1. Lectura preliminar

El objetivo de esta lista de ejercicios es la familiarización con preguntas desarrolladas anteriormente en certámenes. Se sugiere encarecidamente que se aplique la metodología IDEA a cada ejercicio. Algunos ejercicios solicitan explícitamente la implementación de los algoritmos propuestos y otro no, sin embargo, todos los ejercicios contienen una componente de desarrollo algebraico primero y luego el desarrollo se puede implementar en Python. En particular se sugiere adquirir práctica utilizando NumPy adecuadamente en las implementaciones de sus respuestas. Esto se discute extensamente en el reglamento de tareas. Por ejemplo, considere las siguientes propuestas de implementación para obtener A^2, es decir, elevar una matriz al cuadrado:

- `np.power(A,2)`
- `np.dot(A,A)`
- `A @ A`
- `np.linalg.matrix_power(A,2)`

¿Son todas estas alternativas equivalentes? Si no lo fueran, explique.

En Apartado 3 se presenta el desarrollo de algunos ejercicios a modo referencial para que pueda comparar su desarrollo con el desarrollo propuesto.

2. Ejercicios propuestos

2.1. ¿Cómo obtenemos $g(t)$?

Se desea calcular la siguiente integral:

$$f(x) = \int_0^x g(t) e^{-t^2} \mathrm{d}t, \tag{1}$$

donde se conoce que la función $g(t)$ satisface el siguiente Problema de Valor Inicial:

$$\begin{aligned} g''(t) &= -2\, g(t) - 2\, g'(t), \\ g(0) &= 0, \\ g'(0) &= 1. \end{aligned}$$

(a) Proponga un algoritmo que permita calcular el valor de la integral para un valor de x fijo (por ejemplo, $x = x_f$). Asegúrese que su algoritmo resuelva el IVP de manera estable. Considere como parámetros de su algoritmo el valor de x a evaluar y el número de puntos N para estimar la integral.

2.2. ¿Cuál es la estructura polinomial?

Considere que tiene a su disposición un conjunto de datos $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, la tarea a resolver consiste en determinar si los datos cumplen con una estructura polinomial particular. Por ejemplo, si los datos fueran $\{(1,1), (2,2), (3,3)\}$ y la estructura a evaluar es la siguiente $p_1(x) = a_0 + a_1\, x$, entonces la respuesta debiera ser afirmativa, por el contrario, si la estructura fuera $p_2(x) = a_0$ ó $p_3(x) = a_1\, x$, entonces la respuesta debe ser negativa. Notar que los valores a_i no son requeridos explícitamente, pero si son importantes para verificar si se cumple $p(x_j) = y_j$ para $(x_j, y_j) \in D$, con un umbral de error de 10^{-10}, es decir $|p(x_j) - y_j| < 10^{-10}$ para cada par ordenado (x_j, y_j).

Para definir una estructura polinomial generalizada, consideremos la siguiente notación,

$$p(x) = \sum_{i=0}^{m} a_i\, \delta_i\, x^i,$$

donde el parámetro δ_i puede ser 0 ó 1, y determina si el i-ésimo término de la sumatoria debe ser incluido en la estructura o no. Por ejemplo, para la estructura de ejemplo $p_3(x)$ la definición de los coeficientes δ_i, escrito de forma vectorial, sería la siguiente $\boldsymbol{\delta} = [0, 1]$, esto debido a que,

$$\begin{aligned} p_3(x) &= a_0\, \delta_0 + a_1\, \delta_1\, x \\ &= a_0\, 0 + a_1\, 1\, x \\ &= a_1\, x. \end{aligned}$$

Note que el valor m se obtiene de la dimensión de $\boldsymbol{\delta}$, el cual sería en este caso $m = \dim(\boldsymbol{\delta}) - 1 = 1$.

(a) Explique teóricamente como resolverá el problema. Su explicación debe desarrollarse en LaTeX.

(b) Implemente su solución considerando como input el conjunto D, en la forma de dos vectores, $\mathbf{x}$ y $\mathbf{y}$, y el vector $\boldsymbol{\delta}$. Por simplicidad considere que se cumple la siguiente relación: $m < n$.

(c) Indique si su respuesta es afirmativa o negativa para los siguientes valores de entrada:

```
x = np.array( [-1., 0., 1., 2., 3.] )
y = np.array( [1.1,0.1,1.1,4.1,9.1] )
delta = np.array( [1,0,1] )
```

2.3. Mínimos IVP

Un grupo de investigadores ha construido un sistema dinámico que modela la trayectoria de material particulado generado por el tráfico vehicular. El modelo construido determina la posición de una partícula en el tiempo. Por simplicidad, el modelo se ha reducido a 1D, es decir, para la i-ésima partícula se registrará su posición $x_i(t) \in \mathbb{R}$. Desafortunadamente los investigadores no han llegado a un consenso en la formulación del modelo. Para analizar la propuesta, se considerará que el sistema solo tiene dos partículas. Las ecuaciones obtenidas para las partículas 1 y 2 para $t \in [0, T]$ son las siguientes:

$$\dot{x}_1 = \phi(x_2 - x_1) + \sum_{i=1}^{n} \alpha_{1,i}\, \phi(w_i - x_1) - \gamma\, x_1, \tag{1}$$

$$\dot{x}_1 = \phi(x_2 - x_1) + \sum_{i=1}^{n} \beta_{1,i}\, \phi(w_i - x_1) - \gamma\, x_1, \tag{2}$$

$$\dot{x}_2 = \phi(x_1 - x_2) + \sum_{i=1}^{n} \alpha_{2,i}\, \phi(w_i - x_2) - \gamma\, x_2, \tag{3}$$

$$\dot{x}_2 = \phi(x_1 - x_2) + \sum_{i=1}^{n} \beta_{2,i}\, \phi(w_i - x_2) - \gamma\, x_2, \tag{4}$$

$$\dot{x}_1 + \dot{x}_2 = \sum_{i=1}^{n} \delta_i\, \phi\left(w_i - \frac{x_1 + x_2}{2}\right) + \gamma\,(x_1 + x_2), \tag{5}$$

donde $\phi(x)$ es una función que se entrega implementada y cuantifica el efecto de la interacción entre partículas. **Considere que los coeficientes reales γ, w_i, α's, β's son coeficientes conocidos.** Una posible alternativa sería solo utilizar 2 ecuaciones que involucren $\dot{x}_1$ y $\dot{x}_2$, sin embargo, esta alternativa es descartada. Por lo tanto se le solicita que determine la evolución de $x_1(t)$ y $x_2(t)$ minimizando el error cuadrático, es decir, debe construir el sistema dinámico equivalente para $\overline{\dot{x}}_1$ y $\overline{\dot{x}}_2$. Adicionalmente, los investigadores sí se pusieron de acuerdo en que las condiciones iniciales era únicas, es decir $x_1(0) = x_{1,0}$ y $x_2(0) = x_{2,0}$.

(a) Explique teóricamente como obtendrá $\overline{\dot{x}}_1$ y $\overline{\dot{x}}_2$.

(b) Implemente la función `f_Particles`. Esta función implementa de forma explícita el lado derecho del sistema dinámico resultante para $\overline{\dot{x}}_1$ y $\overline{\dot{x}}_2$. Notar que el parámetro n se define en función de la dimensión de los vectores inputs αs, βs, y δ.

(c) Considere los siguiente datos:

```
alpha1 = np.array( [1.2701176891,0.28811319  ,0.7046913486,1.6134431034,1.3446417529] )
alpha2 = np.array( [-0.7036400735, 0.6840636606,-0.10897719  ,-0.0743175733, 0.2956309214] )
beta1 = np.array( [0.1037113712,1.047076925 ,0.5479471621,0.0876060119,0.3195815276] )
beta2 = np.array( [ 0.2402455157, 1.0757369327,-0.1477139499, 0.2254087452,-0.6149489323] )
delta = np.array( [-1.8381526674, 0.4706053887, 0.6223940632,-0.5343588147, 1.6342233293] )
gamma = 0.1
```

Entregue el valor $\overline{x}_1(10)$ y $\overline{x}_2(10)$ considerando que $\overline{x}_1(0) = 1.6$ y $\overline{x}_2(0) = 1.8$, y que utiliza `solve_ivp` de SciPy.

2.4. QR incremental

Considere la siguiente secuencia de matrices tridiagonales $T_n \in \mathbb{R}^{(n+1)\times n}$,

$$T_n = \begin{bmatrix} a_1 & c_1 & 0 & \dots & \dots & \dots & 0 \\ b_1 & a_2 & c_2 & 0 & \dots & \dots & \vdots \\ 0 & b_2 & a_3 & c_3 & 0 & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & b_{n-2} & a_{n-1} & c_{n-1} \\ \vdots & \ddots & \ddots & \ddots & 0 & b_{n-1} & a_n \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 & b_n \end{bmatrix},$$

donde los coeficientes a_i, b_i y c_i son conocidos. Note que T_n no es una matriz cuadrada, dado que tiene una fila más que la cantidad de columnas. Considere que usted tiene acceso a la factorización QR reducida de la matriz T_n, es decir, tiene acceso a,

$$T_n = \widehat{Q}_n \widehat{R}_n = \begin{bmatrix} \mathbf{q}_1^{[n]} & \mathbf{q}_2^{[n]} & \dots & \mathbf{q}_n^{[n]} \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & \dots & r_{1,n} \\ 0 & r_{2,2} & r_{2,3} & \dots & r_{2,n} \\ 0 & 0 & r_{3,3} & \dots & r_{3,n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & r_{n,n} \end{bmatrix},$$

donde $\widehat{Q}_n \in \mathbb{R}^{(n+1)\times n}$ y $\widehat{R}_n \in \mathbb{R}^{n\times n}$. En este caso se usa el súper-índice $[n]$ para denotar que el vector $\mathbf{q}_i^{[n]} \in \mathbb{R}^{n+1}$, y está asociado a la matriz T_n. Como se indicó al principio, T_n denota una secuencia de matrices tridiagonales, esto significa que la siguiente matriz de esta secuencia es,

$$T_{n+1} = \begin{bmatrix} a_1 & c_1 & 0 & \dots & \dots & \dots & \dots & 0 \\ b_1 & a_2 & c_2 & 0 & \dots & \dots & \dots & \vdots \\ 0 & b_2 & a_3 & c_3 & 0 & \dots & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \ddots & 0 & \\ \vdots & \ddots & \ddots & 0 & b_{n-2} & a_{n-1} & c_{n-1} & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 & b_{n-1} & a_n & c_n \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 & b_n & a_{n+1} \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 & b_{n+1} \end{bmatrix},$$

la cual se diferencia de T_n solamente porque se agregó una columna y una fila adicional. Este patrón puede representarse de la siguiente forma,

$$T_{n+1} = \left(\begin{array}{ccc|c} & & & 0 \\ & T_n & & \vdots \\ & & & 0 \\ & & & c_n \\ & & & a_{n+1} \\ \hline 0 & \dots & 0 & b_{n+1} \end{array}\right),$$

es decir, la mayor parte de la matriz T_{n+1} corresponde a la matriz T_n. La tarea a resolver es obtener la factorización QR reducida de la matriz T_{n+1} dado que conocemos la factorización QR reducida de la matriz T_n. Para su desarrollo considere la siguiente

identidad,

$$
\begin{aligned}
T_{n+1} &= \left(\begin{array}{ccc|c} & & & 0 \\ & T_n & & \vdots \\ & & & 0 \\ & & & c_n \\ \hline 0 & \dots & 0 & a_{n+1} \\ & & & b_{n+1} \end{array}\right) \\
&= \left(\begin{array}{ccc|c} & & & 0 \\ & \widehat{Q}_n\,\widehat{R}_n & & \vdots \\ & & & 0 \\ & & & c_n \\ \hline 0 & \dots & 0 & a_{n+1} \\ & & & b_{n+1} \end{array}\right) \\
&= \widehat{Q}_{n+1}\,\widehat{R}_{n+1} \\
&= \begin{bmatrix} \mathbf{q}_1^{[n+1]} & \mathbf{q}_2^{[n+1]} & \dots & \mathbf{q}_n^{[n+1]} & \mathbf{q}_{n+1}^{[n+1]} \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & \dots & r_{1,n} & r_{1,n+1} \\ 0 & r_{2,2} & r_{2,3} & \dots & r_{2,n} & r_{2,n+1} \\ 0 & 0 & r_{3,3} & \dots & r_{3,n} & r_{2,n+1} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & r_{n,n} & r_{n,n+1} \\ 0 & \dots & \dots & 0 & 0 & r_{n+1,n+1} \end{bmatrix}.
\end{aligned}
$$

A partir de las identidades anterior, usted debe determinar los vectores $\mathbf{q}_i^{[n]}$ para $i \in \{1, 2, 3, \dots, n+1\}$ y la última columna de R_{n+1}.

Recuerde que $\mathbf{q}_i^{[n]} \in \mathbb{R}^{n+1}$, sin embargo $\mathbf{q}_i^{[n+1]} \in \mathbb{R}^{n+2}$. Es decir, tiene una diferencia de una unidad en la dimensión del espacio vectorial asociado. Lo importante es que esta diferencia puede ser resuelta convenientemete para obtener las primeras n columnas de la matriz $\widehat{Q}_{n+1}$.

Hint 1: You may consider useful this relationship: $\mathbf{q}_j^{[n+1]} = [(\mathbf{q}_j^{[n]})^T, 0]^T$, for $j \in \{1, 2, 3, \dots, n\}$ and ^{T} is the transpose operator.
Hint 2: Notice that the coefficients $r_{i,j}$ do not have a super index, this means that the only missing part of R_{n+1} is actually its last column!

(a) Construya la función `incrementalQR` que obtiene la factorización QR reducida de la matriz T_{n+1} a partir de la factorización reducida de T_n. El input de esta función corresponde a Q_n, R_n, a_{n+1}, b_{n+1}, y c_n. IMPORTANTE: Esta función se considerará correcta si obtiene adecuadamente la factorización QR reducida de T_{n+1} a partir de la factorización QR reducida de T_n, si se obtiene la factorización QR reducida directamente de la matriz T_{n+1} no se considerará correcta. La importancia de esto es que obtener la factorización QR reducida de este forma es que reduce significativamente la cantidad de operaciones elementales requerida.

(b) Considere los siguiente datos Q_1, R_1, a_2, b_2, y c_1. Indique con 5 decimales $r_{2,2}$.

```
Q1 = np.array([[0.67324134],[0.73942281]])
R1 = np.array([[0.81518093]])
a2 = 0.71518937
b2 = 0.54488318
c1= 0.4236548
```

2.5. ODE periodica

Considere la siguiente ecuación diferencial ordinaria:

$$
\begin{aligned}
y''(x) &= f(x), \\
y(x+0) &= y(x+2\,\pi).
\end{aligned}
$$

Considere que la solución que se busca es periódica.

(a) Proponga un algoritmo basado en diferencias finitas que obtenga una aproximación numérica de $y(x)$.

(b) ¿Existe alguna condición adicional que se necesite imponer? ¿Es única la solución?

(c) ¿Es posible asegurar que el máximo de la aproximación sea 1? Si fuera posible, explique cómo lo obtendrá.

(d) ¿Es posible asegurar que la integral de la aproximación en $[0, 2\,\pi]$ sea 0? Si fuera posible, explique cómo lo obtendrá.

2.6. Soluciona no nulas

Determine numéricamente todas las soluciones **no nulas** de la siguiente ecuación diferencial ordinaria considerando una discretización equiespaciada de $n + 1$ puntos:

$$\begin{aligned}((x^2+1)\,y'(x))' + y(x) &= \lambda\, y(x),\\ y(0) &= 0,\\ y(2\,\pi) &= 0.\end{aligned}$$

2.7. IVP integro-diferencial

Considere la siguiente ecuación integro-diferencial,

$$\begin{aligned}&\dot{y}(t) = \sin(y(t)) - \int_0^t y(s)\,\mathrm{d}s, \text{ para } t \in]0, 10[,\\ &y(0) = 1.\end{aligned} \tag{2}$$

Derivando la ecuación (2) con respecto a t se obtiene lo siguiente,

$$\ddot{y}(t) = \cos(y(t))\,\dot{y}(t) - y(t). \tag{3}$$

Adicionalmente se conoce que $\dot{y}(10) = 0.1$. **La pregunta consiste en resolver Ecuación (3) con las condiciones de borde $y(0) = 1$ y $\dot{y}(10) = 0.1$.**

(a) Proponga un algoritmo basado en el **método de Euler** para resolver el IVP en la ecuación (3).

(b) Proponga un algoritmo que permita encontrar la primera raíz de la función $y(t)$, es decir, la raíz positiva más cercana a 0. Notar que no se solicita encontrar el valor explícito, sino que explique completamente cómo abordaría el problema, indicando claramente cada componente del algoritmo, como se utilizan y como se inicializan. Hint: The first root is in the interval $[2, 4]$.

(c) Explique cómo afecta la elección de los parámetros definidos en su algoritmo propuesto en la pregunta 1.(a) con la exactitud de la raíz encontrada en la pregunta 1.(b). Asegúrese de explicar claramente cuales son las condiciones (parámetros) para aumentar la exactitud de la raíz versus el costo computacional asociado. Por favor use letra clara.

