

Métodos de Monte Carlo - 2

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

2 de mayo de 2018

Contenido

1

Métodos Monte Carlo

- Integración Monte Carlo
- Integración en varias variables
- Calculando áreas con puntos aleatorios

Integración Monte Carlo

Una de las primeras aplicaciones con el uso de números aleatorios ha sido el cálculo numérico de integrales, es decir, un problema no aleatorio (determinista).

Veamos un primer método para calcular

$$\int_a^b f(x) \, dx$$

Integración estándar Monte Carlo

Sea x_1, x_2, \dots, x_n la secuencia de números aleatorios uniformemente distribuidos entre a y b .

Entonces

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

Aproximación a la integral

Es una aproximación a la integral

$$\int_a^b f(x)dx$$

Este método se denomina generalmente
integración de Monte Carlo.

Aproximación a la integral

Es fácil de interpretar la ec. (1): Un resultado bien conocido del cálculo es que la integral de una función f sobre $[a, b]$ es igual al valor medio de f sobre $[a, b]$ multiplicado por la longitud del intervalo, $(b - a)$.

Aproximación a la integral

Si aproximamos el valor medio de $f(x)$ por la media de n evaluaciones de la función distribuidas al azar $f(x_i)$, obtenemos el método de integración.

Estimando la integral con python

Podemos implementar la ec. (1) en una pequeña función de python:

Código 1: Función para la aproximar la integral

```
1 import random
2
3 def MCint(f, a, b, n):
4     s = 0
5     for i in range(n):
6         x = random.uniform(a, b)
7         s += f(x)
8
9     I = (float(b-a)/n) * s
10    return I
```

Versión más rápida del código

Normalmente se necesita un valor de n grande para obtener buenos resultados con este método, por lo que una versión vectorizada más rápida de la función anterior es útil:

Versión más rápida del código

Código 2: Función vectorizada para la aproximación de la integral

```
1 import random
2 from numpy import *
3
4 def MCintvec(f, a, b, n):
5     x = random.uniform(a, b, n)
6     s = sum(f(x))
7     I = (float(b-a)/n) * s
8     return I
```

Ejemplo

Vamos a probar el método de integración de Monte Carlo en una función simple, sea $f(x) = 2 + 3x$, y calculemos la integral en $[1, 2]$.

La mayoría de los otros métodos de integración numérica resolverán exactamente esa función lineal, independientemente del número de evaluaciones de la función.

Ejemplo

Este no es el caso de la integración de Monte Carlo.

Sería interesante ver cómo la calidad de la aproximación de Monte Carlo se incrementa, conforme crece el valor de n .

Ejemplo

Para graficar la evolución de la aproximación con la integral, debemos almacenar los valores intermedios I , por lo que debemos de modificar el código:

Código 3: Código para el ejercicio

```
1 def MCint2(f, a, b, n):
2     # se guardan las aproximaciones
3     # intermedias de la integral en el
4     # arreglo I,
5     # donde I[k-1] es k veces la
6     # funcion evaluada
7     s = 0
8
9     I = np.zeros(n)
10
11    for k in range(1, n+1):
12        x = random.uniform(a, b)
13        s += f(x)
14        I[k-1] = (float(b-a) / k) * s
15
16    return I
```

Consideraciones para la solución

Toma en cuenta que hacemos que k vaya de 1 a n mientras que los índices en I , como de costumbre, van de 0 a $n - 1$.

Consideraciones para la solución

Dado que n puede ser muy grande, el arreglo I puede consumir más memoria de lo que tenemos disponible en la computadora.

Por lo tanto, decidimos almacenar sólo cada N valores de la aproximación.

Consideraciones para la solución

El determinar si un valor debe ser almacenado o no puede ser calculado por la función `mod`:

Código 4: Almacenamiento de valores

```
1 for k in range(1, n+1):
2     if k % N == 0:
3         #se guarda
```

Es decir, cada vez que k se divide N sin ningún residuo, almacenamos el valor.

