PRÁCTICA 0

MEMORIA

Aaron Reboredo Vázquez y Pablo Martín García

Objetivo:

El objetivo de esta práctica es calcular la integral de una función entre dos puntos ‘a’ y ‘b’ usando el método de Monte Carlo. Para ello calcularemos primero el máximo de la función. Con ello generamos un área rectangular con los valores de a, b y M que recoge el área de la función a calcular.

A continuación, generaremos puntos aleatorios dentro del área rectangular y veremos cuántos caen por debajo de la función, serán estos puntos los que nos permitan calcular el área. Tras esto, aplicaremos la fórmula de Monte Carlo:

$$\int_a^b \mathrm{f}\,(x)\,dx = \frac{N_{debajo}}{N_{totales}} * (b-a) * M$$

Otro de los objetivos de la práctica consistía en hacer la función de dos formas, utilizando bucles y utilizando los métodos de Numpy.

Resolución:

```python
import numpy as np
import random
from scipy.integrate import quad
import time
import matplotlib.pyplot as plt


# Definition of sin function
def sin_function(x):
    return np.sin(x)


# Definition of x2 function
def x2_function(x):
    y = x*2
    return y


#
```

```
def compare_times(fun, a, b):

    x_axis = np.linspace(10, 100000, 10)

    time_loops = []
    time_np_fast = []

    for _ in x_axis:
        time_loops += [integra_mc(fun, a, b, int(_))]
        time_np_fast += [integra_mc_vectorial(fun, a, b, int(_))]

    plt.figure()
    plt.scatter(x_axis, time_loops, c = 'red', label = 'loops' )
    plt.scatter(x_axis, time_np_fast, c = 'blue', label = 'np_methods' )
    plt.legend()
    plt.savefig('time.png')
    plt.show()


# returns the max element of an array, using loops
def function_max(fun, x_values_array):

    y_value  = fun(x_values_array[0])
    max_value =  y_value

    for i in range(len(x_values_array)):
        y_value = fun(x_values_array[i])
        if  y_value > max_value:
            max_value = y_value

    return max_value


# Returns an array with random points for the x-axis
def x_values(a,b, num_points):

    length = b - a

    if length < 1:
        num_of_divisions = num_points
    else:
        num_of_divisions = num_points * length

    x_values_array =  np.linspace(a, b, int(num_of_divisions))

    return x_values_array


# Returns the area under the function using monte carlo
```

```
def area_calculator(a,b,num_points,points_inside_area,function_maximum):
    integral = (points_inside_area/num_points)*(b-a)*function_maximum
    return integral


# Returns the number of points that are under the function
def points_behind_function_area(fun, a, b, num_points, function_maximum):

    points_inside_area = 0

    for j in range(num_points):

        x_value_area_point = np.random.uniform(a,b)
        y_value_area_point = np.random.uniform(0,function_maximum)

        if fun(x_value_area_point) > y_value_area_point:
            points_inside_area += 1


    return points_inside_area


# Integration using monte carlo and loops
def integra_mc(fun, a, b, num_points):

    tic = time.process_time()

    x_values_array = x_values(a, b, num_points)

    max_value = function_max(fun, x_values_array)

points_inside_area = points_behind_function_area(fun, a, b, num_points,
max_value)

    toc = time.process_time()
    fun_time =  1000 * (toc-tic)

    print("Area using loops : ", area_calculator(a, b, num_points,
points_inside_area, max_value))
    print("Time using loops : ", fun_time)

    return fun_time


# Integration using monte carlo with no loops
def integra_mc_vectorial(fun, a ,b , num_points = 10000):

    tic = time.process_time()
```

