Metodo de Montecarlo

Washington Yandun
Alexander Quintana
David Trujillo
Ivan Tulcan
Jonathan Miño

Ingenieria en Software - Modelos y Simulacion
Facultad de ingenieria y ciencias aplicadas
UNIVERSIDAD DE LAS AMERICAS

e-mail: washington.yandun@udla.edu.ec
e-mail: alexander.quintana@udla.edu.ec
e-mail: david.trujillo.robalino@udla.edu.ec
e-mail: ivan.tulcan@udla.edu.ec
e-mail: jonathan.mino@udla.edu.ec

This report presents an experimental study on the Monte Carlo method for numerical approximation, focusing on the "hit and miss" and "sampling" approaches. The objective of the experiment was to estimate the value of an integral using these methods and compare their performance and accurancy.

Keywords: **Montecarlo, integracion, python, modelos, simulacion, sampling, hit and miss**

I. INTRODUCCIÓN

El método de Monte Carlo es una poderosa herramienta computacional utilizada en una amplia gama de disciplinas científicas para resolver problemas que implican cálculos numéricos y probabilísticos. Este método se basa en la generación de números aleatorios para aproximar soluciones a problemas complejos que pueden ser difíciles de abordar mediante métodos analíticos tradicionales.

I.I CONTEXTO

En el campo de la simulación y la computación científica, el método de Monte Carlo ha demostrado ser especialmente útil en la resolución de problemas que involucran cálculos de integrales y estimaciones de probabilidades. Al simular múltiples experimentos aleatorios y analizar sus resultados, se pueden obtener aproximaciones numéricas precisas y eficientes.

I.II OBJETIVOS

El objetivo de este informe es presentar un experimento computacional en Python que emplea los métodos de Monte Carlo: Hit and Miss y Sampling con barras. Estos métodos son ampliamente utilizados para estimar integrales y resolver problemas de probabilidad.

El experimento se centrará en la estimación de una integral específica de un cuadrado con cuatro cuartos de circulos dentro del mismo, esto servirá como caso de estudio para comparar y evaluar la precisión y eficiencia de ambos métodos. A través de este estudio, se busca responder a las siguientes preguntas de investigación:

1. ¿Cómo se implementan los métodos de Monte Carlo (Hit and Miss y Sampling con barras) en Python?

2. ¿Cuál de los dos métodos ofrece una mejor aproximación numérica de la integral/area objetivo?

3. ¿Cuál de los dos métodos es más eficiente en términos de tiempo de ejecución y uso de recursos computacionales?

4. ¿Qué consideraciones prácticas deben tenerse en cuenta al utilizar los métodos de Monte Carlo en aplicaciones reales?

Se espera que este informe proporcione una visión clara y detallada de los métodos de Monte Carlo y su aplicación en un experimento computacional concreto. Los resultados y conclusiones obtenidos servirán como base para futuros estudios y aplicaciones en el campo de la simulación numérica y la resolución de problemas complejos en ciencia y ingeniería.

II. MARCO TEÓRICO

En esta sección se describe en detalle las concetos utilizada para llevar a cabo el experimento computacional basado en el metodo de Monte Carlo: Hit and Miss y Sampling con barras:

II.I Circunferencia (Ecuacion centro-radio)

La circunferencia es una figura geométrica en dos dimensiones que se encuentra dentro de un plano cartesiano. Se define como el conjunto de todos los puntos en el plano que se encuentran a una distancia constante, conocida como radio (r), de un punto fijo denominado centro (h, k). En el marco de este

informe de laboratorio, la circunferencia se utiliza para analizar y modelar ciertos fenómenos físicos y matemáticos relacionados.

Para describir matemáticamente la circunferencia, se emplea su ecuación general (centro-radio):

$$(x-h)^2 + (y-k)^2 = r^2$$

Donde:

- (x, y): son las coordenadas de un punto cualquiera en la circunferencia.
- (h, k): son las coordenadas del centro de la circunferencia.
- r: es el radio de la circunferencia.

Es importante destacar que la ecuación de la circunferencia puede expresarse de diferentes formas equivalentes, según las necesidades del análisis realizado. Entre ellas, se encuentran la forma general, la forma centro-radio y la forma punto-pendiente. Estas formas alternativas permiten abordar distintos aspectos de la circunferencia y facilitan su comprensión en diferentes contextos, sin embargo en el presente se udara la ecuacion centro-radio.

La ecuación de la circunferencia resulta fundamental en este informe de laboratorio, ya que nos permite modelar y calcular diversos parámetros relacionados con el perimetro de la figura de la cual se quiere calcular el area.

II.II Método de Montecarlo

El método de Montecarlo es una técnica computacional utilizada en el campo de la informática para estimar resultados numéricos mediante simulaciones aleatorias. Se basa en el principio de generar números aleatorios y utilizarlos para aproximar soluciones a problemas complejos. Este enfoque es ampliamente utilizado en áreas como la computación científica, la inteligencia artificial y la optimización.

En el contexto informático, el método de Montecarlo se aplica para abordar problemas que no pueden resolverse de manera analítica o que son demasiado complejos para ser tratados con métodos tradicionales. A continuación se presenta una descripción general adaptada de los pasos involucrados:

1. Definición del problema: En el contexto informático, se establecen las condiciones y los parámetros del problema que se desea resolver, como por ejemplo, la optimización de algoritmos o la simulación de sistemas complejos.
2. Generación de números aleatorios: Se generan números aleatorios utilizando algoritmos específicos o bibliotecas de generación de números aleatorios. Estos números se utilizan como entrada para la simulación o el cálculo del problema en cuestión.
3. Realización de la simulación o cálculo: Se lleva a cabo la simulación o el cálculo utilizando los números aleatorios generados. Esto implica ejecutar el algoritmo o el modelo correspondiente y recopilar los resultados obtenidos.
4. Análisis de los resultados: Se analizan los resultados obtenidos de la simulación o el cálculo para obtener una estimación numérica del problema original. Esto puede implicar calcular estadísticas relevantes, como promedios, desviaciones estándar o probabilidades.