Código completo I

La función completa sería:

Código 5: Código completo para el ejercicio

```
1 def MCint3(f, a, b, n, N=100):
2     s = 0
3     Ivalores = []
4     kvalores = []
5
6     for k in range(1, n+1):
7         x = random.uniform(a, b)
8         s += f(x)
9         if k % N == 0:
10             I = (float(b-a)/k) * s
```

Código completo II

```
11     Ivalores.append(I)
12     kvalores.append(k)
13
14     return kvalores, Ivalores
```

Estimación del error

Ahora podemos revisar el error que se genera al usar la función:

Código 6: Estimación del error del procedimiento

```
1 def f1(x):
2     return 2 + 3*x
3
4
5 k, I = MCint3(f1, 1, 2, 1000000, N=1
6 0000)
7 error = 6.5 - np.array(I)
```

Gráfica del error vs N

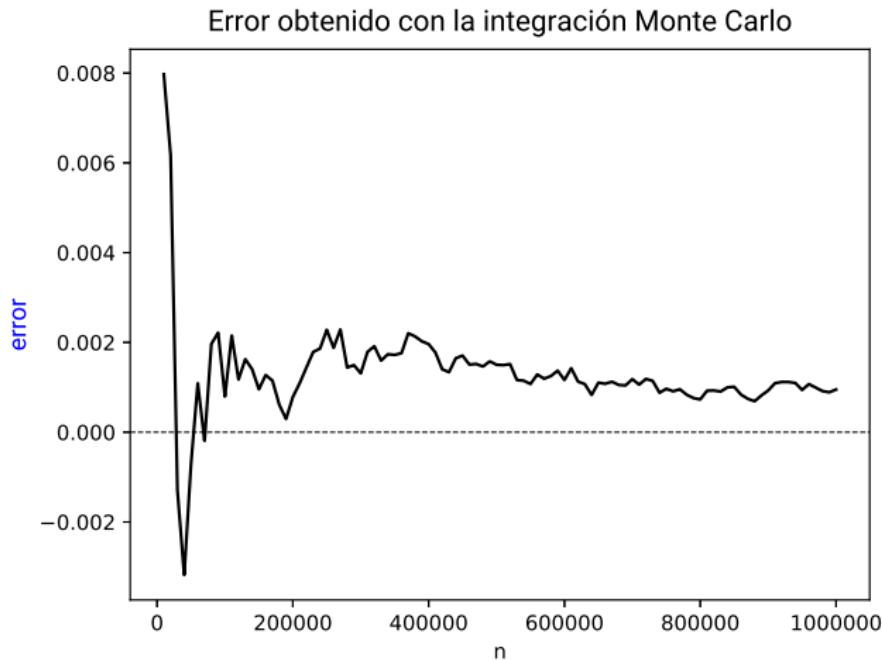


Figura 1: Convergencia de la integración de Monte Carlo para $f(x)$

Velocidad de convergencia

Para funciones de una variable, el método (1) requiere de muchos puntos y es ineficiente comparado con otras reglas de integración.

La mayoría de las reglas de integración tienen un error que se reduce mientras se incrementa n , normalmente de la manera n^{-r} para un $r > 0$.

Velocidad de convergencia

Para la regla del trapecio, $r = 2$, mientras que para la integración de Monte Carlo $r = 1/2$

Esto significa que este método converge muy lentamente en comparación con la regla del trapecio.

Integración en varias variables

Sin embargo, para funciones de varias variables, la integración de Monte Carlo en espacios de dimensiones mayores supera completamente a métodos como la regla del trapecio y la regla de Simpson.

Mejoras para el cálculo de la integral

Existen diferentes maneras de mejorar el rendimiento de la ec. (1), básicamente por ser “aplicados” al momento de graficar los números aleatorios, conocidas como técnicas de reducción de la varianza.

