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. ¿Qué es el argmin?

Considere la siguiente función vectorial $\mathbf{F}(\mathbf{x}) = \underset{\mathbf{y}}{\operatorname{argmin}} \|A\,B\,\mathbf{x} - B\,\mathbf{y}\|_2$, donde $A \in \mathbb{R}^{n\times n}$ y densa; $B \in \mathbb{R}^{n\times m}$, sparse y es full rank, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^m$ y $m < n$.

(a) Construya un algoritmo que pueda evaluar $\mathbf{F}(\mathbf{x})$ para cualquier vector $\mathbf{x}$.

2.2. Función inversa

En la primera parte del curso estudiamos varios algoritmos de búsqueda de ceros. Una posible aplicación de búsqueda de ceros es la construcción de la función inversa de cualquier función $f(x)$ inyectiva, es decir $f^{-1}(y)$. Por ejemplo, si consideramos la búsqueda de ceros de la siguiente función $g(x) = \exp(x) - y$ dado y como un valor conocido, entonces lo que estamos obteniendo con la búsqueda de ceros es el valor $x = \log(y)$. Esto corresponde efectivamente a la función inversa de la función exponencial $f(x) = \exp(x)$, es decir $f^{-1}(y) = \log(y)$. Recuerde que la relación entre una función y su inversa cumple que $f^{-1}(f(x)) = x$ ó $f(f^{-1}(x)) = x$. Entonces la búsqueda de ceros de la función $g(x) = f(x) - y$ entrega $x = f^{-1}(y)$. Este procedimiento es bastante efectivo cuando uno quiere aplicarlo unas pocas veces, sin embargo podría ser muy costoso cuando se requiere aplicar muchas veces. Por ejemplo, si quisiéramos aplicar la función inversa $f^{-1}(x)$ para valores de x en un intervalo definido, digamos $[a, b]$, entonces podría convenir utilizar otra familia de algoritmos, como por ejemplo interpolación polinomial.

En particular, utilizaremos $f(x) = \exp(x)$ para $x \in [-1, 1]$ y su función inversa $f^{-1}(y) = \log(y)$. Esto significa que solo aplicaremos la función inversa $f^{-1}(y)$ a valores de la función $f(x) = \exp(x)$ restringiendo x al intervalo indicado. Esto es muy útil ya que podemos definir una cota de los valores que utilizaremos en el dominio de la función inversa $f^{-1}(y)$. Por ejemplo, si nosotros recibimos el valor $y = 1.6487212707001282$, entonces nos interesa obtener el valor de x tal que $f(x) = 1.6487212707001282$, o también podemos decir $f^{-1}(1.6487212707001282) = x$. En este caso la solución es 0.5 porque $f^{-1}(1.6487212707001282) = 0.5$.

La tarea que se encomienda es construir un polinomio interpolador de la función $f^{-1}(y)$, digamos el polinomio $\mathrm{pLog}_n(y)$, donde n corresponde a la cantidad de puntos utilizados en la interpolación. Además se agrega la restricción que se pueden usar como mínimo 2 puntos de interpolación y como máximo 100 puntos de interpolación, este rango es muy útil para restringir nuestra búsqueda del número de puntos de interpolación.

Para evaluar la calidad del interpolador obtenido consideraremos la siguiente definición de error,

$$\mathrm{Error}(n) = \max_{x\in[-1,1]} |\mathrm{pLog}_n(\exp(x)) - x|.$$

Sin embargo, como queremos obtener una estimación solamente, se propone utilizar la siguiente discretización de la función de error $\mathrm{Error}(n)$,

$$\mathrm{ErrorD}(n) = \max_{i\in\{0,1,2,\ldots,N\}} |\mathrm{pLog}_n(\exp(x_i)) - x_i|,$$

donde $x_i = -1 + \dfrac{2\,i}{N}$ y $N = 100000$, es decir los x_i son puntos equiespaciados en $[-1, 1]$.

Preguntas:

(a) Construya una interpolación polinomial que interpole $\log(y)$ para el rango indicado anteriormente sin que se vea afectado por el fenómeno de Runge. El algoritmo debe retornar una función "callable" y debe tener la siguiente firma:

```
def interpolateLog(n):
    # Your code
    return pLog
```

Por ejemplo, con la función `interpolateLog` podemos construir nuestro interpolador `pLog=interpolateLog(n)` y luego la evaluación *vectorial* se realiza de la siguiente forma `out=pLog(x)`, donde `x=np.linspace(a,b,10)`, para un intervalo $[a, b]$. Como caso particular considere que `pLog=interpolateLog(5)`, entonces el resultado al evaluar `pLog([0.9,1.5])` debe ser `[-0.102978,0.40662723]`.

(b) Implemente ErrorD(n), debe considerar la siguiente firma:

```
def ErrorD(n):
    # Yout code
    return discrete_error
```

(c) Evalúe el error de interpolación discreto ErrorD(n) para $n = 1$.

(d) Construya la siguiente función:

```
def find_n_log(threshold):
    # Your code
    return n
```

Esta función debe recibir la variable `threshold` que corresponde al máximo error discreto permitido y debe retornar el mínimo valor de n tal que se cumpla con el error `threshold` definido como parámetro. Por ejemplo si la variable `threshold` se define como $1e-1$ y usted retornar el mínimo `n` tal que se cumpla que ErrorD(n) <`threshold`. En caso de que no pueda encontrar un valor de `n` que cumpla con el `threshold` requerido en el rango de valores de n indicado, la función debe retornar el valor -1.

(e) Evalúe `find_n_log(threshold)` con `threshold=1e-8`.

(f) ¿Sigue siendo correcto el análisis anterior si se alternar el orden de evaluación entre pLog(x) y exp(x) en la definición de ErrorD(n), es decir pasar de $|\text{pLog}_n(\exp(x_i)) - x_i|$ a $|\exp(\text{pLog}_n(x_i)) - x_i|$?

2.3. ¿Cómo sería conveniente aplicar la teoría acá?

Considere la siguiente función $f(x) = \sin(\alpha \cos(x))$, donde α es un parámetro. La tarea a resolver es la implementación de una aproximación de esta función en un computador de capacidad limitada, es decir, en este computador solo tenemos a nuestra disposición las operaciones elementales suma, resta, multiplicación y división, y adicionalmente la función módulo, es decir, el resto de la división, por ejemplo considere la implementación de NumPy del módulo que nos entrega `np.mod(7.2,np.pi)=0.7168146928204138`.

