Práctica 0: Python

Rafael Herrera Troca
Rubén Ruperto Díaz

Aprendizaje Automático y Big Data
Doble Grado en Ingeniería Informática y Matemáticas

18 de febrero de 2020

1. Introducción

En esta primera práctica hemos implementado un algoritmo de integración de Monte Carlo, que aproxima el valor numérico de una integral en un intervalo, como una primera toma de contacto con Python en la asignatura. El algoritmo consiste en la elección de varios puntos aleatorios sobre un área que contiene a la gráfica de la función, de tal forma que al comprobar qué proporción de puntos quedan por encima o debajo de la gráfica se puede estimar el valor de la integral.

Hemos realizado dos versiones del algoritmo, una primera que trabaja iterativamente, es decir, calcula un punto en cada vuelta de un bucle, y otra que trabaja con todos los puntos simultáneamente haciendo uso de vectores de la librería *Numpy*. Después, hemos estudiado el rendimiento de cada aproximación comparando sus tiempos de cómputo en función del número de puntos que calculan.

Hemos programado el código en el entorno *Spyder* con Python 3.7.

2. Procedimiento

En nuestro código hemos calculado la integral de la función $f(x) = \sin x + 1$ en el intervalo $[0, 2\pi]$. La función es positiva en todo $\mathbb{R}$ y su valor máximo es 2, por lo que se puede aplicar el método de Monte Carlo sobre el rectángulo $[0, 2\pi] \times [0, 2]$, como se muestra en la figura 1.

Para el cálculo de la integral hemos definido dos funciones que aplican el método de Monte Carlo de formas distintas. En ambos casos, primero se calcula el máximo de $f(x)$ en el intervalo dado evaluándola en una partición de dicho intervalo lo suficientemente fina (en nuestro caso de 10000 puntos) que podemos obtener utilizando la función *numpy.linspace*.

Una vez se tiene el máximo se toma una cierta cantidad de puntos del espacio de forma aleatoria, donde dicha cantidad viene dada como parámetro de la función, y se cuenta cuántos de ellos quedan por debajo de la gráfica. Como obtenemos los puntos utilizando una distribución uniforme, la probabili-

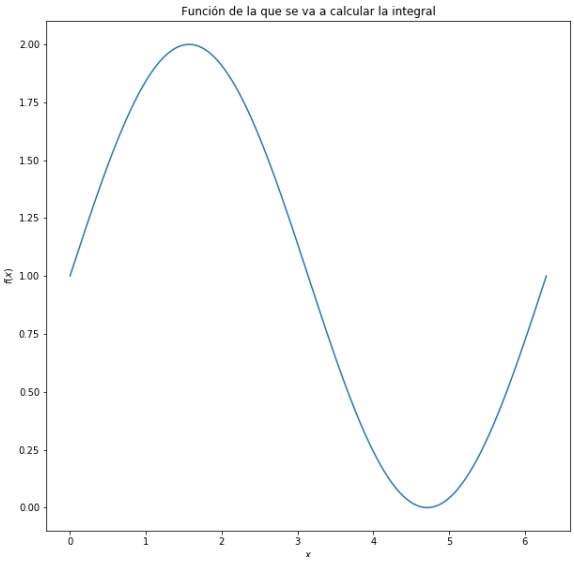

Figura 1: Función utilizada para aplicar el método de Monte Carlo

dad de que estén debajo de la gráfica es
la misma que la proporción del área bajo la gráfica con respecto al área total del rectángulo. Así pues, multiplicando dicha probabilidad, que se calcula como $\frac{N_{debajo}}{N_{total}}$, por el área total del rectángulo, se obtiene el área bajo la gráfica, esto es, la integral de f en el intervalo.

Las dos funciones que hemos definido difieren en la forma de trabajar con los puntos aleatorios. En la función *integra_mc_it*, el proceso se hace de forma iterativa. Se utiliza un bucle en el que se calculan uno a uno los puntos, eligiendo aleatoriamente cada una de sus dos coordenadas de modo que quede dentro del rectángulo. A continuación se evalúa la función en la coordenada x del punto y se comprueba si la coordenada y queda por encima o debajo de la gráfica. Sin embargo, en la función *integra_mc_vc*, se obtienen directamente los *arrays* con el número de puntos deseados haciendo uso de la función *random.rand* de *numpy*. Finalmente se comprueba cuántos puntos quedan bajo la gráfica de una forma muy compacta, en una sola línea de código.

Para comparar el rendimiento de ambas funciones, las ejecutamos dando varios valores entre 1000 y 1000000 al número de puntos. Guardamos el tiempo que tardan en cada ocasión y comparamos la respuesta que dan con el valor real calculado por *scipy.integrate.quad*.

3. Resultados

A continuación mostramos las gráficas comparativas con el tiempo que tardan las dos versiones del algoritmo en función del número de puntos calculados y la precisión obtenida por cada una de ellas.