Calculando áreas con puntos aleatorios

Pensemos en alguna región geométrica G en el plano y una caja circundante B con geometría $[x_L, x_H] \times [y_L, y_H]$.

Una forma de calcular el área de G es dibujar N puntos aleatorios dentro de B y contar cuántos de ellos M , están dentro de G .

Puntos dentro de la región

El área de G es entonces la fracción M/N (fracción de G en el área de B) veces el área de B , $(x_H - x_L)(y_H - y_L)$.

Puntos dentro de la región

De forma diferente, este método es una especie de juego de dardos en el que se cuentan los que caen dentro de G , si cada lanzamiento llega de manera uniforme dentro de B .

Calculando una integral

Veamos cómo será la expresión para calcular la integral

$$\int_a^b f(x) dx$$

Como nota relevante, consideremos que esta integral es el área debajo de la curva $y = f(x)$ y sobre el eje x , entre $x = a$ y $x = b$.

Calculando una integral

Introducimos un rectángulo B , tal que

$$B = \{(x, y) | a \leq x \leq b, 0 \leq y \leq m\}$$

donde $m \leq \max_{x \in [a, b]} f(x)$

Calculando una integral

El algoritmo para calcular el área bajo la curva se basa en dibujar N puntos aleatorios dentro de B y contar cuántos de ellos, M , están por encima del eje $x - y$ y cuántos por debajo de la curva $y = f(x)$

Método del dardo

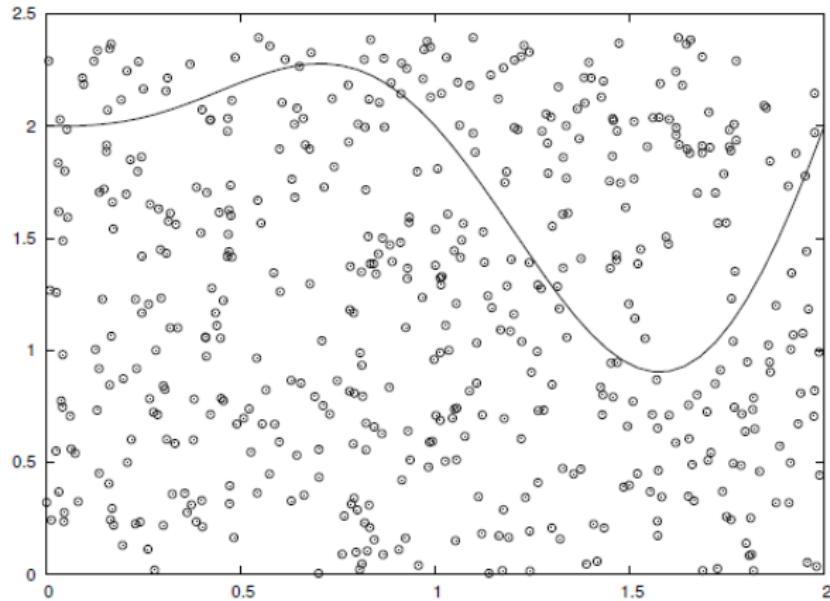


Figura 2: Cuando M de N puntos aleatorios en el rectángulo $[0, 2] \times [0, 2.4]$ se encuentran bajo la curva, el área bajo la curva se estima como la fracción M/N del área del rectángulo.

Valor de la integral

El área o el valor de la integral se estima por la expresión

$$\frac{M}{N} m (b - a)$$

El código sería

Código para la integral

Código 7: Código para el método del dero

```
1 def MCint-area(f, a, b, n, m):  
2     porDeabajo = 0  
3     for i in range(n):  
4         x = random.uniform(a, b)  
5         y = random.uniform(0, m)  
6         if y <= f(x):  
7             porDeabajo += 1  
8         area = porDeabajo/float(n) * m *  
9             (b - a)  
10    return area
```

Toma en cuenta que este método opera con el doble de números aleatorios que el método anterior.