Para la implementación de $f(x)$ en este computador estudiaremos 2 algoritmos:

- Algoritmo 1: Construir un interpolador polinomial (interpolación baricéntrica) con n puntos de Chebyshev en el dominio $[0, 2\,\pi]$.
- Algoritmo 2: Construir un interpolador polinomial "compuesto", es decir, primero se construyen dos interpoladores polinomiales (interpolación baricéntrica) con n puntos de Chebyshev de las funciones $\sin(x)$ y $\cos(x)$, ambas en el intervalo $[0, 2\,\pi]$, denominadas `p_sin(x)` y `p_cos(x)`, y luego se aproxima $f(x)$ como `p_sin(`α`p_cos(x))`. Note que en este caso hay que tener un cuidado especial al evaluar los interpoladores dado que las funciones $\sin(x)$ y $\cos(x)$ son interpoladas en el intervalo $[0, 2\,\pi]$ solamente.

Para la evaluación de los algoritmos anteriores se solicita las implementaciones de las siguientes funciones en Python:

- `polynomialInterpolation_f`: Esta función realiza una interpolación polinomial Baricéntrica *tradicional* de la función $f(x)$, esto significa evaluar $f(x)$ en n puntos de Chebyshev y construir el interpolador polinomial utilizando la interpolación Baricéntrica. En este estudio se sugiere utilizar `BarycentricInterpolator` de SciPy.
- `compositeConstructionSinCos`: Esta función recibe como parámetro n, la cantidad de puntos de Chebyshev a utilizar, y retorna 2 funciones *callables* que interpolan polinomialmente, utilizando la interpolación Baricéntrica, las funciones $\sin(x)$ y $\cos(x)$, denominadas `p_sin(x)` y `p_cos(x)`, respectivamente.

- `compositeEvaluationSinCos`: Esta función implementa la evaluación compuesta de la función $f(x)$ por medio de las aproximaciones polinomiales `p_sin(x)` y `p_cos(x)` de las funciones $\sin(x)$ y $\cos(x)$, respectivamente. Esta función recibe como parámetros las interpolaciones polinomiales `p_sin(x)` y `p_cos(x)`, el coeficiente α, y un NumPy array unidimensional `x`, que es donde se debe evaluar la aproximación compuesta de $f(x)$, es decir debe reemplazar $\sin(x)$ y $\cos(x)$ por sus aproximaciones polinomiales y teniendo en cuenta todos los supuestos implícitamente considerados en las interpolaciones.

Preguntas:

(a) Utilizando el "Teorema de Chebyshev" (incluido en los apuntes del curso), obtenga una cota para el peor caso del error de interpolación polinomial para el Algoritmo 1 considerando el intervalo $[0, 2\,\pi]$. Este es un resultado algebraico.

(b) Implemente `polynomialInterpolation_f`.

(c) Implemente `compositeConstructionSinCos`.

(d) Implemente `compositeEvaluationSinCos`.

(e) Obtenga el error de las interpolación antes definidas utilizando la siguientes cantidades de puntos de interpolación: $\{10, 15, 20, 30, 40, 5$ este es claramente un trabajo computacional pero es muy bueno resolverlo para entender que es lo que ocurre. Notar que el error se debe realizar en una grilla mucho más final, por ejemplo considerando 1000 equiespaciados en el intervalo de interpolación. Los errores deben graficarse en escala logarítmica en el eje-y y lineal en el eje-x.

(f) Para el Algoritmo 1 podemos indicar que el error decae de la siguiente forma: $E_1(n) = A_1\, n^{B_1}$, donde n es el número de puntos utilizados en la interpolación y $E_1(n)$ corresponde al máximo error obtenido con n puntos de interpolación. Explique como puede obtener A_1 y B_1, y obténgalos. Se solicita utilizar solo los "datos" hasta antes que se sature el error, sin embargo si usted prefiere no seguir esta solicitud, debe justificar claramente la razón.

(g) Para el Algoritmo 2 podemos también indicar que el error decae de la siguiente forma: $E_2(n) = A_2\, n^{B_2}$, donde n es el número de puntos utilizados en la interpolación y $E_2(n)$ corresponde al máximo error obtenido con n puntos en la interpolación compuesta. Explique como puede obtener A_2 y B_2, y obténgalos. Se solicita utilizar solo los "datos" hasta antes que se sature el error, sin embargo si usted prefiere no seguir esta solicitud, debe justificar claramente la razón.

2.4. Ajedrez

Considere el siguiente patrón matricial del "tipo" *tablero de ajedrez* para los coeficientes no-nulos,

$$C^{(3)} = \begin{bmatrix} \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & \times & \times \end{bmatrix} \in \mathbb{R}^{6\times 6}, \quad C^{(5)} = \begin{bmatrix} \times & 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times & \times \\ \times & 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 & \times & 0 & \times & \times & \times \end{bmatrix} \in \mathbb{R}^{10\times 10},$$

donde "$\times$" corresponde a un coeficiente no-nulo (no necesariamente todos iguales) y "0" denota un coeficiente nulo (es decir, igual a 0). Notar que las 2 últimas columnas contienen coeficientes no-nulos completamente. El patrón puede extenderse fácilmente a matrices de menor y mayor tamaño con dimensión par. En el caso general uno puede considerar una matriz $C^{(n)} \in \mathbb{R}^{2\,n\times 2\,n}$, es decir,

$$C^{(n)} = \begin{bmatrix} \times & 0 & \times & 0 & \dots & \dots & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & \dots & \dots & 0 & \times & \times & \times \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \times & 0 & \times & 0 & \dots & \dots & \times & 0 & \times & \times \\ 0 & \times & 0 & \times & \dots & \dots & 0 & \times & \times & \times \end{bmatrix} \in \mathbb{R}^{2\,n\times 2\,n}.$$

Adicionalmente conocemos que los coeficientes no-nulos son positivos y que la suma de los coeficientes de cada fila, excluyendo el coeficiente de la diagonal, es exactamente la mitad del valor del coeficiente de la diagonal, es decir, si denotamos los coeficientes de la matriz $C^{(n)}$ como $C^{(n)}_{i,j}$ para $i, j \in \{1, 2, 3, \ldots, 2\,n\}$, entonces tenemos,

$$\frac{C^{(n)}_{i,i}}{2} = \left(\sum_{j=1, j\neq i}^{2\,n} C^{(n)}_{i,j} \right) > 0, \quad \forall i \in \{1, 2, \ldots, 2\,n\}.$$

Otra característica importante de la matriz $C^{(n)}$ es que necesitamos almacenar en memoria solo los coeficientes no-nulos dado la regularidad del patrón tipo *tablero de ajedrez*, los cuales son $n+1$ coeficientes por cada fila. Esto significa que podemos almacenar los coeficientes no-nulos de la matriz $C^{(n)}$ en la matriz $D^{(n)} \in \mathbb{R}^{2\,n\times(n+1)}$, es decir,

$$D^{(n)} = \begin{bmatrix} \times & \times & \ldots & \times \\ \times & \times & \ldots & \times \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \times & \times & \ldots & \times \\ \times & \times & \ldots & \times \end{bmatrix} \in \mathbb{R}^{2\,n\times(n+1)}.$$

¡Lo cual reduce los requerimientos de memoria a aproximadamente la mitad! Ahora, el desafío es resolver sistemas de ecuaciones lineales de la forma $C^{(n)}\,\mathbf{x} = \mathbf{b}$. En particular, estamos interesados en utilizar el método de Jacobi.