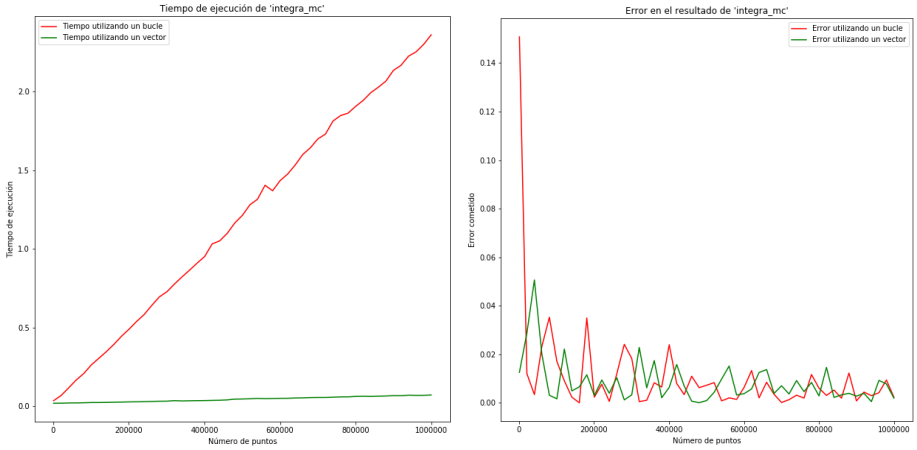

Figura 2: A la izquierda, tiempo de ejecución en segundos de la función en cada una de sus versiones. A la derecha, error cometido en el cálculo de la integral.

Observamos una diferencia sustancial de tiempo entre la forma iterativa y la vectorial. Mientras que con vectores el tiempo apenas crece al incrementar el número de puntos (de 0,019 s con 1000 elementos a 0,07 s con 1000000), con la forma iterativa el tiempo se incrementa mucho mas rápido (de 0,023 s a 2,44 s para los mismos casos). Esto supone un incremento en tiempo de 3,5 veces para la versión vectorial y de algo más de 100 veces para la iterativa, siendo el incremento del tamaño del vector de 1000 veces.

La precisión obtenida por ambas funciones es la misma en líneas generales, y es mayor cuanto más puntos se calculen. A partir de 10^5 el error está en las centésimas y de ahí hasta 10^6 la mejora es muy pequeña.

4. Conclusión

Esta práctica nos ha permitido comprobar hasta qué punto es eficiente la implementación de *arrays* de *numpy*. Nos ha sorprendido mucho que a pesar de agrandar en varios ordenes de magnitud el tamaño de los *arrays* el tiempo que se tarda en operar con ellos no ha aumentado significativamente.

5. Anexo: Código

```
'''
Práctica 0

Rubén Ruperto Díaz y Rafael Herrera Troca
'''

import time
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as scpint

# Función con la que trabajaremos f(x) = sin(x) + 1
def f(x):
    return (np.sin(x)+1)

# Integración por el método de Monte Carlo utilizando un bucle
def integra_mc_it(fun, a, b, num_puntos=10000):
    M = max(map(fun, np.linspace(a, b, 10000)))
    Ndeb = 0
    for i in range(num_puntos):
        x = (b-a) * np.random.rand() + a
        y = M * np.random.rand()
        if fun(x) > y:
            Ndeb += 1
    return (Ndeb / num_puntos) * (b - a) * M

# Integración por el método de Monte Carlo utilizando arrays de
    numpy
def integra_mc_vc(fun, a, b, num_puntos=10000):
    M = max(map(fun, np.linspace(a, b, 10000)))
    x = (b-a) * np.random.rand(1,num_puntos) + a
    y = M * np.random.rand(1,num_puntos)
    Ndeb = np.count_nonzero(fun(x) > y)
    return (Ndeb / num_puntos) * (b - a) * M

# Representamos la función que vamos a integrar
plt.figure(figsize=(10,10))
plt.plot(np.linspace(0,2*np.pi,100000),list(map(f,np.linspace(0,2*
    np.pi,100000))))
plt.title("Función de la que se va a calcular la integral")
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.savefig("func.png")
plt.show()

# Calculamos en un bucle el tiempo de ejecución para cada una de
    las opciones
# y el error cometido, haciéndolo cada vez con más precisión
times = ([],[])
error = ([],[])
res = scpint.quad(f,0,2*np.pi)[0]
for n in np.linspace(1000,1000000,51):
```

```
    time1 = time.process_time()
    res1 = integra_mc_it(f,0,2*np.pi, int(n))
    time2 = time.process_time()
    res2 = integra_mc_vc(f,0,2*np.pi, int(n))
    time3 = time.process_time()
    times[0].append(time2 - time1)
    times[1].append(time3 - time2)
    error[0].append(abs(res - res1))
    error[1].append(abs(res - res2))

# Representamos el tiempo que tardan las dos funciones en ejecutarse según la
# precisión que utilicemos
plt.figure(figsize=(10,10))
plt.plot(np.linspace(1000,1000000,51), times[0], 'r', label="Tiempo utilizando un bucle")
plt.plot(np.linspace(1000,1000000,51), times[1], 'g', label="Tiempo utilizando un vector")
plt.title("Tiempo de ejecución de 'integra_mc'")
plt.xlabel("Número de puntos")
plt.ylabel("Tiempo de ejecución")
plt.legend(loc="upper left")
plt.savefig("times.png")
plt.show()
```