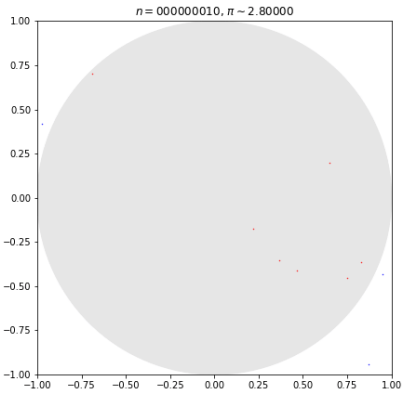
Ventajas del método de Montecarlo

- Flexibilidad: El método de Montecarlo es altamente flexible y se puede aplicar a una amplia gama de problemas informáticos complejos. No requiere una formulación matemática precisa, lo que permite abordar problemas que no tienen una solución analítica directa.
- Tratamiento de la incertidumbre: El método de Montecarlo es especialmente útil cuando se deben tener en cuenta fuentes de incertidumbre o variabilidad en los datos o parámetros del problema. Permite obtener estimaciones probabilísticas y evaluar el impacto de diferentes variables aleatorias en los resultados.
- Optimización y toma de decisiones: El método de Montecarlo puede utilizarse para optimizar algoritmos o sistemas informáticos, evaluando diferentes configuraciones y seleccionando la opción óptima. También puede ayudar en la toma de decisiones informáticas al proporcionar una estimación de riesgos y probabilidades.

Limitaciones del método de Montecarlo

- Recursos computacionales: El método de Montecarlo puede ser computacionalmente intensivo, especialmente si se requieren un gran número de simulaciones o cálculos. Puede requerir una capacidad de procesamiento significativa y tiempo de ejecución prolongado.
- Convergencia y precisión: La precisión del método de Montecarlo depende del número de simulaciones realizadas. En algunos casos, puede requerir una cantidad considerable de simulaciones para obtener resultados precisos, lo que puede ser
- Dependencia de números aleatorios: La calidad de los números aleatorios generados puede afectar la precisión y confiabilidad de los resultados obtenidos mediante el método de Montecarlo

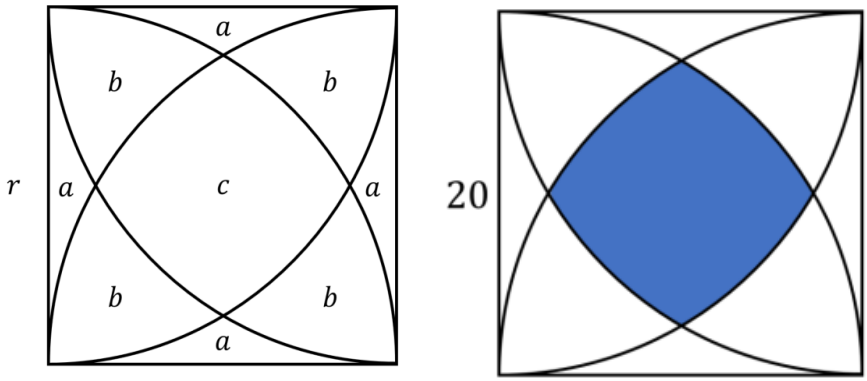
El ejemplo clasico de uso del metodo es en la aproximacion del area de un circulo inscrito en un cuadrado, como se muestra a continuacion:



II.IV Cuartos de circunferencia en un Cuadrado

Esta es una figura que se forma al trazar 4 cuartos de circunferencias dentro de un cuadrado, se toma en cuenta como centro de cada una de las circunferencias cada uno de los vertices del cuadrado.

Sea un cuadrado de longitud "r", entonces:



Como se puede observar se tiene un figura simetrica en el eje "y", "x" e inclusive en la diagonal, en el caso de estudio del presente nos enfocaremos en la aproximacion del area "c" que se puede observar en el centro de la figura de la derecha de color azul, dentro de un cuadrado cuyo lado mide 20 unidades.

II.V Integral de Riemman

La integral de Riemann es un concepto fundamental en el cálculo y se utiliza para calcular el área bajo una curva en un intervalo dado. Fue desarrollada por el matemático alemán Bernhard Riemann en el siglo XIX.

En términos generales, la integral de Riemann divide el área bajo una curva en rectángulos más pequeños y luego calcula la suma de las áreas de esos rectángulos para aproximar el área total.

La integral de Riemann se define mediante una partición del intervalo en subintervalos más pequeños. Estos subintervalos se determinan mediante puntos de división y se representa como:

$$\mathrm{P} = \{x_0 = a, x_1, x_2, \ldots, x_n = b\}$$

Donde:

- P: representa el intervalo [a,b]
- a: es el extremo izquierdo del intervalo.
- b: es el extremo derecho del intervalo.

La partición del intervalo divide el intervalo en subintervalos, asi:

$$\{[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]\}$$

Luego, se elige un punto de muestra en cada subintervalo, denotado como:

$\{c_0, c_1, c_2, \ldots, c_{n-1}\}$

Estos puntos de muestra determinan las alturas de los rectángulos que se utilizan para aproximar el área bajo la curva. La altura de cada rectángulo se determina tomando el valor de la función en el punto de muestra correspondiente.

La suma de las áreas de los rectángulos se calcula multiplicando la altura de cada rectángulo por la longitud del subintervalo correspondiente y luego sumando todas las áreas parciales. Formalmente, la integral de Riemann se define asi:

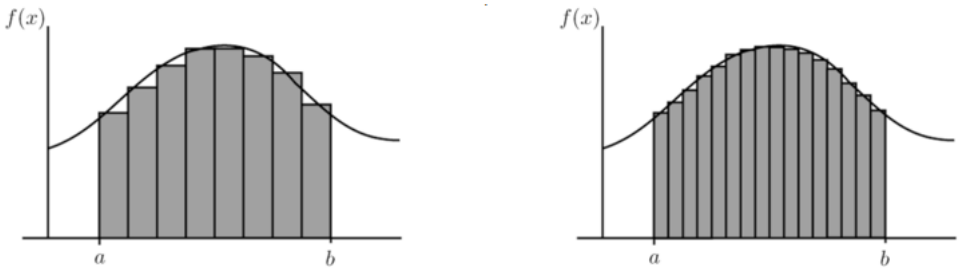
$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} f(c_i)(x_{i+1} - x_i)$$

Donde:

- f(x): es la función que queremos integrar.
- [a, b]: es el intervalo en el que se realiza la integración.
- n: es el número de subintervalos de la partición:
- c_i: es el punto de muestra en el subintervalo [x_i, x_{i+1}]
- (x_{i+1} - x_i): es la longitud del subintervalo.

