Vectorizacion

Guillermo Ovejero
Fernando Bellot

18 de febrero de 2020

1. Práctica 1 Vectorización

1.1. Calculo de integrales mediante el método de Monte Carlo

Importamos las librerías que nos harán falta:

- matplotlib para hacer gráficas
- numpy para generar números aleatorios y operaciones con arrays
- time para calcular el tiempo de cómputo
- scipy para comprobar el área de una función

Para calcular el área de una función compleja de integrar usaremos este método que se basa en la probabilidad para aproximar el calculo de una integral Se usan dos métodos diferentes para calcular el numero de puntos que caen debajo de la función

- Bucle, calculando 1 a 1 todos los puntos que están debajo
- Vectores, calculando la longitud del vector de los elementos por debajo de la función

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import time
from scipy import integrate
```

[2]:
```
#return -(x)**2+100 #puede ser cualquier funcion ej:ln(x)+3x+x^2

f = lambda x : x**2

def plot_func(fun,a,b):
    """plots the function fun from point a to b"""
    x = np.linspace(a,b,50)
    y = fun(x)
    plt.plot(x,y)
```

1.2. Funciones usadas para el calculo de puntos debajo de la función

La función loop itera sobre los n puntos y comprueba si están por debajo de la función.
La función vector simplemente devuelve la longitud de el array el cual contiene los puntos que están por debajo.
Para 1 millón de puntos el tiempo de calculo es de unas 10 veces mayor en la función del bucle

- Vector: media de tiempo con 1 millón de puntos es de 93ms
- Loop : media de tiempo con 1 millón de puntos es de 953ms

[3]:
```
def vector(fun,a,b,num_puntos,x_rand,y_rand): #vector method
    return np.count_nonzero(y_rand < fun(x_rand))

def loop(fun,a,b,num_puntos,x_rand,y_rand): #loop method
    y_fun = fun(x_rand)
    debajo = 0
    for i in range(num_puntos):
        if y_rand[i] < y_fun[i]:
            debajo += 1
    return debajo
```

[4]:
```
def integra_mc(fun, a, b, num_puntos=10000, print_out=True, method=vector):
    VAL = 10000
    x = np.linspace(a,b,VAL)
    y = np.array(fun(x))
    tic = time.process_time()
    x_rand = np.random.uniform(a,b,num_puntos)
    y_rand = np.random.uniform(min(y),max(y),num_puntos)
    debajo = method(fun,a,b,num_puntos,x_rand,y_rand)
    toc = time.process_time()
    if print_out:
        print("Tiempo: {}".format(1000*(toc-tic)))
        print("Nº Debajo: {}".format(debajo))
        print("Nº Total: {}".format(num_puntos))
        print("(b-a)M = {}".format((b-a)*max(y)))
        print("Area Vector: {}".format((debajo/num_puntos)*(b-a)*max(y)))
        plt.scatter(x_rand,y_rand,c='red',marker='x', alpha=0.5)
    return (debajo/num_puntos)*(b-a)*max(y)
```

1.3. Prueba del metodo montecarlo

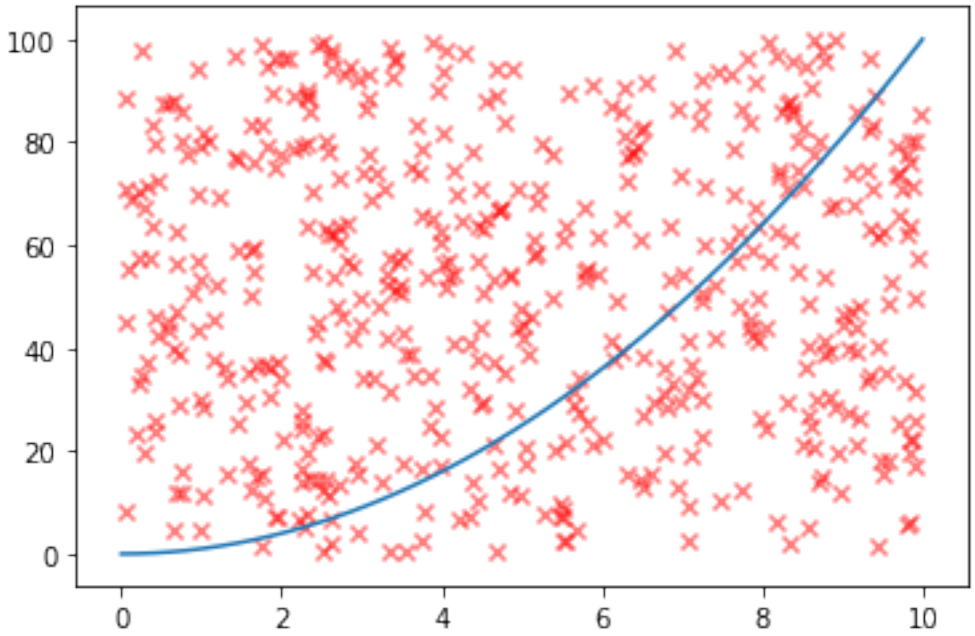
Imprimiendo todos los valores obtenidos.
Gráfica de puntos aleatorios y de la función.

[5]:
```
func = f
a = 0
b = 10
puntos = 500