Una implementación vectorizada del código es

Código 8: Método del dardo en modo vectorizado

```
1 def MCint-area-vec(f, a, b, n, m):  
2     x = random.uniform(a, b, n)  
3     y = random.uniform(0, m, n)  
4     porDeabajo = np.sum(y < f(x))  
5     area = porDeabajo/float(n) * m *  
6         (b - a)  
7  
    return area
```

Podemos ejecutar el código para un conjunto de 2 millones de números al azar, la versión de bucle sencillo no es tan lenta.

Sin embargo, si necesita que la integración se repita muchas veces dentro de otro cálculo, puede ser importante la eficacia superior de la versión vectorizada.

Función completa I

Código 9: Función para el método del dardo

```
1 def MCint3area(f, a, b, n, m, N=1000
  ) :
2     Ivalores = []
3     kvalores = []
4     porDeabajo = 0
5     for k in range(1, n+1):
6         x = random.uniform(a, b)
7         y = random.uniform(0, m)
8         if y <= f(x):
9             porDeabajo += 1
10            area = porDeabajo/float(k) *
m * (b-a)
```

Función completa II

```
11     if k % N == 0:  
12         I = area  
13         Ivalores.append(I)  
14         kvalores.append(2 * k)  
15     return kvalores, Ivalores
```

Implementación del código

Al contar con los elementos necesarios, podemos implementar el código completo para estimar el valor de la integral a partir de generar puntos aleatorios.

Haremos una estimación del tiempo que tarda en resolverse el problema usando el bucle y la versión vectorizada.

Código 10: Implementación del código

```
1 def f1(x):
2     return 2 + 3*x
3
4 a = 1; b = 2; n = 1000000; N = 10000
5 ; fmax = f1(b)
6
7 t0 = time.clock()
8
9
10 print (MCintarea(f1, a, b, n, fmax))
11
12 print (MCintareavec(f1, a, b, n,
13 fmax))
```

```
13  
14 t2 = time.clock()  
15  
16 print ('fraccion bucle/vectorizada:'  
     , (t1 - t0) / (t2 - t1))  
17  
18 k, I = MCint3area(f1, a, b, n, fmax,  
     N)  
19  
20 print (I [-1] )  
21  
22 error = 6.5 - np.array(I)
```

Gráfica de la función

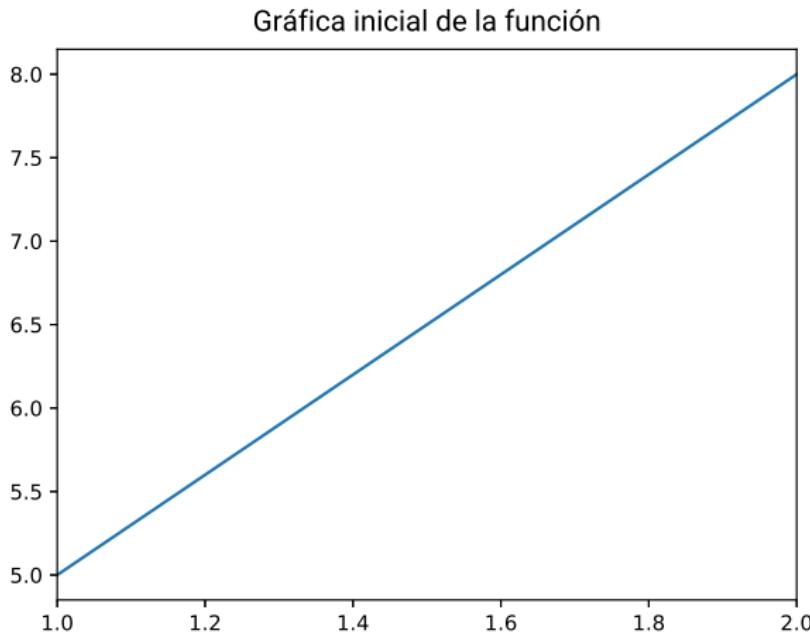


Figura 3: Función inicial.

