

Memoria Practica 0

October 4, 2018

Adrián Alcántara Delgado y Jose María Monreal González

1 Memoria Práctica 0 - MonteCarlo

Para esta práctica 0 hemos dividido el trabajo en dos partes: Adrián se ha encargado de la parte de llevar a cabo el algoritmo mediante bucles, mientras que Jose María se ha encargado de la parte de vectorización.

A continuación se muestra el ejercicio completo con la solución. Vamos a explicar qué hace cada parte del código empezando por la parte de vectorización.

```
In [13]: import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import time

def createFunction(fun, a, b, num_puntos):
    M = vX = np.linspace(a, b, num_puntos)
    Y = fun(M)
    return M, Y;

def createPoints(fun, a, b, M, Y):
    #Ajustamos la altura de la representación de los puntos en Y
    #para que no se salgan de los valores de la gráfica
    alt = max(fun(M))
    size = len(M)
    plt.xlim(a-0.1, b+0.1)
    plt.ylim(-0.1, alt+0.1)
    puntos = np.random.uniform(low=0.0, high=alt, size=(size))
    valor = monteCarlo(Y, puntos, a, b, alt)
    return valor, puntos;

def monteCarlo(Y, puntos, a, b, alt):
    correctos = np.sum(Y < puntos)
    print (correctos)
    return (correctos/(2*len(puntos)))*(b-a)*alt
```

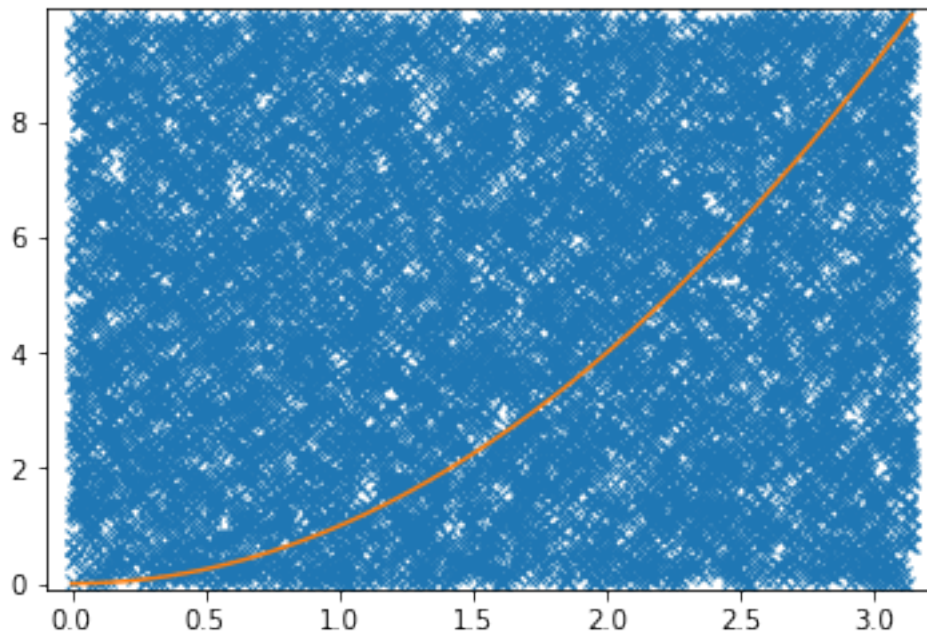
```
def integra_mc(fun, a, b, num_puntos=10000):
    tic = time.process_time()
    M, Y = createFunction(fun, a, b, num_puntos)
    valor, puntos = createPoints(fun, a, b, M, Y)
    toc = time.process_time()
    print("Área: " + str(valor))
    print("Tiempo: " + str(1000*(toc-tic)))
    #Pintamos la función por último sino se queda detrás de
    #todos los puntos
    plt.plot(M, puntos, 'x')
    plt.plot(M, Y)
    plt.show()

integra_mc(np.square, 0, np.pi)
```

6617

Área: 10.258426639677193

Tiempo: 15.625



Ese es todo el código perteneciente a la parte de vectorización. Aunque los métodos auxiliares se repiten en ambas versiones, voy a explicar un poco cada uno y por último los resultados obtenidos.

```
In [11]: def createFunction(fun, a, b, num_puntos):
    M = vX = np.linspace(a, b, num_puntos)
    Y = fun(M)
    return M, Y;
```

La función "createFunction" implementa simplemente la función mediante el método de Monte Carlo. Le pasamos los parámetros necesarios (a, b) y el número de puntos para dividir el espacio de puntos entre esos dos valores. Con la función "linspace" de numpy creamos la función en el intervalo que ahí le pasamos.

