Práctica 0

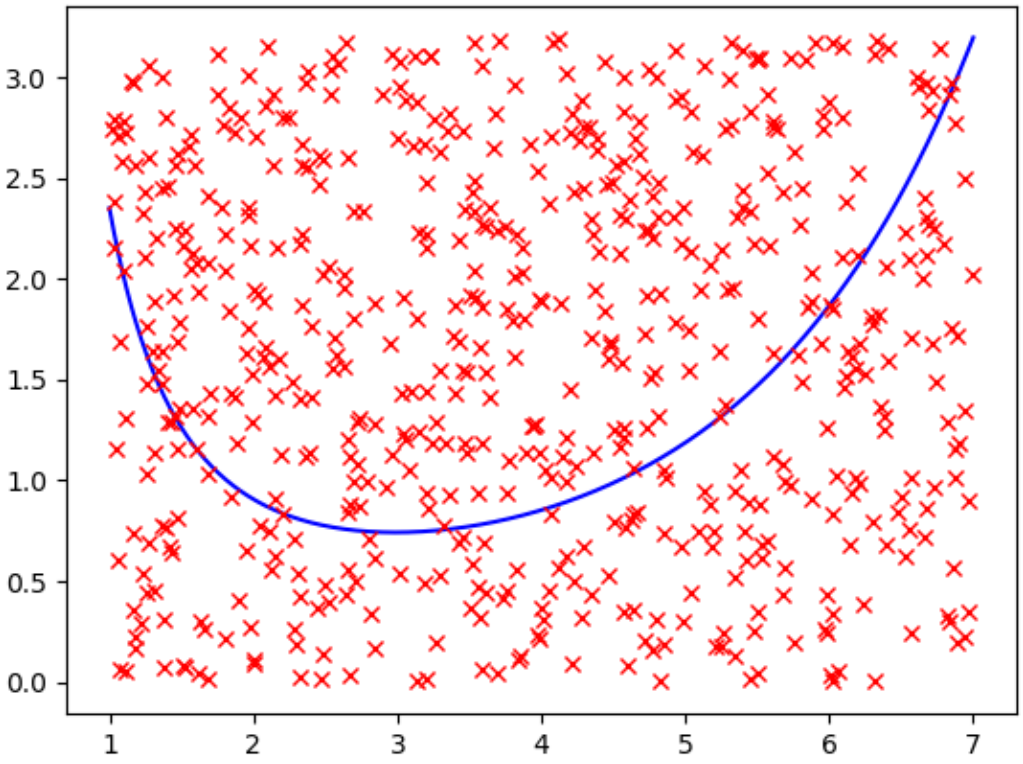
Alejandro Aizel Boto
Miguel Robledo Blanco

Índice

1. Descripción General

En esta práctica hemos desarrollado una implementación del método de Monte Carlo. Para ello hemos utilizado el lenguaje de programación Python junto a las librerías Numpy y Matplot que nos han facilitado bastante la resolución.

El método de Monte Carlo es un método numérico que nos permite de forma aproximada calcular la integral de una función en un intervalo, que se corresponde con el área bajo la curva. Primero calculamos el mayor valor de la función en dicho intervalo y "dibujamos" un cuadrado con dimensión `altura_máxima * (b - a)`. Ahora tenemos que ir "lanzando" dardos dentro de este cuadrado de forma aleatoria y contar el número de estos que han quedado por debajo de la curva. La relación entre los dardos que han quedado por debajo y el total de los que hemos lanzado coincide de forma aproximada con el área bajo la curva de la función.



El objetivo de la práctica es apreciar la diferencia en tiempo entre el uso de bucles frente al de operaciones de arrays (uno de los puntos fuertes de Python). Para ello hemos implementado la misma solución dos veces, una utilizando bucles, y otra utilizando operaciones de arrays.

2. Explicación de la Solución

Como hemos comentado anteriormente hemos creado dos funciones, no obstante ambas tienen los mismos parámetros:

- `fun` = la función que vamos a utilizar para el ejercicio.
- `a` = parte inferior del intervalo en el que la función es positiva.
- `b` = parte superior del intervalo en el que la función es positiva.
- `num_puntos` = el número de puntos que vamos a generar.