2.8. Newton+GMRes

Considere el siguiente sistemas de ecuaciones no-lineales:

$$\mathbf{F}_1(\mathbf{x}) = \mathbf{1}, \tag{4}$$

donde $\mathbf{F}_1(\mathbf{x}) : \mathbb{R}^{2\,n} \to \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^{2\,n}$, y $\mathbf{1}$ corresponde al vector de 1's de dimensión n. Lamentablemente no se puede resolver directamente con el método de Newton en alta dimensión porque tenemos una inconsistencia con el número de ecuaciones disponibles y la cantidad de incógnitas. En particular, tenemos el doble de incógnitas que la cantidad de ecuaciones. Para resolver este problema, se tiene el siguiente sistema de ecuaciones no-lineales complementario,

$$\mathbf{F}_2(\mathbf{x}) = \mathbf{0}, \tag{5}$$

donde $\mathbf{F}_2(\mathbf{x}) : \mathbb{R}^{2\,n} \to \mathbb{R}^n$, y $\mathbf{0}$ corresponde al vector de 0's de dimensión n. Ahora, considerando al mismo tiempo las ecuaciones (4) y (5), sí tenemos la misma cantidad de incógnitas que de ecuaciones. El paso natural sería aplicar el método de Newton en alta dimensión de la siguiente forma, por simplicidad solo incluiremos un paso,

$$\begin{aligned}\mathbf{x}_0 &= \text{“Initial guess”}.\\ J_{\mathbf{F}}(\mathbf{x}_0)\,\Delta\mathbf{x}_0 &= -\mathbf{F}(\mathbf{x}_0),\\ \mathbf{x}_1 &= \mathbf{x}_0 + \Delta\mathbf{x}_0,\end{aligned}$$

Notar que la explicación anterior corresponde para una función $\mathbf{F}(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^m$, pero no aplica directamente para $\mathbf{F}_1(\mathbf{x})$ ni para $\mathbf{F}_2(\mathbf{x})$ de manera independiente. También indicar que $J_{\mathbf{F}}(\mathbf{x}_0)$ corresponde a la matriz Jacobiana de $\mathbf{F}(\mathbf{x})$ evaluada en $\mathbf{x}_0$. Por otro lado, es importante notar que solo se tiene acceso a la evaluar las funciones $\mathbf{F}_1(\mathbf{x})$ y $\mathbf{F}_2(\mathbf{x})$, por lo que no es posible

obtener algebraicamente el gradiente de ninguna de las componentes de esas funciones. Para resolver este problema, se propone obtener una aproximación de primer orden del producto **matriz-vector** entre la matriz Jacobiana y un vector arbitrario $\mathbf{v}$ de la siguiente forma,

$$J_{\mathbf{F}}(\mathbf{x}_0)\,\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x}_0 + \varepsilon\,\mathbf{v}) - \mathbf{F}(\mathbf{x}_0)}{\varepsilon}, \qquad \varepsilon = 10^{-10}. \tag{6}$$

Entonces, la tarea a resolver consiste en determinar $\mathbf{x}_1$, para lo cual debe responder las siguientes preguntas:

(a) Explique cómo se debe definir $\mathbf{F}(\mathbf{x})$ matemáticamente con la información entregada.

(b) Explique cómo se debe definir $\mathbf{F}(\mathbf{x})$ computacionalmente con la información entregada. Considere que tiene acceso a las funciones `F1(x)` y `F2(x)` en Python, por lo cual las puede evaluar en cualquier vector `x`. Se sugiere escribir en "pseudo"-Python su explicación, de la manera más clara posible. "pseudo"-Python se refiere a que su explicación debe definir **explícitamente** todas las componentes necesarias para implementarse en Python.

(c) Explique cómo obtendrá $\mathbf{x}_1$. Su explicación debe ser muy clara y se debe conectar con la "implementación computacional en pseudo-Python" de la pregunta anterior. La única restricción que se tiene es que no se puede almacenar la matriz Jacobiana $J_{\mathbf{F}}(\mathbf{x}_0)$ en memoria, pero si se puede utilizar la aproximación (6) convenientemente.

2.9. Función Implícita

Considere la siguiente función implícita,

$$|x|^{\alpha} + |y|^{\alpha} = 1, \quad \text{para } \alpha \in \mathbb{R}. \tag{7}$$

La gráfica de la ecuación (7) corresponde a un círculo si $\alpha = 2$ y a un rombo si $\alpha = 1$. En particular, si definimos el perímetro o "largo de curva" (arclength) de la función anterior como $\mathcal{L}(\alpha)$, entonces $\mathcal{L}(2) = 2\,\pi$ dado que es un círculo con radio igual a 1 y $\mathcal{L}(1) = 4\,\sqrt{2}$ dado que es un rombo. Más aún, sabemos que $\mathcal{L}(\alpha) \leq 8$. El resultado anterior es posible obtenerlo solo considerando el primer cuadrante, es decir, cuando x e y son positivos, lo que implica que no necesitamos el uso del valor absoluto. En este caso, solo necesitamos multiplicar por 4 el "largo de curva" en el primer cuadrante para obtener el perímetro total. Dado lo anterior, definimos $\mathscr{L}(\alpha)$ como el "largo de curva" en el primer cuadrante, lo cual nos entrega la siguiente identidad:

$$\mathcal{L}(\alpha) = 4\,\mathscr{L}(\alpha).$$

Esto nos permite determinar $\mathcal{L}(\alpha)$ por medio de la computación de $\mathscr{L}(\alpha)$. A modo de recordatorio, se indica que el "largo de curva" de una función $f(x)$ entre $x = a$ y $x = b$ corresponde a la siguiente expresión,

$$S = \int_a^b \sqrt{1 + \left(f'(x)\right)^2}\,\mathrm{d}x.$$

(a) Proponga un algoritmo que utilice el **método de la secante** para obtener el valor de α tal que el perímetro total obtenido de la ecuación (7) sea β, donde sabemos que $4\,\sqrt{2} \leq \beta < 8$. Hint: If you need to perform a numerical quadrature, don't forget to define everything!

2.10. Newton+GMRes Again?

Considere el siguiente sistemas de ecuaciones no-lineales:

$$\mathbf{F}_1(\mathbf{x}) = \mathbf{1},$$

donde $\mathbf{F}_1(\mathbf{x}) : \mathbb{R}^{2\,n} \to \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^{2\,n}$, y $\mathbf{1}$ corresponde al vector de 1's de dimensión n. Lamentablemente no se puede resolver directamente con el método de Newton en alta dimensión porque tenemos una inconsistencia con el número de ecuaciones disponibles y la cantidad de incógnitas. En particular, tenemos el doble de incógnitas que la cantidad de ecuaciones. Para resolver este problema, se propone "congelar" un conjunto de incógnitas, es decir, se mantienen como variables las primeras n incógnitas y las segundas n incógnitas se fijan en un valor conocido, es decir considere la siguiente descomposición de vector de incógnitas $\mathbf{x}$,

$$\mathbf{x} = \langle \mathbf{y}, \mathbf{w} \rangle\,,$$

donde $\mathbf{y} \in \mathbb{R}^n$ y $\mathbf{w} \in \mathbb{R}^n$. Por simplicidad "congelaremos" el vector $\mathbf{w}$ y lo denotaremos como $\widetilde{\mathbf{w}}$. Ahora sí tenemos la misma cantidad de incógnitas que de ecuaciones. El paso natural sería aplicar el método de Newton en alta dimensión de la siguiente forma, por simplicidad solo incluiremos un paso,

$$\begin{aligned} \mathbf{y}_0 &= \text{"Initial guess"}. \\ J_{\mathbf{F}}(\mathbf{y}_0)\,\Delta\mathbf{y}_0 &= -\mathbf{F}(\mathbf{y}_0), \\ \mathbf{y}_1 &= \mathbf{y}_0 + \Delta\mathbf{y}_0, \end{aligned}$$

Notar que la explicación anterior corresponde para una función $\mathbf{F}(\mathbf{y}) : \mathbb{R}^n \to \mathbb{R}^n$, pero no aplica para $\mathbf{F}_1(\mathbf{x})$. También indicar que $J_{\mathbf{F}}(\mathbf{y}_0)$ corresponde a la matriz Jacobiana de $\mathbf{F}(\mathbf{y})$ evaluada en $\mathbf{y}_0$. Por otro lado, es importante notar que solo se tiene acceso a la evaluar la $\mathbf{F}_1(\mathbf{x})$, por lo que no es posible obtener algebraicamente el gradiente de ninguna de las componentes de esas funciones. Para resolver este problema, se propone obtener una aproximación de primer orden del producto **matriz-vector** entre la matriz Jacobiana y un vector arbitrario $\mathbf{v}$ de la siguiente forma,

$$\begin{aligned} J_{\mathbf{F}}(\mathbf{y}_0)\,\mathbf{v} &\approx \frac{\mathbf{F}(\mathbf{y}_0 + \varepsilon\,\mathbf{v}) - \mathbf{F}(\mathbf{y}_0)}{\varepsilon}, \\ \varepsilon &= 10^{-10}. \end{aligned} \tag{8}$$

Entonces, la tarea a resolver consiste en determinar $\mathbf{y}_1$, para lo cual debe responder las siguientes preguntas:

(a) Explique cómo se debe definir $\mathbf{F}(\mathbf{y})$ matemáticamente con la información entregada.

(b) Explique cómo se debe definir $\mathbf{F}(\mathbf{y})$ computacionalmente con la información entregada. Considere que tiene acceso a la función `F1(x)` en Python, por lo cual la puede evaluar en cualquier vector `x`. Se sugiere escribir en "pseudo"-Python su explicación, de la manera más clara posible. "pseudo"-Python se refiere a que su explicación debe definir **explícitamente** todas las componentes necesarias para implementarse en Python.

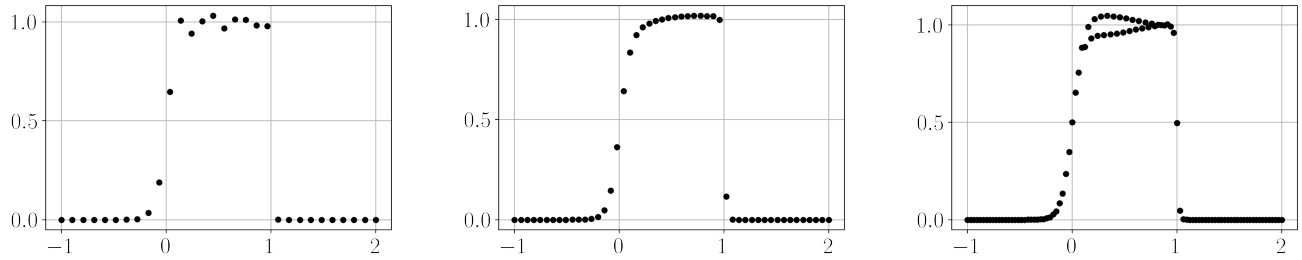
(c) Explique cómo obtendrá $\mathbf{y}_1$. Su explicación debe ser muy clara y se debe conectar con la "implementación computacional en pseudo-Python" de la pregunta anterior. La única restricción que se tiene es que no se puede almacenar la matriz Jacobiana $J_{\mathbf{F}}(\mathbf{y}_0)$ en memoria, pero si se puede utilizar la aproximación (8) convenientemente.

2.11. Graficando puntos

El procedimiento clásico que uno pudiera utilizar para graficar una función unidimensional en Python consiste en los siguientes pasos:

```
# Paso 1: Importar librerías adecuadas
import numpy as np
import matplotlib.pyplot as plt
# Paso 2: Definición de la función a graficar
f = lambda x: np.sin(x)*np.cos(x)
# Paso 3: Definición de la cantidad de puntos equiespaciados a graficar
n = 20
# Paso 4: Definición del intervalo a graficar y generación de los puntos equiespaciados
x = np.linspace(0, 10, n)
# Paso 5: Evaluación 'vectorial' de la función a graficar
y = f(x)
# Paso 6: Construcción de la gráfica
plt.figure()
plt.plot(x, y, '.')
plt.grid()
plt.show()
```

Lo cual generará una gráfica de la función de ejemplo $f(x) = \sin(x)\,\cos(x)$. Sin embargo, al construir la gráfica se tuvo que decidir la *cantidad de puntos* n a utilizar. En general, no existe una regla absoluta para definir n, sino uno debe modificar el valor de n hasta obtener una gráfica adecuada y la capacidad computacional lo permita. Por ejemplo, considere las 3 gráficas de una función $g(x)$ en la Figura 1. En la Figura 1a se observa el resultado cuando se consideraron 30 puntos equiespaciados, en la Figura 1b con 50 puntos y finalmente en la Figura 1c con 100 puntos, también equiespaciados. De la secuencia de gráficas indicadas, podemos incluso decir que no queda muy claro que la gráfica sea de la misma función, aunque sí lo son. Otra observación del experimento es que al considerar puntos equiespaciados en la abscisa puede inducir puntos consecutivos muy distantes en la ordenada.



(a) 30 puntos equiespaciados (b) 50 puntos equiespaciados (c) 100 puntos equiespaciados

Figura 1: Gráfica de $g(x)$ con distinta cantidad de puntos equiespaciados.

Una primera alternativa utilizada para reducir el efecto anterior es aumentar la cantidad de puntos de forma significativa en la abscisa, es decir, en el eje x, sin embargo esto podría traer consigo una representación numérica de la función con muchos puntos innecesario.

Para reducir la cantidad de puntos innecesarios y al mismo tiempo evitar que queden puntos consecutivos muy distantes en la ordenada se propone elegir puntos en la abscisa tal que el *arc-length* (largo de curva) entre puntos sea constante, es decir, l. Esto significa que los puntos en la abscisa, x_i para $i \in \{1, 2, \ldots, m-1\}$, deben satisfacer la siguientes condiciones:

$$\begin{aligned} l &= \int_{x_i}^{x_{i+1}} \sqrt{1 + \left(g'(x)\right)^2}\, \mathrm{d}x, \\ x_i &\in [a, b], \\ x_1 &= a, \\ x_m &= b, \\ x_i &< x_{i+1}. \end{aligned}$$

La única excepción corresponde al último intervalo que debe cumplir $\displaystyle\int_{x_{m-1}}^{b} \sqrt{1 + \left(g'(x)\right)^2}\, \mathrm{d}x < l$. Notar también que en este caso la cantidad de puntos m es desconocida, por lo tanto se debe determinar también.

Antes de abordar las preguntas, considere lo siguiente:

- Si necesita resolver un sistema de ecuaciones lineales, debe definir claramente la matriz y el lado derecho respectivo, e indicar qué algoritmo utilizaría y las razones para elegirlo.
- Si necesita construir una interpolación polinomial, solo defina los los puntos x_j y y_j respectivos y mencione el algoritmo a utilizar, no es necesario que lo describa completamente.
- Si necesita realizar una búsqueda de raíces en 1D, utilice el método de Newton, para lo cual es necesario que describa explícitamente el *input* requerido, es decir, si consideramos la siguiente firma `r=Newton1D(f,fp,r0)`, usted debe explicitar los parámetros de `Newton1D`, donde `f` corresponde a la implementación *callable* de la función $f(x)$ a la cual se le busca la raíz, `fp` corresponde corresponde a la implementación *callable* de la derivada de $f(x)$, es decir, $f'(x)$, `r0` corresponde al *initial guess*, y `r` corresponde a la raíz, es decir se cumple que `f(r)=0`.
- Si necesita integrar numéricamente alguna función, debe especificar claramente el algoritmo que utilizará, la función que integrará, la definición explícita de los nodos y de los pesos que se utilizarán. En este caso se sugiere definir como q la cantidad de nodos y pesos a utilizar, es decir debería ser un parámetro del algoritmo.
- Si necesita utilizar algún otro algoritmo, por favor incluir claramente todos los detalles respectivos y definir todos los *inputs* necesarios.

(a) Proponga un algoritmo que dado una función $g(x)$, su derivada $g'(x)$, un punto x_i, un largo de curva l y el extremo derecho del intervalo b, entregue el siguiente valor x_{i+1} y su evaluación $g(x_{i+1})$. *Recuerde que usted debe definir explícitamente todas las componentes del algoritmo para recibir el puntaje completo, llamar a funciones genéricas con parámetros abstractos o funciones no definidas, no reciben puntaje.*

(b) Implemente en Python el algoritmo anteriormente descrito utilizando adecuadamente la capacidad de vectorización de NumPy. Si fuera necesario, considere que tiene a su disposición la función `x,w=gaussian_nodes_and_weights(q)`, que recibe como input la cantidad de puntos `q` a utilizar en la cuadratura Gaussiana y retorna los nodos `x` y pesos `w` en el intervalo $[-1, 1]$.

```
'''
input:
xi  : (float)    Point x_i.
g   : (callable) Function g(x).
gp  : (callable) Function g'(x).
l   : (float)    Arclength target.
b   : (float)    Right limit of interval.
q   : (integer)  Number of points for numerical quadrature, if needed

output:
x_next  : (float) Point x_{i+1}.
g_next  : (float) g(x_{i+1}).
'''
def find_next_point(xi, g, gp, l, b, q):
    # Your own code.
    return x_next, g_next
```

2.12. ODEs acopladas

Considere el siguiente problema definido en $x \in [-1, 1]$:

$$\begin{aligned} u''(x) - \sin(v(x)) &= 0, \quad x \in]-1, 1[, \\ v''(x) + \cos(u(x)) &= 0, \quad x \in]-1, 1[, \\ u(-1) &= 1, \\ u'(1) &= 0, \\ v'(-1) &= 0, \\ v(1) &= 0. \end{aligned}$$

Antes de abordar las preguntas, considere lo siguiente:

- Si necesita resolver un sistema de ecuaciones lineales, debe definir claramente la matriz y el lado derecho respectivo, e indicar qué algoritmo utilizaría y las razones para elegirlo.
- Si necesita realizar una búsqueda de raíces en alta dimensión, utilice el método de Newton o Newton-Krylov.
- En caso de utilizar el método de Newton considere que tiene acceso a la siguiente función, `x_next=Newton(J,b,x_previous)`, donde `J` corresponde a la matriz a la matriz Jacobiana de $\mathbf{F}$ evaluada en `x_previous`, `b` corresponde a $-\mathbf{F}(\cdot)$ evaluado en `x_previous`, `x_previous` corresponde a la aproximación numérica de la raíz a ser mejorada en la siguiente iteración del Método de Newton, y `x_next` corresponde a vector solución luego de una iteración del método de Newton. Notar que esta función solo implementa 1 paso del método de Newton.
- En caso de utilizar el método de Newton-Krylov considere que tiene acceso a la siguiente función, `x_next=NewtonKrylov(Jw,b,x_previous)`, donde `Jw` corresponde a la función *callable* que recibe el vector $\mathbf{w}$ y retorna el producto entre la matriz Jacobiana de $\mathbf{F}$ evaluada en `x_previous` y el vector $\mathbf{w}$, `b` corresponde a $-\mathbf{F}(\cdot)$ evaluado en `x_previous`, `x_previous` corresponde a la aproximación numérica de la raíz a ser mejorada en la siguiente iteración del Método de Newton, y `x_next` corresponde a vector solución luego de una iteración del método de Newton-Krylov. Notar que esta función solo implementa 1 paso del método de Newton-Krylov.
- Si necesita resolver un IVP, considere que tiene acceso a `Y=solveIVP(fun,t_span,y0,t_eval)`, donde `fun` corresponde a la función escalar o vectorial que define el lado derecho del IVP o sistema dinámico correspondiente, esta debe recibir como parámetros el escalar t y el vector `y`; `t_span` corresponde a una 2-tupla de *floats* que definen el tiempo inicial y final del sistema dinámico; `y0` corresponde al vector de condición inicial; `t_eval` corresponde a los tiempos donde se obtendrá una aproximación numérica de la solución; y `Y` corresponde al vector solución para un IVP o una matriz solución para un sistema dinámico, en el caso del sistema dinámico, retornará tantas filas como funciones incógnitas existan y la misma cantidad de columnas que la dimensión de `t_eval`.
- Si necesita utilizar algún otro algoritmo, por favor incluir claramente todos los detalles respectivos y definir todos los *inputs* necesarios.