Gráfica al incluir puntos aleatorios

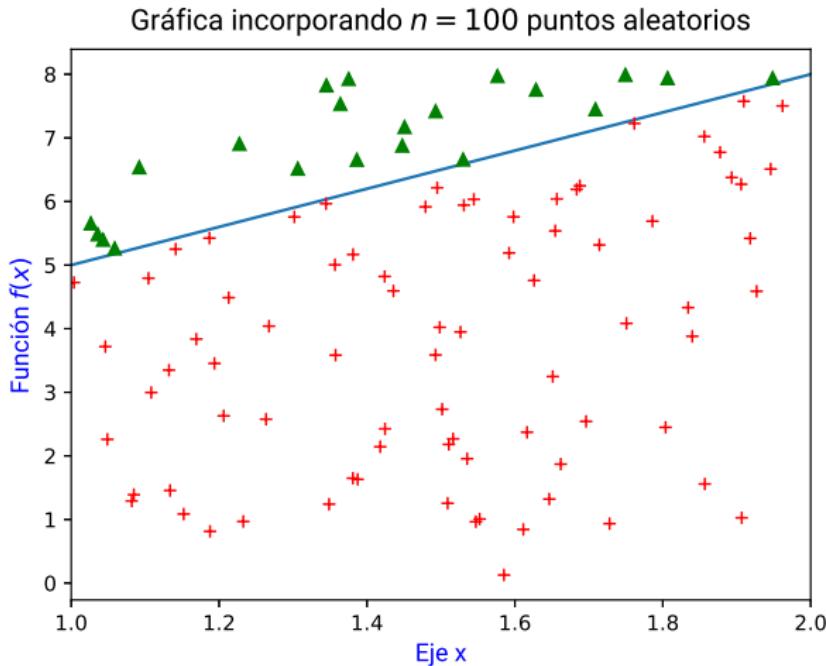


Figura 4: Puntos por arriba y por debajo de la función.

Gráfica al incluir puntos aleatorios

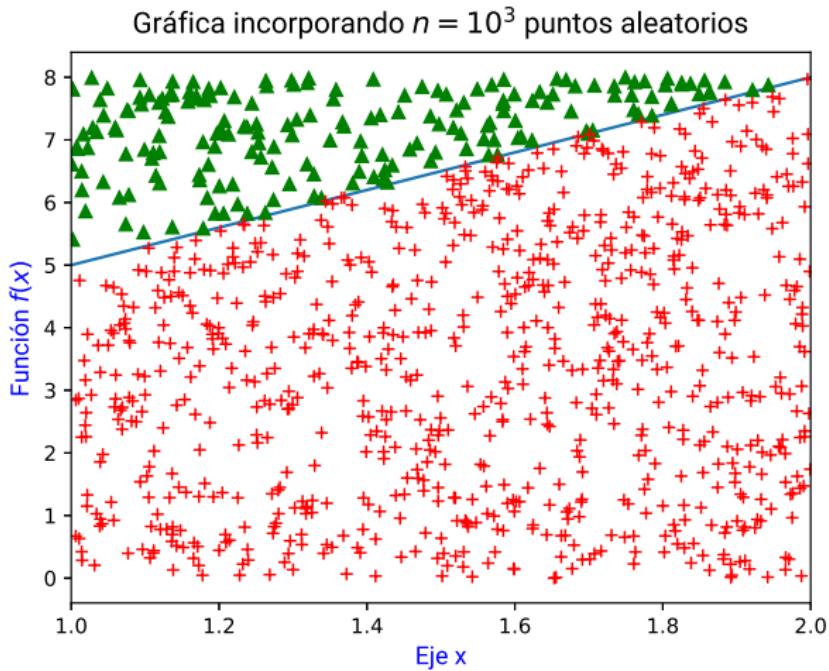


Figura 5: Se va saturando el área debajo de la curva.