A continuación mostramos el código de las dos funciones:

```
def integra_mc_eff(fun, a, b, num_puntos=1000):
    tic = time.process_time()
    x = np.linspace(a, b, num_puntos)
    y = fun(x)

    x_random = np.random.uniform(low=a, high=b, size=int(num_puntos))
    y_random = np.random.uniform(low=0, high=y.max(), size=int(num_puntos))

    print("The area is: {} %".format((np.sum(y_random < fun(x_random)) /
num_puntos) * 100))

    toc = time.process_time()

    return 1000 * (toc - tic)
```

```
def integra_mc_in(fun, a, b, num_puntos=1000):
    tic = time.process_time()
    x = np.linspace(a, b, num_puntos)
    y = []

    for i in x:
        y += [fun(i)]

    x_random = np.random.uniform(low=a, high=b, size=int(num_puntos))
    y_random = np.random.uniform(low=0, high=max(y), size=int(num_puntos))

    under_graph_points = 0

    for i in range(int(num_puntos)):
        if y_random[i] < fun(x_random[i]):
            under_graph_points += 1

    print("The area is: {} %".format((under_graph_points / num_puntos) *
100))

    toc = time.process_time()

    return 1000 * (toc - tic)
```

Para controlar el tiempo en ambas funciones hemos utilizado la librería `time`. Cada función va a devolver el tiempo que ha tardado en hacer todas sus operaciones. Utilizamos una variable `tic` (para guardar el tiempo en el que comienza la ejecución de la función), y una variable `toc` (que guarda el tiempo en el que termina la ejecución). Simplemente tenemos que restar `toc - tic` para conocer el tiempo.

En ambas funciones creamos dos arrays. El primero lo creamos con `np.linspace`, que nos va a dar un array de `num_puntos` puntos entre `a` y `b`. El segundo, en el caso de la función eficiente, hacemos `y = fun(x)` que nos va a hallar el valor de la función en cada punto y lo va a guardar en y. En el caso de la solución ineficiente hemos usado un bucle para ir agregando el valor de la función en cada punto.

Ahora vamos a generar dos arrays de puntos aleatorios con dimensiones del cuadrado que hemos mencionado anteriormente (`altura_máxima * (b - a)`). Vamos a guardar las coordenadas de dichos puntos en `x_random` e `y_random`.

Una vez tengamos esto calculado solamente tenemos que quedarnos con los "dardos" que han caído por debajo de la gráfica. Lo que estamos haciendo en la eficiente es quedarnos con un array en los que cada punto de `y_random` es menor que el valor de la función de cada punto de `x_random`. Este array de booleanos nos va indicar en cada punto si el "dardo" queda por debajo o por encima (mediante `true` o `false`). Utilizando la operación `np.sum()` obtenemos el número de "trues" que hay, es decir el número de "dardos" que han quedado por debajo de la gráfica. Para la ineficiente vamos a ir recorriendo con un bucle los puntos y realizando la misma operación que la anterior para cada punto. Vamos guardando en una variable `under_graph_points` el número que quedan por debajo.

Por último hemos creado la función `hyper_like_function` para ver cómo funcionaba el paso de funciones por parámetro con una función definida por nosotros. Está basada en la función del seno hiperbólico en el que hemos sustituido el denominador por x^3. En resultados mostraremos la gráfica de tiempo de esta función y la del coseno.

```
def hyper_like_func(x):
    return (np.e ** x - np.e ** (-x)) / (x ** 3)
```

En el main() lo que hemos hecho primeramente es crear las tres variables que vamos a usar para llamar a cada función. Seguidamente hemos creado un array de 20 puntos distribuidos de forma uniforme desde 100 hasta 10^7. Este array es el que vamos a llamar en cada función indicando los puntos para ver la diferencia en tiempo. Lo que tarda cada función lo vamos a almacenar en otros dos arrays, uno para la eficiente y otro para la ineficiente.

```
def main():
    fun, a, b = np.cos, 1, 1.5

    points = np.linspace(100, 1000000, 20)

    y_efficient = []
    y_inefficient = []
```

