Práctica 0

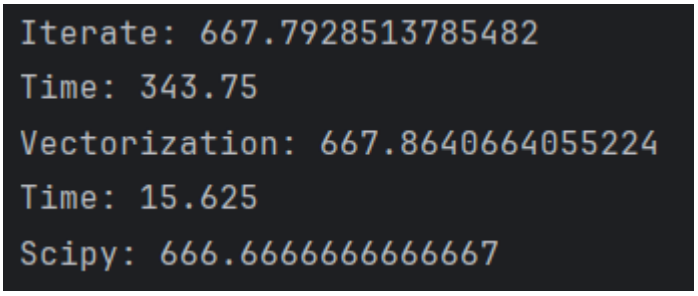
Escribimos dos funciones, una usando la iteración (integra_mc_it()) y otra usando vectorización (integra_mc_vec()). Ejecutamos las dos funciones por separado, calculamos sus tiempos de ejecución y luego los comparamos.

Implementación

```
import random
import time

import scipy
import numpy as np
import matplotlib.pyplot as plt


def cuadrado(x):
    return x * x


def integra_mc_it(fun, a, b, num_puntos=10000):
    puntos = []
    puntos_func = []
    puntos_random = []
    resultados = []

    # Generar coordenadas aleatorias del eje x
    for i in range(num_puntos):
        puntos.append(random.random() * (b - a) + a)

    # Generar puntos aleatorios ubicados en la función
    for i in range(num_puntos):
        puntos_func.append(fun(puntos[i]))

    # Calcule el valor máximo de esta función
    maximo = max(puntos_func)

    # Generar puntos aleatorios
    for i in range(num_puntos):
        puntos_random.append(random.random() * maximo)

    # Compare cada punto generado aleatoriamente con el punto corresponde a la
función
    for i in range(num_puntos):
        resultados.append(puntos_random[i] < puntos_func[i])

    # Cuente el número de puntos aleatorios ubicados debajo de la función
    num_bajo = sum(resultados)

    # Genera una figura
    #generate_figure(a,b,puntos,puntos_func,puntos_random)
    return num_bajo / num_puntos * (b - a) * maximo
```

```python
def integra_mc_vec(fun, a, b, num_puntos=10000):
    # Generar coordenadas aleatorias del eje x
    puntos = np.random.uniform(a, b, num_puntos)

    # Generar puntos aleatorios ubicados en la función
    puntos_func = fun(puntos)

    # Calcule el valor máximo de esta función
    maximo = np.max(puntos_func)

    # Generar puntos aleatorios
    puntos_random = np.random.uniform(0, maximo, num_puntos)

    # Compare cada punto generado aleatoriamente con el punto corresponde a la
    # función y cuente el número de puntos aleatorios ubicados debajo de la
función
    num_bajo = np.sum(puntos_random < puntos_func)

    # Genera una figura
    #generate_figure(a,b,puntos,puntos_func,puntos_random)
    return num_bajo / num_puntos * (b - a) * maximo


def generate_figure(a, b, puntos, puntos_func, puntos_random):
    plt.figure()
    plt.scatter(puntos, puntos_func, s=1, c='blue', marker='x', label='Puntos
Funciones')
    plt.scatter(puntos, puntos_random, s=1, c='red', marker='x', label='Puntos
Randomicos')
    plt.legend()
    plt.show()


def main():
    tic = time.process_time()
    print("Iterate: " + str(integra_mc_it(cuadrado, -10, 10)))
    toc = time.process_time()
    print("Time: " + str(1000 * (toc - tic)))

    tic = time.process_time()
    print("Vectorization: " + str(integra_mc_vec(cuadrado, -10, 10)))
    toc = time.process_time()
    print("Time: " + str(1000 * (toc - tic)))

    print("Scipy: " + str(scipy.integrate.quad(cuadrado, -10, 10)[0]))


if __name__ == '__main__':
    main()
```

Captura de ejecución



Resultado

Como podemos ver, se necesita mucho más tiempo para calcular usando iteración que usando vectorización, la librería numpy nos ahorra mucho tiempo.