Devolvemos los valores derivados de la función para utilizarlos después en el cálculo de los puntos y ver cuántos quedan por debajo o encima de ella.

```
In [12]: def createPoints(fun, a, b, M, Y):
        #Ajustamos la altura de la representación de los puntos en Y
        #linspace para que no se salgan de los valores de la gráfica
        alt = max(fun(M))
        size = len(M)
        plt.xlim(a-0.1, b+0.1)
        plt.ylim(-0.1, alt+0.1)
        puntos = np.random.uniform(low=0.0, high=alt, size=(size))
        valor = monteCarlo(Y, puntos, a, b, alt)
        return valor, puntos;
```

Este es el método que más dista de la otra versión, ya que forma los puntos aleatorios de manera aleatoria mediante vectorización.

Lo que hacemos aquí es sacar el máximo de la función en altura y anchura para representar los puntos que saquemos en esas medidas. A continuación sacamos los num_puntos aleatoriamente utilizando la función numpy.random.uniform.

Por último, dentro de esta misma función llamamos al método MonteCarlo que nos dará el porcentaje de puntos que se encuentran por encima de la función M.

```
In [13]: def monteCarlo(Y, puntos, a, b, alt):
        correctos = np.sum(Y < puntos)
        return (correctos/(2*len(puntos)))*(b-a)*alt
```

"monteCarlo", por su parte, hace exactamente lo ya explicado: suma todos los puntos que cumplen ($Y < \text{puntos}$) y devuelve ese número entre el total en porcentaje.

```
In [14]: def integra_mc(fun, a, b, num_puntos=10000):
        tic = time.process_time()
        M, Y = createFunction(fun, a, b, num_puntos)
        valor, puntos = createPoints(fun, a, b, M, Y)
        toc = time.process_time()
        print("Área: " + str(valor))
        print("Tiempo: " + str(1000*(toc-tic)))
        #Pintamos la función por último sino se queda detrás de
        #todos los puntos
        plt.plot(M, puntos, 'x')
        plt.plot(M, Y)
        plt.show()
```

Por último, la función "integra_mc1" crea la función de MonteCarlo y forma todos los puntos aleatorios. Finalmente, muestra el Porcentaje, pinta los puntos y finalmente la función (está hecho en este orden ya que al ser 10.000 puntos la función no se visualiza debajo de todos los

puntos). Diferenciamos "integra_mc1" de "integra_mc2" aunque hagan prácticamente lo mismo para calcular el tiempo de ejecución.

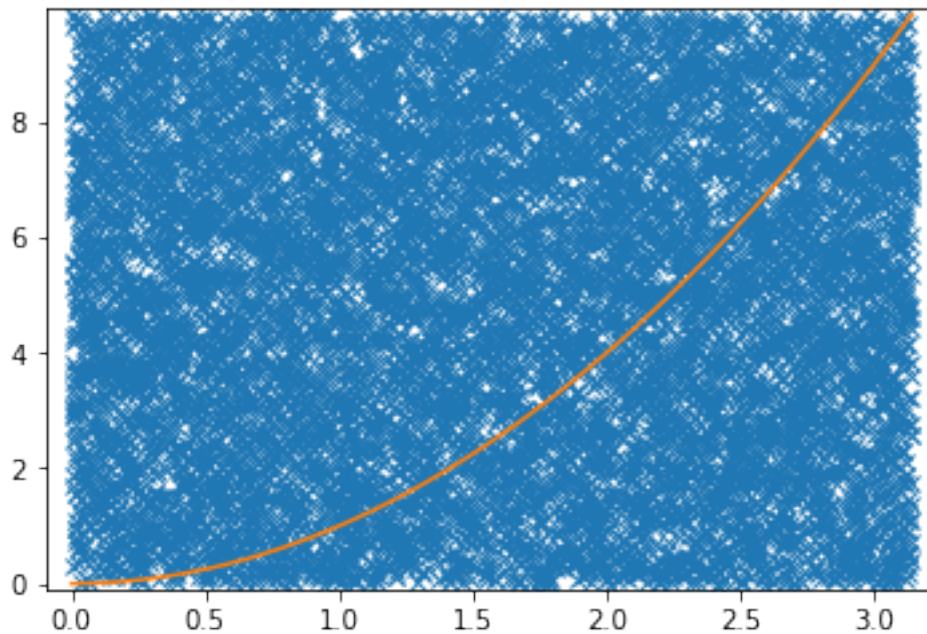
Al terminar, obtenemos la siguiente gráfica:

```
In [34]: integra_mc(np.square, 0, np.pi)
```

6650

Área: 10.30958699619969

Tiempo: 15.625



Donde podemos observar la parte naranja de puntos y la línea verde representando la función. El porcentaje de puntos que se encuentran por encima de la función es aproximadamente 36%.

