Entrega 0: iteración vs vectorización

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 0 y la diferencia entre vectorización e iteración.

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1. Para ver las diferencias de eficiencia entre vectorización e iteración usaremos el cálculo de la integral por el método de montecarlo.

```python
from io import TextIOWrapper  # abrir y/o crear archivos de datos
import numpy as np  # vectorizaciones
from math import sin, pi  # para la funcion seno y su intervalo
import matplotlib.pyplot as plt  # para hacer los distintos graficos
import random  # para generar puntos aleatorios en las funciones iterativas
import time  # para medir los tiempos de ejecucian
import csv  # para guardar los datos de manera ordenada
```

Figura 0.1: Código de las bibliotecas usadas

1. Iteración

En la parte iterativa del problema usaremos las funciones estándar de bucle de python y listas intensionales.

La función calc_max_func (1.1) nos da el máximo valor que alcanza la función haciendo un muestreo uniforme del intervalo $[a, b]$, este máximo será el que usaremos para generar los puntos aleatorios para el algoritmo de montecarlo. Tras conseguir el máximo, la función integra_mc_iter (1.2) genera los puntos aleatorios y compara con la función para ver si el punto está por encima o por debajo. Luego usamos la fórmula $\frac{num\ debajo}{num\ total}(b-a)M$ para calcular el area aproximada de la integral.

Podemos ver que este método es bastante lento cuando empezamos a tener un mayor número de puntos.

```python
def calc_max_func(fun: float, a: float, b: float, num_puntos: int = 1000) -> float:
    """Calculates the maximum y value of a function using random points in the x axis

    Args:
        fun (float): function to calculate
        a (float): lower bound of the interval
        b (float): upper bound of the interval
        num_puntos (int, optional): number of random points generated. Defaults to 1000.

    Returns:
        float: max value of the y axis
    """
    func_steps: float = (b - a) / num_puntos
    # lo ponemos como el minimo ya que el enunciado dice que usemos funciones con resultado positivo
    max_func: float = 0

    points: list[float] = [round(random.uniform(a, b+func_steps), num_puntos)
                           for _ in range(0, num_puntos)]
    for i in points:
        max_func = max(max_func, fun(i))
    return max_func
```

Figura 1.1: Código de la función calc_max_func

```python
def integra_mc_iter(fun: float, a: float, b: float, num_puntos: int = 1000) -> float:
    """Calculates the integral of a function by the montecarlo method. Iterative way.

    Args:
        fun (float): function to integrate
        a (float): lower bound of the interval
        b (float): upper bound of the interval
        num_puntos (int, optional): number of random points to use. Defaults to 1000.

    Returns:
        float: Value of the calculated integral
    """
    max_func: float = calc_max_func(fun, a, b, num_puntos)

    coord_x: list[float] = [random.uniform(a, b) for _ in range(0, num_puntos)]
    coord_y: list[float] = [random.uniform(
        0, max_func) for _ in range(0, num_puntos)]

    n_debajo: int = 0
    for i in range(0, num_puntos):
        if (check_point(fun, coord_x[i], coord_y[i])):
            n_debajo += 1
    montecarlo: float = (n_debajo / num_puntos) * (b-a) * max_func
    return montecarlo
```

Figura 1.2: Código de la función integra_mc_iter

2. Vectorizado

Para la vectorización usamos la librería *numpy* y simplificamos todo el código a una sola función integra_mc_vect (2.1). La función np.vectorize() nos permite que cualquier función se pueda aplicar a vectores, simplificando así la computación de la función. con np.linspace() generamos una distribución de puntos en el eje x a los que aplicarles la función vectorizada, tras esto usamos np.max() para hallar el máximo valor que alcanza la función.