Gráfica al incluir puntos aleatorios

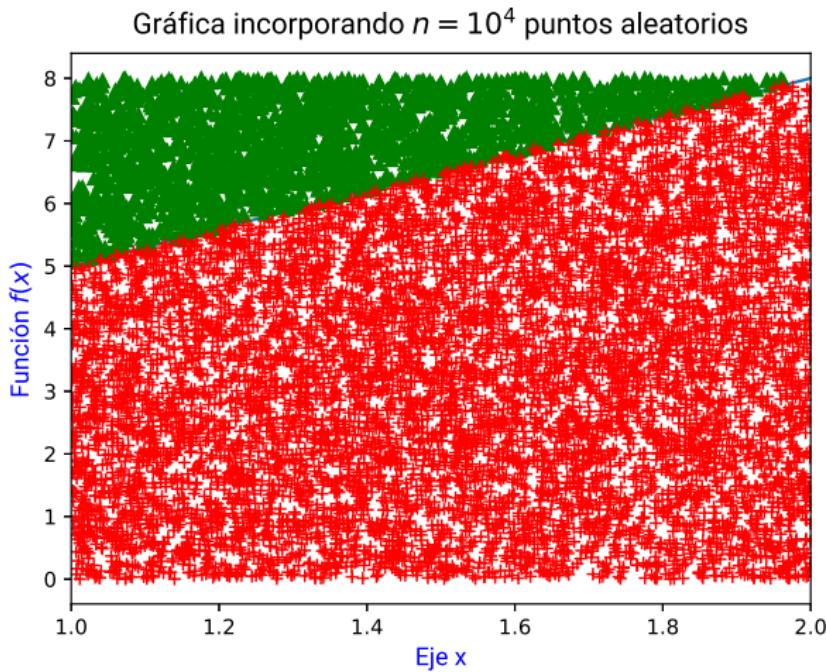


Figura 6: Casi se cubre el área debajo de la curva.

Gráfica al incluir puntos aleatorios

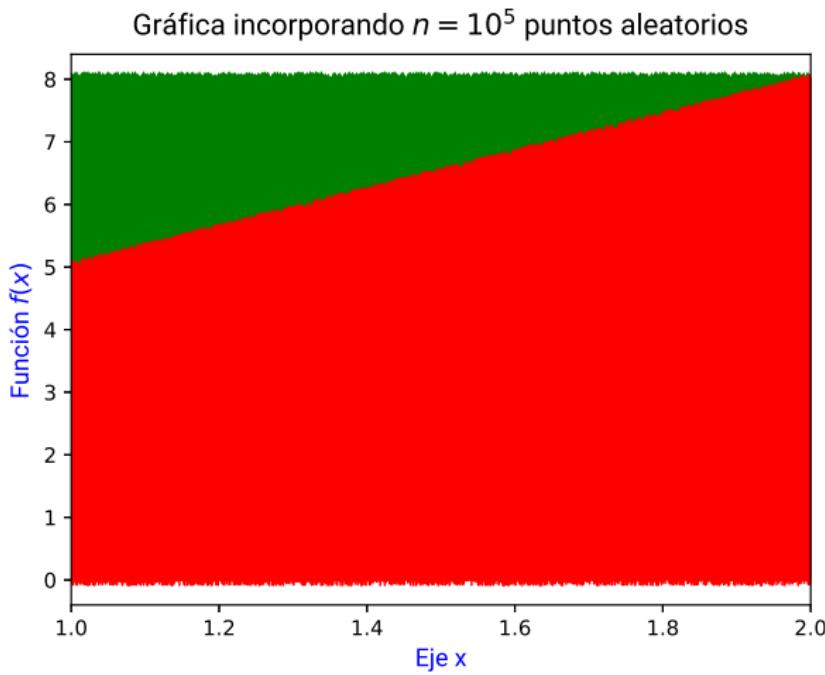


Figura 7: Podríamos pensar que ya tenemos el valor de la integral.