plot_func(func,a,b)
mc_area = integra_mc(func,a,b,num_puntos=puntos)
print("\nArea de scipy: {}".format(integrate.quad(func,a,b)[0]))
print("Error: {}".format(abs(mc_area - integrate.quad(func,a,b)[0])))
plt.show()
```

```
Tiempo: 15.625
Nº Debajo: 159
Nº Total: 500
(b-a)M = 1000.0
Area Vector: 318.0

Area de scipy: 333.33333333333326
Error: 15.333333333333258
```



[6]:
```
def integra_mc_time(fun, a, b, num_puntos=10000, method=vector):
    """Devuelve una tupla de (area,tiempo)"""
    VAL = 1000
    x = np.linspace(a,b,VAL)
    y = np.array(fun(x))
    tic = time.process_time()
    x_rand = np.random.uniform(a,b,num_puntos)
    y_rand = np.random.uniform(min(y),max(y),num_puntos)
    debajo = method(fun,a,b,num_puntos,x_rand,y_rand)
    toc = time.process_time()
    return (((debajo/num_puntos)*(b-a)*max(y)),1000*(toc-tic))

def compara_tiempos(fun,a,b,puntos_min=100,puntos_max=1000000):
    sizes = np.linspace(puntos_min,puntos_max,20) #100000000
    vector_time = []
    vector_area = []
    loop_time = []
    for size in sizes:
        #Devuelven una tupla de (area,tiempo)
        v = integra_mc_time(fun,a,b,int(size),method=vector)
        vector_time.append(v[1])
        vector_area.append(v[0])
        loop_time.append(integra_mc_time(fun,a,b,int(size),method=loop)[1])
    actual_area = integrate.quad(fun,a,b)[0]
    return (sizes, loop_time, vector_time, vector_area, actual_area)

#result -> sizes, loop_time, vector_time, vector_area, actual_area
#result -> 0,     1,         2,           3,           4

def plot_dif_time(result):
    plt.figure()
    plt.scatter(result[0],result[1], c='red')
    plt.plot(result[0],result[1], c='red', label='bucle')
    plt.scatter(result[0], result[2], c='blue')
    plt.plot(result[0], result[2], c='blue', label='vector')
    plt.legend()
    plt.show()
def plot_vector_time(result):
    plt.scatter(result[0], result[2], c='blue', label='vector')
    plt.plot(result[0], result[2], c='blue', label='vector')
    plt.show()

