambios en los parámetros del modelo. El análisis de sensibilidad consiste en variar sistemáticamente uno o más parámetros del modelo y observar cómo cambia la solución óptima. Esto nos permite:

- Identificar qué parámetros tienen mayor impacto en la solución
- Evaluar la robustez de nuestra decisión
- Preparar planes de contingencia para diferentes escenarios

Metodología

1. **Seleccionar el parámetro a analizar:** Elegimos una variable que puede cambiar (demanda, costos, tiempos, etc.)
2. **Definir el rango de variación:** Establecemos un rango realista de valores para el parámetro

3. **Simular para cada valor:** Para cada valor del parámetro, ejecutamos la simulación completa

4. **Registrar la solución óptima:** Anotamos la decisión óptima y su costo para cada escenario

5. **Analizar los resultados:** Observamos patrones y tendencias en cómo cambia la solución

5. Problemas

5.1. Problema 1: Gestión de inventario con demanda incierta

Enunciado Tenemos que decidir cuántas unidades pedir cada mes. La demanda es incierta: en promedio se venden 100 unidades pero puede variar (desviación estándar de 20). Los costos son:

- Comprar: \$50 por unidad
- Guardar inventario: \$5 por unidad por mes
- Quedarse sin stock: \$100 por unidad no servida

Si pedimos poco, nos quedamos sin stock y perdemos ventas. Si pedimos mucho, nos sobra inventario y pagamos costos de almacenamiento. Queremos encontrar la cantidad que minimiza el costo esperado.

Planteo del problema

- **Variable de decisión:** Q = cantidad a pedir
- **Variable aleatoria:** $D \sim N(100, 20^2)$, truncada en 0 para evitar demandas negativas
- **Parámetros:** $c_c = 50$ (compra), $c_h = 5$ (holding mensual), $c_s = 100$ (stockout)

Función objetivo Para una realización de demanda D y una decisión Q:

$$C(Q, D) = c_c\, Q + c_h\ \text{máx}(Q - D, 0) + c_s\ \text{máx}(D - Q, 0)$$

Objetivo: Encontrar Q^* que minimiza $E[C(Q, D)]$ mediante simulación.

5.1.1. Simulación base

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

def simulacion_inventario(n_simulaciones=10000):
    # Parametros del problema
    media_demanda = 100
    desvio_demanda = 20
    costo_compra = 50
    costo_almacenamiento = 5
    costo_stockout = 100

    # Rango de cantidades a evaluar
    cantidades = range(60, 141, 5)
    costos_promedio = []

    for Q in cantidades:
        costos_totales = []

        for _ in range(n_simulaciones):
            # Generar demanda aleatoria
            demanda = max(0, np.random.normal(media_demanda,
    desvio_demanda))

            # Calcular costos
            costo_total = Q * costo_compra

            if demanda <= Q:
                # Hay inventario sobrante
                inventario_sobrante = Q - demanda
                costo_total += inventario_sobrante *
    costo_almacenamiento
            else:
                # Hay stockout
                stockout = demanda - Q
                costo_total += stockout * costo_stockout

            costos_totales.append(costo_total)
```

```
36
37          costos_promedio.append(np.mean(costos_totales))
38
39      return cantidades, costos_promedio
40
41  # Ejecutar simulacion
42  cantidades, costos = simulacion_inventario()
43
44  # Encontrar cantidad optima
45  indice_optimo = np.argmin(costos)
46  cantidad_optima = list(cantidades)[indice_optimo]
47  costo_optimo = costos[indice_optimo]
48
49  print(f"Cantidad optima: {cantidad_optima} unidades")
50  print(f"Costo promedio optimo: ${costo_optimo:.2f}")
51
52  # Graficar resultados
53  plt.plot(list(cantidades), costos, 'b-', linewidth=2)
54  plt.axvline(cantidad_optima, color='r', linestyle='--',
55              label=f'Optimo: {cantidad_optima} unidades')
56  plt.xlabel('Cantidad a pedir (Q)')
57  plt.ylabel('Costo promedio total')
58  plt.title('Problema de Inventario - Simulacion Montecarlo')
59  plt.legend()
60  plt.grid(True, alpha=0.3)
61  plt.show()
```

5.1.2. Sensibilidad vs. demanda media