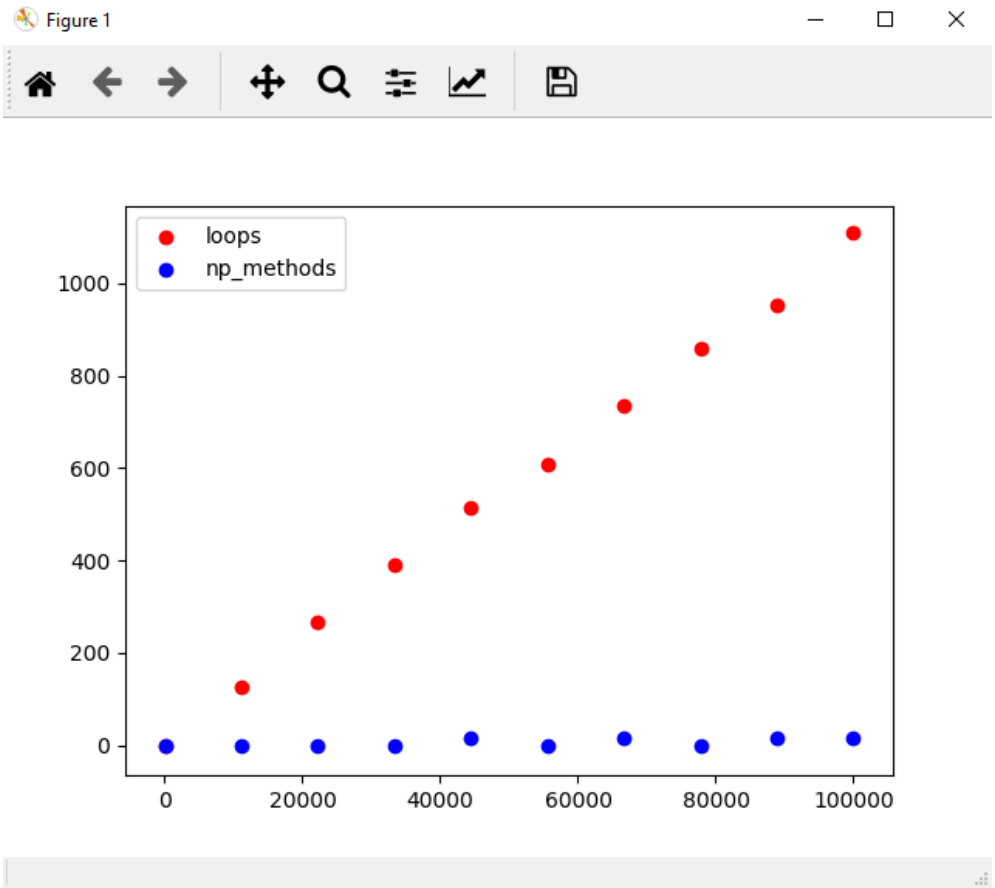
```
    x_values_array = x_values(a, b, num_points)
    y_values_function = fun(x_values_array)

    max_value = np.amax(y_values_function)

    x_random_values = np.random.uniform(a, b, num_points)
    y_random_values = np.random.uniform(0, max_value, num_points)

    y_random_values_function = fun(x_random_values)

    elements_within_area = y_random_values[y_random_values <
y_random_values_function] #el vector resultante contiene tantos elementos
como elementos del array sobre el que se aplica cumplan la condicion dada
    num_of_points_behind_fun = len(elements_within_area)

    toc = time.process_time()

    fun_time =  1000 * (toc-tic)

    print("Area using numpy vector methods : ", area_calculator(a, b,
num_points, num_of_points_behind_fun, max_value))
    print("Time using np ... : ", fun_time)

    return fun_time


compare_times(sin_function, 0, 3)
```

Hemos comprobado que efectivamente, el cálculo del área es correcto para funciones que **siempre son positivas** en el intervalo [a, b].

Para verificar nuestros resultados usamos la función scipy.integrate.quad que nos devuelve la integral exacta entre dos puntos.

Además, hemos comparado los tiempos que tardan en calcular la integral ambas funciones y nos hemos apoyado de gráficas comparativas utilizando diversas funciones para estudiar las diferencias utilizando como variable dependiente el tiempo e independiente el número de puntos aleatorios proyectados sobre el área rectangular.

Observando los resultados, nos damos cuenta de que el coste en tiempo utilizando bucles es mucho mayor que usando los métodos de Numpy, siendo casi despreciable en el segundo caso.



Gráfica para la función sin(x)

Observando las gráficas, hemos podido comprobar que el tiempo de ejecución incrementa cuando aumentamos el número de puntos aleatorios a proyectar en el área rectangular (variable independiente).

Teóricamente la precisión aumenta a la hora de calcular el área según incrementamos el número de puntos aleatorios (variable num_puntos). Sin embargo, hemos podido verificar que el coste en tiempo también aumenta.

Concluimos que un aumento en la precisión supone un aumento en el tiempo de ejecución que es mucho más pronunciado en el caso de utilizar bucles.