(a) Proponga un algoritmo que entregue una aproximación numérica de $u(x)$ y de $v(x)$ sobre una grilla equiespaciada de puntos, es decir, sobre los puntos definidos en el siguiente vector $\mathbf{x} = [x_1, \ldots, x_n]$. Por simplicidad considere que la cantidad de puntos

equiespaciados n es un parámetro. *Recuerde que usted debe definir explícitamente todas las componentes del algoritmo para recibir el puntaje completo, llamar a funciones genéricas con parámetros/funciones abstractos o no definidas, no reciben puntaje.*

(b) Implemente en Python el algoritmo anteriormente descrito utilizando adecuadamente la capacidad de vectorización de NumPy.

```
'''
input:
n   : (integer) Number of equalspaced points x_i.

output:
u  : (ndarray) Numerical approximation of u(x_i).
v  : (ndarray) Numerical approximation of v(x_i).
x  : (ndarray) The discretization points x_i
'''
def find_ui_vi(n):
    # Your own code.
    return u, v, x
```

2.13. ¡Indiana Jones necesita su ayuda!

El famoso arqueólogo ha encontrado una tabla de la cultura Sansana que contiene una serie de valores como una secuencia de pares ordenados (x_k, y_k) donde $x_0 = -1 < x_1 < x_2 < \cdots < x_n < 1 = x_{n+1}$ e $y_k = f(x_k)$ para un $f(x)$ desconocido y $k \in \{0, 1, 2, \ldots, n+1\}$. Al ver estos valores, el Dr. Jones se percata que la secuencia x_k no está equiespaciada. Para entender una pieza clave de la cultura Sansana, Indiana necesita resolver dos acertijos:

- Determinar una aproximación de la variación total de $f(x)$ definida como $\displaystyle\int_{-1}^{1} |f'(x)|\,\mathrm{d}x$ con el **método del Trapecio**, y
- Verificar si para el j-ésimo punto x_j, donde $j \in \{1, 2, \ldots, n\}$, es decir sin incluir los extremos x_0 y x_{n+1}, la siguiente ecuación diferencial,
$$\alpha\, f''(x_j) + \beta\, f'(x_j) + \gamma\, f(x_j) = 1, \qquad \text{donde } \alpha, \beta, \gamma \in \mathbb{R}, \tag{9}$$
es aproximada con una tolerancia δ, es decir $|\alpha\, f''(x_j) + \beta\, f'(x_j) + \gamma\, f(x_j) - 1| < \delta$. Notar que como tenemos datos no equiespaciados, la tradicional fórmula para aproximar la segunda derivada no funciona, en este caso se dispone de una variante, es decir, la Aproximación Generalizada de la Segunda Derivada (AGSD), la cual, para los puntos arbitrarios pero consecutivos, corresponde a: $\{(x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1})\}$,

$$\begin{aligned}\mathrm{AGSD}(x_{i-1}, y_{i-1}, x_i, y_i, x_{i+1}, y_{i+1}) = &-\frac{2\,y_{i-1}}{(x_{i-1} - x_i)\,(x_{i+1} - x_{i-1})} + \frac{2\,y_i}{(x_{i-1} - x_i)\,(x_{i+1} - x_{i-1})} \\ &+ \frac{2\,y_i}{(x_i - x_{i+1})\,(x_{i+1} - x_{i-1})} - \frac{2\,y_{i+1}}{(x_i - x_{i+1})\,(x_{i+1} - x_{i-1})}.\end{aligned}$$

A modo de **ejemplo**, si se evalúa AGSD$(\cdot)$ para los siguientes datos $\{(x_i - h, y_{i-1}), (x_i, y_i), (x_i + h, y_{i+1})\}$ obtenemos:

$$\mathrm{AGSD}(x_i - h, y_{i-1}, x_i, y_i, x_i + h, y_{i+1}) = \frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2},$$

que es nuestra aproximación tradicional de la segunda derivada sobre datos equiespaciados. Por otro lado, sí conocemos algunas aproximaciones de la primera derivada que nos pueden servir directamente en la aproximación numérica requerida.

(a) Detalle completamente cómo calcular la aproximación de la variación total de $f(x)$ definida como $\displaystyle\int_{-1}^{1} |f'(x)|\,\mathrm{d}x$ con el **método del Trapecio**

(b) Detalle completamente cómo determinar si para el j-ésimo punto x_j, donde $j \in \{1, 2, \ldots, n\}$, es decir sin incluir los extremos x_0 y x_{n+1}, la data disponible aproxima la ecuación diferencial (9) con una tolerancia δ, es decir $|\alpha\, f''(x_j) + \beta\, f'(x_j) + \gamma\, f(x_j) - 1| < \delta$. Considere que tiene a su disposición la función AGSD$(\cdot)$, para lo cual solo requiere pasarle los parámetros adecuados.

(c) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el procedimiento propuesto anteriormente. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `np.diff(x)`: Entrega la diferencia discreta del vector `x`, es decir, retorna un vector donde cada componente `i` corresponde a `x[i+1] - x[i]`. Notar que la dimensión del vector resultante se reduce en una unidad respecto al vector original, por ejemplo si `x=[1,2,4]` entonces `np.diff(x)=[1,2]`.
- `np.zeros((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `0`. En caso de que solo se entrega un número entero como *input*, es decir, `np.zeros(n)`, entonces retorna un vector de largo n con 0s en cada coeficiente.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- `AGSD(x1,y1,x2,y2,x3,y3)`: Entrega la Aproximación Generalizada de la Segunda Derivada, explicada anteriormente, en el punto `x2`. Notar que se usaron las variables `x1,y1,x2,y2,x3,y3` por simplicidad de la explicación pero corresponden a $x_{i-1}, y_{i-1}, x_i, y_i, x_{i+1}, y_{i+1}$, respectivamente.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input:
n         : (integer) Value of n, where the length of the sequence in the table is n + 2.
x_k       : (ndarray) Array with the values of x_k.
y_k       : (ndarray) Array with the values of y_k.
alpha     : (float) Value of alpha.
beta      : (float) Value of beta.
gamma     : (float) Value of gamma.
j         : (integer) Index where the differential equation is verified for the data.
delta     : (float) Tolerance.

output:
total_var : (float) Approximation of total variation of f(x).
check     : (bool)  Boolean value that indicates if the data approximates the differential
equation at x_j. If the data satisfied the tolerance, the output must be True, otherwise False.
'''
def jones(n,x_k,y_k,alpha,beta,gamma,j,delta):
    # Your own code.
    return total_var,check
```

2.14. Potencias de una matriz

Considere que usted quiere resolver el siguiente sistema de ecuaciones lineales:

$$A^l \mathbf{x} = \mathbf{b},$$

donde $l \in \mathbb{N}$ y es conocido, $A \in \mathbb{R}^{n\times n}$, lo que implica que $A^l \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^n$, y $n \in \mathbb{N}$ corresponde a la dimensión del problema. En particular, es importante notar que conocemos explícitamente $\mathbf{b}$ y l. Sin embargo, no tenemos acceso explícito a la matriz A. La matriz A se define como la suma de 2 matrices, es decir, $B_1 + B_2$. La matriz B_1 es *sparse* con a lo más k $(1 < k \ll n)$ elementos distintos de 0 por fila, y se tiene acceso explícito, es decir se puede almacenar en la memoria RAM. Por otro lado, la matriz B_2 no se puede almacenar en memoria y solo se puede acceder al producto entre la matriz B_2 y un vector $\mathbf{v}$ por medio de la función *callable* `SFP(v)`, donde *SFP* corresponde a la abreviación *Super Fast Product*. La función `SFP(v)` recibe como *input* un vector "`v`" y retorna el vector "`w`", el cual corresponde al producto entre B_2 y el vector `v`. Note que uno podría eventualmente reconstruir cada columna de la matriz B_2 por medio de la función `SFP(v)` y los vectores canónicos $\mathbf{e}_j$, es decir `SFP(e_j)` nos entrega la j-ésima columna de B_2, por lo que si uno ejecuta un ciclo desde $j = 1$ hasta $j = n$ puede recuperar cada columna de B_2. Lamentablemente la matriz B_2 es densa, es decir, la mayoría de sus coeficientes son distintos de 0, y, por restricciones de memoria, no se puede almacenar explícitamente B_2 en la memoria RAM. Esto significa que uno tiene solo acceso a B_1 y a `SFP(v)` para *interactuar* con A y resolver el problema presentado.

(a) Construya un algoritmo que permita obtener una aproximación $\widetilde{\mathbf{x}}$ de la solución $\mathbf{x}$ dado n, l, B_1, `SFP(v)` y $\mathbf{b}$, y que se asegure que el residuo $\left\|\mathbf{b} - A^l\,\widetilde{\mathbf{x}}\right\|_2 \leq \delta$. Si considera utilizar un algoritmo iterativo, se sugiere utilizar como *initial guess* `x0` el vector nulo y debe argumentar por qué el algoritmo es convergente.

En resumen se solicita:

(I) Construir un algoritmo que obtenga una aproximación numérica $\widetilde{\mathbf{x}}$ de $\mathbf{x}$.

(II) Usted conoce n (dimensión del problema), l (exponente de A^l), el vector $\mathbf{b}$, B_1 (matriz *sparse*, recuerde que las matrices *sparse* requiere muy poca memoria para ser almacenadas y operar con ellas es muy rápido), y se tiene a disposición `SFP(v)`, que para un vector input "`v`" entrega como resultado el vector "`w`" que corresponde al producto entre la matriz B_2 y el vector $\mathbf{v}$.

(III) La aproximación numérica $\widetilde{\mathbf{x}}$ debe satisfacer $\left\|\mathbf{b} - A^l\,\widetilde{\mathbf{x}}\right\|_2 \leq \delta$ y se debe explicar por qué se logrará esto en su propuesta de algoritmo.

(b) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el procedimiento propuesto anteriormente. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.sort(x)`: Entrega el arreglo unidimensional `x` ordenado de menor a mayor.
- `np.argsort(x)`: Entrega en el vector de salida `y` los índices de las entradas de `x` tal que al evaluar vectorialmente `x[y]` se obtienen las entradas ordenadas de menor a mayor.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `q,r=np.linalg.qr(A, mode='reduced')`: Factorización reducida QR de A.
- `x=np.linalg.solve(A,b)`: Resuelve $A\,\mathbf{x} = \mathbf{b}$ para $\mathbf{x}$. La matriz `A` debe estar almacenada explícitamente en memoria.
- `np.transpose(A)`: Entrega la matriz transpuesta de `A`, es decir `A^T`. Esto también se obtiene con la operación `A.T`.
- `np.ones((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `1`. En caso de que solo se entrega un número entero como *input*, es decir, `np.ones(n)`, entonces retorna un vector de largo n con 1s en cada coeficiente.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- `OneStepJacobiMethod(A,b,x0)`: Esta función aplica un paso del método de Jacobi aplicado al sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$. El *input* considera tener acceso explícito en memoria de la matriz `A`, `b` corresponde al vector del lado derecho y `x0` corresponde al *initial guess* del paso. Esta función retorna `x1`, el cual puede ser utilizado como *initial guess* para obtener `x2`, y así sucesivamente.
- `OneStepGaussSeidelMethod(A,b,x0)`: Esta función aplica un paso del método de Gauss-Seidel aplicado al sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$. El *input* considera tener acceso explícito en memoria de la matriz `A`, `b` corresponde al vector del lado derecho y `x0` corresponde al *initial guess* del paso. Esta función retorna `x1`, el cual puede ser utilizado como *initial guess* para obtener `x2`, y así sucesivamente.
- `GMRes_explicit_matrix(A,b,x0,threshold)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que se tiene acceso a la matriz A almacenada explícitamente en la memoria RAM, al vector $\mathbf{b}$, `x0` como *initial guess* y que retorna la aproximación numérica $\widetilde{\mathbf{x}}$ tal que se asegura que $\|\mathbf{b} - A\,\widetilde{\mathbf{x}}\| \leq$`threshold`, es decir menor o igual que el valor `threshold`.
- `GMRes_matrix_free(afun,b,x0,threshold)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que solo se tiene acceso a la función `afun`, que recibe como parámetro un vector `v` y retorna el vector `w` que corresponde al producto entre la matriz A y el vector `v`, al vector $\mathbf{b}$, `x0` como *initial guess* y que retorna la aproximación numérica $\widetilde{\mathbf{x}}$ tal que se asegura que $\|\mathbf{b} - A\,\widetilde{\mathbf{x}}\| \leq$`threshold`, es decir menor o igual que el valor `threshold`.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input:
n       : (integer) dimension of the problem.
l       : (integer) Integer value of l.
B_1     : (ndarray) Sparse matrix B_1. Notice this is defined for simplicity as ndarray.
```

```
SFP     : (callable) Procedure that receives the vector 'v' and
computes the 'fast' product between B_2 and the vector 'v'.
b       : (ndarray) Right-hand-side vector.
delta   : (double) Numerical threshold for computation of residual.

output:
xt       : (ndarray) Array with the values of the numerical approximation x_tilde.
'''
def find_x(n,l,B_1,SFP,b,delta):
    # Your own code.
    return xt
```

2.15. GMRes ayuda a Sylvester conjugado

Sea $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{C}^{n\times n}$, $C \in \mathbb{C}^{n\times n}$, y $Z \in \mathbb{C}^{n\times n}$. Considere la siguiente variante de la ecuación de Sylvester:

$$A\,Z + \mathrm{Conj}(Z)\,B = C \tag{10}$$

donde A es simétrica y definida positiva, B es Hermitiana, C es una matriz no nula, y $\mathrm{Conj}(\cdot)$ corresponde al operador de conjugación, i.e. si $z = x + \mathrm{i}\,y$ donde x e y son números reales y $\mathrm{i}^2 = -1$, entonces $\mathrm{Conj}(z) = x - \mathrm{i}\,y$ y en el caso matricial se aplica elemento a elemento. Para estandarizar la notación, se sugiere considerar $Z = X + \mathrm{i}\,Y$, $B = B_1 + \mathrm{i}\,B_2$, y $C = C_1 + \mathrm{i}\,C_2$, donde X, Y, B_1, B_2, C_1, y C_2 pertenecen a $\mathbb{R}^{n\times n}$. El desafío es proponer un algoritmo que solo dependa de aritmética real para encontrar Z, el cual se define al obtener X e Y, es decir, es suficiente obtener X e Y. **Recordar que no está permitido obtener la inversa explícita de ninguna de las matrices involucradas, lo que se debe hacer es resolver el sistema de ecuaciones lineales asociado**.

(a) Dada la restricción de que solo se puede utilizar aritmética real, re-escriba la ecuación (10) para que solo dependa de aritmética real. *Hint: You should get a linear system of equations where the unknowns are the matrices X and Y.*

(b) Proponga e implemente en Python un algoritmo para encontrar X y Y utilizando adecuadamente la capacidad de vectorización de NumPy. En la parte inferior se incluye solamente la función principal que se solicita, pero usted puede definir funciones adicionales si fuera necesario. Si es que fuera útil, considere que tiene a su disposición las funciones `GMRes1(M,x0,b,rel_res=1e-10)` y `GMRes2(afun,x0,b,rel_res=1e-10)`, las cuales retornan el vector $\mathbf{x}_a$, i.e. la aproximación numérica de la solución del sistema de ecuaciones lineales $M\,\mathbf{x} = \mathbf{b}$ tal que el residuo relativo es menor o igual a `rel_res`, i.e. $\|M\,\mathbf{x}_a - \mathbf{b}\|/\|\mathbf{b}\| \leq$`el_res` . Notar que `afun` es una función que recibe un vector $\mathbf{v}$ y retorna el producto matriz-vector $M\,\mathbf{v}$. *Hint: GMRes may be a good alternative.*

```
'''
input:
A   : (ndarray) Matrix "A".
B1  : (ndarray) Matrix "B1".
B2  : (ndarray) Matrix "B2".
C1  : (ndarray) Matrix "C1".
C2  : (ndarray) Matrix "C2".
rel_res : (float) Relative residue to be achieved.

output:
X  : (ndarray) Matrix "X".
Y  : (ndarray) Matrix "Y".
'''
def solve_Complex_Sylvester(A, B1, B2, C1, C2, rel_res):
    # Your own code.
    return X, Y
```

2.16. Matrix polynomials are back!

Considere que usted tiene acceso a una secuencia de imágenes en escala de grises denotadas por $Y_i \in \mathbb{R}^{n\times n}$, para $i \in \{1, 2, \ldots, m\}$. En particular, esta secuencia de imágenes Y_i corresponde al pre-procesamiento de las X_i por medio del filtro $Y = P(X)$, es decir, $Y_i = P(X_i)$.