```
1  import numpy as np
2
3  sim = 10000
4  desvio_demanda = 20
5  costo_compra = 50
6  costo_almacenamiento = 5
7  costo_stockout = 100
8
9  # Rango de demandas medias a evaluar
10 demandas_medias = np.arange(80, 121, 5)
11 Q_vals = np.arange(60, 141, 5)
12 Q_opts = []
13 C_mins = []
14
15 for media_demanda in demandas_medias:
```

```
    costos_promedio = []

    for Q in Q_vals:
        costos_totales = []

        for _ in range(sim):
            # Generar demanda aleatoria
            demanda = max(0, np.random.normal(media_demanda,
    desvio_demanda))

            # Calcular costos
            costo_total = Q * costo_compra

            if demanda <= Q:
                inventario_sobrante = Q - demanda
                costo_total += inventario_sobrante *
    costo_almacenamiento
            else:
                stockout = demanda - Q
                costo_total += stockout * costo_stockout

            costos_totales.append(costo_total)

        costos_promedio.append(np.mean(costos_totales))

    # Encontrar Q optimo para esta demanda media
    idx_optimo = np.argmin(costos_promedio)
    Q_opt = Q_vals[idx_optimo]
    C_min = costos_promedio[idx_optimo]

    Q_opts.append(Q_opt)
    C_mins.append(C_min)

# Graficar sensibilidad
import matplotlib.pyplot as plt
plt.plot(demandas_medias, Q_opts, 'bo-', linewidth=2,
    markersize=6)
plt.xlabel('Demanda media')
plt.ylabel('Cantidad optima (Q*)')
plt.title('Sensibilidad: Cantidad optima vs Demanda media')
plt.grid(True, alpha=0.3)
plt.show()
```

5.2. Problema 2: Optimización de reparto

Enunciado Una empresa de logística busca optimizar la cantidad de repartidores para su servicio de entrega diario. La empresa estima una demanda promedio de 80 pedidos por día. Cada repartidor puede completar 12 entregas diarias, con un costo de contratación de \$120 por día. Adicionalmente, cada pedido que no se logre satisfacer representa un costo de penalización de \$25.

Planteo del problema

- **Variable de decisión:** M = número de repartidores a contratar
- **Variable aleatoria:** $D \sim \text{Poisson}(\lambda = 80)$ = demanda diaria
- **Parámetros:** $c = 12$ (capacidad por repartidor), $c_m = 120$ (costo por repartidor), $c_p = 25$ (costo de penalización)

Función objetivo Para un día dado, con M repartidores y una demanda D, el costo total es:

$$C(M, D) = \underbrace{c_m \cdot M}_{\text{Costo Fijo}} + \underbrace{c_p \cdot \text{máx}(D - c \cdot M, 0)}_{\text{Costo de Desabastecimiento}}$$

Objetivo: Encontrar M^* que minimice $E[C(M, D)]$ mediante simulación.

5.2.1. Simulación base

```python
import numpy as np

# parametros del problema
lam = 80
cap_por_repartidor = 12
costo_repartidor = 120
costo_penal = 25
sim = 10000
np.random.seed(42)

# rango de M
M_vals = np.arange(0, 16)
costos_prom = []

```

```
for M in M_vals:
    # 1) simular demanda
    demanda = np.random.poisson(lam=lam, size=sim)
    # 2) calcular demanda insatisfecha
    capacidad_total = cap_por_repartidor * M
    pedidos_no_servidos = np.maximum(demanda -
   capacidad_total, 0)
    # 3) costo total
    costo_operativo = costo_repartidor * M
    costo_penalizacion = costo_penal * pedidos_no_servidos
    costo_total = costo_operativo + costo_penalizacion
    costos_prom.append(costo_total.mean())