(a) Determine el número de operaciones elementales requeridas para realizar una iteración del método de Jacobi para resolver $C^{(n)}\,\mathbf{x} = \mathbf{b}$ tomando ventaja del patrón del "tipo" *tablero de ajedrez* presentado. Recuerde que debe incluir todo el desarrollo para obtener todo el puntaje.

(b) Determine si el método de Jacobi convergerá o no al utilizarlo en el sistema de ecuaciones lineales $C^{(n)}\,\mathbf{x} = \mathbf{b}$.

(c) Implemente una iteración del método de Jacobi antes propuesto en Python utilizando adecuadamente la capacidad de vectorización[1] de NumPy y considerando que el *input* de su algoritmo es n, la matriz $D^{(n)}$, el *right-hand-side* $\mathbf{b}$ y un *initial guess* $\mathbf{x}_0$ para el método de Jacobi. Para este caso no está permitido en la implementación reconstruir la matriz $C^{(n)}$ a partir de la matriz $D^{(n)}$ por restricciones de memoria. La firma de la función es la siguiente[2]:

```
'''
input:
n   : (int64) Super index of C^{(n)} and D^{(n)}
Dn  : (ndarray) Matrix of coefficients of dimension 2*n times (n+1).
b   : (ndarray) Array of dimension 2*n with the coefficientes of the right-hand-side of Cn*x=b.
x0  : (ndarray) Array of dimension 2*n with the initial guess x0.

output:
x1  : (ndarray) Array of dimension 2*n that stores the output of one iteration of the Jacobi method.
'''
def OneStepJacobi(n, Dn, b, x0):
    # Your own code.
    return x1
```

2.5. ¿Lineal o no-lineal?

Considere la siguiente función en varias variables:

$$f(x, a, b, \omega) = a\,\sin(x) + b\,\cos(\omega\,x).$$

La cual se quiere utilizar para aproximar el siguiente conjunto de datos: (x_1, y_1) y (x_2, y_2), es decir se requiere que se cumplan las siguientes ecuaciones:

$$f(x_1, a, b, \omega) = a\,\sin(x_1) + b\,\cos(\omega\,x_1) = y_1,$$
$$f(x_2, a, b, \omega) = a\,\sin(x_2) + b\,\cos(\omega\,x_2) = y_2.$$

De inmediato notamos que tenemos 3 coeficientes desconocidos, es decir a, b y ω, sin embargo solo tenemos 2 ecuaciones. Para resolver este problema, considere la información adicional que se entregará en cada pregunta.

Preguntas:

[1] Es decir, reducir al máximo el uso de "loops".

[2] Para el desarrollo no es necesario que repita completamente la descripción de inputs y outputs.

(a) Considere que $b = \widehat{b}$, es decir, conocemos el valor de b. Entregue todas las componentes necesarias para utilizar el método de Newton en $\mathbb{R}^2$, es decir, debe indicar explícitamente la función $\mathbf{F}(\cdot) : \mathbb{R}^2 \to \mathbb{R}^2$ a la cual se le buscará la raíz, la matriz Jacobiana respectiva y cómo se utiliza para realizar una iteración considerando como initial guess a_0 y ω_0. Notar que no se solicita la inversa de la matriz asociada, solo la descripción explícita de la matriz. Justifique su resultado.

(b) Considere que $\omega = \widehat{\omega}$, es decir, conocemos el valor de ω, y además se nos entrega un par ordenado adicional, es decir, (x_3, y_3). Determine el sistema de ecuaciones lineales **cuadrado** que entrega los coeficientes $\overline{a}$ y $\overline{b}$ que minimizan el error cuadrático correspondiente. No es necesario que resuelva el sistema de ecuaciones lineales, solo que lo indique explícitamente la matriz de 2×2 respectiva y el lado derecho asociado. Justifique su resultado.

2.6. Secuencia de factorizaciones $Q\,R$

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,

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

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,

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

A partir de las identidades anteriores, 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 convenientemente 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!*

Preguntas

(a) Explique teóricamente como obtendrá la factorización QR reducida de T_{n+1} a partir de la factorización QR reducida de T_n.

(b) 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 esta forma es que reduce significativamente la cantidad de operaciones elementales requerida, por lo tanto acelera la computación significativamente.

```
,,,
INPUT:
Qn    : (ndarray) Matriz unitaria Q_n de la factorización QR reducida de la matriz T_n=Q_n*R_n.
Rn    : (ndarray) Matriz triangular superior R_n de la factorización QR reducida
de la matriz T_n=Q_n*R_n.
a     : (float)   Coeficiente a_{n+1} de la matriz T_{n+1}.
b     : (float)   Coeficiente b_{n+1} de la matriz T_{n+1}.
c     : (float)   Coeficiente c_n de la matriz T_{n+1}.

OUTPUT:
Q_next : (ndarray) Matriz unitaria Q_{n+1} de la factorización QR reducida de la
matriz T_{n+1}=Q_{n+1}*R_{n+1}.
R_next : (ndarray) Matriz triangular superior R_{n+1} de la factorización QR
```

```
reducida de la matriz T_{n+1}=Q_{n+1}*R_{n+1}.
'''
def incrementalQR(Qn,Rn,a,b,c):
    # Your own code.
    Q_next = np.zeros(2,1) # Remover esta línea si usted implementa esta función.
    R_next = np.zeros(1,1) # Remover esta línea si usted implementa esta función.
    return Q_next, R_next
```

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

2.7. Polinomio exponencial

Considere la siguiente función,

$$f(x) = P_n(x)\, \exp\left(Q_m(x)\right), \quad x \in [-1, 1],$$

donde,

$$P_n(x) = \sum_{i=0}^{n} a_i\, x^i, \quad a_i \in \mathbb{R},\, n \in \mathbb{N},$$

$$Q_m(x) = \sum_{j=0}^{m} b_j\, x^j, \quad b_i \in \mathbb{R},\, m \in \mathbb{N}.$$