En esta ocasión, el problema consiste en que a partir de una imagen pre-procesada Y_i se obtenga su imagen original X_i.

Para nuestra ventaja, se conoce la forma explícita del filtro $P(X)$, el cual corresponde a:

$$P(X) = U^{-1}\,(X_0 - X) + (X_0 - X)\,L + B,$$

donde las matrices U, L, X_0, y B son conocidas y no singulares. Adicionalmente se sabe que U es una matriz triangular superior y L es triangular inferior.

(a) Proponga un algoritmo que permita obtener la matriz X_i considerando que se conoce la matriz Y_i, U, X_0, L, y B. Recuerde que no está permitido (¡ni es recomendado!) la posibilidad de obtener la inversa de una matriz de forma explícita. Por completitud, se repite la ecuación que debe resolverse:

$$Y_i = U^{-1}\,(X_0 - X_i) + (X_0 - X_i)\,L + B.$$

(b) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo anteriormente propuesto para obtener la matriz X_i. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.zeros((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `0`. En caso de que solo se entrega un número entero como *input*, es decir, `np.zeros(n)`, entonces retorna un vector de largo n con 0s en cada coeficiente.
- `np.ones((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `1`. En caso de que solo se entrega un número entero como *input*, es decir, `np.ones(n)`, entonces retorna un vector de largo n con 1s en cada coeficiente.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- `P,L,U=palu(A)`: Entrega las matrices `P`, `L` y `U`, correspondiente a la factorización LU con pivoteo parcial de la matriz A, es decir $PA = LU$.

- `GMRes_explicit_matrix(A,b)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que se tiene acceso a la matriz A almacenada explícitamente en la memoria RAM y al vector $\mathbf{b}$. Por simplicidad se omite criterio de detención.
- `GMRes_matrix_free(afun,b)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que solo se tiene acceso a la función `afun`, que recibe como parámetro un vector `v` y retorna el vector `w` que corresponde al producto entre la matriz A y el vector `v`, y el vector $\mathbf{b}$. Por simplicidad se omite criterio de detención.
- `np.ravel(X,order='F')`: Entrega la versión vectorizada de la matriz `X` siguiendo la indexación *column-major*, es decir, el estilo de ordenamiento de Fortran. Por ejemplo:

```
>>> X=np.array([[1, 2], [3, 4]])
>>> np.ravel(X,order='F')
array([1, 3, 2, 4])
```

- `np.reshape(x,(n,n),order='F')`: Entrega el re-ordenamiento del vector `x` de dimensión n^2 en una matriz cuadrada de dimensión $n \times n$ siguiendo la indexación *column-major*, es decir, el estilo de ordenamiento de Fortran. Por ejemplo:

```
>>> x = np.array([1, 3, 2, 4])
>>> np.reshape(x,(2,2),order='F')
array([[1, 2],
       [3, 4]])
```

- `solve_triangular(A, b, lower=False)`: Esta función aplica *Backward Substitution* al sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$ para $A \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$ y $b \in \mathbb{R}^n$. Si el parámetro `lower` es definido como `True`, entonces aplica *Forward Substitution*. Notar que la matriz `A` debe ser cuadrada de dimensión `n` y el parámetro `b` puede ser un vector de dimensión `n` o una matriz de dimensión `n` por `m`, para $m \geq 1$. En el caso que el parámetro `b` sea una matriz, este aplica *Backward Substitution* o *Forward Substitution* columna a columna.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas.

Considere la siguiente firma:

```
'''
input:
Yi   : (ndarray) Matrix Yi.
X0   : (ndarray) Matrix X0.
U    : (ndarray) Matrix U.
L    : (ndarray) Matrix L.
B    : (ndarray) Matrix B.
n    : (int)     Each matrix has size n x n.

output:
Xi   : (ndarray) Matrix Xi.
'''
def compute_Xi(Yi,X0,U,L,B,n):
    # Your own code.
    return Xi
```

2.17. Curiosity wants to send data!

El famoso robot Curiosity que se encuentra explorando Marte, después de tantas horas de expedición, necesita transmitir los datos que ha recolectado de forma periódica. Para poder transmitir los datos almacenados, necesita buscar los puntos más altos de su entorno, por simplicidad se considerará un entorno unidimensional. El dato que está a nuestra disposición es la función $f : \mathbb{R} \to \mathbb{R}$, la cual describe la altura *relativa* respecto al robot. Notar que $f(x)$ puede tomar valores positivos, cuando está *sobre* el robot, o valores negativos cuando está *bajo* el robot. La variable x también describe la distancia *relativa* respecto al robot, por ejemplo si el valor de x es positivo, corresponde que está *frente* al robot, y si es negativo, significa que está *detrás* del robot.

En el último tiempo, el robot ha estado utilizando el mismo punto para transmitir datos, sin embargo la pérdida de paquetes en la transmisión ha superado el umbral permitido, por lo cual es necesario buscar otros puntos.

El desafío a enfrentar es obtener un conjunto de valores de x donde el robot pueda encontrar un vecindario distinto al actual para que, de forma posterior, pueda buscar el máximo local y así transmita los datos almacenados hasta ese punto.

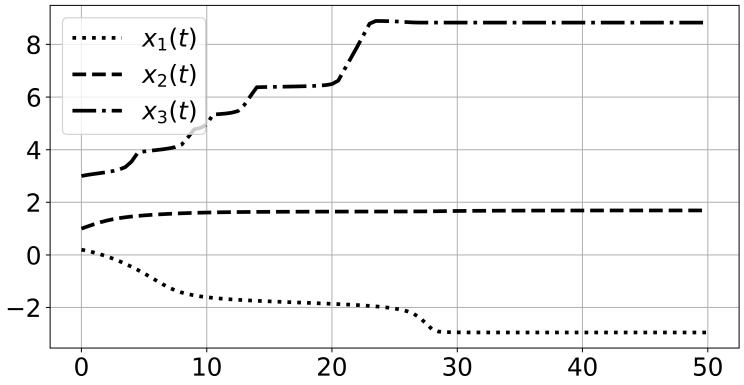
Para cumplir con el objetivo, se propone utilizar un algoritmo de maximización basado en partículas sobre la función $f(x)$, tal que busque distintos vecindarios de máximos locales. En este caso, se usarán 3 partículas $x_1(t)$, $x_2(t)$ y $x_3(t)$. La variación en el tiempo de las posiciones de las partículas viene dada por:

$$\begin{aligned}
\dot{x}_1 &= (1-\alpha)\, s\left(f'(x_1)\right) - \alpha\ \left(s(x_2 - x_1) + s(x_3 - x_1)\right) \\
\dot{x}_2 &= (1-\alpha)\, s\left(f'(x_2)\right) - \alpha\ \left(s(x_1 - x_2) + s(x_3 - x_2)\right) \\
\dot{x}_3 &= (1-\alpha)\, s\left(f'(x_3)\right) - \alpha\ \left(s(x_1 - x_3) + s(x_2 - x_3)\right)
\end{aligned}$$

donde $\alpha \in [0, 1]$, $s(x)$ es una función conocida y $f'(x)$, que es la derivada de $f(x)$, también es conocida.

(a) Proponga un algoritmo que permita obtener una aproximación de la trayectoria de las partículas $x_1(t)$, $x_2(t)$ y $x_3(t)$ para $t \in [0, T]$. En particular se necesita una aproximación numérica de $x_1(t_i)$, $x_2(t_i)$ y $x_3(t_i)$ en cada tiempo $t_i = i\,\dfrac{T}{N}$ para $i \in \{0, 1, 2, 3, \ldots, N\}$. Las posiciones iniciales de cada partícula son conocidas y vienen dadas por $x_{1,0}$, $x_{2,0}$ y $x_{3,0}$. Debe especificar claramente cada paso y componente de su algoritmo.

(b) Proponga un algoritmo que permita determinar el menor tiempo τ_k, para $k \in \{1, 2, 3\}$, donde la partícula x_k ha alcanzado un *estado estacionario numérico*. Una partícula x_k alcanza un *estado estacionario numérico* cuando $|\dot{x}_k(\tau_k)| < \varepsilon$ para algún $\tau_k \in \{t_0, t_1, \ldots, t_N\}$ y con $\varepsilon \ll 1$. En caso de que la partícula x_k no alcance el *estado estacionario numérico* en la grilla discreta de tiempos $\{t_0, t_1, \ldots, t_N\}$, entonces el algoritmo debe retornar $\tau_k = -1$. Por ejemplo, en la siguiente Figura se pueden observar las trayectorias de cada partícula en función del tiempo y aparentemente alcanzan un estado estacionario numérico para valores cercanos a $t = 50$, o incluso en valores menores, por ejemplo cerca de $t = 30$. Se indica *aparentemente* porque no se puede asegurar que $|\dot{x}_k(\tau)| < \varepsilon$ solo visualizando este gráfico, por lo que se debe verificar numéricamente.



(c) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) los procedimientos propuestos anteriormente con el menor error de aproximación posible. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `np.zeros((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `0`. En caso de que solo se entrega un número entero como *input*, es decir, `np.zeros(n)`, entonces retorna un vector de largo n con 0s en cada coeficiente.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- {`eulerMethod`,`RK2`,`RK4`}`(t0,T,N,y0,F)`: Implementa el {método de Euler, RK2, RK4} para IVP (y sistemas dinámicos) donde `y0` $\in \mathbb{R}^m$ corresponde al vector de estado en el tiempo inicial `t0`, `F`: $\mathbb{R}^m \to \mathbb{R}^m$ es la función del lado derecho del IVP (o sistema dinámico), `N` es la cantidad de *timesteps* y `T` es el tiempo final. Esta función retorna la discretización del tiempo en el intervalo `[t0,T]` con `N + 1` puntos equiespaciados y la aproximación numérica de la solución $\mathbf{y}(t_i)$ para $i \in \{0, \ldots, N\}$ en una matriz de dimensión $(N+1) \times m$. Notar que si la dimensión m del problema es 1 entonces es un IVP pero si $m > 1$ entonces es un sistema dinámico, considerando m un número entero. La implementación es

trasparente a la dimensión del problema, por lo cual se puede usar para ambos casos. **Notar que la notación de paréntesis de llaves es usada para reducir el espacio utilizado aprovechando que para las 3 alternativas de *solver* la descripción de los parámetros y *output* es la misma**, la única diferencia es lo que hacen los *solvers* como tal.

- `Newton1D(f,fp,x0)`: Implementa el método de Newton en 1D que entrega la aproximación de la raíz de `f`. Esta función recibe como parámetros la función `f`, la derivada `fp` de `f` y el *initial guess* `x0`. Por simplicidad se omite criterio de detención.
- `Bisection(a,b,f)`: Implementa el método de la Bisección para búsqueda de la raíz de `f` en el intervalo [`a`,`b`]. Por simplicidad se omite criterio de detención.
- `FPI(g,x0)`: Implementa una iteración de punto fijo para la función `g(x)` que se inicializa en `x0`. Por simplicidad se omite criterio de detención.
- `Secant(f,x0,x1)`: Implementa el método de la Secante para la búsqueda de la raíz de `f`, y requiere los *initial guesses* `x0` y `x1`. Por simplicidad se omite criterio de detención.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas.

Considere la siguiente firma:

```
'''
input:
x10   : (float)    Initial condition for x_1.
x20   : (float)    Initial condition for x_2.
x30   : (float)    Initial condition for x_3.
alpha : (float)    Value of alpha.
s     : (callable) Function s(x).
fp    : (callable) Derivative of the function f.
t0    : (float)    Initial time of trajectories.
T     : (float)    Final time of trajectories.
eps   : (float)    Tolerance related to the numerical stationary state.
N     : (int)      Number of timesteps.

output:
t     : (ndarray) Discretization of time in the interval [t0,T] with (N + 1) equidistant points.
x_1   : (ndarray) Trajectory of particle x_1.
x_2   : (ndarray) Trajectory of particle x_2.
x_3   : (ndarray) Trajectory of particle x_3.
tau_1 : (float)   First time when particle x_1 reaches a numerical stationary state.
tau_2 : (float)   First time when particle x_2 reaches a numerical stationary state.
tau_3 : (float)   First time when particle x_3 reaches a numerical stationary state.
'''
def particles(x10,x20,x30,alpha,s,fp,t0,T,eps,N):
    # Your own code.
    return t,x_1,x_2,x_3,tau_1,tau_2,tau_3
```

2.18. Bessel is here

Considere la ecuación diferencial de Bessel,

$$x^2\,y''(x) + x\,y'(x) + (x^2 - m^2)\,y(x) = 0, \quad \text{para este caso } x \in]1, 2\pi[, \tag{11}$$

donde m corresponde a un parámetro y representa el orden de la función de Bessel asociada. Por simplicidad se considera que $m \in \mathbb{N}_0$. Se define como $J_m(x)$ la solución de la ecuación diferencial ordinaria anterior que no es singular en el origen. Para el caso de que m sea entero, es decir el caso en estudio, se conoce la siguiente representación integral,

$$J_m(x) = \frac{1}{\pi}\int_0^{\pi} \cos\left(m\,\tau - x\,\sin(\tau)\right)\,\mathrm{d}\tau.$$

Adicionalmente considere la siguiente ecuación diferencial en el mismo intervalo espacial anterior,

$$(x^2 + 4\,\sin(x))\,w''(x) + x\,w'(x) + (x^2 - 1)\,w(x) = 0, \quad \text{para } x \in]1, 2\,\pi[,$$
$$w(1) = J_1(1) \approx 0.4400505857449335159596822037189149131273723019927 7\ldots,$$
$$w(2\,\pi) = J_1(2\,\pi) \approx -0.2123825300763622028586556710849962272560258536 9\ldots.$$

La cual es una perturbación de la Ecuación (11) considerando $m = 1$, dado que se agrega el término $4\,\sin(x)$ multiplicando a $w''(x)$, lo cual genera una ecuación diferencial distinta a la Ecuación (11). Por lo tanto generará un solución distinta, pero ¿Qué tan distinta?

El problema en estudio consiste en comparar ambas aproximaciones numéricas, es decir, comparar la función de Bessel $J_1(x)$ y $w(x)$ en el intervalo $x \in [1, 2\,\pi]$. La comparación se realizará por medio de la computación del vector diferencia $\mathbf{d} \in \mathbb{R}^M$, $M \in \mathbb{N}$ y $M > 10$. El vector $\mathbf{d}$ se define como:

$$\mathbf{d}_j = J_1(x_j) - w(x_j), \quad x_j = 1 + j\frac{2\pi - 1}{M}, \quad \text{para } j \in \{0, 1, 2, \ldots, M\}.$$

En particular se considera que tanto $J_1(x_j)$ como $w(x_j)$ deberán ser aproximados numéricamente y queda a su libertad el como *obtenerlos*.

(a) Proponga un algoritmo que permita obtener una aproximación de $\mathbf{d}_j = J_1(x_j) - w(x_j)$, donde $x_j = 1 + j\frac{2\pi - 1}{M}$, para $j \in \{0, 1, 2, \ldots, M\}$. En particular, usted debe estimar numéricamente $J_1(x_j)$ y $w(x_j)$ para $j \in \{0, 1, 2, \ldots, M\}$. Notar que pueden existir varias alternativas, y que podrían ser complementarias. Seleccione la alternativa que más le acomode, pero asegúrese de obtener ambos valores.

(b) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo propuesto anteriormente. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `np.zeros((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `0`. En caso de que solo se entrega un número entero como *input*, es decir, `np.zeros(n)`, entonces retorna un vector de largo n con 0s en cada coeficiente.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- `{eulerMethod,RK2,RK4}(t0,T,N,y0,F)`: Implementa el {método de Euler, RK2, RK4} para IVP (y sistemas dinámicos) donde `y0` $\in \mathbb{R}^m$ corresponde al vector de estado en el tiempo inicial `t0`, `F`: $\mathbb{R}^m \to \mathbb{R}^m$ es la función del lado derecho del IVP (o sistema dinámico), `N` es la cantidad de *timesteps* y `T` es el tiempo final. Esta función retorna la discretización del tiempo en el intervalo `[t0,T]` con `N + 1` puntos equiespaciados y la aproximación numérica de la solución $\mathbf{y}(t_i)$ para $i \in \{0, \ldots, N\}$ en una matriz de dimensión $(N + 1) \times m$. Notar que si la dimensión m del problema es 1 entonces es un IVP pero si $m > 1$ entonces es un sistema dinámico, considerando m un número entero. La implementación es trasparente a la dimensión del problema, por lo cual se puede usar para ambos casos. **Notar que la notación de paréntesis de llaves es usada para reducir el espacio utilizado aprovechando que para las 3 alternativas de *solver* la descripción de los parámetros y *output* es la misma**, la única diferencia es lo que hacen los *solvers* como tal.
- `Newton1D(f,fp,x0)`: Implementa el método de Newton en 1D que entrega la aproximación de la raíz de `f`. Esta función recibe como parámetros la función `f`, la derivada `fp` de `f` y el *initial guess* `x0`. Por simplicidad se omite criterio de detención.
- `Bisection(a,b,f)`: Implementa el método de la Bisección para búsqueda de la raíz de `f` en el intervalo [`a`,`b`]. Por simplicidad se omite criterio de detención.
- `FPI(g,x0)`: Implementa una iteración de punto fijo para la función `g(x)` que se inicializa en `x0`. Por simplicidad se omite criterio de detención.
- `Secant(f,x0,x1)`: Implementa el método de la Secante para la búsqueda de la raíz de `f`, y requiere los *initial guesses* `x0` y `x1`. Por simplicidad se omite criterio de detención.
- `P,L,U=palu(A)`: Entrega las matrices `P`, `L` y `U`, correspondiente a la factorización LU con pivoteo parcial de la matriz A, es decir $PA = LU$.