# elegir M optimo
idx_optimo = np.argmin(costos_prom)
M_opt, C_min = M_vals[idx_optimo], costos_prom[idx_optimo]
print(M_opt, C_min)
```

5.2.2. Sensibilidad vs. λ

```
import numpy as np

sim = 10000
cap_por_repartidor = 12
costo_repartidor = 120
costo_penal = 25

lambdas = np.arange(60, 121, 5)
M_vals = np.arange(0, 20)
M_opts = []
C_mins = []

for lam in lambdas:
    costos_prom = []
    for M in M_vals:
        demanda = np.random.poisson(lam=lam, size=sim)
        capacidad_total = cap_por_repartidor * M
        pedidos_no_servidos = np.maximum(demanda -
   capacidad_total, 0)
        costo_operativo = costo_repartidor * M
        costo_penalizacion_total = costo_penal *
   pedidos_no_servidos
        costo_total_promedio = (costo_operativo +
   costo_penalizacion_total).mean()
```

```
        costos_prom.append(costo_total_promedio)
    idx_optimo = np.argmin(costos_prom)
    M_opt, C_min = M_vals[idx_optimo], costos_prom[idx_optimo
   ]
    M_opts.append(M_opt)
    C_mins.append(C_min)
```

5.3. Problema 3: Optimización de mantenimiento

Enunciado Una fábrica busca optimizar el programa de mantenimiento para una de sus máquinas críticas. Esta máquina sufre una falla catastrófica después de acumular exactamente 5 fallos menores, cuya probabilidad es del 20 %. Para evitar las paradas costosas asociadas a esta falla, se realizan inspecciones preventivas a intervalos fijos de T días. Los costos asociados son:

- Costo Falla Catastrófica (c_f): \$10,000 (si falla entre inspecciones)
- Costo Inspección (c_i): \$400 (por cada visita)
- Costo Mantenimiento Preventivo (c_p): \$2,000 (adicional si la inspección coincide con el día de la falla)

Planteo del problema

- **Variable de decisión:** T = intervalo entre inspecciones (días)
- **Variable aleatoria:** D = duración del ciclo hasta la falla catastrófica
- **Modelo:** $D = D_{fracasos} + r$ donde $D_{fracasos} \sim \text{NegBin}(r = 5, p = 0{,}20)$
- **Parámetros:** $c_f = 10000$, $c_i = 400$, $c_p = 2000$

Función objetivo Para un ciclo de duración D y un intervalo T:

$$C(T, D) = \underbrace{C_{\text{mantenimiento}}(T, D)}_{\text{Preventivo o Catastrófico}} + \underbrace{C_{\text{inspección}}(T, D)}_{\text{Visitas}}$$

Donde:

- $C_{\text{mantenimiento}}(T, D) = \begin{cases} c_p & \text{si } D \pmod T = 0 \\ c_f & \text{si } D \pmod T \neq 0 \end{cases}$

- $C_{\text{inspección}}(T, D) = c_i \cdot \lceil D/T \rceil$

Objetivo: Minimizar el Costo Diario Esperado: $E[C_{\text{diario}}(T)] = \frac{E[C(T,D)]}{E[D]}$

Nota: $E[D] = r/p = 5/0{,}20 = 25$ días.

5.3.1. Simulación base

```
import numpy as np

r, p = 5, 0.20
costo_falla_catastrofica = 10000
costo_preventivo = 2000
costo_inspeccion = 400
sim = 20000
np.random.seed(42)

T_vals = np.arange(1, 15)
costos_por_dia = []

for T in T_vals:
    # 1) simular D
    D_fracasos = np.random.negative_binomial(r, p, size=sim)
    D = D_fracasos + r
    # 2) costos de mantenimiento
    mantenimiento_catastrofico = (D % T != 0) *
   costo_falla_catastrofica
    mantenimiento_preventivo = (D % T == 0) *
   costo_preventivo
    costo_mantenimiento = mantenimiento_catastrofico +
   mantenimiento_preventivo
    # 3) costos de inspeccion
    num_inspecciones = np.ceil(D / T)
    costo_inspeccion_ciclo = num_inspecciones *
   costo_inspeccion
    # 4) costo diario esperado
    costo_total_ciclo = costo_mantenimiento +
   costo_inspeccion_ciclo
    costo_diario_promedio = costo_total_ciclo.mean() / D.mean
   ()
    costos_por_dia.append(costo_diario_promedio)

idx_optimo = np.argmin(costos_por_dia)
T_opt, C_min = T_vals[idx_optimo], costos_por_dia[idx_optimo]
print(T_opt, C_min)
```

5.3.2. Sensibilidad vs. c_f