Sin embargo, se sabe que los coeficientes a_i y b_j fueron obtenidos resolviendo numéricamente un sistema de ecuaciones lineales donde la matriz asociada era una matriz del tipo Vandermonde, que se sabe es muy *mal condicionada* y por lo tanto los coeficientes pueden contener un error significativo. El problema que genera esto es que al evaluar los polinomios $P_n(x)$ y $Q_m(x)$ los resultados obtenidos no son confiables y por ende la evaluación de la función $f(x)$ tampoco sería confiable. Afortunadamente se nos ha informado que tenemos acceso a las funciones `P_expensive(x)` y `Q_expensive(x)` que nos entregan en valor exacto de $P_n(x)$ y $Q_m(x)$, respectivamente. En principio, las funciones `P_expensive(x)` y `Q_expensive(x)` podrían utilizarse para implementar la función $f(x)$, sin embargo son muy costosas de evaluar, por lo que no está permitido usarlas para evaluar $f(x)$ de forma intensiva pero si podrían usarse para obtener valores particulares. Entonces, el desafío es construir un algoritmo y una implementación que permita la rápida y correcta evaluación de $f(x)$, es decir, reduciendo el error o incluso haciéndolo 0.

(a) Proponga un algoritmo utilizando interpolación polinomial para aproximar la función $f(x)$ donde el error máximo sea ε, el cual incluso puede ser 0; que no se vea afectado por el fenómeno de Runge; y que minimice la cantidad de operaciones elementales para su evaluación. Considere que usted tiene a su disposición la posibilidad de evaluar la función exponencial sin un costo significativo. En su propuesta de algoritmo debe explicar claramente los parámetros que utilizará y las razones de la elección de el/los mismo/s. En caso de que usted pueda hacer el error de aproximación exactamente 0, explique claramente como lo logrará. *Hint: We strongly suggest you to make the error equal to 0.*

(b) Implemente en Python utilizando adecuadamente la capacidad de vectorización de NumPy el procedimiento propuesto anteriormente para la construcción de la aproximación de $f(x)$. Notar que en este caso no es necesario implementar el procedimiento en una única función. Para su implementación, considere que tiene a su disposición las siguientes funciones:

- `pV=Vandermonde(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` y y_i en el vector `yi`, y retorna la función *callable* `pV` que se construyó utilizando la interpolación con la matriz de Vandermonde.
- `pL=Lagrange(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` y y_i en el vector `yi`, y retorna la función *callable* `pL` que se construyó utilizando la interpolación de Lagrange.
- `pB=BarycentricInterpolation(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` y y_i en el vector `yi`, y retorna la función *callable* `pB` que se construyó utilizando la interpolación Baricéntrica.

2.8. Schmidt-Gram

Recordando a los ilustres profesores Jørgen Pedersen Gram y Erhard Schmidt, los cuales desarrollaron el proceso de ortonormalización de vectores en un espacio vectorial equipado con un producto interno. Conocido como la descomposición QR, en particular la descomposición QR reducida de una matriz $A \in \mathbb{R}^{m\times n}$ con $m \geq n \geq 1$, genera la matriz ortonormal $\widehat{Q} \in \mathbb{R}^{m\times n}$, i.e. sus columnas son ortonormales, y la matriz $\widehat{R} \in \mathbb{R}^{n\times n}$ es triangular superior, como se muestra a continuación:

$$A = \begin{pmatrix} \mathbf{a}_1 & | & \mathbf{a}_2 & | & \cdots & | & \mathbf{a}_n \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{q}_1 & | & \mathbf{q}_2 & | & \cdots & | & \mathbf{q}_n \end{pmatrix}}_{\widehat{Q}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}}_{\widehat{R}}$$

Sin embargo, en uno de los posibles futuros de la humanidad, se ha decretado que no se podrán utilizar matrices triangulares superiores en los procesos de ortonormalización. Lo cual deja a la humanidad de forma inmediata sin acceso a resolver problemas de mínimos cuadrados por medio de la clásica factorización QR...

Esto genera el inicio de la revolución científica liderada por los estudiantes de Computación Científica, donde su más fuerte habilidad es la construcción de algoritmos sofisticados que velen por el continuo avance de la Ciencia y la Ingeniería considerando la restricción definida, i.e. no utilizar matrices triangulares superiores.

Para resolver este problema, se propone construir la siguiente factorización matricial:

$$A = \begin{pmatrix} \mathbf{a}_1 & | & \mathbf{a}_2 & | & \cdots & | & \mathbf{a}_n \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{t}_1 & | & \mathbf{t}_2 & | & \cdots & | & \mathbf{t}_n \end{pmatrix}}_{\widehat{T}} \underbrace{\begin{pmatrix} u_{11} & 0 & 0 & \cdots & 0 \\ u_{21} & u_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ u_{n1} & u_{n2} & \cdots & \cdots & u_{nn} \end{pmatrix}}_{\widehat{U}}$$

(a) Construya un algoritmo que determine la factorización TU propuesta, i.e. el input del algoritmo es la matriz A y retorna la matriz $\widehat{T}$ donde sus columnas son ortonormales y la matriz $\widehat{U}$ que es triangular inferior.

(b) Implemente en Python su algoritmo.

2.9. Dólares

El precio del dólar ha estado fluctuando bastante este último tiempo. Este tipo de fluctuaciones ha sido visto como una interesante posibilidad de inversión por expertos del área de High Frequency Trading de finanzas. Sin embargo, para poder rentabilizar su negocio ellos necesitan saber si el dólar subirá o bajará en el futuro inmediato, ya que en ambos casos ellos ganan, pero la estrategia es diferente. Para poder responder a su pregunta, ellos han propuesto utilizar un algoritmo autoregresivo de segundo orden para estimar el valor futuro de dólar. Este algoritmo consiste en estimar el precio del dolar de un instante como una combinación lineal del precio del dolar en 2 instantes inmediatamente anteriores. En términos matemáticos esto significa lo siguiente:

$$X^{(n+1)} = \alpha_0 X^{(n)} + \alpha_1 X^{(n-1)},$$

donde $X^{(n)}$ corresponde al precio del dólar en el tiempo n. Considere la Tabla 2 para el precio del dólar en los tiempos discretos indicados.

n	$X^{(n)}$
0	812
1	806
2	797
3	791
4	791

Tabla 1: Precio del dolar

Por simplicidad solo nos referiremos a instantes discretos y equiespaciados en el tiempo.

(i) Explique cómo determinaría las constantes α_0 y α_1, y encuentrelas. Hint: You should get a rectangular matrix of 3×2 and use an algorithm for that.

(ii) Utilice la aproximación con los α's estimados en **(i)** para estimar el valor del dolar en el tiempo $n = 5$.

