Reporte 5:
Método Monte-Carlo

Jorge Torres

13 de marzo de 2022

Capítulo 1

Estimación de una Integral Definida

1.1. Objetivo

El objetivo de esta actividad es el estudiar estadísticamente la convergencia de la precisión del estimado de una integral definida (ecuación 1.1), con el método Monte Carlo, comparándolo con el valor producido por Wolfram Alpha, en términos de 1) el error absoluto, 2) el error cuadrado y 3) la cantidad de decimales correctos, aumentando el tamaño de muestra en tres niveles.

$$\int_3^7 \frac{1}{\exp(x) + \exp(-x)}\, dx \tag{1.1}$$

1.2. Desarrollo

El desarrollo de la actividad está basado en el código implementado por E. Schaeffer [2]. En el código 1.1 se establecen los parámetros para la ejecución del programa, que consisten en 1) el valor estimado por Wolfram Alpha, `wolfram`, 2) el rango en que se define la integral, `desde` y `hasta`, 3) la cantidad de números pseudoaleatorios que se producen con la distribución definida por la integral, `pedazo`, y 4) la lista de niveles en que se varía la cantidad de iteraciones del experimento, `cuantos`.

Código 1.1: Parámetros

```
wolfram = 0.048834111126049311
desde = 3
hasta = 7
pedazo = 50000
cuantos = [500, 5000, 50000]
```

Se define la normalización de la integral por medio de la función `g(x)` dada por la ecuación 1.2 y se vectoriza la distribución resultante en el código 1.2.

$$\frac{2}{\pi}\int_{-\infty}^{\infty} \frac{1}{\exp(x) + \exp(-x)}\, dx \tag{1.2}$$

Código 1.2: Normalización de la Integral

```
def g(x):
    return (2  / (pi * (exp(x) + exp(-x))))

vg = np.vectorize(g)
X = np.arange(-8, 8, 0.05)
Y = vg(X
```

Importando una receta y utilizándola en conjunto con la función `parte()` del código 1.3, se genera una cantidad de números aleatorios con una distribución de acuerdo a la integral normalizada, visualizada en la figura 1.1.

Código 1.3: Generación de Números Aleatorios

```
generador = GeneralRandom(np.asarray(X), np.asarray(Y))

def parte(replica):
```

```
    V = generador.random(pedazo)[0]
    return ((V >= desde) & (V <= hasta)).sum()
```