```
import numpy as np

sim = 10000
r, p = 5, 0.20
costo_preventivo = 2000
costo_inspeccion = 400

cf_vals = np.arange(5000, 20001, 2500)
T_vals = np.arange(1, 15)
T_opts = []
C_mins = []

# media teorica de D
D_mean_fracasos = r * (1 - p) / p
D_mean = D_mean_fracasos + r

for costo_falla_catastrofica in cf_vals:
    costos_diarios_prom = []
    # simular D una sola vez (CRN)
    D_fracasos_sim = np.random.negative_binomial(r, p, size=
   sim)
    D_sim = D_fracasos_sim + r
    for T in T_vals:
        mantenimiento_catastrofico = (D_sim % T != 0) *
   costo_falla_catastrofica
        mantenimiento_preventivo = (D_sim % T == 0) *
   costo_preventivo
        costo_mantenimiento = mantenimiento_catastrofico +
   mantenimiento_preventivo
        num_inspecciones = np.ceil(D_sim / T)
        costo_inspeccion_ciclo = num_inspecciones *
   costo_inspeccion
        costo_total_ciclo = costo_mantenimiento +
   costo_inspeccion_ciclo
        costo_diario_promedio = costo_total_ciclo.mean() /
   D_mean
        costos_diarios_prom.append(costo_diario_promedio)
    idx_optimo = np.argmin(costos_diarios_prom)
    T_opt, C_min = T_vals[idx_optimo], costos_diarios_prom[
   idx_optimo]
    T_opts.append(T_opt)
    C_mins.append(C_min)
```

5.4. Problema 4: Riesgo en proyecto

Enunciado Una constructora está planificando un proyecto que se desarrolla en tres etapas secuenciales: Diseño, Construcción y Pruebas. Tras consultar con especialistas de cada área, se estima que la duración de cada etapa es incierta pero se dieron estimativos del tiempo mínimo y máximo de cada una. Debido a restricciones de la empresa, el proyecto completo tiene un plazo límite estricto de 25 días. Cada día de retraso más allá de este límite genera un costo de penalización de $500.

Tarea	**Distribución**	**Mínimo (a_i)**	**Máximo (b_i)**
A (Diseño)	Unif(a_A, b_A)	5 días	10 días
B (Construcción)	Unif(a_B, b_B)	8 días	15 días
C (Pruebas)	Unif(a_C, b_C)	4 días	7 días

Planteo del problema

- **Variable aleatoria:** $D_{total} = D_A + D_B + D_C$ donde cada $D_i \sim$ Unif(a_i, b_i)
- **Parámetros:** $L = 25$ (límite), $c_p = 500$ (penalización por día de retraso)
- **Objetivos:**
  - Estimar $P(D_{total} > 25)$ = probabilidad de retraso
  - Estimar $E[c_p \cdot \text{máx}(0, D_{total} - L)]$ = costo esperado de penalización

5.4.1. Simulación base

```
import numpy as np

fecha_limite = 25
costo_penalizacion = 500
sim = 100000
np.random.seed(42)

tareas = {'A': (5, 10), 'B': (8, 15), 'C': (4, 7)}

duracion_A = np.random.uniform(tareas['A'][0], tareas['A'
    ][1], size=sim)
```