Al tomar el límite cuando "n" tiende a infinito, se obtiene una aproximación más precisa del área bajo la curva y se define la integral de Riemann. Computacionalmente, esto es imposible, pero sí se puede obtener una muy buena aproximación gracias a este concepto combinado con el método de Montecarlo.

Un ejemplo visual del funcionamiento en conpeto de la integral de Riemman es la siguiente:



Es importante tener en cuenta que la integral de Riemann es solo una de las formas de definir la integral en el cálculo. También existen otras definiciones, de integral pero esta es la que sera usada en el metodo de sampling.

II.VI Métodos derivados de Monte Carlo

El método de Monte Carlo originalmente se estableció para dar solución a problemas relacionados con integrales que no podían ser evaluadas de forma analítica. Pero puede tener otras aplicaciones como en este caso para los programas informáticos. Utilizando Monte Carlo se puede analizar datos y predecir una serie de resultados futuros en función de una elección de datos. Una simulación de Monte Carlo es un modelo probabilístico que puede incluir un elemento de aleatoriedad en su predicción. Cuando se utiliza un modelo probabilístico para simular una salida, se obtienen resultados diferentes cada vez. Los métodos que se estudiarán en este trabajo son:

- Sampling
- Hit and miss

II.VI.I Método Sampling

El método Sampling de Monte Carlo es una técnica de simulación estadística, que es utilizada para aproximar características de una determinada muestra aleatoria de datos de un conjunto. Para implementar este método se debe establecer ciertos parámetros, como:

- N: Es el tamaño de la muestra
- xi: Los puntos (en estevcaso son randomicos)
- [a, b]: Intervalo de evaluación

$$A = \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

Ejemplo de aplicacion: Supongamos que deseas investigar la opinión de los empleados de una empresa sobre el ambiente laboral. La población objetivo es de 500 empleados. Para realizar el estudio, decides utilizar un muestreo estratificado proporcional basado en los diferentes departamentos de la empresa.

1. Definición de la población: Los empleados de la empresa (población objetivo) que consta de 500 personas.
2. Determinación del tamaño de la muestra: Decides que deseas obtener una muestra representativa de tamaño 100 empleados. Además, decides asignar un número proporcional de empleados a cada departamento según su tamaño relativo.
3. Selección de la muestra: Identificas los departamentos de la empresa y determinas la proporción que representan en la población total. Supongamos que la empresa tiene tres departamentos: ventas (40% de la población), recursos humanos (30% de la población) y contabilidad (30% de la población).

Para el departamento de ventas (40% de 500 empleados = 200 empleados), debes seleccionar una muestra de 40 empleados (el 20% de 200 empleados). Para el departamento de recursos humanos (30% de 500 empleados = 150 empleados), debes seleccionar una muestra de 30 empleados (el 20% de 150 empleados). Para el departamento de contabilidad (30% de 500 empleados = 150 empleados), debes seleccionar una muestra de 30 empleados (el 20% de 150 empleados). Puedes seleccionar los empleados de cada departamento utilizando un método aleatorio simple dentro de cada estrato. Por ejemplo, podrías utilizar un generador de números aleatorios para seleccionar aleatoriamente a los empleados de cada departamento.

1. Recopilación de datos: Una vez que hayas seleccionado a los empleados de la muestra, puedes proceder a recopilar los datos. Podrías administrar un cuestionario a cada empleado, solicitando su opinión sobre el ambiente laboral en la empresa.
2. Análisis de datos: Una vez que hayas recopilado los datos, puedes realizar análisis estadísticos para obtener conclusiones sobre la opinión de los empleados en cada departamento y en general. Por ejemplo, podrías calcular la media de las respuestas en cada departamento, comparar los resultados entre los departamentos y realizar pruebas estadísticas para determinar si existen diferencias significativas en la opinión sobre el ambiente laboral entre los departamentos.

II.VI.II Metodo Hit and Miss

El método Hit and Miss, también es una técnica de simulación estadística que es de utilidad para estimar la probabilidad de que ocurran eventos en una determinada región, lo cual generalmente representa lo que está por debajo de una curva. Hit and miss maneja pares de coordenadas en los ejes "x" e "y" dentro de un área de interés. Cuando dicha coordenada se encuentra en el área sugerida entonces se lo contaría como un HIT; por otro lado, si la coordenada se encuentra fuera de la región, esto contaría como un MISS. Dicho de otra manera, se deben generar números aleatorios y observar que puntos caen en el área de exito y que puntos caen fuera, para después calcular la probabilidad.

Probabilidad = Aciertos (Hit) / Total de intentos (Hit and miss)

Ejemplo de aplicacion: Supongamos que se quiere estimar el área de un círculo utilizando el método "hit and miss" de Monte Carlo. Sabemos que el círculo está inscrito en un cuadrado de lado 2 unidades y queremos calcular el área del círculo utilizando una simulación Monte Carlo. Los pasos a seguir serían:

1. Definir el área de la región de interés: En este caso, el área del círculo es π * r^2, donde r es el radio del círculo.
2. Generar puntos aleatorios dentro del cuadrado: Generamos una cantidad suficiente de puntos aleatorios (por ejemplo, 10,000 puntos) dentro del cuadrado de lado 2 unidades.
3. Contar los aciertos, es decir la cantidad de puntos que caen dentro del círculo (aciertos).
4. Calcular la probabilidad y estimar el área del círculo. Además, se debe tomar en cuenta que la probabilidad la multiplicamos por el área del cuadrado (4 unidades cuadradas) para obtener una estimación del área del círculo.

Entonces si generamos 10,000 puntos aleatorios dentro del cuadrado de lado 2 unidades. De esos puntos, 7,850 puntos caen dentro del círculo.

- Probabilidad = 7,850 / 10,000 = 0.785

Estimación del área del círculo = 0.785 * 4 = 3.14 unidades cuadradas

Por lo tanto, la estimación del área del círculo utilizando el método "hit and miss" de Monte Carlo es de aproximadamente 3.14 unidades cuadradas, que se acerca al valor real de π (pi) para un círculo unitario.

III. METODOLOGÍA

Para la implementación del experimento, se siguió la metodología de trabajo de solución de problemas de Montecarlo de cuatro puntos. Es importante tener en cuenta que, con el fin de lograr uniformidad en la experimentación y comparación de métodos, se inicializó una semilla (2401).