Con np.random.uniform generamos un vector de valores de la función desde 0 hasta el máximo, con esto y los primeros valores generados de y podemos ver que puntos quedan por arriba y cuales por debajo de la función. Con esto y la fórmula anterior, volvemos a calcular la integral aproximada de la función.

```python
def integra_mc_vect(fun: float, a: float, b: float, num_puntos: int = 1000) -> float:
    """Calculates the integral of a function by the montecarlo method. Vectorized way.

    Args:
        fun (float): function to integrate
        a (float): lower bound of the interval
        b (float): upper bound of the interval
        num_puntos (int, optional): number of random points to use. Defaults to 1000.

    Returns:
        float: Value of the calculated integral
    """

    vec_fun = np.vectorize(fun)
    coord_x: ndarray[float] = np.linspace(a, b, num_puntos)
    real_y = vec_fun(coord_x)
    max_func: float = np.max(real_y)

    coord_y: ndarray[float] = np.random.uniform(0, max_func, num_puntos)

    n_debajo = np.sum(coord_y < real_y)

    montecarlo = (n_debajo / num_puntos) * (b-a) * max_func

    return montecarlo
```

Figura 2.1: Código de la función integra_mc_vect

3. Funciones y generación de datos

En esta sección se explican otras funciones que se han usado en el programa pero que no afectan a la métrica o las propias funciones que se usan para probar la integral.

3.1. Pruebas

Para las pruebas tenemos dos funciones, una que generará los test (3.1) y otra que correrá cada test individual cinco veces para sacar una media (3.2).

La función test_cases se encarga de generar un vector de tiempos (tanto vectoriales como iterativos) conseguidos de cada caso de prueba, también genera dichos casos de prueba (con una función de *numpy*). Esta función también se encarga de dibujar los gráficos de tiempo de los casos de prueba.

La función test_increment se encarga de coger un caso (número de puntos) y correrlo cinco veces de manera vectorizada e iterativa, midiendo sus tiempos y haciendo la media de todas las ejecuciones. También nos muestra por pantalla el valor conseguido para asegurarnos de que es un valor correcto y guarda dichos datos en un csv para exportarlos más fácilmente a documentos (como este) o para su estudio posterior.

En la función test_cases generamos un pequeño gráfico de cada función probada para ver que es la que queremos calcular realmente, para ello usamos la función gen_fun_graph (3.3).

```python
def test_cases() -> None:
    """Battery of tests to generate time data for the report
    """
    fun_list = [cuadrado, sin, fun2]  # functions to test
    upper: list[float] = [5, pi, 100]  # upper bound for each function
    lower: list[float] = [-5, 0, 0]  # lower bound for each function

    # initial test cases
    casos_iniciales: list[int] = np.linspace(10, 10000000, 20)
    print(casos_iniciales)

    for i in range(0, len(fun_list)):
        gen_fun_graph(fun_list[i], lower[i], upper[i])
        tiempo_iter: list[float] = []
        tiempo_vect: list[float] = []
        casos: list[int] = []
        print(f'{fun_list[i].__name__}: [{lower[i]},{upper[i]}]')
        file: TextIOWrapper = open(
            f"./memoria/recursos/{fun_list[i].__name__}.csv", "w")
        writer = csv.writer(file)
        writer.writerow(['', 'Iterativo', 'Vectorizado'])
        file.close()
        for caso in casos_iniciales:
            (tiempo_iter_aux, tiempo_vect_aux) = test_increment(
                int(caso), fun_list[i], lower[i], upper[i])

            tiempo_iter.append(tiempo_iter_aux)
            tiempo_vect.append(tiempo_vect_aux)
            # last case is duplicate so we don't use it
            casos.append(int(caso))

        plt.figure(i, figsize=(8, 5))  # different plot for each function
        plt.scatter(casos, tiempo_iter, label="iterativo")
        plt.scatter(casos, tiempo_vect, label="vectorizado")
        plt.xlabel('Número de puntos')
        plt.ylabel('Tiempo (ms)')
        plt.legend()

        plt.savefig(
            f'./memoria/imagenes/tiempos_{fun_list[i].__name__}.png', dpi=300)
        plt.clf()
```