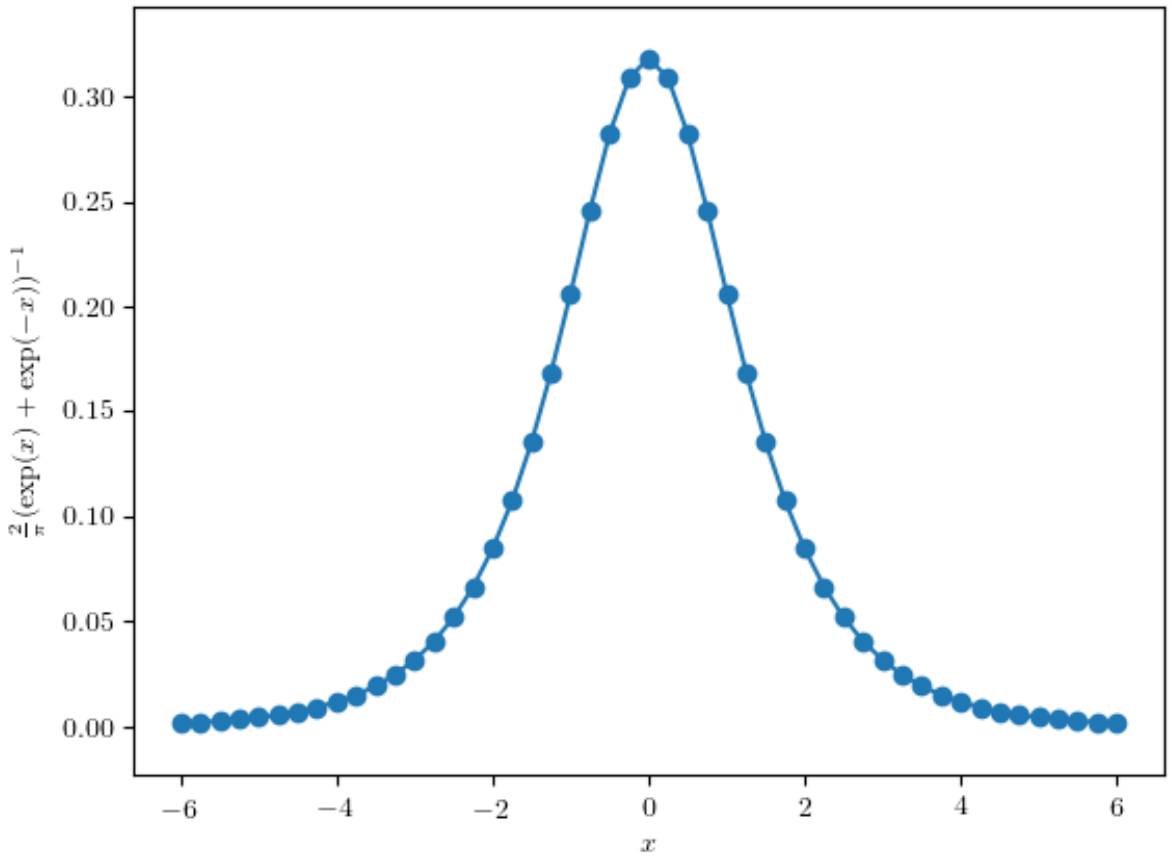


Figura 1.1: Distribución de la integral normalizada. Obtenida de P5 - Simulación

Para determinar la cantidad de decimales correctos entre el estimado de la integral y el valor de Wolfram Alpha, se define la función `compare_strings` en el código 1.4, que compara ambos valores caracter por caracter hasta que el decimal es diferente.

Código 1.4: Comparación de Decimales

```
def compare_strings(a, b):
    a = str(a)
    b = str(b)

    if a is None or b is None:
        return 0

    size = min(len(a), len(b))
    count = 0

    for i in range(size):
        if a[i] == b[i]:
            count += 1
        else:
            break
    return count
```

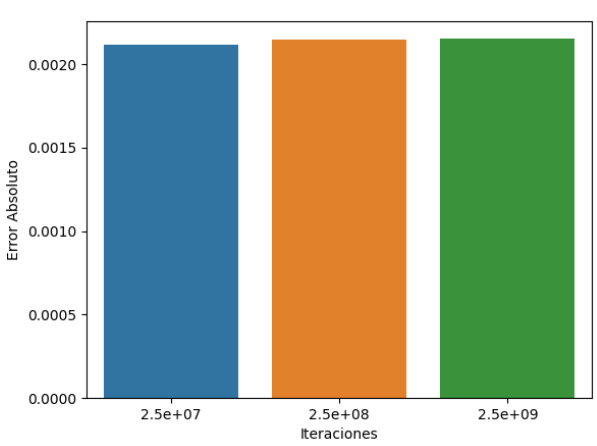
Por último, por medio del código 1.5 se ejecutan las iteraciones del experimento y se calculan los valores de los errores para poder comparar las estimaciones de la integral definida dependiendo de la cantidad de iteraciones. El desarrollo completo del código se puede revisar en el repositorio en GitHub de J. Torres [3].

Código 1.5: Estimación de la Integral y Cálculo de Errores

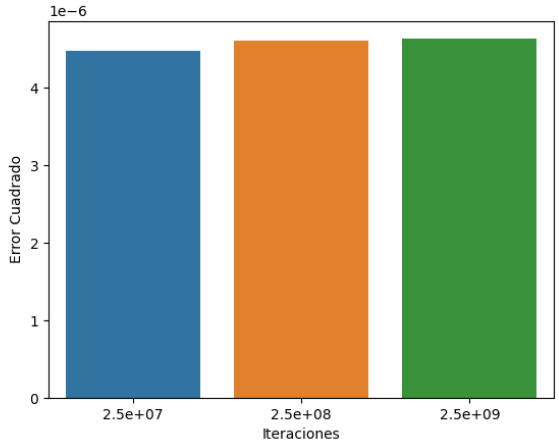
```
if __name__ == "__main__":
    with multiprocessing.Pool() as pool:
        for c in cuantos:
            p = c * pedazo
            puntos.append('{:.1e}'.format(p))
            montecarlo = pool.map(parte, range(c))
            integral = sum(montecarlo) / p
            valor = (pi / 2) * integral
            ae.append(abs(valor - wolfram))
            se.append(((valor - wolfram)**2))
            dec.append(compare_strings(wolfram, valor) - 2)
        resultados = {'Iteraciones': puntos,
                      'Error Absoluto': ae,
                      'Error Cuadrado': se,
                      'Decimales Correctos': dec}
        df = pd.DataFrame(resultados)
```

