COMPUTACIÓN AVANZADA

Práctica 1: Determinación de la constante de Madelung

19 DE FEBRERO DE 2023

UNIVERSIDAD AUTÓNOMA DE MADRID

Pablo Gradolph Oliva

1. OBTENCIÓN DE LA CONSTANTE DE MADELUNG (M)

En el enunciado de la práctica se nos proporciona la siguiente ecuación:

$$M = \sum_{\substack{i,j,k=-L \\ no\ i=j=k=0}}^{L} \pm \frac{1}{\sqrt{i^2+j^2+k^2}}$$

Donde M es la constante de Madelung y donde los términos con *i+j+k* par son positivos y los términos con *i+j+k* impar son negativos.

Se pide hacer un programa en Python y otro en C++ para calcular dicha constante para valores de L = 20, 50 ,100 y 200, y comparar los resultados obtenidos con el valor real de M (M = -1.74756) y, también, comparar los tiempos de ejecución entre ambos lenguajes.

Para ello, he hecho los dos programas principales "Practica1V1.py" y "Practica1V1.cpp" en los que la lógica del programa es la misma para los dos lenguajes de programación, obteniéndose así los mismos resultados, que son los siguientes:

```
Para L = 20 --> M = -1.719401169; Error absoluto = 0.02815883055; Error relativo = 1.611322675%
Para L = 50 --> M = -1.736131916; Error absoluto = 0.01142808394; Error relativo = 0.6539451542%
Para L = 100 --> M = -1.741819816; Error absoluto = 0.00574018416; Error relativo = 0.3284685024%
Para L = 200 --> M = -1.744685042; Error absoluto = 0.002874957829; Error relativo = 0.1645126822%
```

Comprobamos que las soluciones convergen hacia el valor real de M. Vemos, además, como para valores más altos de L, la precisión en la solución obtenida es mayor y, por tanto, el error cometido es menor. Algo lógico, puesto que para longitudes más altas del cubo considerado (leer enunciado), estamos teniendo en cuenta más coordenadas del espacio lo cual mejora la aproximación (valor real cuando $L \to \infty$). Por último, decimos que la convergencia es lenta puesto que para aumentar en 1 el número de decimales correctos se requiere de un número elevado de iteraciones (el error absoluto para L=100 y L=200 es del mismo orden y gracias al programa de C++ sabemos que el orden en el error absoluto cambia para L=600).

2. COMPARACIÓN EN LOS TIEMPOS DE EJECUCIÓN

Respecto a los tiempos de ejecución, encontramos una gran diferencia entre los dos lenguajes, y es que, aunque el tiempo de ejecución no siempre es el mismo y depende del ordenador del que se disponga o del número de tareas que esté realizando en ese momento, vemos que, en Python, el cálculo de M para los 4 valores de L toma entorno a 1 minuto y medio, mientras que en C++ siempre está por debajo de los 2 segundos. Ejemplo de una de las medidas:

Python:

```
* Tiempo de ejecución del programa (s) = 96.17665949999355
```

C++:

```
* Tiempo de ejecucion del programa (s) = 1.0193798
```

Podemos considerar que es una diferencia significativa y, por esta razón, he creado los ficheros “Practica1V2.py” y “Practica1V2.cpp”, en los que he creado una versión 2 de los primeros programas trabajando de la siguiente manera:

Dado que en C++ la ejecución es bastante rápida, en la versión 2 del programa lo que he hecho es un bucle do-while de forma que el programa finalice cuando el error en la aproximación de M esté por debajo de una tolerancia dada (a partir de una tolerancia 10e-5 no se va a alcanzar puesto que es el número de decimales que posee el valor real de la constante). De esta forma, conseguimos un programa que realice una aproximación mucho más precisa para la constante de Madelung (L se incrementa de 100 en 100 por una cuestión de eficiencia, pero este parámetro es modificable). Datos obtenidos para valores de L más grandes:

```
Para L = 100 --> M = -1.741819816; Error absoluto = 0.00574018416; Error relativo = 0.3284685024%
Para L = 200 --> M = -1.744685042; Error absoluto = 0.002874957829; Error relativo = 0.1645126822%
Para L = 300 --> M = -1.745643296; Error absoluto = 0.001916704087; Error relativo = 0.1096788715%
Para L = 500 --> M = -1.746411048; Error absoluto = 0.001148952268; Error relativo = 0.06574608413%
Para L = 600 --> M = -1.746603145; Error absoluto = 0.0009568546082; Error relativo = 0.05475374855%
Para L = 700 --> M = -1.746740397; Error absoluto = 0.0008196028295; Error relativo = 0.04689983917%
Para L = 800 --> M = -1.746843357; Error absoluto = 0.0007166425738; Error relativo = 0.04100818134%
Para L = 900 --> M = -1.74692345; Error absoluto = 0.0006365496968; Error relativo = 0.03642505532%
Para L = 1000 --> M = -1.746987533; Error absoluto = 0.0005724673772; Error relativo = 0.0327580957%
```