- `GMRes_explicit_matrix(A,b)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que se tiene acceso a la matriz A almacenada explícitamente en la memoria RAM y al vector $\mathbf{b}$. Por simplicidad se omite criterio de detención.
- `GMRes_matrix_free(afun,b)`: Esta función implementa el algoritmo GMRes para resolver un sistema de ecuaciones lineales de la forma $A\,\mathbf{x} = \mathbf{b}$ considerando que solo se tiene acceso a la función `afun`, que recibe como parámetro un vector `v` y retorna el vector `w` que corresponde al producto entre la matriz A y el vector `v`, y el vector $\mathbf{b}$. Por simplicidad se omite criterio de detención.
- `np.ravel(X,order='F')`: Entrega la versión vectorizada de la matriz `X` siguiendo la indexación *column-major*, es decir, el estilo de ordenamiento de Fortran. Por ejemplo:

  ```
  >>> X=np.array([[1, 2], [3, 4]])
  >>> np.ravel(X,order='F')
  array([1, 3, 2, 4])
  ```

- `np.reshape(x,(n,n),order='F')`: Entrega el re-ordenamiento del vector `x` de dimensión n^2 en una matriz cuadrada de dimensión $n \times n$ siguiendo la indexación *column-major*, es decir, el estilo de ordenamiento de Fortran. Por ejemplo:

  ```
  >>> x = np.array([1, 3, 2, 4])
  >>> np.reshape(x,(2,2),order='F')
  array([[1, 2],
         [3, 4]])
  ```

- `solve_triangular(A, b, lower=False)`: Esta función aplica *Backward Substitution* al sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$ para $A \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$ y $b \in \mathbb{R}^n$. Si el parámetro `lower` es definido como `True`, entonces aplica *Forward Substitution*. Notar que la matriz `A` debe ser cuadrada de dimensión `n` y el parámetro `b` puede ser un vector de dimensión `n` o una matriz de dimensión `n` por `m`, para $m \geq 1$. En el caso que el parámetro `b` sea una matriz, este aplica *Backward Substitution* o *Forward Substitution* columna a columna.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas.

Considere la siguiente firma:

```
'''
input:
M     : (int) Number of discretization intervals, which implies M+1 points.
J1_1  : (float) Numerical value of J_1(1)
J1_2pi: (float) Numerical value of J_1(2\,\pi)

output:
d     : (ndarray) Difference vector d_j=J_1(x_j)-w(x_j)
'''
def compareBessel(M, J1_1, J1_2pi):
    # Your own code.
    return d
```

2.19. Criptomonedas

Una de las criptomonedas más interesantes del último tiempo es la *Ethereum*. En la Tabla 1 usted puede observar **algunos** de los valores más altos y más bajos alcanzados por esta moneda en un día específico. Un indicador interesante que los operadores de bolsa prestan atención, es el `Spread`, el cual consiste en la diferencia entre el valor más alto y más bajo de la moneda en un determinado tiempo, es decir $\text{Spread}(t) = \text{High}(t) - \text{Low}(t)$. Particularmente, los operadores de bolsa están interesados en obtener un valor medio ponderado del `Spread` en una ventana de tiempo, de tal forma que les permita decidir su próxima estrategia financiera. El problema surge cuando los operadores de bolsa notan que los datos con los cuales necesitan trabajar están incompletos. Por ejemplo, en la Tabla 1, hay datos con valor -1, lo cual significa que no están disponibles, es decir, es un *missing value*. En resumen, los operadores de bolsa se deben enfrentar a dos problemas: (i) dada las series $y_{\text{low}}(t_k)$ e $y_{\text{high}}(t_k)$ con $k \in \{0, \ldots, N\}$, se debe completar los datos faltantes que se necesitan y (ii) calcular el valor medio ponderado del `Spread`, que viene dada por la siguiente expresión:

$$I_{\text{spread}} = \frac{\displaystyle\int_{t_0}^{t_N} \text{Spread}(t)\,\omega(t)\,\mathrm{d}t}{\displaystyle\int_{t_0}^{t_N} \omega(t)\,\mathrm{d}t}$$

Fecha	High	Low
27-nov-23	2069.14	2002.88
26-nov-23	2094.10	2038.60
25-nov-23	2091.34	-1
24-nov-23	2132.48	2061.00
23-nov-23	-1	2041.46
22-nov-23	2089.51	1933.16
21-nov-23	2035.04	-1
20-nov-23	2066.41	1996.04
19-nov-23	-1	1944.90
18-nov-23	1971.46	1921.06
17-nov-23	1990.05	-1
16-nov-23	2088.66	1940.57
15-nov-23	2061.99	1968.77
⋮	⋮	⋮

Tabla 1: Tabla con *algunos* valores de la criptomoneda Ethereum.

donde $\omega(t) > 0$ para $t \in [t_0, t_N]$. Considere que los datos faltantes solamente pueden aparecer en los tiempos t_j para $j \in \{2, \ldots, N-2\}$.

(a) Proponga un algoritmo que permita aproximar para algún tiempo t_k el *missing value* de $\texttt{Low}(t_k)$ y/o $\texttt{High}(t_k)$ considerando los dos valores anteriores, en los tiempos t_{k-1} y t_{k-2}, y los dos valores posteriores, en los tiempos t_{k+1} y t_{k+2}. Las aproximaciones, tanto para $\texttt{Low}(t_k)$ como para $\texttt{High}(t_k)$, se deben realizar con la función $p(t) = a + b\,(t - t_k) + c\,(t - t_k)^2$, que luego se evalúa en t_k para obtener el valor faltante en $\texttt{Low}(t_k)$ o $\texttt{High}(t_k)$, respectivamente. Luego, para un siguiente *missing value*, se realiza lo mismo pero con sus propios datos vecinos. *Hint: Notice that we have two time series of data but it seems they are very close problems, actually, they look like identical twins!*

(b) Proponga un algoritmo que permita calcular el valor ponderado del `Spread`, es decir I_{spread}, considerando las series $y_{\text{low}}(t_k)$ e $y_{\text{high}}(t_k)$ con $k \in \{0, \ldots, N\}$ y la función de ponderación $\omega(t)$. Por completitud se repite la definición I_{spread},

$$I_{\text{spread}} = \frac{\displaystyle\int_{t_0}^{t_N} \text{Spread}(t)\,\omega(t)\,\mathrm{d}t}{\displaystyle\int_{t_0}^{t_N} \omega(t)\,\mathrm{d}t}.$$

(c) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) los procedimientos propuestos anteriormente. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos y condicionales propios de Python:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `np.zeros((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `0`. En caso de que solo se entrega un número entero como *input*, es decir, `np.zeros(n)`, entonces retorna un vector de largo n con 0s en cada coeficiente.
- `np.ones((n_rows, n_cols))`: Entrega un `ndarray` de dimensión `n_rows`×`n_cols` donde cada coeficiente es igual a `1`. En caso de que solo se entrega un número entero como *input*, es decir, `np.ones(n)`, entonces retorna un vector de largo n con 1s en cada coeficiente.
- `np.concatenate((a1,a2,...),axis}=0)`: Entrega un `ndarray` el cual permite concatenar una secuencia de `ndarray` a lo largo de un eje existente `axis` que por defecto es `0` (concatena a lo largo de las filas). Por ejemplo:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7,8]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
```

```
              [5, 6],
              [7,8]])
       >>> np.concatenate((a, b), axis=1)
       array([[1, 2, 5, 6],
              [3, 4, 7, 8]])
```

- `q,r=np.linalg.qr(A, mode='reduced')`: Factorización reducida QR de A.
- `x=np.linalg.solve(A,b)`: Resuelve $A\,\mathbf{x} = \mathbf{b}$ para $\mathbf{x}$. La matriz `A` debe estar almacenada explícitamente en memoria.
- `trapezoidDiscrete(t, y, a, b)`: Esta función entrega el resultado de aplicar el método del trapecio para integración numérica sobre las evaluaciones discretas de la abscisa t_i y ordenada y_i almacenadas en los vectores `t` e `y`, respectivamente, `a` corresponde al límite inferior de la integral definida y `b` corresponde al límite superior de la integral definida.

Notar que al momento de implementar usted debe decidir qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
   ,,,
   input:
   N        : (integer)  that defines the N+1 timesteps.
   t        : (ndarray)  (N+1)-dimensional vector data $\mathbf{t}$.
   i_low    : (ndarray)  indices where there is a missing value in the time series Low.
              For instance, in Table 1, i_low = [2,6,10,...].
   i_high   : (ndarray)  indices where there is a missing value in the time series High.
              For instance, in Table 1, i_high = [4,8,...].
   w        : (callable) weighting function w(t).
   y_low    : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{low}$.
   y_high   : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{high}$.

   output:
   i_spread        : (float)   weighted average value of the Spread.
   y_low_complete  : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{low}$ with the
                               missing values approximated.
   y_high_complete : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{high}$ with the
                               missing values approximated.
   ,,,
   def compute_spread(N,t,i_low,i_high,w,y_low,y_high):
       # Your own code.
       return i_spread, y_low_complete, y_high_complete

,,,
input:
N        : (integer)  that defines the N+1 timesteps.
t        : (ndarray)  (N+1)-dimensional vector data $\mathbf{t}$.
i_low    : (ndarray)  indices where there is a missing value in the time series Low.
           For instance, in Table 1, i_low = [2,6,10,...].
i_high   : (ndarray)  indices where there is a missing value in the time series High.
           For instance, in Table 1, i_high = [4,8,...].
w        : (callable) weighting function w(t).
y_low    : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{low}$.
y_high   : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{high}$.

output:
i_spread        : (float)   weighted average value of the Spread.
y_low_complete  : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{low}$ with the
                            missing values approximated.
y_high_complete : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{high}$ with the
                            missing values approximated.
,,,
def compute_spread(N,t,i_low,i_high,w,y_low,y_high):
```

```
    # Your own code.
    return i_spread, y_low_complete, y_high_complete
```

3. Desarrollos de referencia

¡MUY IMPORTANTE!

En esta selecciones de ejercicios se incluyen resoluciones teóricas (desarrollo) y computacionales (código), y también algunas que solo contienen la parte teórica o de código, y para esas respuestas se sugiere trabajar en la implementación computacional o teórica asociada.

3.1. Desarrollo Pregunta "Mínimos IVP"

Código principal:

```
import numpy as np
import scipy as sp
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def phi(x):
    return x/(np.abs(x)+0.05)

def f_Particles(t, y, w, alpha1, alpha2, beta1, beta2, delta, gamma):
    A = np.array([[1,1,0,0,1],[0,0,1,1,1]]).T
    RHS = np.zeros(5)
    x1 = y[0]
    x2 = y[1]
    RHS[0] = phi(x2-x1)+np.dot(alpha1,phi(w-x1))-gamma*x1
    RHS[1] = phi(x2-x1)+np.dot(beta1,phi(w-x1))-gamma*x1
    RHS[2] = phi(x1-x2)+np.dot(alpha2,phi(w-x2))-gamma*x2
    RHS[3] = phi(x1-x2)+np.dot(beta2,phi(w-x2))-gamma*x2
    RHS[4] = np.dot(delta,phi(w-0.5*(x1+x2)))+gamma*(x1+x2)
    y_dot = np.linalg.solve(A.T @ A, np.dot(A.T,RHS))
    return y_dot

np.random.seed(0)
n = 5
w = np.linspace(1.5,2,n)
sigma = 0.72
alpha1 = np.random.normal(0,sigma,n)
alpha2 = np.random.normal(0,sigma,n)
beta1  = np.random.normal(0,sigma,n)
beta2  = np.random.normal(0,sigma,n)
delta  = np.random.normal(0,sigma,n)
gamma  = 0.1
w = np.array( [1.5  ,1.625,1.75 ,1.875,2.   ] )
alpha1 = np.array( [1.2701176891,0.28811319  ,0.7046913486,1.6134431034,1.3446417529] )
alpha2 = np.array( [-0.7036400735, 0.6840636606,-0.10897719  ,-0.0743175733, 0.2956309214] )
beta1 = np.array( [0.1037113712,1.047076925 ,0.5479471621,0.0876060119,0.3195815276] )
beta2 = np.array( [ 0.2402455157, 1.0757369327,-0.1477139499, 0.2254087452,-0.6149489323] )
delta = np.array( [-1.8381526674, 0.4706053887, 0.6223940632,-0.5343588147, 1.6342233293] )
gamma = 0.1

out = solve_ivp(f_Particles, [0,10], [1.6,1.8], t_eval=np.linspace(0,10,1000), args=(w, alpha1, alpha2, beta1, b
x1 = out.y[0,:]
```

```
x2 = out.y[1,:]
t  = out.t
```

Se agrega una función para graficar el resultado a modo referencial:

```
plt.figure(figsize=(10,10))
plt.plot(t,x1,'-',label=r'$x_1$')
plt.plot(t,x2,'-',label=r'$x_2$')
for i in range(n):
    plt.plot(t,w[i]+t*0,'-')
plt.grid(True)
plt.legend(loc='best')
plt.show()
```

3.2. Desarrollo Pregunta "Newton+GMRes"

Este desarrollo corresponde a la pregunta en Apartado 2.8.

(a) Se define $\mathbf{F}(\mathbf{x})$ de la siguiente forma:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \mathbf{F}_1(\mathbf{x}) - \mathbf{1} \\ \mathbf{F}_2(\mathbf{x}) \end{bmatrix}.$$

En este caso es muy importante restar el vector $\mathbf{1}$ a $\mathbf{F}_1(\mathbf{x})$ en las primeras n componentes de la definición de $\mathbf{F}(\mathbf{x})$, de otra forma no se encontrará la raíz solicitada.

(b) La implementación en Python queda de la siguiente forma:

```
def F(x):
    w = np.zeros(2*n)
    w[:n] = F1(x)-np.ones(n)
    w[n:-1] = F2(x)
    return w
```

(c) Para encontrar $\mathbf{x}_1$ se necesita resolver el siguiente sistema de ecuaciones lineales,

$$J_{\mathbf{F}}(\mathbf{x}_0)\,\Delta\mathbf{x}_0 = -\mathbf{F}(\mathbf{x}_0).$$

Según lo indicado, no se tiene acceso explícito a la matriz $J_{\mathbf{F}}(\mathbf{x}_0)$, sin embargo se explica que se puede acceder al producto matriz-vector entre $J_{\mathbf{F}}(\mathbf{x}_0)$ y un vector arbitrario $\mathbf{v}$. Para poder utilizar esta información, se procederá a implementar lo indicado en Python,

```
def afun(v):
    epsilon = 1e-10
    w = (F(x0+epsilon*v)-F(x0))/epsilon
    return w
```

Notar que se considera que se tiene acceso a `x0` el cual corresponde a $\mathbf{x}_0$ indicado en la pregunta y `F(x)` es la función implementada en la pregunta anterior. Ahora que tenemos acceso a `afun`, notamos que el único algoritmo que conocemos (existen otros) que puede utilizar `afun` es GMRes. Recuerde que tiene la restricción que no puede almacenar en memoria la matriz Jacobiana, pero sí podemos utilizar `afun`. Entonces, podemos utilizar GMRes de la siguiente forma conectandolo con el problema que debemos resolver,

```
b = -F(x0)
Delta_x_i = GMRes(afun,b)
x1 = x0 + Delta_x_i
```

Por lo tanto, hemos logrado determinar $\mathbf{x}_1$, que es exactamente lo solicitado.

3.3. Desarrollo Pregunta "Función Implícita"

Este desarrollo corresponde a la pregunta en Apartado 2.9.

(I) Se procederá a despejar $y(x)$ considerando que $x \geq 0$ y $y \geq 0$ dado que se considera el primer cuadrante,

$$\begin{aligned} |x|^{\alpha} + |y|^{\alpha} &= 1, \\ x^{\alpha} + y^{\alpha} &= 1, \\ y^{\alpha} &= 1 - x^{\alpha}, \\ y(x) &= (1 - x^{\alpha})^{1/\alpha}. \end{aligned}$$

(II) Se determina la derivada de $y(x)$,

$$y'(x) = -(1 - x^{\alpha})^{\frac{1}{\alpha}-1} x^{\alpha-1}.$$

(III) Se define el "largo de curva" en el primer cuadrante para la función implícita,

$$\begin{aligned} \mathscr{L}(\alpha) &= \int_0^1 \sqrt{1 + (y'(x))^2}\, \mathrm{d}x, \\ &= \int_0^1 \underbrace{\sqrt{1 + \left(-(1 - x^{\alpha})^{\frac{1}{\alpha}-1} x^{\alpha-1}\right)^2}}_{\widehat{f}(x)}\, \mathrm{d}x. \end{aligned}$$

(IV) Se aproxima numéricamente la integral $\mathscr{L}(\alpha)$ con el método del trapecio, la cual se denota por $\mathscr{L}_n(\alpha)$. Para realizar la aproximación se requieren las siguientes definiciones,

$$\begin{aligned} &n : \text{Cantidad de intervalos a utilizar en la integración numérica.} \\ &h = \frac{1-0}{n}, \quad \Delta x \text{ utilizado.} \\ &x_i = h\, i,\ i \in \{0, 1, 2, \ldots, n\}, \quad \text{nodos de la integración numérica.} \\ &w_i = \begin{cases} \dfrac{h}{2}, & \text{si } i \in \{0, n\}, \\ h, & \text{si } i \in \{1, 2, \ldots, n-1\}. \end{cases} \\ &\mathscr{L}(\alpha) \approx \mathscr{L}_n(\alpha) = \sum_{i=0}^{n} w_i\, \widehat{f}(x_i), \quad \text{Aproximación numérica de la integral por el método del trapecio.} \end{aligned}$$

(V) Se define la función $p(\alpha) = 4\,\mathscr{L}_n(\alpha) - \beta$. A esta función se le buscará una raíz para determinar el valor de α tal que cumpla que tenga el perímetro β. A continuación, se define la información necesaria para aplicar el método de la secante,

$$\begin{aligned} \alpha_0 &= 1, \\ \alpha_1 &= 2, \\ \alpha_{k+2} &= \alpha_{k+1} - \frac{p(\alpha_{k+1})\ (\alpha_{k+1} - \alpha_k)}{p(\alpha_{k+1}) - p(\alpha_k)}, \quad k \in \{0, 1, 2, \ldots\}, \end{aligned}$$

donde la iteración se detiene cuando $|\alpha_{K+2} - \alpha_{K+1}| \leq 10^{-10}$, para algún $K \leq 10^6$. Si se cumple el máximo valor de K, se retorna α_{K+2}.