1.3. Resultados

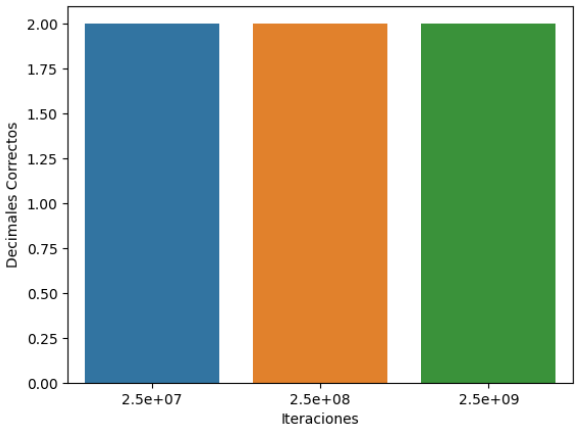
En los diagramas de la figura 1.2 se puede apreciar la variación en los valores de error absoluto (figura 1.2a), error cuadrado (figura 1.2b), y cantidad de decimales correctos (figura 1.2c), al comparar entre los valores de la integral definida estimados por Wolfram Alpha y por el código desarrollado en la actividad.



(a) Error absoluto.



(b) Error cuadrado.



(c) Decimales correctos.

Figura 1.2: Errores determinados entre la estimación de Wolfram Alpha y la calculada por el código desarrollado.

1.4. Conclusiones

Como se puede apreciar en los diagramas de la sección 1.3, aumentar la cantidad de iteraciones de $2{,}5 \times 10^7$ hasta $2{,}5 \times 10^9$ no presenta una diferencia apreciable en ningún tipo de error. Lo que es más, tanto el error absoluto como el error cuadrado son mínimos y no representarían un error estrictamente significativo. Sin embargo, se puede ver que la cantidad de decimales correctos no excede un máximo de dos, por lo que el rango de error comienza a crecer en las milésimas. En muchas aplicaciones, este rango podría causar diferencias bastante apreciables en cálculos que impliquen más precisión.

Capítulo 2

Estimación de π

2.1. Objetivo

El primer reto de esta actividad consiste en utilizar el método Monte-Carlo para estimar el valor de π a partir de una serie de puntos, determinando si las coordenadas de estos puntos se encuentran dentro o fuera del área de un círculo de radio r.

2.2. Desarrollo

El código 2.1 es tomado directamente del método descrito por W. Kurt [1], y en él se utiliza la relación entre el área de un círculo y el cuadrado que lo circunscribe para aproximar el valor de *pi* a partir de una serie de puntos de números generados aleatoriamente dentro de una distribución normal. Más específicamente, se suma la cantidad de puntos que queda dentro del círculo, se divide entre la cantidad total de puntos, y se multiplica por 4 para obtener la aproximación.

Código 2.1: Cálculo de π por Método Monte-Carlo

```
runs <- 1000000
xs <- runif(runs,min=-0.5,max=0.5)
ys <- runif(runs,min=-0.5,max=0.5)
in.circle <- xs^2 + ys^2 <= 0.5^2
mc.pi <- (sum(in.circle)/runs)*4)
plot(xs,ys,pch='.',col=ifelse(in.circle,"blue","grey")
     ,xlab='',ylab='',asp=1,
     main=paste("MC Approximation of Pi =",mc.pi))
```

2.3. Resultados

En la figura 2.1 se visualiza de manera clara la serie de puntos ubicados aleatoriamente, que caen tanto dentro como fuera del círculo. También se puede apreciar la aproximación calculada, $\pi = 3{,}141956$.

2.4. Conclusiones

Inmediatamente queda claro que éste método es útil para aproximar valores de procesos que son complejos de calcular analíticamente con una precisión aceptable, ya que se ha podido estimar π correctamente en un rango tres decimales. Es de esperar que la precisión aumente con la cantidad de puntos que se generan, pero en esta actividad no se han realizado pruebas para confirmarlo.