Figura 3.1: Código de la función test_cases

```python
def test_increment(caso_inicial: int, fun: float, a: float, b: float) -> (float, float):
    """Test a set number of cases, incrementing 9 time by the initial case

    Args:
        caso_inicial (int): initial case
        fun (float): function to test
        a (float): lower bound of the interval
        b (float): upper bound of the interval

    Returns:
        float, float: times of the iterative tests, times of the vectorized tests
    """
    total_it = 0
    total_vec = 0

    for _ in range(0, 5):
        # Caso iterativo
        it_start: int = time.process_time()
        sol_it: float = integra_mc_iter(fun, a, b, caso_inicial)
        it_end: int = time.process_time()
        total_it += 1000 * (it_end - it_start)
        print(
            f'caso iterativo {caso_inicial}: {sol_it} -> {1000*(it_end-it_start)}ms')

        # Caso vectorizado
        vec_start = time.process_time()
        sol_vect: float = integra_mc_vect(fun, a, b, caso_inicial)
        vec_end = time.process_time()
        total_vec += 1000 * (vec_end - vec_start)
        print(
            f'caso vectorial {caso_inicial}: {sol_vect} -> {1000*(vec_end - vec_start)}ms')
    print(f'tiempo medio it: {total_it/5}ms')
    print(f'tiempo medio vec: {total_vec/5}ms')
    file: TextIOWrapper = open(
        f"./memoria/recursos/{fun.__name__}.csv", "a")
    writer = csv.writer(file)
    writer.writerow([caso_inicial, round(
        total_it/5, 8), round(total_vec/5, 8)])
    file.close()
```

Figura 3.2: Código de la función test_increment

```python
def gen_fun_graph(fun: float, a: float, b: float) -> None:
    """Generates the graph of a given function on a given interval. Graph also includes some
    sample points for the montecarlo method

    Args:
        fun (float): function given
        a (float): lower bound of the interval
        b (float): upper bound of the interval
    """
    num_puntos = 100
    step = (b - a) / num_puntos
    func_x = np.arange(a, b + step, step)
    func_y = np.vectorize(fun)(func_x)

    plt.plot(func_x, func_y)

    max_func = np.max(func_y)

    coord_x: ndarray[float] = (b-a) * np.random.random_sample(num_puntos) + a
    coord_y: ndarray[float] = max_func * np.random.random_sample(num_puntos)
    plt.scatter(coord_x, coord_y, color="red", marker="x")
    plt.savefig(f"./memoria/imagenes/{fun.__name__}.png", dpi=300)
    plt.close()
```

Figura 3.3: Código de la función gen_fun_graph

3.2. Funciones probadas

Las funciones probadas han sido el cuadrado de un número (3.4), la función sin de la biblioteca *math* y fun2 (3.5), una función simplona porque no se me ocurría otra cosa. Las visualizaciones de dichas funciones se pueden ver en las figuras 3.6, 3.7 y 3.8 respectivamente

```python
def cuadrado(x: float) -> float:
    """Test function to calculate the square of a number

    Args:
        x (float): number to operate with

    Returns:
        float: x*x
    """
    return x * x
```

Figura 3.4: Código de la función cuadrado

```python
def fun2(x: float) -> float:
    """Test function that calculates a random function

    Args:
        x (float): value to calculate

    Returns:
        float: 1 + x / 10
    """
    return 1 + x / 10
```