Gráfica al incluir puntos aleatorios

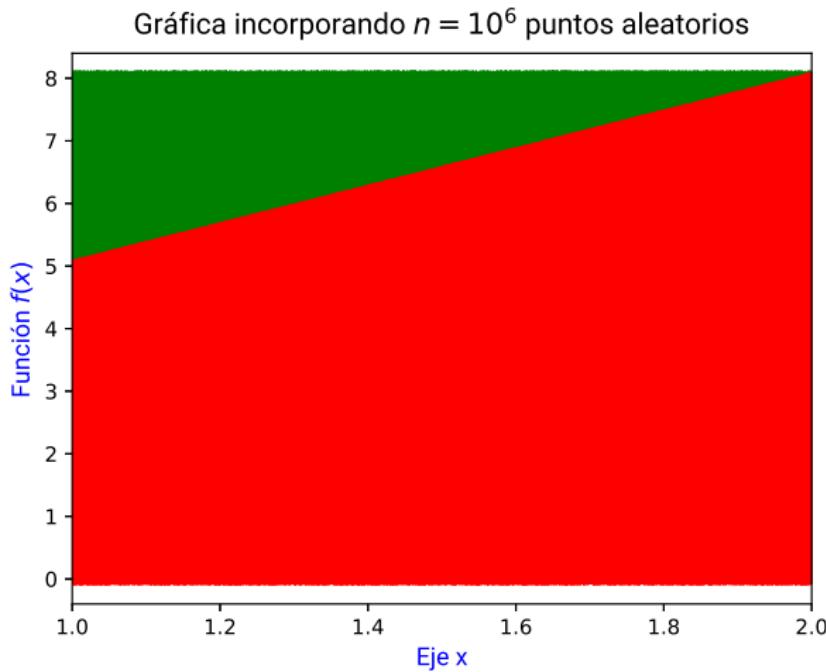


Figura 8: El resultado del cálculo de la integral es exacto.

Gráfica del error vs. número de muestras

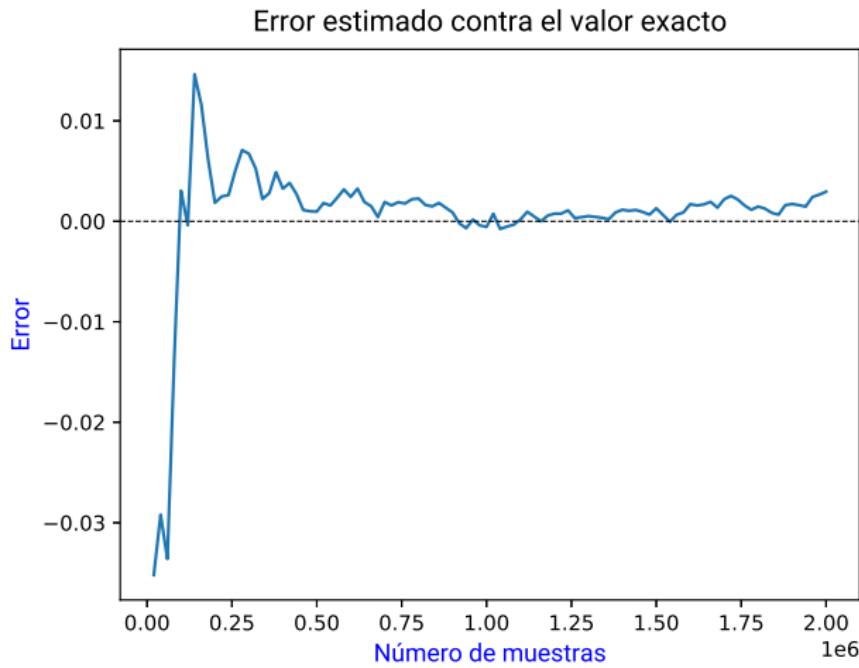


Figura 9: Comportamiento del error.