(iii) Considerando que el valor del dólar verdadero para $n = 5$ fue 797. ¿Considera buena su estimación realizada en **(ii)**? Justifique. Recuerde que la empresa gana si sabe si el dólar subirá o bajará solamente.

2.10. Preciso y conciso

Construya un algoritmo basado en la descomposición QR que determine si una matriz cuadrada es singular o no, e impleméntelo en Python con Numpy.

2.11. ¿Más ecuaciones que incógnitas?

Sea $A \in \mathbb{R}^{m\times n}$ con $m > n$ y $\mathbf{b} \in \mathbb{R}^m$. Considere el siguiente sistema de ecuaciones lineales sobre-determinado:

$$A\,\mathbf{x} = \mathbf{b}. \tag{1}$$

El procedimiento tradicional para encontrar $\bar{\mathbf{x}}$ es por medio de minimizar el error cuadrático de (1), es decir,

$$\bar{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{b} - A\,\mathbf{x}\|.$$

El cual puede ser obtenido a través de las ecuaciones normales,

$$A^*\,A\,\bar{\mathbf{x}} = A^*\,\mathbf{b},$$

donde * corresponde al operador de trasposición y conjugación. Este método es muy utilizado en Machine Learning, pero una variante es quizás más utilizada, i.e. la regularización de Tikhonov. La cual consiste en resolver el siguiente sistema de ecuaciones lineales,

$$(A^*\,A + \delta\,I)\,\bar{\mathbf{x}}_\delta = A^*\,\mathbf{b},$$

donde I es la matriz identidad de orden n y $0 < \delta \in \mathbb{R}$. Ahora, considerando la descomposición QR-reducida de $A = \widehat{Q}\,\widehat{R}$, se propone la siguiente regularización,

$$(A^*\,A + \lambda\,\widehat{R})\,\widehat{\bar{\mathbf{x}}}_\lambda = A^*\,\mathbf{b}. \tag{2}$$

(a) Considerando que ya se tiene acceso a la descomposición QR-reducida de A, proponga un algoritmo $\mathcal{O}(n^2)$ que obtenga $\widehat{\bar{\mathbf{x}}}_\lambda$.

2.12. ¿Es o no es tridiagonal?

Considere el siguiente sistema de ecuaciones lineales:

$$\underbrace{\begin{pmatrix} 0 & \dots & \dots & \dots & 0 & b & a \\ 0 & \dots & \dots & 0 & b & a & c \\ 0 & \dots & 0 & b & a & c & 0 \\ \vdots & \iddots & \iddots & \iddots & \iddots & \iddots & \vdots \\ 0 & \iddots & \iddots & \iddots & \iddots & \dots & \vdots \\ b & a & c & 0 & \dots & \dots & 0 \\ a & c & 0 & \dots & \dots & \dots & 0 \end{pmatrix}}_{A} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \\ \vdots \\ 1/(n-1) \\ 1/n \end{pmatrix} \tag{3}$$

donde se conoce que $|c| < |b| < \dfrac{|a|}{2}$ y $A \in \mathbb{R}^{n\times n}$.

(a) Proponga un algoritmo **nuevo** basado en los algoritmos vistos en el curso que tome ventaja de la estructura y de la información entregada del sistema de ecuaciones lineales (3) que permita obtener los coeficientes x_1, x_2, …, y x_n, dado a, b, c y n. Hint: You have complete freedom in how to find x_1, x_2, …, and x_n! Make sure you propose an algorithm that does find them. Hint2: You may consider aditional parameters, but you should justify their use.

(b) Implemente el algoritmo propuesto considerando que el **input** es: a, b, c, y n; y el **output** es un NumPy Array con los coeficientes x_1, x_2, …, y x_n.

(c) Obtenga los coeficientes x_1, x_2, …, y x_n para el siguiente input: $a = 10$, $b = -4$, $c = 1/2$ y $n = 3000$.

(d) Implemente un algoritmo que dado los parámetros $a = 10$, $b = -4$, $c = 1/2$ y $n = 3000$ pueda determinar si unos coeficientes x_1, x_2, …, y x_n, satisfacen la ecuación matricial (3), y utilícelo para verificar si en (c) obtuvo los coeficientes esperados.

2.13. ¿Se puede obtener la inversa de la matriz de Vandermonde?

El primer algoritmo que tradicionalmente se discute cuando uno estudia interpolación polinomial en una variable utiliza la matriz de Vandermonde y rápidamente uno llega a la conclusión que la matriz es mal condicionada, por lo cual es poco recomendable trabajar directamente con esta matriz utilizando aritmética de punto flotante. El segundo algoritmo que uno tradicionalmente estudia es la Interpolación de Lagrange y luego el algoritmo de Diferencias Divididas de Newton, ambos algoritmos son capaces de encontrar el polinomio interpolador en cuestión.

Considere el problema de interpolar 3 puntos: (x_1, y_1), (x_2, y_2) y (x_3, y_3), donde $x_1 \neq x_2$, $x_1 \neq x_3$ y $x_2 \neq x_3$. Si uno utiliza la matriz de Vandermonde uno obtiene el siguiente sistema de ecuaciones lineales:

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix}}_{V_3} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}. \tag{4}$$

Donde a, b y c, son los coeficientes del polinomio interpolador $p_3(x) = a + b\,x + c\,x^2$. Un dato importante sobre la matriz de Vandermonde cuando interpola n puntos es que su determinante puede ser expresado de la siguiente forma:

$$\det(V_n) = \prod_{1 \leq i < j \leq n} (x_j - x_i).$$

Lo que indica que el determinante no es nulo siempre y cuando $x_i \neq x_j \ \forall i \neq j$. En nuestro caso particular nosotros tenemos $\det(V_3) = \prod_{1 \leq j < i \leq 3} (x_j - x_i) \neq 0$. Ahora, dado que sabemos que V_3 no es singular, podemos estudiar la inversa de esta matriz, i.e. V_3^{-1}.

(a) Construya la interpolación de Lagrange o de Diferencias de Newton para obtener el polinomio interpolador de (x_1, y_1), (x_2, y_2) y (x_3, y_3).

(b) Explique como puede usted obtener V_3^{-1} a partir de las interpolaciones anteriormente encontradas.

(c) Obtenga V_3^{-1} utilizando el procedimiento descrito anteriormente. Note que V_3^{-1} satisface la siguiente ecuaci´n:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \underbrace{\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}}_{V_3^{-1}} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}. \tag{5}$$

Donde $\left(V_3^{-1}\right)_{ij} = b_{ij}$ para $i, j = 1, 2, 3$, i.e. b_{ij} son los coeficientes de la matriz V_3^{-1}.