III.I Método: Sampling

Para la implementación del método de muestreo en Python, se utilizaron bibliotecas como:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time

# Configuremos una semilla cualquiera
np.random.seed(2401)
```

Cada una de ellas tiene una función específica en el experimento:

- NumPy: Manejo numérico eficiente (computación).
- Matplotlib: Graficación del experimento y sus resultados.
- Pandas: Manejo eficiente de datos.
- Time: Medición del tiempo de ejecución.

Después de importar las librerías necesarias para el método de muestreo, es importante definir el problema y crear un modelo basado en eso.

Como se mencionó anteriormente, el área buscada está formada por cuatro cuartos de círculo dentro de un cuadrado. Basándonos en esto, podemos obtener una función que modele uno de los círculos que lo componen. Además, sabemos que esta figura es simétrica en los ejes "x" e "y". Por lo tanto, el enfoque

seguido fue dividir la figura en cuatro secciones a lo largo de su centro en los ejes "x" e "y", luego obtener el área bajo la curva de una de las cuatro partes y multiplicar ese área por cuatro.

Siguiendo esta línea de pensamiento, podemos obtener la siguiente función:

```
def f(x):
    """Define la función a integrar."""
    return (400 - (x-20) ** 2) ** (1/2)
```

Como se puede apreciar, se utilizó una ecuación que describe el centro y el radio de la circunferencia. Tomando como radio el lado del cuadrado, que en este caso es igual a 20, y se eligió como centro las coordenadas (x, y) en (0, 20).

Por lo tanto se tiene una circunferencia del tipo:

$$(x - 20)^2 + y^2 = 400$$

De la cual se despejo "x" dejando como variable dependiente a "y":

$$y = \sqrt{400 - (x - 20)^2}$$

Donde:

- (x, y): son las coordenadas de un punto cualquiera en la circunferencia.
- (0, 20): son las coordenadas del centro de la circunferencia.
- r = 20: radio de la circuenferencia y lado del cuadrado.

Como se puede observar, sólo se tomó en cuenta la sección donde f(x) es positivo.

Luego de ello, también crearemos una función que nos devuelva el area analitica de la figura

```
def area_analitica_cuadrantes(n):
    '''
    Devuelve el area real de la figura buscada de forma analitica
    de la figura en base a el lado del cuadrado que la contiene
    '''
    return (n**2) * (1 - (3 ** (1 / 2)) + np.pi / 3)
```

La obtención de dicha ecuación es trivial y se logra aplicando conceptos de geometría euclidiana en la figura y su contenedora.

También se uso una distribución aleatoria uniforme de números aleatorios en que es obtenida de la siguiente función:

```
def generate_random_points(a, b, n):
    """Genera n puntos aleatorios en el intervalo [a, b]."""
    return np.random.uniform(a, b, size=n)
```

Para mostrar el funcionamiento del metodo se realizaron algunas funciones con el fin de graficar el mismo, de lo que:

```
def plot_function(f, a, b):
    """Grafica la función f(x) en el intervalo [a, b]."""
    x = np.linspace(a, b)
    plt.title('Metodo sampling', fontsize=14)
    plt.xlabel('x', fontsize=12)
    plt.ylabel('f(x)', fontsize=12)
    plt.plot(x, f(x), color = "r")
    plt.ylim(0, 20)
    plt.xlim(0, 20)

def plot_bars(x, y):
    """Grafica los puntos (x, y) en el plano cartesiano."""
    plt.vlines(x, 10, y, colors='g')
```

Después de ello con las funciones de apoyo listas se procedió a crear una función que calcule el área bajo la curva de la función objetivo usando el método sampling y tomando como concepto la integral de Riemman:

```
def approximate_integral(f, a, b, n, flag = False):
    """Aproxima la integral de f(x) en el intervalo [a, b] con n puntos aleatorios."""
    w = (b - a) / n
    random_points = generate_random_points(a, b, n)
```

```
    integral = 4*(w*f(random_points).sum() - (b - a)*f(a))

    # Graficar la función y los puntos aleatorios cuando es solicitado
    if flag:
        random_points = generate_random_points(a, b, n)
        plot_bars(random_points, f(random_points))
        plot_function(f, a, b)

    return integral
```

Como se puede observar en el código y se explicó en secciones anteriores, se tomó solo un cuarto de la figura. Se crearon "barras" con altura f(x), teniendo en cuenta los límites de integración (2.68, 10), eliminando así la sección rectangular sobrante, es decir, el área que no forma parte de la figura. Después de esto, recordando que la figura es simétrica, se multiplicó el área obtenida por cuatro para obtener el área de la figura objetivo.

Siguiendo con el método de muestreo, debemos determinar el rango de integración de la función. De manera analítica, se obtuvo que el vértice izquierdo de la figura se encuentra en x = 2.68, y su vértice superior se encuentra en x = 10. La obtención de estos valores es trivial y se obtienen fácilmente encontrando las intersecciones con otras dos funciones que conforman la figura. Estas funciones son:

$$
\begin{aligned}
&\mathrm{x}^2 + y^2 = 400 \\
&(x - 20)^2 + y^2 = 400 \\
&x^2 + (y - 20)^2 = 400 \\
&(x - 20)^2 + (y - 20)^2 = 400
\end{aligned}
$$

Además, en la misma sección se crearon algunas variables:

- n_bars: esta lista contiene cada uno de los números de puntos a simular.
- n_areas: esta lista contiene el área correspondiente a cada elemento de n_bars.
- n_simulations: esta lista representa el número de simulaciones para cada número de puntos (cada elemento en n_bars). Esto se hace para observar las variaciones en los resultados y medirlas.
- errors: esta lista representa el error relativo de cada n_bar en sus respectivas simulaciones.
- results_df: en este DataFrame se guarda toda la información utilizando la biblioteca Pandas.

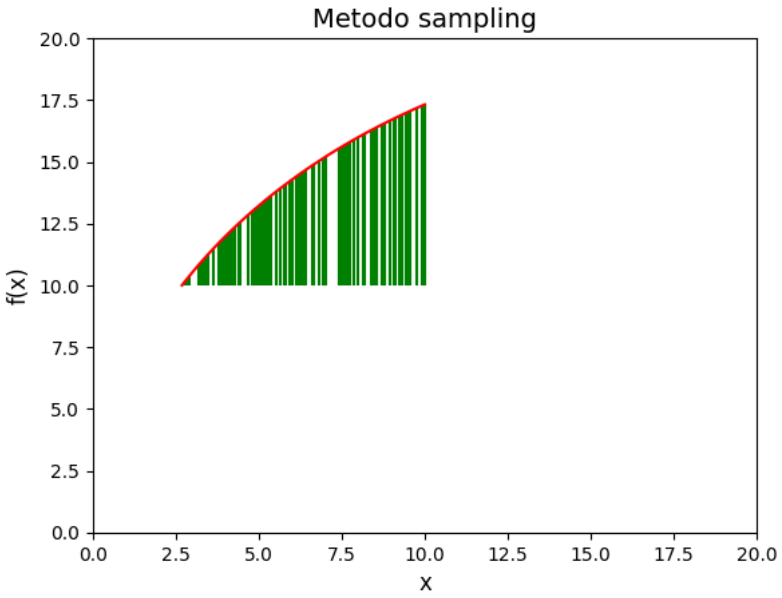
```
# Definir los límites de integración
a, b = 2.680, 10