def plot_areas(result):
    #quitamos el primero para que no desvie demasiado el eje del error
    plt.scatter(result[0][1:],result[3][1:], c='blue', label='results vector')
    plt.plot([min(result[0]),max(result[0])],[result[4],result[4]], c='green',␣
 ↪label='actual')
```

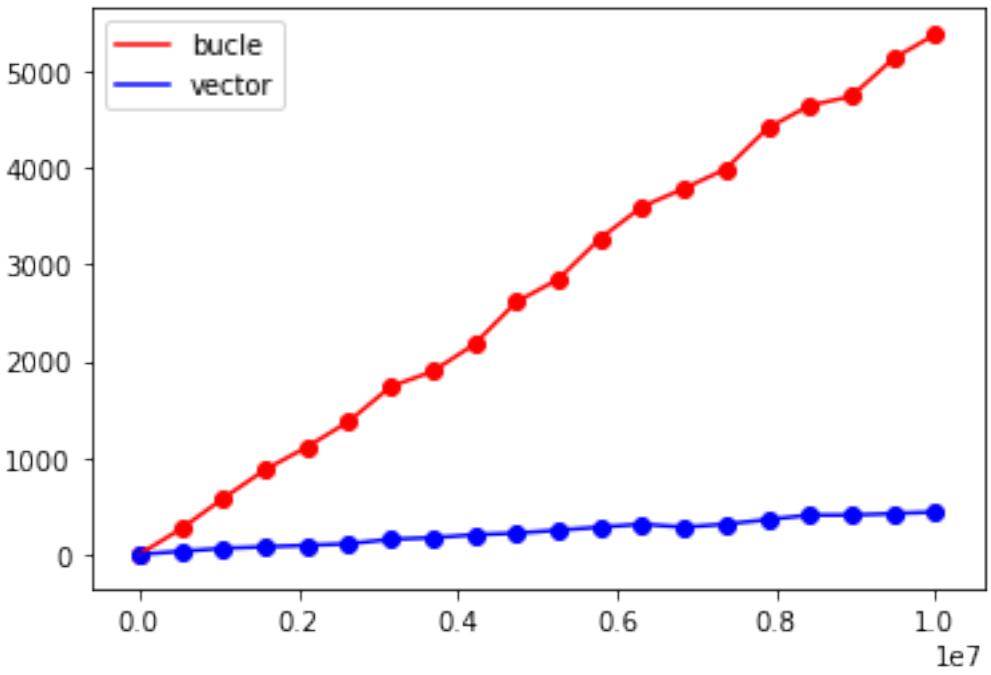
```
    plt.legend()
    plt.show()
```

1.4. Gráfica de tiempos

[7]:
```
result = compara_tiempos(f,a,b,puntos_min=100,puntos_max=10000000)
```

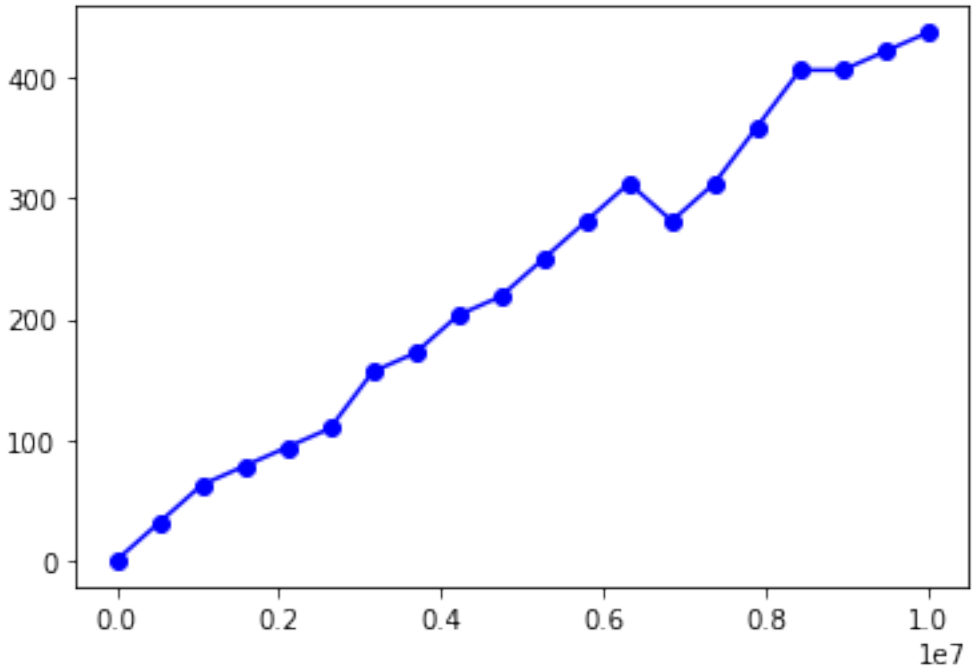
Se puede observar que ambas gráficas crecen de forma lineal, pero sin embargo la del vector (linea azul) lo hace con menor pendiente

[8]:
```
plot_dif_time(result)
```



En esta gráfica vemos mas detalladamente como el tiempo del vector aumenta cuantos mas puntos añadamos Pero vemos que tienen una diferencia menor entre min y max comparada con la del bucle

[9]:
```
plot_vector_time(result)
```



Como podemos ver aquí, el error, al aumentar el numero de puntos, disminuye.
Para que el método sea preciso necesitamos aumentar el numero de puntos incluso mas, para ello es recomendable usar el método de los vectores pues es unas 10 veces mas rápido.

[10]:
```
plot_areas(result)
```