2.14. Sistemas de ecuaciones lineales de sistemas de ecuaciones lineales

En general, el primer sistema de ecuaciones lineales que uno estudia es un sistema de 2×2, es decir 2 ecuaciones con 2 incógnitas, por ejemplo:

$$y = a_1\,x + b_1,$$
$$y = a_2\,x + b_2,$$

donde y, x, a_1, a_2, b_1, y b_2 pertenecen a $\mathbb{R}$, es decir son números reales. En particular sabemos que en este caso si,

$$\det \begin{pmatrix} 1 & -a_1 \\ 1 & -a_2 \end{pmatrix} = a_1 - a_2 \neq 0,$$

existe una única solución para x y y que satisface ambas ecuaciones. Una forma de resolver el sistema de ecuaciones lineales anterior es simplemente **igualando** la variables y, es decir:

$$a_1\,x + b_1 = a_2\,x + b_2.$$

De la cual podemos despejar x de la siguiente forma:

$$x = \frac{b_2 - b_1}{a_1 - a_2}.$$

Ahora, ¿qué ocurriría si reemplazamos los coeficientes e incógnitas por matrices de dimensión 2×2? Esto significa que las ecuaciones se transforman a la siguiente forma:

$$Y = A_1\,X + B_1,$$
$$Y = X\,A_2 + B_2,$$

donde Y, X, A_1, A_2, B_1, y B_2 pertenecen a $\mathbb{R}^{2\times 2}$. Note que el orden de la multiplicación de X por las matrices A_1 y A_2 no es un error, así se propone la generalización. Considere las siguientes definiciones para X, Y, A_k, y B_k, para $k \in \{1, 2\}$:

$$X = \begin{pmatrix} x_1 & x_3 \\ x_2 & x_4 \end{pmatrix}, Y = \begin{pmatrix} y_1 & y_3 \\ y_2 & y_4 \end{pmatrix}, A_k = \begin{pmatrix} a_{k,1} & a_{k,3} \\ a_{k,2} & a_{k,4} \end{pmatrix}, \text{ y } B_k = \begin{pmatrix} b_{k,1} & b_{k,3} \\ b_{k,2} & b_{k,4} \end{pmatrix}.$$

Al aplicar el mismo procedimiento anterior para el caso con coeficientes reales, uno no encuentra directamente un sistema de ecuaciones lineales en su forma tradicional. Sin embargo sí se puede re-escribir como un sistema de ecuaciones lineales de dimensión 4×4, solo considerando como incógnita la matriz X. Para el manejo del problema considere el siguiente orden de los coeficientes. Poner especial atención a los coeficientes del lado derecho ya que definen el orden de las filas de la matriz C y el orden de los x_i ya que define el orden de las columnas de la matriz C,

$$\underbrace{\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}}_{C} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_{2,1} - b_{1,1} \\ b_{2,2} - b_{1,2} \\ b_{2,3} - b_{1,3} \\ b_{2,4} - b_{1,4} \end{pmatrix}.$$

Notar que los coeficientes $c_{i,j}$, para $i, j \in \{1, 2, 3, 4\}$, son desconocidos y los debe obtener. Para el desarrollo de las siguientes preguntas, considere las siguientes matrices:

$$A_1 = \begin{pmatrix} a_{1,1} & a_{1,3} \\ a_{1,2} & a_{1,4} \end{pmatrix}, A_2 = \begin{pmatrix} a_{2,1} & a_{2,3} \\ a_{2,2} & a_{2,4} \end{pmatrix}, B_1 = \begin{pmatrix} b_{1,1} & b_{1,3} \\ b_{1,2} & b_{1,4} \end{pmatrix}, \text{ y } B_2 = \begin{pmatrix} b_{2,1} & b_{2,3} \\ b_{2,2} & b_{2,4} \end{pmatrix}.$$

(a) En el caso original, con coeficientes reales, se indicó que si $a_1 = a_2$ no tiene solución única el problema. ¿Es posible que exista solución única en el problema modificado si $A_1 = A_2$? Explique

(b)
- Si la matriz A_1 fuera la matriz nula, ¿sigue siendo no singular la matriz C?
- Si la matriz A_2 fuera la matriz nula, ¿sigue siendo no singular la matriz C?

(c)
- ¿Es siempre posible obtener la factorización LU de C?.

(d) Proponga e implemente un algoritmo que obtenga X basado en la factorización PALU de C.

2.15. Una matriz bidiagonal

Tony Stark se ha dado cuenta al analizar la data de las gemas del infinito que todas usan una variante de la matriz bi-diagonal A_n para poder controlar su poder. Sin embargo, para acceder al poder se debe resolver muchos sistemas de ecuaciones lineales de la forma $A_n \mathbf{x}_i = \mathbf{u}_i$ donde el sub-índice i denota al i-ésimo vector, respectivamente. La ventaja que tenemos es que conocemos la estructura de la matriz no-singular A_n:

$$A_n = \begin{pmatrix} 1 & -2 & 0 & \dots & 0 \\ 0 & 1 & -2 & 0 & \dots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & 1 & -2 \\ 0 & \dots & \dots & 0 & 1 \end{pmatrix} \in \mathbb{R}^{n \times n}.$$

(a) ✍ Determine A_3^{-1} convenientemente, donde:

$$A_3 = \begin{pmatrix} 1 & -2 & 0 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Hint: You may find the letters B.S. useful, we hope.

Hint2: Solve $A_3\,(x_1, x_2, x_3)^T = (b_1, b_2, b_3)^T$ and then write $(x_1, x_2, x_3)^T$ as the product of a matrix B times the vector $(b_1, b_2, b_3)^T$, the matrix B will be A_3^{-1}, and of course ^{T} is the transpose operator.

(b) ✍ Determine A_n^{-1} convenientemente.
Hint: Recall that if you want to solve $A\,\mathbf{x} = \mathbf{b}$ the solution is $\mathbf{x} = A^{-1}\,\mathbf{b}$, i.e. if you know the solution and you can write it as product of certain coefficients and the coefficients of $\mathbf{b}$, then you have the inverse!

(c) ✍ Determine $\kappa_\infty(A_n) = \|A_n\|_\infty \|A_n^{-1}\|_\infty$.
Hint: Just in case $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$, for $a \in \mathbb{R}$.

(d) ⌨ Implemente computacionalmente A_n^{-1} basado en (b). No puede obtener la inversa numéricamente, debe hacerlo algebraicamente.

