INFORME TAREA 2

Benjamín Briceño
RUT: 20.653.153-3
Github: @Benbriel

P1

En esta parte de la tarea se busca analizar los promedios anuales de anomalías de temperatura de la Tierra, y aplicar un método de interpolación para estimar un dato faltante. Además, se exploran los alcances de la extrapolación con un método de interpolación.

Para esto, se utilizó la función `CubicSpline` del módulo `scipy.interpolate`, que recibe como input un array de datos como eje x, y otro como eje y. Con esto, se genera una función que pasa por todos los puntos dados, y predice una función suave y continua que pasa por puntos de entremedio. Además, se usó el argumento `extrapolate=True`, por lo que es posible obtener valores de la función fuera del rango original. Por otro lado, la función `CubicSpline` ocupa por defecto el valor `bc_type='not-a-knot'`, que asume que el polinomio en los extremos es igual al del segmento anterior. Además, se utilizó el módulo `pandas` para importar y tratar los datos del archivo `GLB.Ts+dSST.csv`.

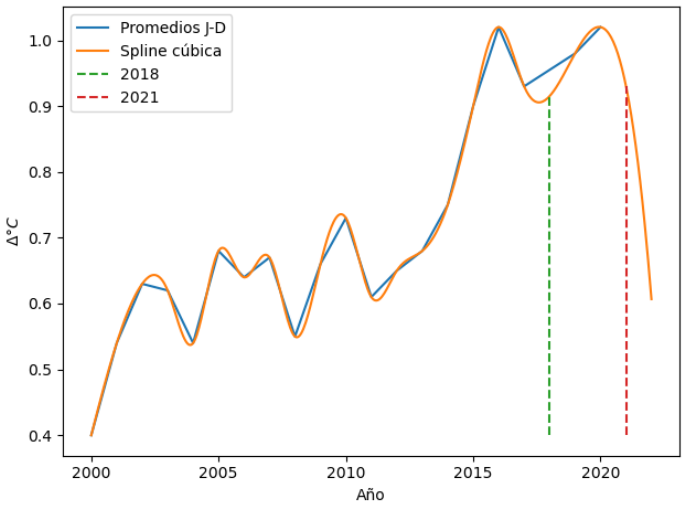

Figura 1: Datos de promedios de la anomalía de temperatura de la Tierra, y la interpolación spline cúbica entre 2000 y 2021. En verde, se estima el valor para 2018, y en rojo se predice para 2021.

Para la temperatura promedio del año 2018, se obtuvo un valor de 0.915, lo que tiene un error del 7.65 %, lo cual es relativamente malo, pues es incluso mayor al 5 %, que es el rango aceptable. Para esto, se utilizó una interpolación spline cúbica, evidenciable en la Figura 1, y se eligió de esta manera asumiendo que la función anomalía de temperatura era suave para todos los años. Por esto, un polinomio de grado alto habría tenido un comportamiento oscilatorio, conocido como fenómeno

de Runge. Esto no habría tenido sentido predictivo al extrapolar o incluso al interpolar.

Por otro lado, la diferencia entre el estimado y el valor real puede tener muchas explicaciones. Por un lado, es posible que la anomalía tenga un comportamiento aleatorio, por lo que ninguna función sería capaz de predecir los datos. Por otro lado, la anomalía podría ser mejor descrita por un polinomio de grado mayor que 3, por lo que la spline cúbica no podría interpolar de forma correcta.

Por último, extrapolando la función spline para el año 2021 resulta en un valor de 0.93. Esto utilizando la extrapolación `not-a-knot`, que predice un polinomio de concavidad negativa, por lo que la anomalía decrecería en el futuro. Este método fue elegido, pues observando los datos como comportamiento sinusoidal, en 2020 se encontraba en un peak de anomalía, por lo que en principio debería disminuir. Sin embargo, aplicando otras extrapolaciones, el valor predicho para 2021 aumenta, pues la spline predice un polinomio de concavidad positiva. Utilizando `bc_type='natural'`, por ejemplo, se predice un valor de la anomalía de 1.06 para 2021. Esta gran diferencia entre los valores predichos según el tipo de extrapolación da a entender que la extrapolación es muy inefectiva utilizando el método spline, y otros métodos interpolantes en general.