3.4. Desarrollo Pregunta "Newton+GMRes Again?"

Este desarrollo corresponde a la pregunta en Apartado 2.10.

(a) Se define $\mathbf{F}(\mathbf{y})$ de la siguiente forma:

$$\mathbf{F}(\mathbf{y}) = \mathbf{F}_1(\langle \mathbf{y}, \widetilde{\mathbf{w}} \rangle) - \mathbf{1}.$$

En este caso es muy importante restar el vector $\mathbf{1}$ a $\mathbf{F}_1(\mathbf{x})$ y considerar la "parte conocida" de $\mathbf{x}$, la cual corresponde a las componentes $n+1$ hasta $2\,n$ denotada por $\widetilde{\mathbf{w}}$.

(b) La implementación en Python queda de la siguiente forma:

```
def F(y):
    x = np.zeros(2*n)
    x[:n] = y # input
    x[n:-1] = widetilde_w # dato conocido.
    return F1(x)-np.ones(n)
```

(c) Para encontrar $\mathbf{y}_1$ se necesita resolver el siguiente sistema de ecuaciones lineales,

$$J_{\mathbf{F}}(\mathbf{y}_0)\,\Delta\mathbf{y}_0 = -\mathbf{F}(\mathbf{y}_0).$$

Según lo indicado, no se tiene acceso explícito a la matriz $J_{\mathbf{F}}(\mathbf{y}_0)$, sin embargo se explica que se puede acceder al producto matriz-vector entre $J_{\mathbf{F}}(\mathbf{y}_0)$ y un vector arbitrario $\mathbf{v}$. Para poder utilizar esta información, se procederá a implementar lo indicado en Python,

```
def afun(v):
    epsilon = 1e-10
    w = (F(y0+epsilon*v)-F(y0))/epsilon
    return w
```

Notar que se considera que se tiene acceso a `y0` el cual corresponde a $\mathbf{y}_0$ indicado en la pregunta y `F(y)` es la función implementada en la pregunta anterior. Ahora que tenemos acceso a `afun`, notamos que el único algoritmo que conocemos (existen otros) que puede utilizar `afun` es GMRes. Recuerde que tiene la restricción que no puede almacenar en memoria la matriz Jacobiana, pero sí podemos utilizar `afun`. Entonces, podemos utilizar GMRes de la siguiente forma conectandolo con el problema que debemos resolver,

```
b = -F(y0)
Delta_y_i = GMRes(afun,b)
y1 = y0 + Delta_y_i
```

Por lo tanto, hemos logrado determinar $\mathbf{y}_1$, que es exactamente lo solicitado.

3.5. Desarrollo Pregunta "Graficando puntos"

Este desarrollo corresponde a la pregunta en Apartado 2.11.

(a) Se consideran los siguientes puntos para cumplir lo solicitado en la pregunta:

- Se define por conveniencia $x_{i+1} = x_i + \delta$ y la variante continua de la función a la cual se le buscará la raíz: $f(\delta) = \int_{x_i}^{x_i+\delta} \sqrt{1 + (g'(x))^2}\,\mathrm{d}x - l$. Notar que aún falta discretizarla.
- Se construye la versión discreta de $f_d(\delta)$.
  - Se consideran los pesos w_j y nodos y_j definidos en el intervalo $[-1, 1]$ de la cuadratura Gaussiana para $j \in \{1, 2, \ldots, q\}$.Notar que se usa la variable y_j para los nodos dado que la variable x_i ya está en uso. Por lo que se hace el siguiente cambio de variables:
  - Considere $a = x_i$ y $b = x_i + \delta$
  - $\widehat{w}_j = w_j\,(b-a)/2 = w_j\,\delta/2$
  - $\widehat{y}_j = ((b-a)\,y_j + b + a)/2 = (\delta\,y_j + 2\,x_i + \delta)/2 = \delta\,(y_j+1)/2 + x_i$

$$f_d(\delta) = \left(\sum_{k=1}^{q} \widehat{w}_j \sqrt{1 + (g'(\widehat{y}_j))^2}\right) - l.$$

- Se obtiene $f'(\delta) = \sqrt{1 + (g'(x_i + \delta))^2}$.

- Se ejecuta el método de Newton con la información entregada:

$$r = \text{Newton1D}(f_d(\delta), f'(\delta), r_0 = l/2).$$

  Se elige un *initial guess* mayor que 0 y relacionado con l, por lo que se propone $r_0 = l/2$.
- Finalmente se verifica que $r < b$, y se retorna $x_{i+1} = x_i + r$ y $g(x_{i+1})$. En caso contrario, se retorna $x_{i+1} = b$ y $g(b)$.

(b) Se listan las componentes del algoritmo:

- Obtener nodos y pesos de la cuadratura Gaussiana.

```
'''
input:
xi  : (float)    Point x_i.
g   : (callable) Function g(x).
gp  : (callable) Function g'(x).
l   : (float)    Arclength target.
b   : (float)    Right limit of interval.
q   : (integer)  Number of points for numerical quadrature, if needed

output:
x_next  : (float) Point x_{i+1}.
g_next  : (float) g(x_{i+1}).
'''
def find_next_point(xi, g, gp, l, b, q):
    # Computing Gaussian nodes and weights
    x_gauss, w_gauss = gaussian_nodes_and_weights(q)
```

- Definir $f(\delta)$.

```
    # Defining an auxiliar function.
    h  = lambda x: np.sqrt(1+(np.power(gp(x),2)))
    # Defining the discrete version of the function that we will find the root.
    f  = lambda delta: np.dot(w_gauss*delta/2,h(delta*(x_gauss+1)/2+xi))-l
```

- Definir $f'(\delta)$.

```
    # Defining the derivative of the function we will find the root.
    fp = lambda delta: h(xi+delta)
```

- Obtener raíz.

```
    # Computing the root.
    r  = Newton1D(f,fp,r0=l/2)
```

- Revisar caso límite y retornar *output* solicitado.

```
    # Checking if root is beyond the end of the interval or not
    if r<b:
        x_next = xi+r
        g_next = g(x_next)
    else:
        x_next = b
        g_next = g(b)
    return x_next, g_next
```

3.6. Desarrollo Pregunta "ODEs acopladas"

Este desarrollo corresponde a la pregunta en Apartado 2.12.

(a) Se consideran los siguientes puntos para cumplir lo solicitado en la pregunta. En caso de haber seguido otro procedimiento que también entrega la respuesta, se evaluará también.

- Se define $h = \dfrac{2}{n-1}$ y $x_i = h\,i + 1 - 2\,\frac{n}{n-1}$.

- Se discretiza la primera ecuación, $u''(x_i) - \sin(v(x_i)) = 0$, considerando $u_i \approx u(x_i)$ y $v_i \approx v(x_i)$, entonces,

$$\frac{u_{i+1} - 2\,u_i + u_{i-1}}{h^2} - \sin(v_i) = 0$$

multiplicando por h^2 se obtiene,

$$u_{i+1} - 2\,u_i + u_{i-1} - h^2\,\sin(v_i) = 0.$$

- Se discretiza la segunda ecuación, $v''(x_i) + \cos(u(x_i)) = 0$, entonces,

$$\frac{v_{i+1} - 2\,v_i + v_{i-1}}{h^2} + \cos(u_i) = 0$$

multiplicando por h^2 se obtiene,

$$v_{i+1} - 2\,v_i + v_{i-1} - h^2\,\cos(u_i) = 0.$$

- Ahora se consideran las condiciones de borde,

$$\begin{aligned} u(-1) = 1 &\to u_1 = 1, \\ u'(1) = 0 &\to \frac{u_n - u_{n-1}}{h} = 0, \\ v'(-1) = 0 &\to \frac{v_2 - v_1}{h} = 0, \\ v(1) = 0 &\to v_n = 0. \end{aligned}$$

- Lo que se ha obtenido es un sistema de ecuaciones no-lineales, por lo cual se procederá a definir la función vectorial respectiva de $\mathbb{R}^{2\,n} \to \mathbb{R}^{2\,n}$.

$$\mathbf{F}(\mathbf{w}) = \mathbf{F}(\langle \mathbf{u}, \mathbf{v} \rangle) = \begin{bmatrix} u_3 - 2\,u_2 + u_1 - h^2\,\sin(v_2) \\ \vdots \\ u_{i+1} - 2\,u_i + u_{i-1} - h^2\,\sin(v_i) \\ \vdots \\ u_n - 2\,u_{n-1} + u_{n-2} - h^2\,\sin(v_{n-1}) \\ v_3 - 2\,v_2 + v_1 + h^2\,\cos(u_2) \\ \vdots \\ v_{i+1} - 2\,v_i + v_{i-1} + h^2\,\cos(u_i) \\ \vdots \\ v_n - 2\,v_{n-1} + v_{n-2} + h^2\,\cos(u_{n-1}) \\ u_1 - 1 \\ u_n - u_{n-1} \\ v_2 - v_1 \\ v_n \end{bmatrix}$$

Notar que en las ecuaciones $2\,n - 2$ y $2\,n - 1$ se multiplicó por h, por esta razón no se incluye.

- Dado que tenemos definido $\mathbf{F}(\mathbf{w})$, es conveniente utilizar el método de Newton-Krylov, donde se define,

$$\begin{aligned} \mathbf{w}_i &= \text{“Initial guess”}, \\ \mathrm{Jw} &= \frac{\mathbf{F}(\mathbf{w}_i + \varepsilon\,\mathbf{w}) - \mathbf{F}(\mathbf{w})}{\varepsilon}, \\ \mathbf{w}_{i+1} &= \text{NewtonKrylov(Jw,b=}-\mathbf{F}(\mathbf{w}_i)\text{,}\mathbf{w}_i\text{)} \end{aligned}$$

Sin embargo esto debe iterarse hasta lograr convergencia de $\mathbf{w}_i$, i.e. la función entregada es útil para una iteración.

(b) Se listan las componentes del algoritmo:

- Inicialización de h y x_i

```
    '''
    input:
    n   : (integer) Number of equalspaced points x_i.

    output:
    u  : (ndarray) Numerical approximation of u(x_i).
    v  : (ndarray) Numerical approximation of v(x_i).
    x  : (ndarray) The discretization points x_i
    '''
    def find_ui_vi(n):
        # Defining h and xi
        h = 2/(n-1)
        x = np.linspace(-1,1,n)
```

- Construcción numérica de $\mathbf{F}(\mathbf{w})$.

```
    # Matrix-free implementation of F
    def F(w):
        # Decomposing input
        u = w[:n]
        v = w[n:]
        # Building output vector
        out = np.zeros(2*n)
        # Computing first n-2 equations
        out[:n-2]   = u[2:]-2*u[1:-1]+u[:-2]-np.power(h,2)*np.sin(v[1:-1])
        # Computing second n-2 equations
        out[n-2:-4] = v[2:]-2*v[1:-1]+v[:-2]+np.power(h,2)*np.cos(u[1:-1])
        # Computing equations for boundary conditions
        out[-4] = u[0]-1
        out[-3] = u[-1]-u[-2]
        out[-2] = v[1]-v[0]
        out[-1] = v[-1]
        return out
```

- Iteración de Newton-Krylov para encontrar una aproximación de la raíz.

```
    # Newton-Krylov iterations - setting upper limit for the number of iterations as 20
    w0 = np.zeros(2*n)
    w0[0] = 1 # This is added because we know it!
    w1 = np.zeros(2*n)
    epsilon = 1e-8
    for i in np.arange(20):
        Jw = lambda w: (F(w0+epsilon*w)-F(w0))/epsilon
        # One step of Newton-Krylov
        w1 = NewtonKrylov(Jw,b=-F(w0),w0)
        if np.linalg.norm(w1-w0)<1e-12 or np.linalg.norm(F(w1))<1e-8:
            break
        w0 = w1
```

- Recuperando las variables originales.

```
    u = w1[:n]
    v = w1[n:]
    return u, v, x
```

3.7. Desarrollo Pregunta "¡Indiana Jones necesita su ayuda!"

Este desarrollo corresponde a la pregunta en Apartado 2.13.

(a) Primero se debe aproximar el valor absoluto de la derivada de $f(x)$ para así obtener los puntos $z_k \approx |f'(x_k)|$ con alguna regla de aproximación. Por ejemplo, se puede aplicar **forward-difference** para aproximar la derivada en cada punto x_k para $k \in \{0, 1, \ldots, n\}$:

$$z_k = \frac{|y_{k+1} - y_k|}{x_{k+1} - x_k} \quad \text{para } k \in \{0, 1, \ldots, n\}$$

Segundo, nos falta la derivada para el punto x_{n+1}, en este caso utilizamos **backward-difference** para la aproximación:

$$z_{n+1} = \frac{|y_{n+1} - y_n|}{x_{n+1} - x_n}$$

Notar que el valor absoluto se aplica solo al numerador de z_k porque se conoce que el denominador es siempre positivo.

Finalmente, utilizando los puntos z_k para $k \in \{0, 1, \ldots, n+1\}$ obtenidos, se aplica el método del Trapecio, pero una versión modificada, ya que la data no está equiespaciada. Para cada intervalo $[x_k, x_{k+1}]$ con $k \in \{0, \ldots, n\}$ se aproxima la integral como:

$$\int_{x_k}^{x_{k+1}} |f'(x)| \approx \frac{(z_k + z_{k+1})\ (x_{k+1} - x_k)}{2} = (z_k + z_{k+1})\ \frac{\Delta x_k}{2}$$

donde $\Delta x_k = (x_{k+1} - x_k)$. Aplicando en todo el intervalo para $|f'(x)|$ se tiene que:

$$\begin{aligned}
\int_{-1}^{1} |f'(x)| &\approx \sum_{k=0}^{n} (z_k + z_{k+1})\ \frac{\Delta x_k}{2} \\
&\approx (z_0 + z_1)\ \frac{\Delta x_0}{2} + (z_1 + z_2)\ \frac{\Delta x_1}{2} + \cdots + (z_n + z_{n+1})\ \frac{\Delta x_n}{2} \\
&\approx \frac{\Delta x_0}{2}\ z_0 + \left(\frac{\Delta x_0}{2} + \frac{\Delta x_1}{2}\right) z_1 + \left(\frac{\Delta x_1}{2} + \frac{\Delta x_2}{2}\right) z_2 + \left(\frac{\Delta x_{n-1}}{2} + \frac{\Delta x_n}{2}\right) z_n + \frac{\Delta x_n}{2}\ z_{n+1} \\
&\approx \frac{\Delta x_0}{2}\ z_0 + \frac{\Delta x_n}{2}\ z_{n+1} + \sum_{k=0}^{n-1} \left(\frac{\Delta x_k}{2} + \frac{\Delta x_{k+1}}{2}\right) z_{k+1}
\end{aligned}$$

Notar que si $h = \Delta x_0 = \Delta x_1 = \cdots = \Delta x_n$ obtenemos los pesos del método del Trapecio para puntos equiespaciados.

(b) Se debe evaluar un punto x_j en la ecuación diferencial, por lo tanto, se debe obtener $f''(x_j)$, $f'(x_j)$ y $f(x_j)$.

- Para un punto x_j se puede utilizar la función AGSD$(\cdot)$ para aproximar $f''(x_j)$. Se debe entregar a la función los parámetros adecuados:
$$f''(x_j) \approx \text{AGSD}(x_{j-1}, y_{j-1}, x_j, y_j, x_{j+1}, y_{j+1})$$
- Para la derivada $f'(x_j)$ se utiliza la aproximación realizada en el ítem anterior $f'(x_j) \approx (y_{j+1} - y_j)/(x_{j+1} - x_j)$ y para evaluar la función en x_j, es decir, $f(x_j)$, simplemente se utiliza los datos disponibles, por lo tanto, $f(x_j) = y_j$
- Finalmente se evalúa si el punto x_j aproxima la ecuación diferencial con una tolerancia δ:
$$\left|\alpha\ \text{AGSD}(x_{j-1}, y_{j-1}, x_j, y_j, x_{j+1}, y_{j+1}) + \beta\ \frac{(y_{j+1} - y_j)}{(x_{j+1} - x_j)} + \gamma\ y_j - 1\right| < \delta$$

(c)
```
'''
input:
n         : (integer) Value of n, where the length of the sequence in the table is n + 2.
x_k       : (ndarray) Array with the values of x_k.
y_k       : (ndarray) Array with the values of y_k.
alpha     : (float) Value of alpha.
beta      : (float) Value of beta.
gamma     : (float) Value of gamma.
j         : (integer) Index where the differential equation is verified for the data.
delta     : (float) Tolerance.

output:
total_var : (float) Approximation of total variation of f(x).
check     : (bool)  Boolean value that indicates if the data approximates the differential
equation at x_j. If the data satisfied the tolerance, the output must be True, otherwise False.
'''
def jones(n,x_k,y_k,alpha,beta,gamma,j,delta):
```

- Calcular $y_{k+1} - y_k$ y $x_{k+1} - x_k$ para $k \in \{0, 1, \ldots, n\}$

```
delta_y = np.diff(y_k)
delta_x = np.diff(x_k)
```

- Aproximación del valor absoluto de la derivada en x_k para $k \in \{0, 1, \ldots, n+1\}$

```
z = np.zeros(n + 2)
z[:n+1] = np.abs(delta_y) / delta_x
% using backward difference for the last node
z[n+1] = np.abs(y_k[n+1] - y_k[n])/delta_x[n]
```

- Utilizar los pesos modificados y calcular la aproximación de la integral mediante el producto punto.

```
w = np.zeros(n+2)
w[0] = delta_x[0] / 2
w[-1] = delta_x[-1] / 2
w[1:-1] = (delta_x[:n] + delta_x[1:]) / 2
total_var = np.dot(w,z)
```

- Evaluar la ecuación diferencial en el punto x_j

```
yp = delta_y[j] / delta_x[j]
ode = alpha*AGSD(x_k[j-1],y_k[j-1],x_k[j],y_k[j],x_k[j+1],y_k[j+1]) + beta*yp + gamma*y_k[j]
```

- Verificar si en el punto x_j, la data aproxima la ecuación diferencial entregada,

```
check = np.abs(ode - 1) < delta
return total_var,check
```

3.8. Desarrollo Pregunta "Potencias de una matriz"

Este desarrollo corresponde a la pregunta en Apartado 2.14.