(e) ⌨ Verifique que puede obtener la solución de $A_{100}\,\mathbf{x} = \mathbf{b}$, donde $\mathbf{b} =$ np.sin(np.linspace$(0, 2 *$ np.pi$, 100)$). Para verificar que se obtuve correctamente $\widehat{\mathbf{x}} = A_{100}^{-1}\,\mathbf{b}$, usted debe obtener $\|\mathbf{b} - A_{100}\,\widehat{\mathbf{x}}\|_2$ y basado en ese resultado concluir.

2.16. Matrices Toeplitz

Una importante familia de matrices son las matrices de Toeplitz:

$$T_n = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \dots & a_{-n+1} \\ a_1 & a_0 & a_{-1} & \dots & \vdots \\ a_2 & a_1 & a_0 & \dots & a_{-2} \\ \vdots & \ddots & \ddots & \ddots & a_{-1} \\ a_{n-1} & \dots & a_2 & a_1 & a_0 \end{pmatrix} \in \mathbb{R}^{n\times n},$$

donde $a_k \in \mathbb{R}$ para $k \in \{-n+1, \dots, n-1\}$. Este tipo de matrices es usada en procesamiento de series de tiempo, análisis de señales e imágenes, entre otros. El problema a tratar es nuestro conocido sistema de ecuaciones lineales $T_n\,\mathbf{x} = \mathbf{b}$, en donde $\mathbf{b}$ se interpreta como la señal de entrada y $\mathbf{x}$ como la señal de salida luego de haber sido procesada por la matriz respectiva, en nuestro caso T_n.

(a) Considerando que $a_j = 0$ para $|j| > 2$, determine el número de operaciones elementales mínimas requeridas para obtener la factorización LU de la matriz T_n. Por simplicidad considere que no es necesario hacer permutaciones. Hint: You don't need to make 0 what it is already 0. Even more, you don't need to add 0 either since you already know it won't change anything.

(b) Considere que usted tiene un computador que se demora aproximadamente $2.3\,[ns]$ por FLOP y que su matriz T_n utiliza $32\,[GB]$ de RAM almacenando todos los elementos de la matriz. ¿Cuanto es la reducción de tiempo al considerar la factorización LU de la pregunta anterior respecto a la factorización LU tradicional?
Hint: We will consider a FLOP (FLoating point OPeration) as the time it takes to compute $+,-,*$ or $/$.

2.17. De $\mathbb{C}^{n\times n}$ a $\mathbb{R}^{2\,n\times 2\,n}$

Considere el siguiente sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$, donde $A \in \mathbb{C}^{n\times n}$ y $\mathbf{b} \in \mathbb{C}^n$, por lo tango $\mathbf{x} \in \mathbb{C}^n$. Es decir, tenemos matrices y vectores donde sus elementos pueden ser número complejos. El uso de un algoritmo tradicional exige que debamos ser capaces de representar y manipular números complejos, sin embargo, esto aún no lo hemos estudiado. Una forma de manejar este tipo de situaciones es descomponer nuestro sistema de ecuaciones lineales en su parte real e imaginaria de la siguiente forma, $(A_r + \mathrm{i}\,A_i)\,(\mathbf{x}_r + \mathrm{i}\,\mathbf{x}_i) = \mathbf{b}_r + \mathrm{i}\,\mathbf{b}_i$, donde el subscript r denota la parte real, el subscript i denota la parte imaginaria y $\mathrm{i}^2 = -1$.

(a) Construya un nuevo sistema de ecuaciones lineales en el cual no se requiera manipulación explicita de números complejos. Describa claramente su nueva matriz, el vector de incógnitas y el lado derecho. Hint: Real is real and imaginary is imaginary!

(b) Considerando ahora que A_r es una matriz diagonal, y que $|(A_r)_{k,k}| > \sum_{j=1}^{n} |(A_i)_{k,j}|$ donde $k = 1:n$, proponga un algoritmo iterativo que asegure convergencia para el sistema de ecuaciones lineales propuesto en la parte **(a)**. Usted debe demostrar que el algoritmo propuesto convergerá. Notese que en la desigualdad anterior, la matriz que está al lado izquierdo es diferente a la matriz que está al lado derecho.

2.18. ¿Qué es A?

Considere el sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$, donde $\mathbf{b} \in \mathbb{R}^n$ es un vector arbitrario y $A \in \mathbb{R}^{n\times n}$ es una matriz cuyos coeficientes se definen de la siguiente forma:

$$A_{i,j} = \begin{cases} a & \text{if } i = 1 \text{ and } j = \{1, 2\} \\ a & \text{and } j = i \pm 1 \text{ and } i = 2 : n-1 \\ 2\,a & \text{if } i = j \text{ and } i > 1 \\ a & \text{if } i = n \text{ and } j = \{n-1, n\} \\ 0 & \text{otherwise.} \end{cases}$$

(a) Construya la descomposición $PA = LU$ de la matriz A.

(b) Si se utilizara el método de Jacobi para el sistemas de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$ antes descrito. ¿Es posible asegurar convergencia de el método de Jacobi? Hint: Use the infinity norm for matrices to do this.

2.19. ¿Se puede hacer eso?¿Cuales son las incógnitas?

Considere el siguiente sistema de ecuaciones lineales para $f_1(x)$ y $f_2(x)$:

$$\begin{aligned} x\, f_1(x) + f_2(x) + \sin(x)\, f_2(x) &= \exp(x) \\ f_1(x) + x^3\, f_2(x) &= p_n(x) \end{aligned} \tag{6}$$

donde $p_n(x)$ es un polinomio de grado $n-1$.

(a) Escriba (6) como un sistema de ecuaciones lineales de la forma $A(x)\,\mathbf{x}(x) = \mathbf{b}(x)$, donde (x) indica la dependencia de la variable independiente x.

(b) Determine la descomposición LU de $A(x)$. Note que L y U también dependerán de x. Considere que $x \neq 0$.

(c) ¿Qué descomposición sugeriría usted tal que existe $\forall\, x$? Justifique.

(d) Obtenga la descomposición que usted sugiere en la pregunta anterior.

(e) Resuelva el sistema de ecuaciones lineales (6) con la descomposición obtenida con alguna de las descomposiciones anteriormente encontradas.

2.20. ¡Llegó el valor absoluto!

Considere la ecuación (7), que corresponde a un *sistema de ecuaciones especial* , donde $A \in \mathbb{R}^{n\times n}$ una matriz estrictamente diagonal dominante, $\mathbf{x}$ y $\mathbf{b} \in \mathbb{R}^n$.

$$A\,\mathbf{x} - |\mathbf{x}| = \mathbf{b} \tag{7}$$

Además, $|\cdot|$ es el operador *element-wise* valor absoluto.

(a) Derive un algoritmo basado en un método iterativo para este tipo de sistemas de ecuaciones. Debe explicitar claramente todas las ecuaciones que usará y utilizar un criterio de detención adecuado.

