**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:

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

**Resolución:**

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

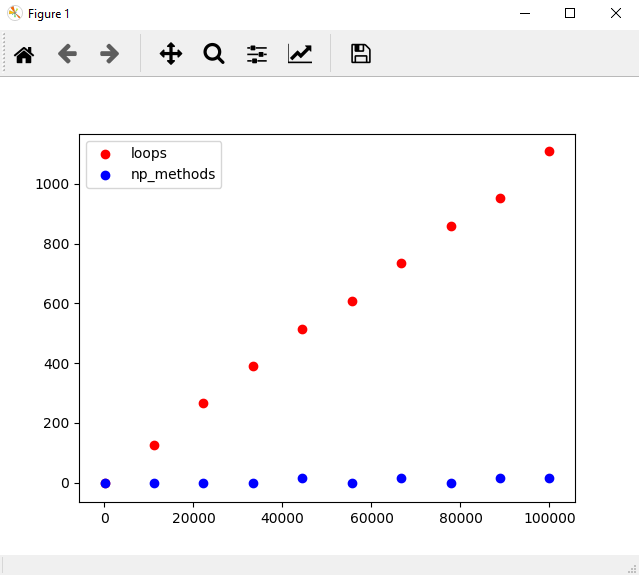
compare\_times(sin\_function, 0, 3)

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.