# Obtención de datos
n_bars = [10**i for i in range(0, 6, 1)]
n_simulations = 1000
n_areas = np.zeros((len(n_bars), n_simulations))
errors = np.zeros((len(n_bars), n_simulations))
results_df = pd.DataFrame(columns=['Número de barras', 'Simulación', 'Aproximación', 'Error relativo'])
```

Una vez definidos los valores y limites necesarios ademas de haber creado las funciones necesarias para el metodo se realizó una prueba con un valor arbitrario para ver el funcionamiento del código:

```
# Grafica de un ejemplo con valor arbitrario
integral = approximate_integral(f, a, b, 90, flag=True)
print(integral)
```

```
133.43106551146087
```



Una vez comprobado el funcinamiento del ejemplo arbitrario, pasamos a realizar varias simulaciones con un número de puntos creciente en base 10 que recordemos, está representado en n_bars, en el mismo proceso se obtendrá errores relativos y desviaciones estandar, las cuales guardaremos en results_df y obtendremos el tiempo de ejecucion del algorítmo. Es importante analizar el tiempo de ejecución el algorítmo, el tiempo de ejecución puede depender de diversas variables pero, debido a la muestra seleccionada se obviarán dichas variables ya que en el caso del presente son inecesarias:

```
area_real = area_analitica_cuadrantes(20)

# Iniciar tiempo de ejecución
start_time = time.time()

# Simulaciones
for i, j in enumerate(n_bars):
    for k in range(n_simulations):
        area = approximate_integral(f, a, b, j, flag=False)
        error = abs(area_real - area) / area_real

        # Agregar los resultados al DataFrame
        results_df = results_df.append({
            'Número de barras': j,
            'Simulación': k+1,
            'Aproximación': area,
            'Error relativo': error
        }, ignore_index=True)

# Imprimir el DataFrame completo
print(results_df)

# Calcular el tiempo de ejecución total
elapsed_time = time.time() - start_time
print(elapsed_time)
```

```
      Número de barras  Simulación  Aproximación  Error relativo
0                  1.0         1.0     72.668533        0.423534
1                  1.0         2.0     67.846488        0.461787
2                  1.0         3.0    149.220825        0.183741
3                  1.0         4.0    134.591442        0.067689
4                  1.0         5.0    135.033272        0.071194
...                ...         ...           ...             ...
5995          100000.0       996.0    126.278361        0.001743
5996          100000.0       997.0    125.923373        0.001074
5997          100000.0       998.0    125.855410        0.001613
5998          100000.0       999.0    125.996836        0.000491
5999          100000.0      1000.0    126.215059        0.001240

[6000 rows x 4 columns]
7.99999737739563
```

Una vez obtenida la información, se calcularon las desviaciones estándar y los promedios para poder mostrar los resultados obtenidos.

```
# Calcular promedios y desviaciones estándar de las simulaciones
avg_areas = results_df.groupby('Número de barras')['Aproximación'].mean()
```

```
std_areas = results_df.groupby('Número de barras')['Aproximación'].std()
avg_errors = results_df.groupby('Número de barras')['Error relativo'].mean()
std_errors = results_df.groupby('Número de barras')['Error relativo'].std()

print(avg_areas)
print(std_areas)
print(avg_errors)
print(std_errors)
```

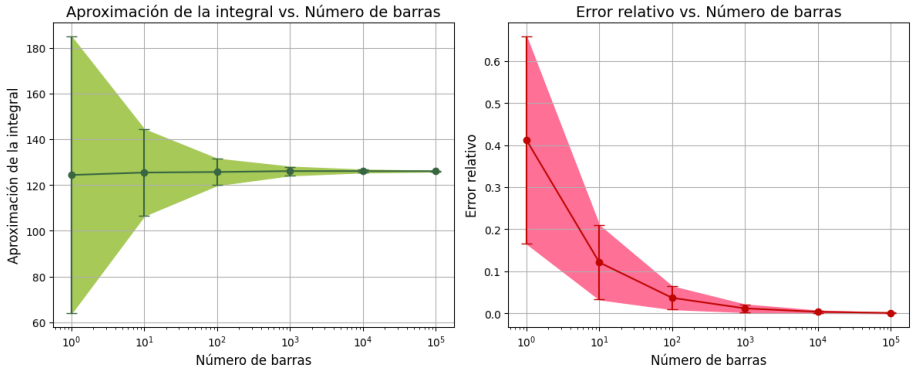
```
10.0        125.429552
100.0       125.676047
1000.0      126.077984
10000.0     126.065365
100000.0    126.030105
Name: Aproximación, dtype: float64
Número de barras
1.0         60.450714
10.0        18.874881
100.0        5.775956
1000.0       1.897540
10000.0      0.609643
100000.0     0.184857
Name: Aproximación, dtype: float64
Número de barras
1.0         0.411820
10.0        0.121219
100.0       0.037085
1000.0      0.012069
10000.0     0.003856
100000.0    0.001190
Name: Error relativo, dtype: float64
Número de barras
1.0         0.245715
10.0        0.087952
100.0       0.027056
1000.0      0.008990
10000.0     0.002916
100000.0    0.000885
Name: Error relativo, dtype: float64
```

De manera prudente se decidió graficar no solo el error, sino también la aproximación de los puntos al resultado, ademas de sus desviaciones estandar para cada número de puntos:

```
# Graficar los resultados
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.errorbar(n_bars, avg_areas, yerr=std_areas, c = "#386641", ecolor="#386641",fmt='o-', capsize=5)
plt.fill_between(n_bars, avg_areas - std_areas, avg_areas + std_areas, color='#a7c957')
plt.xscale('log')
plt.xlabel('Número de barras', fontsize=12)
plt.ylabel('Aproximación de la integral', fontsize=12)
plt.title('Aproximación de la integral vs. Número de barras', fontsize=14)
plt.grid(True)

plt.subplot(1, 2, 2)
plt.errorbar(n_bars, avg_errors, yerr = std_errors, c ="#bf0603",fmt='o-', capsize=5)
plt.fill_between(n_bars, avg_errors - std_errors, avg_errors + std_errors, color='#ff7096')
plt.xscale('log')
plt.xlabel('Número de barras', fontsize=12)
plt.ylabel('Error relativo', fontsize=12)
plt.title('Error relativo vs. Número de barras', fontsize=14)
plt.grid(True)

plt.tight_layout()
plt.show()
```



III.I Metodo: Hit and miss

Para la implementación del metodo "hit and miss" en python se usaron las mismas librerias del metodo "sampling" las cuales son:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time

# Configuremos la misma semilla del metodo sampling
np.random.seed(2401)
```

Recordemos que habíamos planteado a nuestra figura como el resultado de la interseccion de 4 circunferencias, la obtención cada función que compone la figura fu en base a la primera función (la que se uso en el método sampling) a partir de ahí, por lo que la obtención de las funciones fue de manera analítica, la obtención de las mismas no es de interés para el objetivo del presente, sin embargo se detallan las cuatro ecuaciones a continuacion:

$$x^2 + y^2 = 400$$
$$(x-20)^2 + y^2 = 400$$
$$x^2 + (y-20)^2 = 400$$
$$(x-20)^2 + (y-20)^2 = 400$$

De lo cual se implemento una funcion que modele la interseccion de las figuras:

```
def f(x, y):
    """
    Retorna True si el punto (x, y) está dentro de los arcos de cuadrantes y False en otro caso
    """
    return (x ** 2 + y ** 2 <= 400) and ((x - 20) ** 2 + y ** 2 <= 400) and (
            x ** 2 + (y - 20) ** 2 <= 400) and ((x - 20) ** 2 + (y - 20) ** 2 <= 400)
```

Luego, se crearon funciones para obtener el área del cuadrado que contiene la figura, así como una función que calcula el área real de la figura.

```
def area_cuadrado(n):
    """
    Retorna el área real del cuadrado
    """
    return n ** 2


def area_analitica_cuadrantes(n):
    '''
    Devuelve el area real de la figura buscada de forma analitica
    de la figura en base a el lado del cuadrado que la contiene
    '''
    return (n ** 2) * (1 - (3 ** (1 / 2)) + np.pi / 3)
```

Después de eso, se creó una función que modela el experimento de Montecarlo, donde los límites de generación de puntos son 0 como límite inferior y 20 como límite superior. Esto se debe a las funciones utilizadas para modelar la figura.

```
def monte_carlo_hit_and_miss(n):
    """
    Calcula el área de los arcos de cuadrante utilizando el método Hit and Miss con n dardos

    Parameters:
    n (int): número de dardos
```

```
    Returns:
    float: el área de los arcos de cuadrante (aprox, usando montecarlo)
    """
    a = 0
    b = 20
    hits = sum(f(x, y) for x, y in zip(np.random.uniform(a, b, n), np.random.uniform(a, b, n)))
    return hits / n * area_cuadrado(20)
```

Ademas en la misma seccion se crearon algunas variables:

- n_point: la cual es el listado de cada uno de los numeros de puntos de los cuales se simularan
- n_areas: la cual es el area de cada uno de los elementode de n_points
- n_simulations: representa el numero de simulaciones por cada numero de puntos (cada item en n_points), esto con el fin de ver variaciones en los resultados y medirlos
- errors: Que representa el error relativo de cada n_point en sus respectivas simulaciones
- resulrs_df: donde se guardara toda la data usando la libreria pandas

```
# Obtención de datos
n_points = [10**i for i in range(0, 6, 1)]
n_simulations = 1000
n_areas = np.zeros((len(n_points), n_simulations))
errors = np.zeros((len(n_points), n_simulations))
results_df = pd.DataFrame(columns=['Número de puntos', 'Simulación', 'Aproximación', 'Error relativo'])
```

Como se puede observar, la estructura es muy similar a la del método de muestreo. De hecho, es el mismo. El cambio en el código es mínimo. En ese contexto, se procedió a crear una función que grafique un experimento, una simulación, con un valor cualquiera, con el fin de visualizar el modelo.

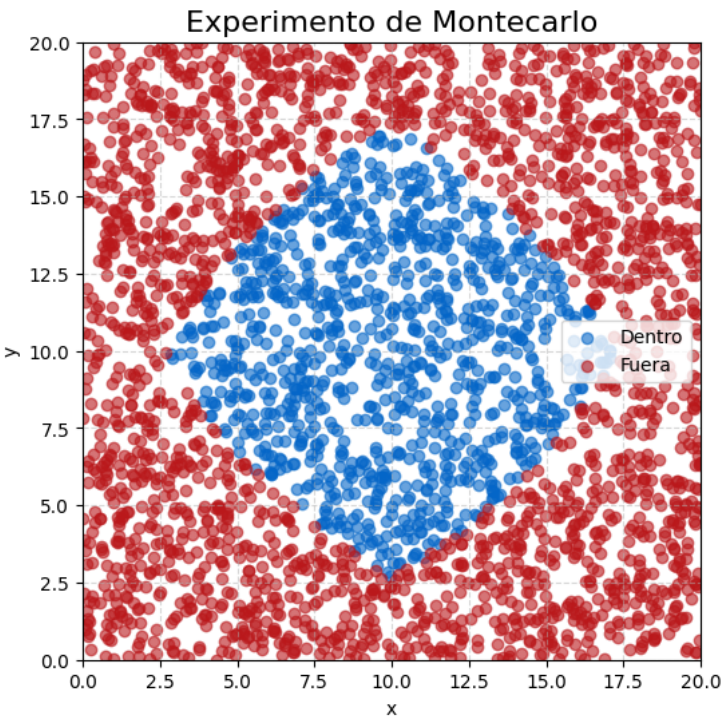
```
def montecarlo_experiment_show(num_points):
    inside_points_x = []
    inside_points_y = []
    outside_points_x = []
    outside_points_y = []

    for _ in range(num_points):
        x = np.random.uniform(0, 20)
        y = np.random.uniform(0, 20)

        if f(x, y):
            inside_points_x.append(x)
            inside_points_y.append(y)
        else:
            outside_points_x.append(x)
            outside_points_y.append(y)

    plt.figure(figsize=(6, 6))
    plt.scatter(inside_points_x, inside_points_y, c='#0466c8', label='Dentro', alpha=0.6)
    plt.scatter(outside_points_x, outside_points_y, c='#ba181b', label='Fuera', alpha=0.6)
    plt.xlim(0, 20)
    plt.ylim(0, 20)
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Experimento de Montecarlo', fontsize=16)
    plt.grid(linestyle='--', alpha=0.5)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

# Ejemplo de uso
montecarlo_experiment_show(3107)
```