(a)

- Dado que no tenemos acceso a la matriz A de forma explícita, es decir no está almacenada en memoria (sea RAM o disco o cualquier medio de almacenamiento), no podemos operar directamente con ella. Esto significa que no es posible hacer operaciones filas o columnas sobre ella. Lo que sí sabemos es que la matriz A se define como la suma de 2 matrices, la matriz B_1 y B_2, donde la matriz B_1 es *sparse* y sí está almacenada en memoria de forma explícita pero la matriz B_2 no. Para operar con la matriz B_2 tenemos acceso a la función $\text{SFP}(\mathbf{v})$, que dado un vector $\mathbf{v}$ entrega el producto entre B_2 y $\mathbf{v}$. Entonces, para poder obtener el producto entre A y un vector arbitrario $\mathbf{v}$ lo podemos construir de la siguiente forma:

$$\begin{aligned} A\,\mathbf{v} &= (B_1 + B_2)\,\mathbf{v} \\ &= B_1\,\mathbf{v} + B_2\,\mathbf{v} \\ &= B_1\,\mathbf{v} + \text{SFP}(\mathbf{v}), \end{aligned}$$

donde A, B_1 y B_2 pertenecen a $\mathbb{R}^{n\times n}$, y $\mathbf{v} \in \mathbb{R}^n$. Lo que implica que tenemos a nuestra disposición el operador $A\,\mathbf{v}$, es decir el producto entre A y el vector $\mathbf{v}$ por medio de B_1 y $\text{SFP}(\cdot)$. Con estos antecedentes a nuestra disposición y dentro de los algoritmos estudiados, el algoritmo que se adapta idealmente a este caso es GMRes.

- Ahora, recordemos que lo que se debe resolver es el sistema $A^l\,\mathbf{x} = \mathbf{b}$, por lo tanto, se necesita un algoritmo que nos permita calcular $A^l\,\mathbf{v}$ para cualquier $\mathbf{v} \in \mathbb{R}^n$ para luego conectarlo con GMRes, lo cual se puede lograr de la siguiente forma considerando que $l \in \mathbb{N}$:

$$\begin{aligned} A^l\,\mathbf{v} = A^{l-1}\,(A\,\mathbf{v}) &= A^{l-1}\underbrace{(B_1\,\mathbf{v} + \text{SFP}(\mathbf{v}))}_{\mathbf{y}_1} = A^{l-2}\,(A\,\mathbf{y}_1) \\ &= A^{l-2}\underbrace{(B_1\,\mathbf{y}_1 + \text{SFP}(\mathbf{y}_1))}_{\mathbf{y}_2} = A^{l-3}\,(A\,\mathbf{y}_2) \\ &= A^{l-3}\underbrace{(B_1\,\mathbf{y}_2 + \text{SFP}(\mathbf{y}_2))}_{\mathbf{y}_3} \\ &\vdots \\ &= A\underbrace{(B_1\,\mathbf{y}_{l-2} + \text{SFP}(\mathbf{y}_{l-2}))}_{\mathbf{y}_{l-1}} = B_1\,\mathbf{y}_{l-1} + \text{SFP}(\mathbf{y}_{l-1}) \end{aligned}$$

Notar entonces que podemos calcular $A^l\,\mathbf{x}$ como una sucesión de productos matriz-vector, sin necesidad de almacenar de forma explícita la matriz A o la matriz A^l. Un algoritmo para calcular $A^l\,\mathbf{v}$ tendría la siguiente forma:

- Calcular el vector $\mathbf{y}_1 = B_1\,\mathbf{v} + \mathrm{SFP}(\mathbf{v})$.
- Calcular $\mathbf{y}_{k+1} = B_1\,\mathbf{y}_k + \mathrm{SFP}(\mathbf{y}_k)$ para k desde 1 hasta $l-1$.

Notar que no es necesario almacenar todos los vectores $\mathbf{y}_k$, simplemente se puede sobre-escribir en un vector único, digamos $\mathbf{y}$, pero por claridad de la explicación se presenta como una secuencia de vectores. El procedimiento anterior se puede almacenar en una función, digamos `afun` que recibe el vector $\mathbf{v}$ y retorna el vector $A^l\,\mathbf{v}$.

- Por lo tanto, podemos utilizar el algoritmo de GMRes para obtener una aproximación $\tilde{\mathbf{x}}$, entregando a GMRes `afun` que obtiene $A^l\,\mathbf{v}$, el lado derecho $\mathbf{b}$, un *initial guess* $\mathbf{x}_0$ y una tolerancia δ, de tal forma que el algoritmo entrega la aproximación $\tilde{\mathbf{x}}$ asegurando que $\|\mathbf{b} - A\,\tilde{\mathbf{x}}\| \leq \delta$.

(b)
```
'''
input:
n       : (integer) dimension of the problem.
l       : (integer) Integer value of l.
B_1     : (ndarray) Sparse matrix B_1. Notice this is defined for simplicity as ndarray.
SFP     : (callable) Procedure that receives the vector 'v' and
computes the 'fast' product between B_2 and the vector 'v'.
b       : (ndarray) Right-hand-side vector.
delta   : (double) Numerical threshold for computation of residual.

output:
xt       : (ndarray) Array with the values of the numerical approximation x_tilde.
'''
def find_x(n,l,B_1,SFP,b,delta):
```

- Calcular $A\,\mathbf{v}$ para algún vector $\mathbf{v}$.

```
afun1 = lambda v: np.dot(B_1,v) + SFP(v)
```

- Calcular $A^l\,\mathbf{v}$ para algún vector $\mathbf{v} \in \mathbb{R}^n$

```
def afun2(v):
    y = afun1(v)
    for i in np.arange(l-1):
      y = afun1(y)
    return y
```

- Definir el *initial guess*, utilizar GMRes y retornar la aproximación numérica `xt`.

```
x0 = np.zeros(n)
xt = GMRes_matrix_free(afun2,b,x0,delta)
return xt
```

3.9. Desarrollo Pregunta "GMRes ayuda a Sylvester conjugado"

Este desarrollo corresponde a la pregunta en Apartado 2.15.

(a) Se consideran los siguientes puntos para cumplir lo solicitado en la pregunta:

- Como primer paso se reemplazará $Z = X + \mathrm{i}\,Y$, $B = B_1 + \mathrm{i}\,B_2$, y $C = C_1 + \mathrm{i}\,C_2$, y se expandirán los términos:

$$\begin{aligned}
A\,Z + \mathrm{Conj}(Z)\,B &= C \\
A\,(X + \mathrm{i}\,Y) + \mathrm{Conj}\,(X + \mathrm{i}\,Y)\,(B_1 + \mathrm{i}\,B_2) &= C_1 + \mathrm{i}\,C_2 \\
A\,(X + \mathrm{i}\,Y) + (X - \mathrm{i}\,Y)\,(B_1 + \mathrm{i}\,B_2) &= C_1 + \mathrm{i}\,C_2 \\
A\,X + \mathrm{i}\,A\,Y + X\,B_1 + \mathrm{i}\,X\,B_2 - \mathrm{i}\,Y\,B_1 - \mathrm{i}^2\,Y\,B_2 &= C_1 + \mathrm{i}\,C_2 \\
A\,X + \mathrm{i}\,A\,Y + X\,B_1 + \mathrm{i}\,X\,B_2 - \mathrm{i}\,Y\,B_1 + Y\,B_2 &= C_1 + \mathrm{i}\,C_2 \\
(A\,X + X\,B_1 + Y\,B_2) + \mathrm{i}\,(A\,Y + X\,B_2 - Y\,B_1) &= C_1 + \mathrm{i}\,C_2
\end{aligned}$$

- Separar la parte componente real y la imaginaria, así se obtiene una ecuación que solo depende de aritmética real:

$$\begin{aligned}
A\,X + X\,B_1 + Y\,B_2 &= C_1, \\
A\,Y + X\,B_2 - Y\,B_1 &= C_2.
\end{aligned}$$

(b) Implementación de la solución usando `GMRes2(afun,x0,b,rel_res=1e-10)`, esta primera parte considera construir `afun`:

```
'''
input:
A   : (ndarray) Matrix "A".
B1  : (ndarray) Matrix "B1".
B2  : (ndarray) Matrix "B2".
C1  : (ndarray) Matrix "C1".
C2  : (ndarray) Matrix "C2".
rel_res : (float) Relative residue to be achieved.

output:
X  : (ndarray) Matrix "X".
Y  : (ndarray) Matrix "Y".
'''
def solve_Complex_Sylvester(A, B1, B2, C1, C2, rel_res):
    n = A.shape[0]
    def afun(x):
        X = np.reshape(x[:n**2],(n,n),order='F')
        Y = np.reshape(x[n**2:],(n,n),order='F')
        out1 = np.dot(A,X)+np.dot(X,B1)+np.dot(Y,B2)
        out2 = np.dot(A,Y)+np.dot(X,B2)-np.dot(Y,B1)
        out = np.zeros(2*n**2)
        out[:n**2] = out1.flatten('F')
        out[n**2:] = out2.flatten('F')
        return out
```

(c) Esta segunda parte construye el lado derecho del sistema de ecuaciones lineales correspondiente:

```
    b = np.zeros(2*n**2)
    b[:n**2] = C1.flatten('F')
    b[n**2:] = C2.flatten('F')
```

(d) Finalmente se define `x0`, se llama a GMRes, se recupera la solución y se retorna lo solicitado:

```
    x0 = np.zeros(2*n**2)
    x_sol = GMRes2(afun,x0,b,rel_res=rel_res)
    X = np.reshape(x_sol[:n**2],(n,n),order='F')
    Y = np.reshape(x_sol[n**2:],(n,n),order='F')
    return X, Y
```

3.10. Desarrollo Pregunta "Matrix polynomials are back!"

Este desarrollo corresponde a la pregunta en Apartado 2.16.

(a) Para poder determinar X_i, lo primero que hacer es dejar los términos con factores desconocidos al lado izquierdo de la igualdad y los factores conocidos al lado derecho, es decir,

$$\begin{aligned} Y_i &= U^{-1}\,X_0 - U^{-1}\,X_i + X_0\,L - X_i\,L + B, \\ U^{-1}\,X_i + X_i\,L &= \underbrace{U^{-1}\,X_0 + X_0\,L + B - Y_i}_{C}. \end{aligned}$$

Entonces reconocemos que tenemos una ecuación de Sylvester, que es un sistema de ecuaciones lineales pero no en la forma estándar $A\,\mathbf{x} = \mathbf{b}$. Para resolver este sistema de ecuaciones lineales, utilizaremos GMRes, para lo cual solo necesitamos construir la función `afun` que realiza el producto matriz vector asociado, y el lado derecho correspondiente.

Primero, notamos que el lado derecho es solo la vectorización de la matriz C definida anteriormente, es decir $C = U^{-1}\,X_0 + X_0\,L + B - Y_i$. La cual se obtiene haciendo las operaciones de suma, resta y multiplicación correspondiente, salvo la computación de $U^{-1}\,X_0$. En este caso definimos el siguiente problema equivalente,

$$\begin{aligned} D &= U^{-1}\,X_0, \\ U\,D &= X_0. \end{aligned}$$

Entonces para obtener D, solo necesitamos resolver los n sistemas de ecuaciones lineales asociados con el algoritmo *Backward Substitution*.

Ahora necesitamos listar las operaciones que definirán a la función `afun`, las cuales son las siguientes:

- Esta función recibe un vector `x` de dimensión n^2 el cual debe re-estructurarse para construir una matriz de dimensión $n \times n$, a la cual llamaremos la matrix X.
- Ahora debemos obtener $U^{-1}\,X$. Lo cual lo resolvemos siguiendo el mismo procedimiento utilizado al construir el lado derecho, es decir $\widehat{D} = U^{-1}\,X$, lo que nos genera $U\,\widehat{D} = X$, el cual se resuelve con el algoritmo *Backward Substitution*.
- El segundo término que debemos obtener es $X\,L$, el cual es una multiplicación matriz-matriz, la cual se puede hacer de forma directa.
- Por último, se retorna la versión vectorizada de la matriz $\widehat{D} + X\,L$.

Finalmente, con este procedimiento definido, podemos **llamar** a GMRes con la función `afun` antes definida y la vectorización del *right-hand-side* C.

(b)
```
'''
input:
Yi   : (ndarray) Matrix Yi.
X0   : (ndarray) Matrix X0.
U    : (ndarray) Matrix U.
L    : (ndarray) Matrix L.
B    : (ndarray) Matrix B.
n    : (int)     Each matrix has size n x n.

output:
Xi   : (ndarray) Matrix Xi.
'''
def compute_Xi(Yi,X0,U,L,B,n):
    # Building RHS
    D = solve_triangular(U, X0, lower=False)
    C = D+X0@L+B-Yi
    RHS = np.ravel(C,order='F')

    # Building afun
    def afun(x):
        X=np.reshape(x,(n,n),order='F')
        Dt = solve_triangular(U, X, lower=False)
        LHS = Dt+X@L
        out = np.ravel(LHS,order='F')
        return out

    # Using GMRes
    xi = GMRes_matrix_free(afun,RHS)

    # Reshaping the output to a matrix form
    Xi=np.reshape(xi,(n,n),order='F')

    return Xi
```

3.11. Desarrollo Pregunta "Curiosity wants to send data!"

Este desarrollo corresponde a la pregunta en Apartado 2.17.

(a) La aproximación numérica de $x_1(t_i)$, $x_2(t_i)$ y $x_3(t_i)$ en cada tiempo $t_i = i\,\dfrac{T}{N}$ para $i \in \{0, 1, 2, 3, \ldots, N\}$ se puede obtener aplicando algún método de resolución de IVP para las condiciones iniciales $x_{1,0}$, $x_{2,0}$ y $x_{3,0}$. El procedimiento se describe a continuación:

a) Definir el vector $\boldsymbol{x}$:

$$\boldsymbol{x} = [x_1(t), x_2(t), x_3(t)]^\top$$

b) Construir la función $\dot{\boldsymbol{x}} = \boldsymbol{F}(t, \boldsymbol{x})$. Esta función viene dada por la variación en el tiempo de las posiciones de las partículas:

$$\boldsymbol{F}(t,\boldsymbol{x}) = \begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ \dot{x}_3\end{bmatrix} = \begin{bmatrix}f_1(\boldsymbol{x})\\ f_2(\boldsymbol{x})\\ f_3(\boldsymbol{x})\end{bmatrix} = \begin{bmatrix}(1-\alpha)\,s\left(f'(x_1)\right) - \alpha\,\left(s(x_2-x_1)+s(x_3-x_1)\right)\\ (1-\alpha)\,s\left(f'(x_2)\right) - \alpha\,\left(s(x_1-x_2)+s(x_3-x_2)\right)\\ (1-\alpha)\,s\left(f'(x_3)\right) - \alpha\,\left(s(x_1-x_3)+s(x_2-x_3)\right)\end{bmatrix}$$

c) Definir el vector $\boldsymbol{x}_0$:

$$\boldsymbol{x}_0 = \begin{bmatrix}x_{1,0}\\ x_{2,0}\\ x_{3,0}\end{bmatrix}$$

d) Obtener los valores de $\boldsymbol{x}(t_i)$ en cada tiempo $t_i = i\,\dfrac{T}{N}$ con algún solver de IVP:

$$t, \boldsymbol{x} = \text{solverIVP}(t_0, T, N, \boldsymbol{x}_0, \boldsymbol{F}(t,\boldsymbol{x}))$$

(b) Al obtener la aproximación numérica de $x_1(t_i)$, $x_2(t_i)$ y $x_3(t_i)$ en cada tiempo $t_i = i\,\dfrac{T}{N}$ para $i \in \{0,1,2,3,\ldots,N\}$, se puede evaluar la variación de la posición de cada partícula en los mismos tiempos t_i. Es decir, se puede evaluar la función $\boldsymbol{F}(t_i, \boldsymbol{x}_i)$ para cada tiempo t_i. En este caso se obtiene una aproximación numérica de la variación de la posición de cada partícula:

$$\boldsymbol{F}(t_i,\boldsymbol{x}_i) = \begin{bmatrix}\dot{x}_1(t_i)\\ \dot{x}_2(t_i)\\ \dot{x}_3(t_i)\end{bmatrix} = \begin{bmatrix}f_1(\boldsymbol{x}(t_i))\\ f_2(\boldsymbol{x}(t_i))\\ f_3(\boldsymbol{x}(t_i))\end{bmatrix} = \begin{bmatrix}(1-\alpha)\,s\left(f'(x_1(t_i))\right) - \alpha\,\left(s(x_2(t_i)-x_1(t_i))+s(x_3(t_i)-x_1(t_i))\right)\\ (1-\alpha)\,s\left(f'(x_2(t_i))\right) - \alpha\,\left(s(x_1(t_i)-x_2(t_i))+s(x_3(t_i)-x_2(t_i))\right)\\ (1-\alpha)\,s\left(f'(x_3(t_i))\right) - \alpha\,\left(s(x_1(t_i)-x_3(t_i))+s(x_2(t_i)-x_3(t_i))\right)\end{bmatrix}$$

Se define el conjunto $T_j = \{t_k \;:\; |\dot{x}_j(t_k)| < \varepsilon\}$ donde $t_k = k\,\dfrac{T}{N}$ para $k \in \{0,1,2,3,\ldots,N\}$ para cada partícula x_j con $j = \{1,2,3\}$. Se debe calcular, para cada $j = \{1,2,3\}$ el tiempo $\tau_j = \min(T_j)$. En caso de que el conjunto T_j sea vacío, entonces $\tau_j = -1$.