El código hecho a partir de bucles tendría esta forma. A continuación se explicarán los pequeños cambios en el código y las grandes diferencias de tiempo respecto a la versión vectorizada

```
In [71]: i = 0
         for x in puntosX:
             if(puntosY[i] < fun(puntosX[i])):
                 ++correctos
             ++i
```

NameError

Traceback (most recent call last)

```
<ipython-input-71-c14294b9f83d> in <module>()
    1 i = 0
----> 2 for x in puntosX:
    3     if(puntosY[i] < fun(puntosX[i])):
    4         ++correctos
    5     ++i
```

NameError: name 'puntosX' is not defined

```
In [14]: def createFunction(fun, a, b, num_puntos):
        vX = np.linspace(a, b, num_puntos)
        vY = [fun(i) for i in vX]
        return vX, vY

def monteCarlo(fun, Y, puntos, a, b, alt):
    correctos = 0
    cont = 0

    while(cont < len(puntos)):
        encontrado = False
        index_Y = 0
        while(index_Y < len(Y) and encontrado == False):
            if(Y[index_Y] < puntos[cont]):
                correctos = correctos + 1
                encontrado = True
            index_Y = index_Y + 1
        cont = cont + 1

    return (correctos/(2*len(puntos)))*(b-a)*alt

def createPoints(fun, a, b, M, Y):
    #Ajustamos la altura de la representación de los puntos en Y
    #para que no se salgan de los valores de la gráfica
    alt = max(fun(M))
    size = len(M)
    plt.xlim(a-0.1, b+0.1)
    plt.ylim(-0.1, alt+0.1)
    puntos = np.random.uniform(low=0.0, high=alt, size=(size))
    #puntosX = np.random.uniform(a, b, size=(size))
    valor = monteCarlo(fun, Y, puntos, a, b, alt)
    #puntos = (puntosX, puntosY)
    return valor, puntos;

def integra_mc(fun, a, b, num_puntos=10000):
```

```

tic = time.process_time()
M, Y = createFunction(fun, a, b, num_puntos)
valor, puntos = createPoints(fun, a, b, M, Y)
toc = time.process_time()
print("Área: " + str(valor))
print("Tiempo: " + str(1000*(toc-tic)))
#Pintamos la función por último sino se queda detrás de
#todos los puntos
plt.plot(M, puntos, 'x')
plt.plot(M, Y)
plt.show()

```

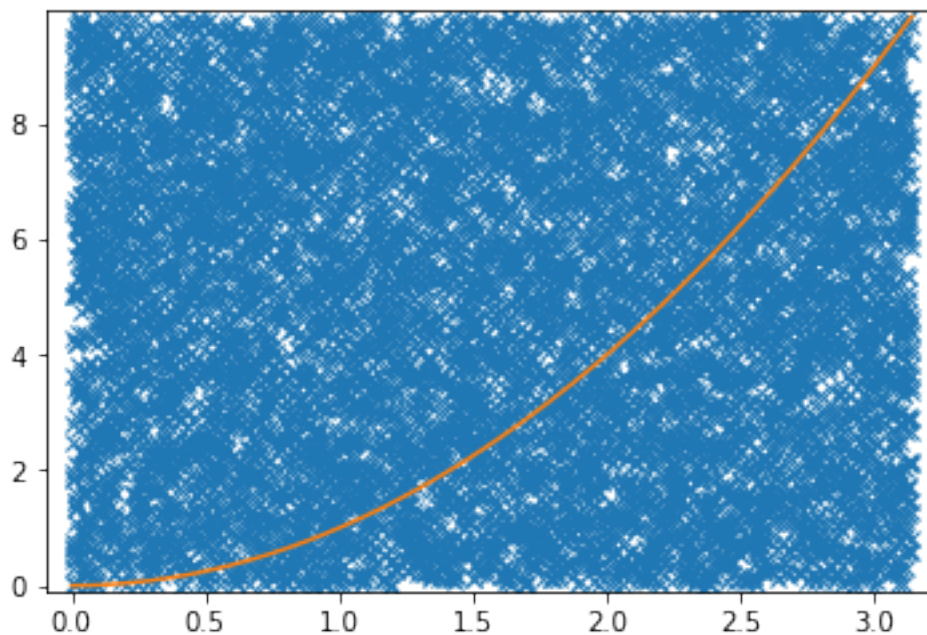
```

integra_mc(np.square, 0, np.pi)

```

Área: 15.503138340149908

Tiempo: 62.5



La única diferencia respecto a la otra implementación es la función MonteCarlo. Al recorrer los dos arrays con bucles y calcular el área de la integral se tarda un tiempo mucho mayor

```

In [74]: from scipy import integrate

```

```

scipy.integrate.quad(np.square, 0, np.pi)

```

```

Out[74]: (10.335425560099939, 1.1474627426122306e-13)

```