Una vez que se ha comprobado y visualizado el funcionamiento del ejemplo arbitrario, procedemos a realizar varias simulaciones con un número creciente de puntos en base 10, similar a la representación en n_points utilizado en el método de muestreo. Durante este proceso, obtendremos los errores relativos y las desviaciones estándar, que se guardarán en results_df. También registraremos el tiempo de ejecución del algoritmo, lo cual es importante para compararlo con el del método de muestreo.

```
area_real = area_analitica_cuadrantes(20)

# Iniciar tiempo de ejecución
start_time = time.time()

for i, j in enumerate(n_points):
    for k in range(n_simulations):
        area = monte_carlo_hit_and_miss(j)
        error = abs(area_real - area) / area_real

        # Agregar los resultados al DataFrame
        results_df = results_df.append({
            'Número de puntos': j,
            'Simulación': k+1,
            'Aproximación': area,
            'Error relativo': error
        }, ignore_index=True)

# Imprimir el DataFrame completo
print(results_df)

# Calcular el tiempo de ejecución total
elapsed_time = time.time() - start_time
print(elapsed_time)
```

```
      Número de puntos  Simulación  Aproximación  Error relativo
0                  1.0         1.0         0.000        1.000000
1                  1.0         2.0         0.000        1.000000
2                  1.0         3.0         0.000        1.000000
3                  1.0         4.0         0.000        1.000000
4                  1.0         5.0         0.000        1.000000
...                ...         ...           ...             ...
5995          100000.0       996.0       126.124        0.000518
5996          100000.0       997.0       126.956        0.007118
5997          100000.0       998.0       125.700        0.002845
5998          100000.0       999.0       125.564        0.003924
5999          100000.0      1000.0       126.300        0.001914

[6000 rows x 4 columns]
126.72608184814453
```

Una vez obtenida la información, se calcularon las desviaciones estándar y los promedios para poder mostrar los resultados obtenidos.

```
# Calcular promedios y desviaciones estándar de las simulaciones
avg_areas = results_df.groupby('Número de puntos')['Aproximación'].mean()
std_areas = results_df.groupby('Número de puntos')['Aproximación'].std()
avg_errors = results_df.groupby('Número de puntos')['Error relativo'].mean()
std_errors = results_df.groupby('Número de puntos')['Error relativo'].std()

print(avg_areas)
print(std_areas)
print(avg_errors)
print(std_errors)
```

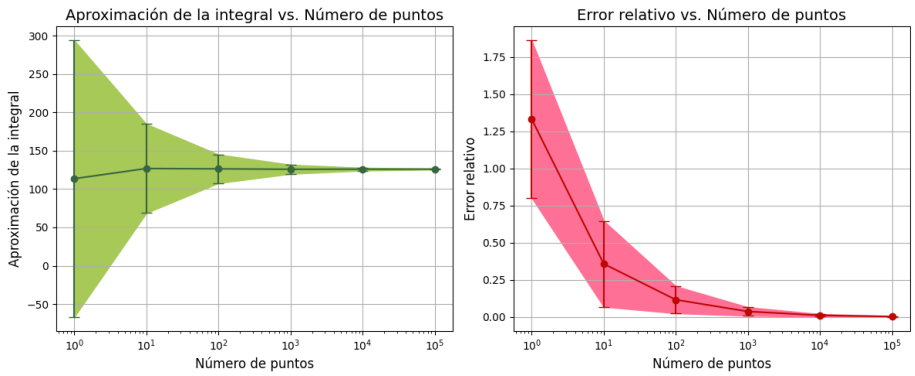
```
10.0        126.840000
100.0       126.432000
1000.0      125.849600
10000.0     125.980320
100000.0    126.051668
Name: Aproximación, dtype: float64
Número de puntos
1.0         180.464976
10.0         58.018795
100.0        18.652219
1000.0        5.958073
10000.0       1.839276
100000.0      0.583625
Name: Aproximación, dtype: float64
Número de puntos
1.0         1.333167
10.0        0.358358
100.0       0.116696
1000.0      0.037595
10000.0     0.011604
100000.0    0.003680
Name: Error relativo, dtype: float64
Número de puntos
1.0         0.529270
10.0        0.288655
100.0       0.090943
1000.0      0.028668
10000.0     0.008859
100000.0    0.002807
Name: Error relativo, dtype: float64
```

De manera prudente se decidio graficar no solo el error, sino tambien la aproximacion de los puntos al resultado, ademas de sus desviaciones estandar para cada numero de puntos:

```
# Graficar los resultados
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.errorbar(n_points, avg_areas, yerr=std_areas, c = "#386641", ecolor="#386641",fmt='o-', capsize=5)
plt.fill_between(n_points, avg_areas - std_areas, avg_areas + std_areas, color='#a7c957')
plt.xscale('log')
plt.xlabel('Número de puntos', fontsize=12)
plt.ylabel('Aproximación de la integral', fontsize=12)
plt.title('Aproximación de la integral vs. Número de puntos', fontsize=14)
plt.grid(True)

plt.subplot(1, 2, 2)
plt.errorbar(n_points, avg_errors, yerr = std_errors, c ="#bf0603",fmt='o-', capsize=5)
plt.fill_between(n_points, avg_errors - std_errors, avg_errors + std_errors, color='#ff7096')
plt.xscale('log')
plt.xlabel('Número de puntos', fontsize=12)
plt.ylabel('Error relativo', fontsize=12)
plt.title('Error relativo vs. Número de puntos', fontsize=14)
plt.grid(True)

plt.tight_layout()
plt.show()
```



IV. RESULTADOS

Es importante destacar que ambos métodos tienen una complejidad temporal de su algoritmo en O(n * m), donde n es la longitud de n_points/n_bars y m es el número de simulaciones (n_simulations).