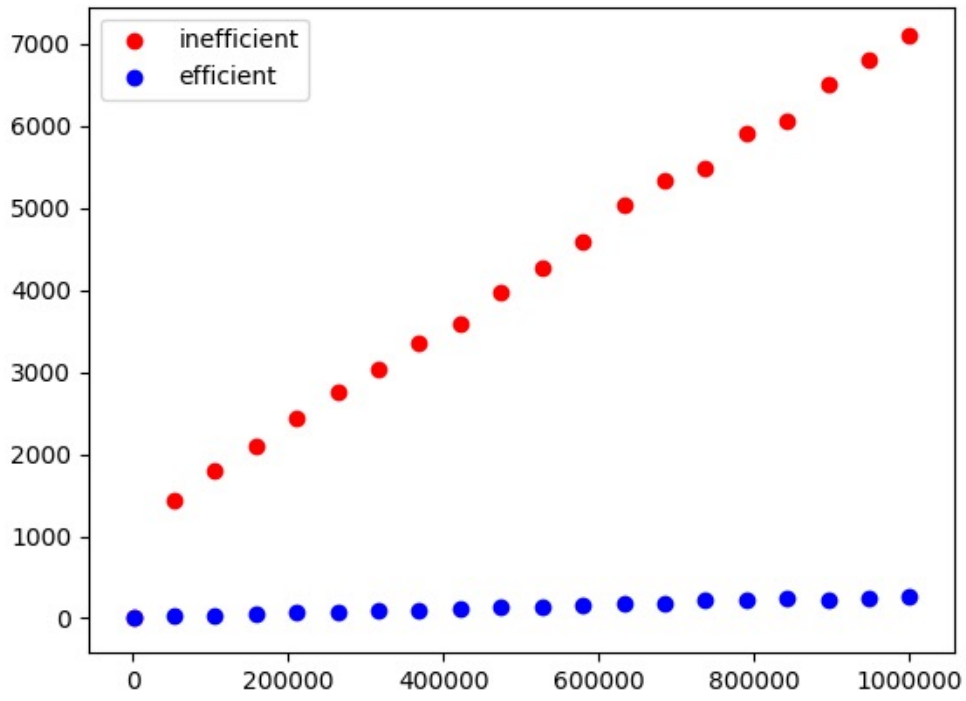
```
for i in points:
    y_efficient += [integra_mc_eff(fun, a, b, num_puntos=i)]
    y_inefficient += [integra_mc_in(fun, a, b, num_puntos=i)]

plt.figure()
plt.scatter(points, y_inefficient, c='red', label='inefficient')
plt.scatter(points, y_efficient, c='blue', label='efficient')
plt.legend()
plt.savefig('time_graph.png')

print("Done!")
```

Mediante un bucle vamos llamando a cada función y obteniendo el resultado. Por último, mediante la librería `pyplot`, guardamos en una imagen el resultado.