P2

1. Introducción

En esta sección de la tarea se buscó resolver un sistema matricial de primer orden, utilizando distintos algoritmos de resolución del problema. En este caso, el problema es el valor de las tensiones de un sistema de 12 cuerdas y 3 masas, que se encuentra en el enunciado. Se conoce que $M_2 = 1$ kg, y $M = 4.153$ kg. Luego, el problema a resolver es

$$A \cdot \vec{T} = b \tag{1}$$

Para esto, se utilizaron las funciones `solve`, `lu_factor`, `lu_solve` e `inv`, provenientes del módulo `scipy.linalg`. Además, se importaron los módulos `numpy` y `matplotlib.pyplot`, y la constante de aceleración de la Tierra `g=scipy.constants.g`. Además, se comparó el tiempo de ejecución de estas funciones para un set de 100 valores de $M_1 \in [0, 2]$ kg.

2. Desarrollo

Primero, se obtuvo el valor de las matrices A y b, planteando las ecuaciones de suma de fuerzas para cada nodo entre las distintas cuerdas y masas. Luego,

$$\begin{pmatrix}
-\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 & 0 & 0 & \sin\frac{\pi}{4} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 & 0 & -\sin\frac{\pi}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 & \sin\frac{\pi}{4} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & -\sin\frac{\pi}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\sin\frac{\pi}{4} & \sin\frac{\pi}{4} \\
\cos\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 & 0 & 0 & 0 & -\cos\frac{\pi}{6} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \cos\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 & 0 & 0 & -\cos\frac{\pi}{6} & -\cos\frac{\pi}{6} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cos\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 & 0 & 0 & -\cos\frac{\pi}{6} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \cos\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 0 & -\cos\frac{\pi}{12} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cos\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & -\cos\frac{\pi}{12} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cos\frac{\pi}{12} & \cos\frac{\pi}{12}
\end{pmatrix} \cdot \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \\ T_{10} \\ T_{11} \\ T_{12} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ M_1 g \\ M_2 g \\ M g \end{pmatrix}$$

A continuación, se importó la función `scipy.linalg.solve`, que al recibir las matrices A y b, entrega el vector $\vec{T}$ del problema 1, usando diversos algoritmos según el argumento `assume_a='gen'`, que por defecto utiliza el método PLU de descomposición de matrices. Cada vez que se llama a la función, esta descompone la matriz A, lo cual es ineficiente, pues A es constante, y no depende de M_1.

Por otro lado, se usaron las funciones `scipy.linalg.lu_factor` y `lu_solve`. La primera toma la matriz A y la descompone en la matriz `lu` y `piv`, donde `lu` es la superposición de las matrices triangulares `l` y `u`, y `piv` es un array que indica el orden de las filas permutadas. Tanto `lu` como `piv` funcionan igual que la descomposición PLU, solo que cambia su representación en son de mayor eficiencia.

Por último, se importó la función `scipy.linalg.inv`, que toma una matriz A y entrega su inversa. Sorprendentemente, esta función también utiliza el método PLU para invertir A. Sin embargo, puede ser más eficiente para múltiples valores de M_1, pues al calcular $A^{-1} \cdot b$ solo debe hacer una multiplicación, mientras que la función `lu_solve` debe hacer múltiples, al tratar con 3 matrices distintas. La implementación de esta función es similar a la anterior, pues primero se define `inv_A=inv(A)`, y luego se itera sobre M_1. Este método hace el código más eficiente, al limitar el cálculo de A^{-1} a una sola vez.