Podemos comparar las diferencias entre los dos métodos analizando las gráficas obtenidas durante la experimentación, teniendo en cuenta que utilizamos la misma muestra y la misma semilla para garantizar igualdad de condiciones para ambos.

Sampling

Hit and Miss

Como se puede observar en las gráficas, aunque en promedio ambos métodos dan un resultado aceptable con un número bajo de puntos o barras, el método de muestreo presenta una mejor aproximación al área. Si el resultado es pobre con pocas barras, su desviación estándar es considerablemente menor, siendo 0.18 en comparación con 0.59 en el método de Hit and Miss, lo cual indica una diferencia considerable a favor del método de muestreo.

De igual manera, se puede observar que el método de muestreo tiene un error relativo promedio considerablemente más bajo que el método de Hit and Miss. Esto está directamente relacionado con el rendimiento mencionado anteriormente con pocas barras o puntos.

Con el aumento del número de puntos, el método de Hit and Miss comienza a dar mejores resultados, pero el método de muestreo también mejora y la diferencia de rendimiento inicial es notable y se mantiene.

Ambos métodos, con un número alto de puntos, dan como resultado un área muy aproximada con un error relativo muy bajo, siendo 0.001190 en el caso del método de muestreo y 0.003793 en el caso del método de Hit and Miss, con una diferencia de 0.002603, la cual es despreciable.

Se puede notar que el método de Hit and Miss ofrece un resultado más preciso, pero esto va de la mano con un costo computacional muy elevado. Tanto es así que los tiempos de ejecución del método de muestreo no superan en promedio los 30 segundos, con un rango de tiempo de ejecución de 8 a 35 segundos con la muestra seleccionada. Por otro lado, el método de Hit and Miss puede llegar a tardar varios minutos en su ejecución, con un tiempo de ejecución promedio mínimo de 162 segundos.

Después de todo lo mencionado, es importante recalcar que, en promedio, ambos métodos ofrecen un resultado muy aproximado. Se puede observar en las gráficas cómo el error disminuye a medida que aumenta el número de puntos o barras utilizadas, lo que muestra una relación inversamente proporcional.

V. CONLCUSIÓN

- El método de muestreo es superior al método de hit and miss, ya que proporciona resultados mucho más precisos con un costo computacional menor. Los resultados son precisos incluso con pocas barras, lo que lo hace escalable. Por otro lado, el método de hit and miss, al aumentar el número de puntos a uno muy elevado, tiende a tener un tiempo de ejecución abismalmente grande.
- La relación entre el número de puntos/barras y el error relativo es inversamente proporcional. A medida que aumenta el número de puntos, el error relativo disminuye. Sin embargo, cuando el número de puntos es ya alto, aumentar aún más la cantidad no resultará en una disminución significativa del error.
- Aumentar el número de puntos/barras se traduce en un resultado más preciso. Sin embargo, cuando se trata de un número bajo de puntos, esto no es inmediato. Es decir, se puede aumentar el número de puntos, pero aún así puede dar un resultado con un error alto. En ese caso, la desviación estándar puede ser un indicador que nos ayude a verificar la efectividad de ese número de puntos.
- En el método de hit and miss, es importante seleccionar una semilla adecuada, ya que podría darse el caso de que no caigan dardos/puntos dentro del perímetro deseado.

VI. RECOMENDACIONES

- Utilizar el método de sampling en lugar del método hit and miss: Basado en las conclusiones presentadas, es recomendable enfocarse en el método de sampling en lugar del método hit and miss. El método de sampling ofrece resultados más precisos con un menor costo computacional. Además, su escalabilidad lo convierte en una opción preferida para aplicaciones con un número elevado de puntos.

- Determinar el número óptimo de puntos/barras: Es importante resaltar que la relación entre el número de puntos/barras y el error relativo es inversamente proporcional. A medida que aumenta el número de puntos, el error relativo tiende a disminuir. Sin embargo, existe un punto de inflexión donde el aumento adicional de puntos no resultará en una reducción significativa del error. Es recomendable realizar análisis de sensibilidad para determinar el número óptimo de puntos/barras que minimice el error relativo sin incurrir en un costo computacional excesivo.
- Considerar la desviación estándar como indicador de efectividad: En casos donde el número de puntos/barras es bajo y se obtienen resultados con un error alto, es importante utilizar la desviación estándar como un indicador adicional para evaluar la efectividad de ese número de puntos. Una desviación estándar alta sugiere una mayor variabilidad en los resultados y puede ser un signo de que el número de puntos utilizado no es suficiente para lograr la precisión deseada.
- Documentar el impacto del método de Montecarlo en el rendimiento: Además de las consideraciones relacionadas con la precisión y el error, es relevante analizar el impacto del método de Montecarlo en el rendimiento computacional. En el informe, se recomienda incluir datos comparativos de los tiempos de ejecución entre el método de sampling y el método hit and miss para respaldar la afirmación de que el método de sampling ofrece tiempos de ejecución considerablemente más bajos en escenarios con un alto número de puntos.

VII. BIBLIOGRAFIA

[1] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the Monte Carlo method is so important today," WIREs Comput Stat., vol. 6, no. 6, pp. 386–392, 2014. doi: 10.1002/wics.1314.

[2] D. Hubbard and D. A. Samuelson, "Modeling Without Measurements," OR/MS Today, pp. 28–33, Oct. 2009.

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," J. Chem. Phys., vol. 21, no. 6, pp. 1087–1092, Jun. 1953. doi: 10.1063/1.1699114.

[4] W. K. Hastings, "Monte Carlo sampling methods using Markov chains and their applications," Biometrika, vol. 57, no. 1, pp. 97–109, Apr. 1970. doi: 10.1093/biomet/57.1.97.

[5] J. S. Liu, F. Liang, and W. H. Wong, "The Multiple-Try Method and Local Optimization in Metropolis Sampling," J. Am. Stat. Assoc., vol. 95, no. 449, pp. 121–134, Mar. 2000. doi: 10.1080/01621459.2000.10473908.