Figura 3.5: Código de la función fun2

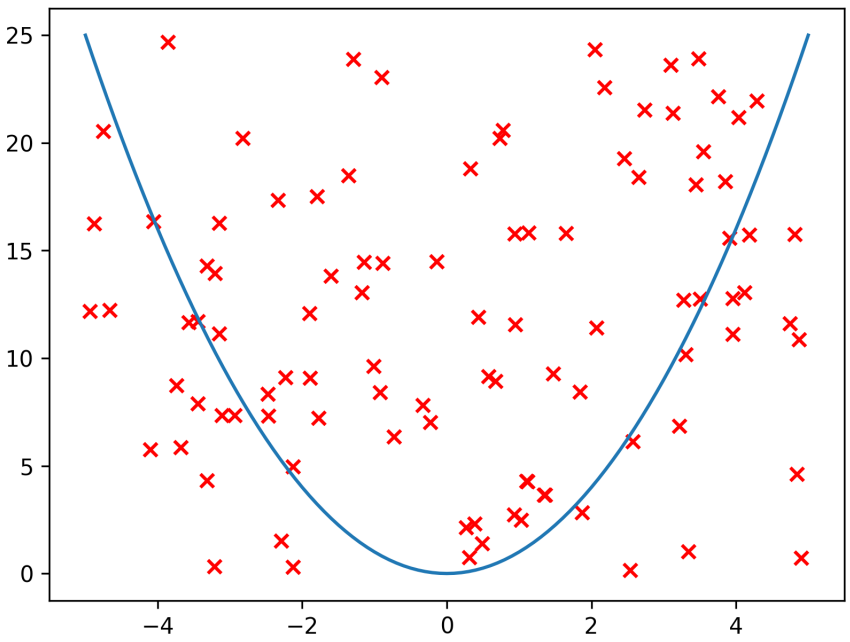

Figura 3.6: Gráfico de la función cuadrado

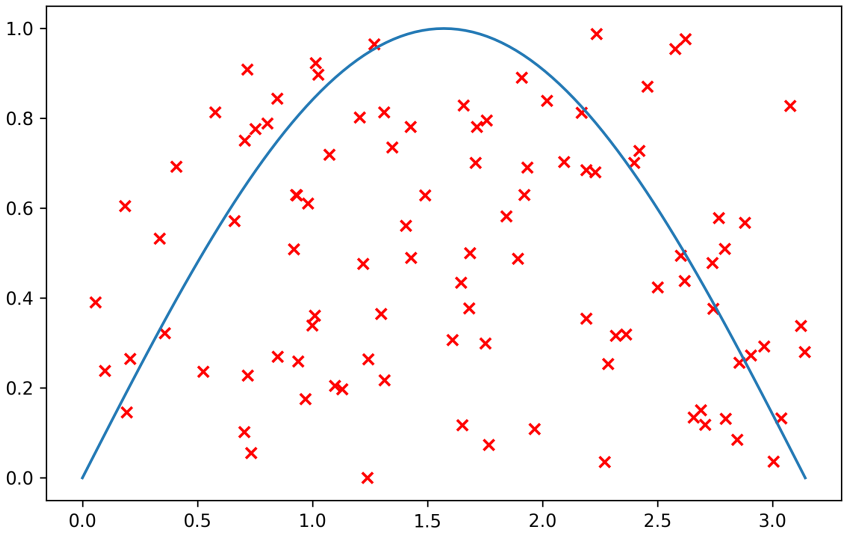

Figura 3.7: Gráfico de la función sin

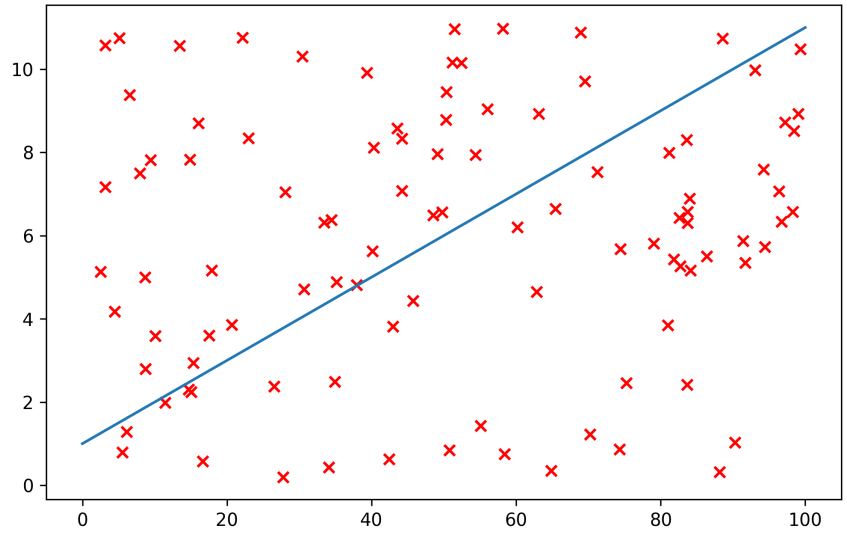

Figura 3.8: Gráfico de la función fun2

4. conclusiones

Viendo los tiempos en las tablas 4.1, 4.2, 4.3 y en las figuras 4.1, 4.2 y 4.3, la vectorización apenas crece en tiempo cuando aumentamos los puntos en comparación con las funciones iterativas. Esto muestra bastante bien que las funciones de *numpy* van a ser muy útiles a la hora de tratar con grandes números de datos.

Número puntos	Tiempo Iterativo(ms)	Tiempo Vectorizado (ms)
10	0.1369756	0.3771346
526325	674.6754046	125.8347064
1052640	1274.558451	300.3626278
1578955	1717.5269202	379.6558324
2105271	2255.6019598	434.6240468
2631586	2884.051705	566.1890712
3157901	3407.3960018	749.923235
3684216	3818.739089	776.2043134
4210532	4594.5391146	969.4505468
4736847	5255.2862856	1027.7914164
5263162	5279.4706534	989.740287
5789477	6312.9308932	1305.0510936
6315793	5591.0079092	1154.575369