```
duracion_B = np.random.uniform(tareas['B'][0], tareas['B'][1], size=sim)
duracion_C = np.random.uniform(tareas['C'][0], tareas['C'][1], size=sim)
D_total = duracion_A + duracion_B + duracion_C

retraso = np.maximum(D_total - fecha_limite, 0)
E_costo = (retraso * costo_penalizacion).mean()
P_retraso = (D_total > fecha_limite).mean()
print(P_retraso, E_costo)
```

5.4.2. Sensibilidad vs. b_B

```
import numpy as np

sim = 100000
fecha_limite = 25
costo_penalizacion = 500

tareas_fijas = {'A': (5, 10), 'C': (4, 7)}
a_B = 8

b_B_vals = np.arange(12, 21, 1)
P_retraso_vals = []
E_costo_vals = []

for b_B in b_B_vals:
    duracion_A = np.random.uniform(tareas_fijas['A'][0], tareas_fijas['A'][1], size=sim)
    duracion_B = np.random.uniform(a_B, b_B, size=sim)
    duracion_C = np.random.uniform(tareas_fijas['C'][0], tareas_fijas['C'][1], size=sim)
    D_total = duracion_A + duracion_B + duracion_C
    retraso = np.maximum(D_total - fecha_limite, 0)
    E_costo = (retraso * costo_penalizacion).mean()
    P_retraso = (D_total > fecha_limite).mean()
    P_retraso_vals.append(P_retraso)
    E_costo_vals.append(E_costo)
```

5.4.3. Comparación teórica vs. simulación

```
import numpy as np

```

```
tareas = {'A': (5, 10), 'B': (8, 15), 'C': (4, 7)}
sim = 1000000

# 1) analitico de media y varianza
E_D_analitica = 0
Var_D_analitica = 0
for (a, b) in tareas.values():
    mu = (a + b) / 2
    var = (b - a) ** 2 / 12
    E_D_analitica += mu
    Var_D_analitica += var
Std_D_analitica = np.sqrt(Var_D_analitica)

# 2) monte carlo
duracion_A = np.random.uniform(tareas['A'][0], tareas['A'
    ][1], size=sim)
duracion_B = np.random.uniform(tareas['B'][0], tareas['B'
    ][1], size=sim)
duracion_C = np.random.uniform(tareas['C'][0], tareas['C'
    ][1], size=sim)
D_total = duracion_A + duracion_B + duracion_C
E_D_mc = D_total.mean()
Std_D_mc = D_total.std()

print(round(E_D_analitica, 4), round(E_D_mc, 4))
print(round(Std_D_analitica, 4), round(Std_D_mc, 4))
```

6. Extra: Código básico

```
# Generar numeros aleatorios
import numpy as np

# Uniforme - valores entre 0 y 1
uniforme = np.random.uniform(0, 1, 1000)

# Normal - media 100, desviacion 15
normal = np.random.normal(100, 15, 1000)

# Exponencial - tasa 2
exponencial = np.random.exponential(2, 1000)

# Poisson - tasa 5
poisson = np.random.poisson(5, 1000)
```

```python

# Binomial - 10 ensayos, probabilidad 0.3
binomial = np.random.binomial(10, 0.3, 1000)

# Binomial Negativa - 5 exitos requeridos, probabilidad 0.7
binomial_negativa = np.random.negative_binomial(5, 0.7, 1000)

# Ejemplo simple: calcular el area de un circulo
def simulacion_montecarlo(n_iteraciones=10000):
    resultados = []

    for _ in range(n_iteraciones):
        # Generar punto aleatorio en cuadrado [-1,1] x [-1,1]
        x = np.random.uniform(-1, 1)
        y = np.random.uniform(-1, 1)

        # Verificar si esta dentro del circulo unitario
        if x**2 + y**2 <= 1:
            resultados.append(1)  # Dentro del circulo
        else:
            resultados.append(0)  # Fuera del circulo

    return np.array(resultados)

# Ejecutar simulacion
resultados = simulacion_montecarlo()
# El area del circulo es aproximadamente 4 * (promedio de
   resultados)
area_estimada = 4 * np.mean(resultados)
print(f"Area estimada del circulo: {area_estimada:.4f}")
print(f"Area real: {np.pi:.4f}")
```