(c)
```
'''
input:
x10   : (float)    Initial condition for x_1.
x20   : (float)    Initial condition for x_2.
x30   : (float)    Initial condition for x_3.
alpha : (float)    Value of alpha.
s     : (callable) Function s(x).
fp    : (callable) Derivative of the function f.
t0    : (float)    Initial time of trajectories.
T     : (float)    Final time of trajectories.
eps   : (float)    Tolerance related to the numerical stationary state.
N     : (int)      Number of timesteps.

output:
t     : (ndarray) Discretization of time in the interval [t0,T] with (N + 1) equidistant points.
x_1   : (ndarray) Trajectory of particle x_1.
x_2   : (ndarray) Trajectory of particle x_2.
x_3   : (ndarray) Trajectory of particle x_3.
tau_1 : (float)   First time when particle x_1 reaches a numerical stationary state.
tau_2 : (float)   First time when particle x_2 reaches a numerical stationary state.
tau_3 : (float)   First time when particle x_3 reaches a numerical stationary state.
'''
def particles(x10,x20,x30,alpha,s,fp,t0,T,eps,N):
```

- Construcción de la función `F`:

```
    f1 = lambda x1,x2,x3: (1. - alpha)*s(fp(x1)) - alpha*(s(x2 - x1) + s(x3 - x1))
    f2 = lambda x1,x2,x3: (1. - alpha)*s(fp(x2)) - alpha*(s(x1 - x2) + s(x3 - x2))
    f3 = lambda x1,x2,x3: (1. - alpha)*s(fp(x3)) - alpha*(s(x1 - x3) + s(x2 - x3))
    def F(t,x):
        x1,x2,x3 = x[0],x[1],x[2]
        x1d = f1(x1,x2,x3)
```

```
    x2d = f2(x1,x2,x3)
    x3d = f3(x1,x2,x3)
    return np.array([x1d,x2d,x3d])
```

- Resolver IVP para la trayectoria de cada partícula en $t \in [0, T]$:

```
x0 = np.array([x10,x20,x30])
t,x = RK4(t0,T,N,x0,F)
```

- Obtener la trayectoria de cada partícula:

```
x_1 = x[:,0]
x_2 = x[:,1]
x_3 = x[:,2]
```

- Obtener el menor tiempo τ_k, para $k \in \{1, 2, 3, \}$:

```
taus1 = t[np.abs(f1(x_1,x_2,x_3))<eps]
tau1 = -1 if len(taus1)==0 else taus1[0]
taus2 = t[np.abs(f2(x_1,x_2,x_3))<eps]
tau2 = -1 if len(taus2)==0 else taus2[0]
taus3 = t[np.abs(f3(x_1,x_2,x_3))<eps]
tau3 = -1 if len(taus3)==0 else taus3[0]

return t,x_1,x_2,x_3,tau_1,tau_2,tau_3
```

3.12. Desarrollo Pregunta "Bessel is here"

Este desarrollo corresponde a la pregunta en Apartado 2.18.

(a) Considerando la discretización espacial $x_j = 1 + j\dfrac{2\pi - 1}{M}$, se propondrá un algoritmo basado en el método de diferencias finitas, considerando la aproximación de la primera derivada por diferencias centradas, es decir,

$$z''(x_j) \approx \frac{z_{j+1}^{[\delta]} - 2\, z_j^{[\delta]} + z_{j-1}^{[\delta]}}{h^2},$$
$$z'(x_j) \approx \frac{z_{j+1}^{[\delta]} - z_{j-1}^{[\delta]}}{2\, h},$$

donde la dependencia de δ indica que $y(x_j) \approx z_j^{[0]}$, es decir aproxima la función de Bessel $J_1(x)$, y $w(x_j) \approx z_j^{[1]}$, es decir la ODE perturbada. Notar que ambas ODEs pueden y necesitan usar la misma discretización espacial, y la única diferencia corresponde al término $4\,\sin(x)$, entonces se usará la misma formulación, pero haciendo el cambio correspondiente en el término adicional.

Ahora, la versión discreta de la ODE es la siguiente,

$$\frac{x_j^2 + \delta\, 4\, \sin(x_j)}{h^2}\left(z_{j+1}^{[\delta]} - 2\, z_j^{[\delta]} + z_{j-1}^{[\delta]}\right) + \frac{x_j}{2\, h}\left(z_{j+1}^{[\delta]} - z_{j-1}^{[\delta]}\right) + (x_j^2 - 1))\, z_j^{[\delta]} = 0,$$

y

$$z_0^{[\delta]} = J_1(1) \approx 0.4400505857449335159596822037189149131273723019927 7\ldots,$$
$$z_M^{[\delta]} = J_1(2\,\pi) \approx -0.2123825300763622028586556710849962272560258536 9\ldots.$$

Considerando las matrices de diferenciación,

$$D_2 = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix},$$

$$D = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}.$$

Podemos escribir el sistema de ecuaciones lineales asociado de la siguiente forma,

$$\frac{1}{h^2}\,\mathrm{diag}(\mathbf{x}^2 + \delta\, 4\,\sin(\mathbf{x}))\,D_2\,\mathbf{z}^{[\delta]} + \frac{1}{2\,h}\,\mathrm{diag}(\mathbf{x})\,D\,\mathbf{z}^{[\delta]} + \mathrm{diag}(\mathbf{x}^2 - \mathbf{1})\,\mathbf{z}^{[\delta]} = \begin{bmatrix} -\dfrac{x_1^2 + \delta\, 4\,\sin(x_1)}{h^2} z_0^{[\delta]} + \dfrac{x_1}{2\,h}\, z_0^{[\delta]} \\ 0 \\ \vdots \\ 0 \\ -\dfrac{x_{M-1}^2 + \delta\, 4\,\sin(x_{M-1})}{h^2}\, z_M^{[\delta]} - \dfrac{x_{M-1}}{2\,h}\, z_M^{[\delta]} \end{bmatrix},$$

donde $\mathbf{z}^{[\delta]} = \left[z_1^{[\delta]}, z_2^{[\delta]}, \ldots, z_{M-1}^{[\delta]}\right]$. $\mathbf{x}^2$ y $\sin(\mathbf{x})$ significa la aplicación de la función de forma *element-wise*, y $\mathbf{1}$ es un vector de puros 1s de dimensión compatible. Entonces, el sistema de ecuaciones lineales resultando es,

$$\underbrace{\left(\frac{1}{h^2}\,\mathrm{diag}(\mathbf{x}^2 + \delta\, 4\,\sin(\mathbf{x}))\,D_2 + \frac{1}{2\,h}\,\mathrm{diag}(\mathbf{x})\,D + \mathrm{diag}(\mathbf{x}^2 - \mathbf{1})\right)}_{A_M^{[\delta]}}\mathbf{z}^{[\delta]} = \underbrace{\begin{bmatrix} -\dfrac{x_1^2 + \delta\, 4\,\sin(x_1)}{h^2} z_0^{[\delta]} + \dfrac{x_1}{2\,h}\, z_0^{[\delta]} \\ 0 \\ \vdots \\ 0 \\ -\dfrac{x_{M-1}^2 + \delta\, 4\,\sin(x_{M-1})}{h^2}\, z_M^{[\delta]} - \dfrac{x_{M-1}}{2\,h}\, z_M^{[\delta]} \end{bmatrix}}_{\mathbf{b}_M^{[\delta]}}.$$

Por lo tanto, se deben resolver los siguientes sistemas de ecuaciones lineales, por ejemplo con PALU,

$$A_M^{[0]}\,\mathbf{z}^{[0]} = \mathbf{b}_M^{[0]},$$
$$A_M^{[1]}\,\mathbf{z}^{[1]} = \mathbf{b}_M^{[1]},$$

donde $\mathbf{z}_j^{[0]}$ aproximará a $J_1(x_j)$ y $\mathbf{z}_j^{[0]}$ aproximará a $w(x_j)$, lo que permite obtener $\mathbf{d}_j = \mathbf{z}_j^{[0]} - \mathbf{z}_j^{[0]}$. Notar que la primera y última componente de $\mathbf{d}$ serán 0 dado que ambas funciones son exactamente iguales en $x = 1$ y $x = 2\,\pi$.

(b)
```
'''
input:
M     : (int) Number of discretization intervals, which implies M+1 points.
J1_1  : (float) Numerical value of J_1(1)
J1_2pi: (float) Numerical value of J_1(2\,\pi)

output:
d     : (ndarray) Difference vector d_j=J_1(x_j)-w(x_j)
'''
def compareBessel(M, J1_1, J1_2pi):
```

```
def build_D_D2(N):
    D = np.zeros((N,N))
    D2 = np.zeros((N,N))
    # Build D
    j = np.arange(N-1)
    D[j,j+1]=1
    D[j+1,j]=-1
    #Build D2
    i = np.arange(N)
    D2[i,i]=-2
    D2[j,j+1]=1
    D2[j+1,j]=1
    return D, D2

def my_diag(x,n):
    A = np.zeros((n,n))
    i = np.arange(n)
    A[i,i] = x
    return A

D,D2 = build_D_D2(M-1)

j  = np.arange(M+1)
h  = (2*np.pi-1)/M
h2 = h**2.
x  = 1+j[1:-1]*h
z0 = J1_1
zM = J1_2pi

A_delta = lambda delta: (my_diag(x**2+delta*4*np.sin(x),M-1)/h2)@D2 \
    +(my_diag(x,M-1)/(2*h))@D \
    +my_diag(x**2-1,M-1)
def b_delta(delta,n):
    b     = np.zeros(n)
    b[0]  = -((x[0]**2+delta*4*np.sin(x[0]))/h2)*z0+(x[0]/(2*h))*z0
    b[-1] = -((x[-1]**2+delta*4*np.sin(x[-1]))/h2)*zM-(x[-1]/(2*h))*zM
    return b

A0 = A_delta(0.)
b0 = b_delta(0.,M-1)

A1 = A_delta(1.)
b1 = b_delta(1.,M-1)

z_delta_0 = np.zeros(M+1)
z_delta_0[0] = z0
z_delta_0[-1] = zM
z_delta_0[1:M] = GMRes_explicit_matrix(A0,b0)

z_delta_1 = np.zeros(M+1)
z_delta_1[0] = z0
z_delta_1[-1] = zM
z_delta_1[1:M] = GMRes_explicit_matrix(A1,b1)

d = z_delta_0-z_delta_1

return d
```

3.13. Desarrollo Pregunta "Criptomonedas"

Este desarrollo corresponde a la pregunta en Apartado 2.19.

(a) Para poder completar un *missing value* en algún tiempo t_k donde exista un valor faltante, debemos considerar los dos valores anteriores y posteriores. Por ejemplo, si falta el dato $\texttt{Low}(t_k)$, debemos considerar los valores $\texttt{Low}(t_{k-2})$, $\texttt{Low}(t_{k-1})$, $\texttt{Low}(t_{k+1})$ y $\texttt{Low}(t_{k+2})$ para la aproximación. Entonces, cada uno de estos valores en t_{k-2}, t_{k-1}, t_{k+1} y t_{k+2} deben cumplir que:

$$\begin{aligned}
\texttt{Low}(t_{k-2}) = p(t_{k-2}) &\approx a + b\,(t_{k-2} - t_k) + c\,(t_{k-2} - t_k)^2 \\
\texttt{Low}(t_{k-1}) = p(t_{k-1}) &\approx a + b\,(t_{k-1} - t_k) + c\,(t_{k-1} - t_k)^2 \\
\texttt{Low}(t_{k+1}) = p(t_{k+1}) &\approx a + b\,(t_{k+1} - t_k) + c\,(t_{k+1} - t_k)^2 \\
\texttt{Low}(t_{k+2}) = p(t_{k+2}) &\approx a + b\,(t_{k+2} - t_k) + c\,(t_{k+2} - t_k)^2
\end{aligned}$$

Podemos escribir el sistema anterior de forma matricial:

$$\begin{bmatrix} 1 & (t_{k-2} - t_k) & (t_{k-2} - t_k)^2 \\ 1 & (t_{k-1} - t_k) & (t_{k-1} - t_k)^2 \\ 1 & (t_{k+1} - t_k) & (t_{k+1} - t_k)^2 \\ 1 & (t_{k+2} - t_k) & (t_{k+2} - t_k)^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -2\Delta tk & 4\,(\Delta t_k)^2 \\ 1 & -\Delta tk & (\Delta t_k)^2 \\ 1 & \Delta tk & (\Delta t_k)^2 \\ 1 & 2\Delta tk & 4\,(\Delta t_k)^2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a \\ b \\ c \end{bmatrix}}_{\mathbf{x}} \approx \underbrace{\begin{bmatrix} \texttt{Low}(t_{k-2}) \\ \texttt{Low}(t_{k-1}) \\ \texttt{Low}(t_{k+1}) \\ \texttt{Low}(t_{k+2}) \end{bmatrix}}_{\mathbf{y}}$$

Luego se resuelve el problema de mínimos cuadrados $A\,\mathbf{x} \approx \mathbf{y}$ que nos permita obtener los coeficientes a, b y c. Al obtener estos coeficientes, se evalúa en la función de aproximación en el tiempo t_k obteniendo:

$$p(t_k) = a + b\,(t_k - t_k) + c\,(t_k - t_k)^2 = a$$

Finalmente, se asigna a $\texttt{Low}(t_k)$ el valor del coeficiente a. Este mismo procedimiento se repite para los *missing values* restantes asociados a la data `Low` como para los *missing values* de `High`.

(b) Considerando que tanto la data $y_{\text{low}}(t_k)$ como $y_{\text{high}}(t_k)$ con $k \in \{0, \ldots, N\}$ han sido completadas con el procedimiento anterior, se realiza lo siguiente:

a) se procede a calcular los valores $\text{spread}(t_k) = y_{\text{high}}(t_k) - y_{\text{low}}(t_k)$ para $k \in \{0, \ldots, N\}$, discretizados en el intervalo $[t_0, t_N]$.

b) se evalúan los tiempos t_k en la función $\omega(t)$ obteniendo los valores $\omega_k = \omega(t_k)$ para $k \in \{0, \ldots, N\}$, discretizados en el intervalo $[t_0, t_N]$.

c) se calculan los valores $\text{spw}(t_k) = \text{spread}(t_k) \cdot \omega_k$ para $k \in \{0, \ldots, N\}$, discretizados en el intervalo $[t_0, t_N]$.

d) se calcula mediante un método de integración numérica el numerador del valor ponderado del `Spread`:

$$N_s = \int_{t_0}^{t_N} \text{Spread}(t)\,\omega(t)\,\mathrm{d}t \approx \sum_{k=0}^{N} \alpha_k\,\text{spw}(t_k)$$

donde $h = (t_N - t_0)/(N+1)$ y α_k es el peso asociado al método de integración numérica, por ejemplo, si el método escogido es trapecio, entonces $\alpha_0 = \alpha_N = h/2$ y $\alpha_k = h$ para $k = 1, \ldots, N-1$.

e) se calcula mediante un método de integración numérica el denominador:

$$D_s = \int_{t_0}^{t_N} \omega(t)\,\mathrm{d}t \approx \sum_{k=0}^{N} \alpha_k\,\omega_k$$

donde $h = (t_N - t_0)/(N+1)$ y α_k es el peso asociado al método de integración numérica, por ejemplo, si el método escogido es trapecio, entonces $\alpha_0 = \alpha_N = h/2$ y $\alpha_k = h$ para $k = 1, \ldots, N-1$.

f) finalmente se calcula el valor ponderado del `Spread` como $I_{\text{spread}} = N_s/D_s$.

(c)
```
'''
input:
N        : (integer)  that defines the N+1 timesteps.
t        : (ndarray)  (N+1)-dimensional vector data $\mathbf{t}$.
i_low    : (ndarray)  indices where there is a missing value in the time series Low.
           For instance, in Table 1, i_low = [2,6,10,...].
```

```
i_high   : (ndarray)  indices where there is a missing value in the time series High.
           For instance, in Table 1, i_high = [4,8,...].
w        : (callable) weighting function w(t).
y_low    : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{low}$.
y_high   : (ndarray)  (N+1)-dimensional vector data $\mathbf{y}_{high}$.

output:
i_spread        : (float)   weighted average value of the Spread.
y_low_complete  : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{low}$ with the
                            missing values approximated.
y_high_complete : (ndarray) (N+1)-dimensional vector data $\mathbf{y}_{high}$ with the
                            missing values approximated.
'''
def compute_spread(N,t,i_low,i_high,w,y_low,y_high):
    dt = t[1] - t[0]
    A = np.ones((4,3))
    A[:,1] = np.array([-2.,-1.,1.,2.])*dt
    A[:,2] = np.array([4.,1.,1.,4.])*dt**2
    Q,R = np.linalg.qr(A,mode="reduced")
    for i in i_low:
        c = np.concatenate((y_low[i-2:i],y_low[i+1:i+3]))
        v = np.linalg.solve(R,np.dot(Q.T,c))
        y_low[i] = v[0]
    for i in i_high:
        c = np.concatenate((y_high[i-2:i],y_high[i+1:i+3]))
        v = np.linalg.solve(R,np.dot(Q.T,c))
        y_high[i] = v[0]
    spread = y_high - y_low
    w_t = w(t)
    spread_w = trapezoidDiscrete(t,w_t*spread,t[0],t[-1])
    int_w = trapezoidDiscrete(t,w_t,t[0],t[-1])
    i_spread = spread_w / int_w
    y_low_complete, y_high_complete = y_low,y_high
    return i_spread, y_low_complete, y_high_complete
```