6842108	7194.746246	1414.1793968
7368423	7473.1107246	1554.4561802
7894738	7113.5983272	1308.080552
8421054	7668.295385	1546.8493824
8947369	9002.321786	1891.4927836
9473684	9811.3381928	1960.1618222
10000000	9910.4064952	1873.6220574

Cuadro 4.1: Comparación de tiempos de ejecución en la función cuadrado

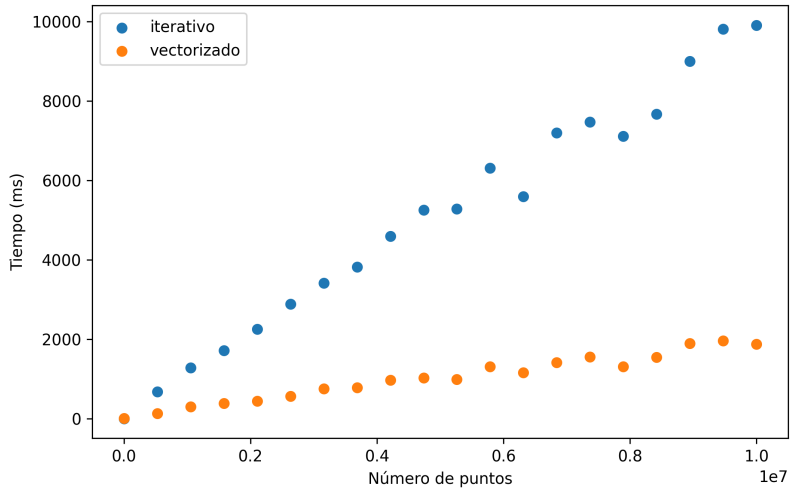

Figura 4.1: Gráfica de tiempo de la función cuadrado

Número puntos	Tiempo Iterativo(ms)	Tiempo Vectorizado (ms)
10	0.0631796	0.1909238
526325	457.9924458	74.8942774
1052640	825.409456	148.7016206
1578955	1269.4013368	223.3296038
2105271	1685.6547786	309.3003262
2631586	2111.9439964	380.9114448
3157901	2505.5602964	462.6987304
3684216	2917.3198848	536.3096212
4210532	3354.8244072	617.4054194
4736847	4093.0994862	776.2384638
5263162	4379.5006232	916.9070134
5789477	5788.8836104	1081.972379
6315793	5961.8761988	1118.8396774
6842108	6259.3834624	1168.4500836
7368423	5950.8662412	1075.4419258
7894738	6386.2250442	1147.1379114
8421054	6775.2562408	1226.9761048
8947369	7450.5532828	1341.32155
9473684	7610.6918024	1378.639595
10000000	9398.1964264	1660.8293858