Por otro lado, en Python he estudiado cuáles son las principales causas por las que la ejecución es tan lenta y he tratado de optimizar el programa al máximo. Para ello, he hecho varias pruebas y he visto lo siguiente:

- La función sqrt del paquete math es similar a elevar a 1/2.
- Multiplicar una variable por sí misma es ligeramente más eficiente que elevarla al cuadrado.
- $(-1)^{i+j+k}$ * nuestra_función es algo más eficiente que hacer un condicional (if/else) en cada iteración del bucle for.
- La función np.arange de numpy es más eficiente que el range de Python.
- Lo que más tiempo le lleva a Python dentro de este programa son los bucles for.

En la versión 2 del programa en Python, he conseguido reducir ligeramente el tiempo de ejecución (ahora se queda entorno a 1 minuto) haciendo los cambios comentados anteriormente. Para cambiar los 3 bucles for, he conseguido crear dos métodos cuyos tiempos de ejecución son similares.

Para ambos he creado todas las posibles coordenadas en una lista llamada bucle, con la función “product” del paquete itertools y he eliminado la coordenada (0,0,0) la cual no tenemos que tener en cuenta y además, evitamos dividir por 0.

Luego, en el primer método he hecho un único bucle for recorriendo todas las coordenadas y aplicando la fórmula para hallar M:

```
for coordenada in bucle:
        i,j,k = coordenada[0],coordenada[1],coordenada[2]
        M += ((-1)**(fabs(i+j+k))) * (1/sqrt(i*i + j*j + k*k))
```

Y en el segundo he aplicado un map a todas las coordenadas con una función lambda que aplica la fórmula para el cálculo de M y luego, mediante "sum" calculo la suma de todas ellas:

```
M = sum(map(lambda coor: ((-1)**(fabs(coor[0]+coor[1]+coor[2]))) *
(1/sqrt(coor[0]*coor[0] + coor[1]*coor[1] + coor[2]*coor[2])),
bucle))
```

Este segundo método se encuentra comentado, habría que comentar el primer método y "descomentar" este para que se ejecute correctamente. Veamos la diferencia en los tiempos de ejecución (Ejemplo de una de las medidas):

Versión 1 del código en Python:

```
* Tiempo de ejecución del programa (s) = 83.93868690007366
```

Versión 2 de Python por el primer método:

```
* Tiempo de ejecución del programa (s) = 65.02973400009796
```

Versión 2 de Python por el segundo método:

```
* Tiempo de ejecución del programa (s) = 67.58111170004122
```

En general, el tiempo de ejecución se reduce ligeramente, aunque en ningún caso alcanzaremos las velocidades del lenguaje C++. Por esta razón, deducimos que la principal desventaja que se le encuentra a Python frente a otros lenguajes (especialmente los compilados) es la velocidad de ejecución. Aunque, se trata de un lenguaje con otras muchísimas características positivas entre las que destacan la sencillez o el multipropósito del lenguaje gracias a las librerías.

Me gustaría comentar también, que he encontrado formas de ejecutar lenguaje Python con distintos compiladores (mypyc, PyPy…), lo cual hace que la velocidad de ejecución se asemeje a la de un lenguaje compilado como C++, o ejecutar tu código mediante Cython que es lenguaje similar a Python que compila código nativo y es mucho más rápido. Sin embargo, considero que el objetivo de la práctica era más bien encontrar que funciones son más o menos eficientes y optimizar nuestro propio código ejecutado con el intérprete de Python y no tanto este tipo de soluciones.

.