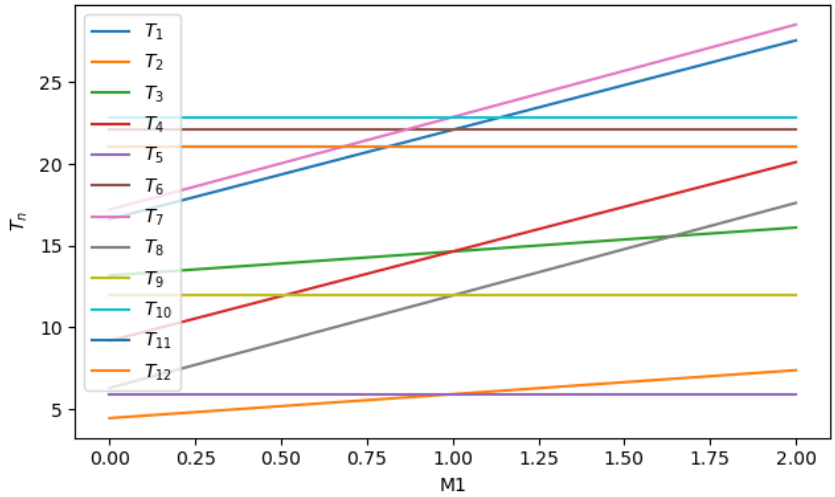

Figura 2: Valor de las 12 tensiones en función de M_1, representando el vector $\vec{T}$. Esto fue calculado usando la función `inv`. Por falta de colores, cabe destacar que T_{12} es la recta naranja superior. Además, la tensión $T_{11} = T_{12}$, por lo que no se ve al solaparse con T_{12}.

Después de haber utilizado estas funciones, se pudo encontrar que los valores de $\vec{T}$ de resolución del problema eran idénticos entre los 3 métodos, lo que era de esperar. Además, como muestra la Figura 2, la tensión máxima del sistema si $M_1 < 1$ es T_{10}. Luego, cuando $M_1 = 1$, ocurre $M_1 = M_2$, por lo que se produce una simetría entre las tensiones $T_1 = T_6$, $T_2 = T_5$, $T_3 = T_4$, $T_7 = T_{10}$, $T_8 = T_9$ y $T_{11} = T_{12}$. Cuando $M_1 > 1$, sin embargo, la tensión máxima pasa a ser T_7. Todo esto hace sentido, pues cuando $M_1 < M_2$, el sistema tiende a cargarse más a la derecha, tensando más las cuerdas de la derecha. Cuando $M_1 > M_2$, ocurre lo contrario.

Método (`func`)	Tiempo de ejecución (`%timeit T_solve(func)`)
`solve`	4.56 ms ± 90 µs per loop
`lu_solve`	1.46 ms ± 43.9 µs per loop
`inv`	579 µs ± 28.7 µs per loop

Cuadro 1: Utilizando la función mágica de `ipython`, `%timeit`, se midió el tiempo de ejecución de una función `T_space(func)`, que toma la función `func` y la utiliza para calcular $\vec{T}$ para 100 valores de M_1.

Comparando los tiempos de ejecución de los 3 métodos, es claro que la función más eficaz es `inv`, seguida de `lu_solve` y `solve` a la hora de entregar un valor para $\vec{T}$. Esto puede explicarse, pues tanto para `inv` como `lu_solve`, se realizaron cálculos fuera de la iteración para cada M_1. Lo anterior eliminó las redundancias que todavía tiene el método `solve`, pues $A = PLU$ es constante y no necesita ser calculada cada vez.

3. Discusión y Conclusiones

Es posible representar un sistema de ecuaciones lineales como una sola ecuación matricial, la cual se puede resolver numéricamente en `python` a través de distintas formas. Como estos problemas involucran arrays con muchos datos, la eficiencia en el código es muy importante.

De los métodos postulados anteriormente, `inv` es el más rápido. Sin embargo, esto ocurre solo puesto que la matrix A es constante, por lo que se puede reducir considerablemente la cantidad de cálculos numéricos. Para otro tipo de sistemas con otras variables, es posible que esto no ocurra.

Las librerías de `python` como `numpy` o `scipy` poseen módulos muy bien optimizados, que permiten realizar cálculos numéricos de forma muy eficiente. Es importante buscar este tipo de librerías u otras en internet al resolver un problema, puesto que probablemente estarán más optimizadas, y funcionarán órdenes de magnitud más rápido que cualquier implementación propia.