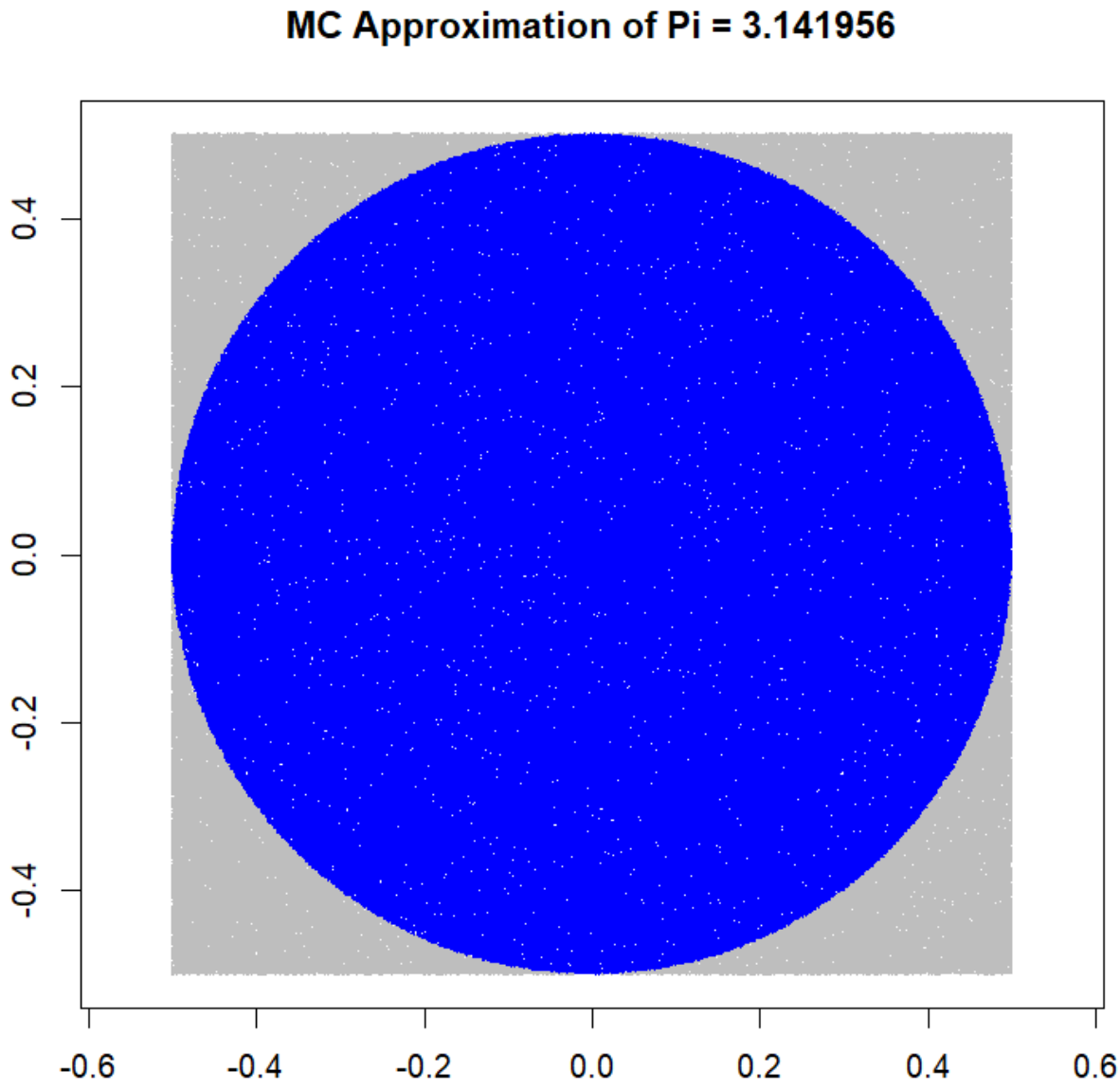


Figura 2.1: Círculo formado por los puntos aleatorios.

Bibliografía

[1] W. Kurt. Count bayesie, 2015. URL https://www.countbayesie.com/blog/2015/3/3/6-amazing-trick-with-monte-carlo-simulations.

[2] E. Schaeffer. Montecarlo, 2019. URL https://github.com/satuelisa/Simulation/tree/master/MonteCarlo.

[3] J. Torres. P5, 2022. URL https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P5.