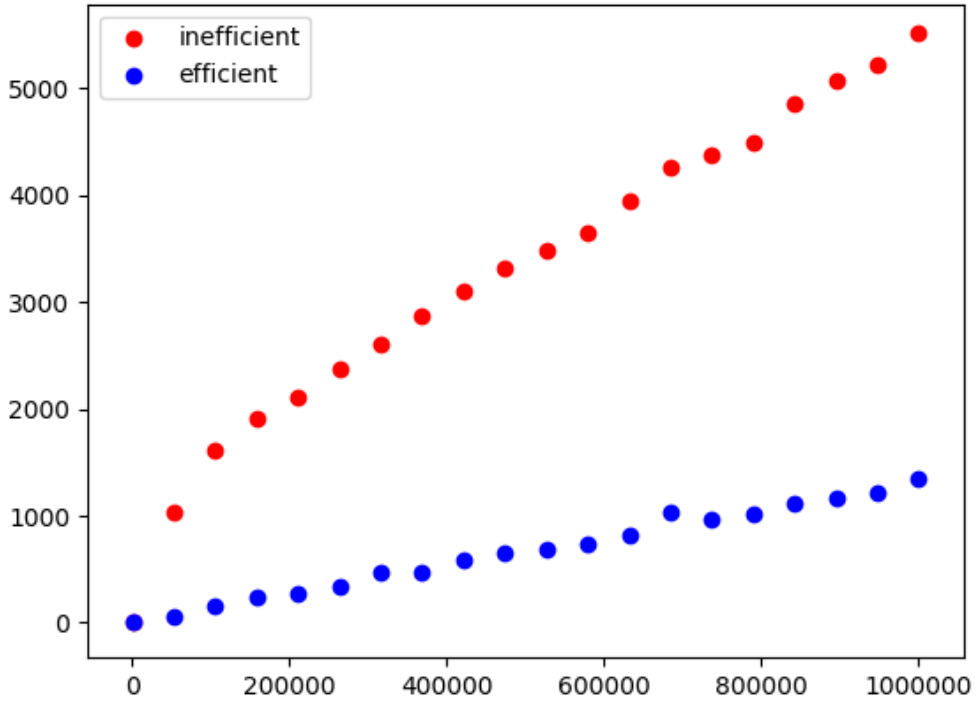
3. Resultados

El resultado que hemos obtenido era el esperado. El tiempo que ha tardado la función eficiente en ejecutarse ha sido considerablemente menor que la ineficiente, observando una gran diferencia en cuanto aumenta el número de puntos.

Hemos ejecutado el código tanto para la función coseno como para la que hemos creado nosotros. El resultado es el siguiente:



Función coseno de la librería numpy



Función hyper_like_func definida por nosotros

4. Conclusión

Para finalizar, ésta práctica nos ha servido para darnos cuenta de la importancia de usar operaciones de arrays en lugar de bucles por la disminución considerable en tiempo entre uno y otro. Hemos podido comprobar que cuanto más aumenta el número de datos de entrada el coste de las operaciones en caso de usar bucles puede llegar a ser muy alto. Además nos ha ayudado a familiarizarnos con el lenguaje de programación Python y las librerías Numpy, Pyplot y Time.

5. Código

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

def hyper_like_func(x):
    return (np.e ** x - np.e ** (-x)) / (x ** 3)

def integra_mc_eff(fun, a, b, num_puntos=1000):
    tic = time.process_time()
    x = np.linspace(a, b, num_puntos)
    y = fun(x)

    x_random = np.random.uniform(low=a, high=b, size=int(num_puntos))
    y_random = np.random.uniform(low=0, high=y.max(), size=int(num_puntos))
```

```
    print("The area is: {} %".format((np.sum(y_random < fun(x_random)) /
num_puntos) * 100))

    toc = time.process_time()

    return 1000 * (toc - tic)

def integra_mc_in(fun, a, b, num_puntos=1000):
    tic = time.process_time()
    x = np.linspace(a, b, num_puntos)
    y = []

    for i in x:
        y += [fun(i)]

    x_random = np.random.uniform(low=a, high=b, size=int(num_puntos))
    y_random = np.random.uniform(low=0, high=max(y), size=int(num_puntos))

    under_graph_points = 0

    for i in range(int(num_puntos)):
        if y_random[i] < fun(x_random[i]):
            under_graph_points += 1

    print("The area is: {} %".format((under_graph_points / num_puntos) *
100))

    toc = time.process_time()

    return 1000 * (toc - tic)

def main():
    fun, a, b = hyper_like_func, 1, 7

    points = np.linspace(100, 1000000, 20)

    y_efficient = []
    y_inefficient = []

    for i in points:
        y_efficient += [integra_mc_eff(fun, a, b, num_puntos=i)]
        y_inefficient += [integra_mc_in(fun, a, b, num_puntos=i)]

    plt.figure()
    plt.scatter(points, y_inefficient, c='red', label='inefficient')
    plt.scatter(points, y_efficient, c='blue', label='efficient')
    plt.legend()
    plt.savefig('time_graph.png')

    print("Done!")

main()
```