Cuadro 4.2: Comparación de tiempos de ejecución en la función sin

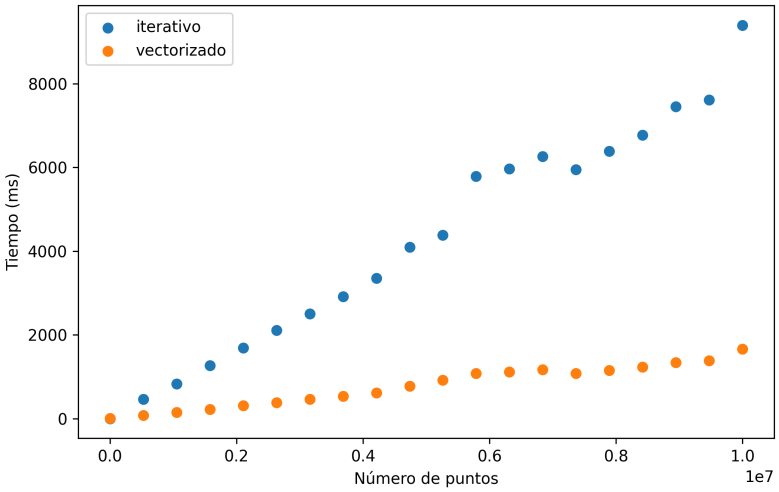

Figura 4.2: Gráfica de tiempo de la función sin

Número puntos	Tiempo Iterativo(ms)	Tiempo Vectorizado (ms)
10	0.078149	0.2461624
526325	612.1670324	115.830224
1052640	1157.641911	240.3914702
1578955	1731.5750798	362.7504006
2105271	2309.3118208	487.8311122
2631586	2879.3533208	612.9847106
3157901	3523.6481424	757.217449
3684216	4017.2214028	859.5417018
4210532	4526.2402242	961.4854278
4736847	4485.2059432	971.9223102
5263162	4633.4739268	1007.061835
5789477	5138.0062738	1112.934594
6315793	5759.3467958	1301.4575466
6842108	7628.0873452	1630.5288604
7368423	6492.34342	1410.333629
7894738	7036.936523	1520.3425232
8421054	7569.2997088	1642.851772
8947369	8188.8997046	1823.0862214
9473684	8514.3442056	1829.5735648
10000000	8964.4238148	1935.803055

Cuadro 4.3: Comparación de tiempos de ejecución en la función 'fun2'

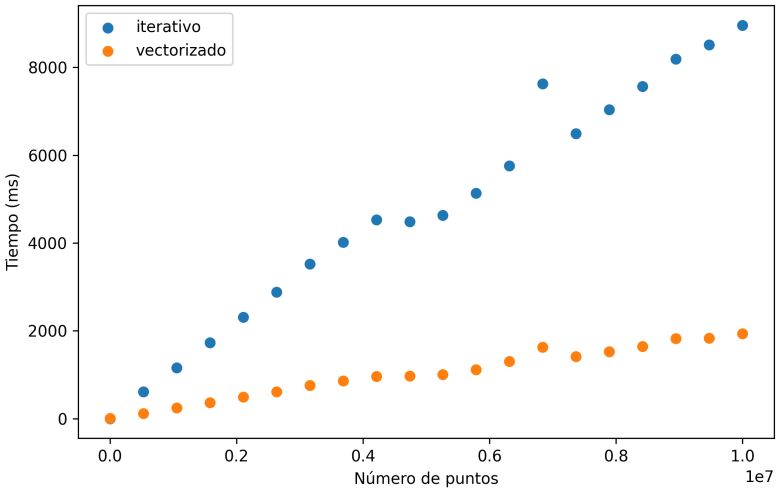

Figura 4.3: Gráfica de tiempo de la función fun2