(b) Suponga que un cierto sistema de ecuaciones de este tipo realiza m iteraciones para converger a una solución usando el algoritmo propuesto en (a). Para $n \gg 0$, ¿cuántas operaciones elementales realiza este algoritmo?

2.21. Ja..bi

Se desea resolver el sistema de ecuaciones lineales $A\,\mathbf{x} = \mathbf{b}$, donde $A \in \mathbb{R}^{3n\times 3n}$, $\mathbf{x} \in \mathbb{R}^{3\,n}$ y $\mathbf{b} \in \mathbb{R}^{3n}$. Se conoce que la matriz A tiene la siguiente estructura:

$$A = \begin{bmatrix} D_1 & B & C \\ B & D_2 & B \\ -C & B & D_3 \end{bmatrix},$$

con B una matriz que cumple $\|B\|_\infty = \varepsilon$ y D_i diagonal, para $i \in \{1,2,3\}$. Además, se sabe que:

$$\begin{aligned} \left|\, (D_1)_{i,i} \,\right| &> \sum_{j=1}^{n} \left|C_{i,j}\right| + 10^{-2}, \\ \left|\, (D_2)_{i,i} \,\right| &> \sum_{j=1}^{n} \left|C_{i,j}\right| + 10^{-2}, \\ \left|\, (D_3)_{i,i} \,\right| &> \sum_{j=1}^{n} \left|C_{i,j}\right| + 10^{-2}. \end{aligned}$$

Sin embargo, no se tiene acceso explícito a la matriz A, si no que solamente a las matrices que la componen, i.e. a D_1, D_2, D_3, B y C. Esto debido a que los valores de estas matrices provienen de un conjunto de mediciones y cálculos de otros programas. Para tener acceso a las matrices que componen A, se tiene la función getElementFromA(i,j), donde $i \in \{0,1,2\}$ y $j \in \{0,1,2\}$ indican la "coordenada" de la matriz que se desea obtener, por ejemplo, getElementFromA$(0,2)$ retorna la matriz C.

(a) ¿Para qué valores de ε se asegura convergencia para resolver $A\mathbf{x} = \mathbf{b}$ con el Método de Jacobi? *Hint: Recall that the infinite-norm of a matrix equals the maximum over the absolute sum of its rows.*

(b) Considere que por temas de disponibilidad de memoria, no es posible almacenar explícitamente más de una matriz que compone a A, es decir, solo es posible acceder a D_1, D_2, D_3, B, C o $-C$ en un momento específico. Proponga un algoritmo que resuelva el problema presentado mediante el método de Jacobi, considerando la restricción de memoria y que ε asegura convergencia en el Método de Jacobi. Considere como parámetros la función getElementFromA, el vector $\mathbf{b}$, un *initial guess* $\mathbf{x}_0$ y un número de iteraciones máximo K.

2.22. Back to the Future

Marty McFly ha ignorado lo que el Dr. Emmett "Doc" Brown le dejó en sus instrucciones: *"No intentes venir a buscarme a 1885"*. Como Marty ha viajado a 1885 y además ha dañado el DeLorean dejándolo sin combustible, el gran Doc se ve aproblemado porque necesita un nuevo combustible para reactivar el *condensador de flujo* generando los 1.2 gigawatts de potencia que necesita. Para esto, debe encontrar el *vector condensador* relacionado a una matriz *tridiagonal* $T_n \in \mathbb{R}^{n\times n}$ muy particular. Específicamente, el Doc debe resolver el sistema de ecuaciones $T_n\,\mathbf{v} = \mathbf{w}$, con $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ y donde $\mathbf{v}$ es el ansiado *vector condensador*. T_n se define a continuación como el producto de una matriz bidiagonal-"triangular inferior" y una matriz bidiagonal-"triangular superior",

$$T_n = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & \dots & 0 \\ -\frac{1}{2} & 1 & \ddots & & & & \vdots \\ 0 & -\frac{2}{3} & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & \ddots & -\frac{(i-1)}{i} & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 1 & 0 \\ 0 & \dots & \dots & \dots & 0 & -\frac{(n-1)}{n} & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & \dots & 0 \\ 0 & \frac{3}{2} & -1 & \ddots & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & & \\ \vdots & & \ddots & \frac{i+1}{i} & -1 & \ddots & \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \frac{n}{n-1} & -1 \\ 0 & \dots & \dots & \dots & \dots & 0 & \frac{n+1}{n} \end{bmatrix}.$$

Por ejemplo para $n = 5$:

$$T_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 & 0 \\ 0 & 0 & 0 & -\frac{4}{5} & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 & 0 \\ 0 & 0 & \frac{4}{3} & -1 & 0 \\ 0 & 0 & 0 & \frac{5}{4} & -1 \\ 0 & 0 & 0 & 0 & \frac{6}{5} \end{bmatrix} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

Además, McFly le indica al Doc que el vector del lado derecho del sistema de ecuaciones lineales se define como $\mathbf{w} = [f(x_1), f(x_2), \dots, f(x_j), \dots, f(x_n)]^T$, donde $f(x)$ es una función continua definida en $x \in [-1, 1]$, y $x_j = \frac{2}{n-1}(j-1) - 1$, con $j \in \{1, 2, \dots, n\}$. Sin embargo, McFly recuerda que no alcanzó a rescatar la definición de $f(x)$. Afortunadamente el Doc recordó que tiene anotado dos conjuntos de datos, con m puntos cada uno, donde registró el valor de la función $f(x)$. Sea $m < n$, entonces los conjuntos son:

(I) El conjunto S_1 de pares ordenados (x_k^e, y_k^e) donde $y_k^e = f(x_k^e)$ para $k \in \{1, 2, \dots, m\}$ donde la variable independiente x_k^e son m puntos equiespaciados en $[-1, 1]$. Por conveniencia se almacenan de la siguiente forma: $\mathbf{x}^e = [x_1^e, x_2^e, \dots, x_m^e]$ e $\mathbf{y}^e = [y_1^e, y_2^e, \dots, y_m^e]$.

(II) El conjunto S_2 de pares ordenados (x_k^c, y_k^c) donde $y_k^c = f(x_k^c)$ para $k \in \{1, 2, \dots, m\}$ donde la variable independiente x_k^c son m puntos de Chebyshev en $[-1, 1]$. Por conveniencia se almacenan de la siguiente forma: $\mathbf{x}^c = [x_1^c, x_2^c, \dots, x_m^c]$ e $\mathbf{y}^c = [y_1^c, y_2^c, \dots, y_m^c]$.

Lamentablemente ni el Doc ni McFly saben como utilizar alguno de los conjuntos de datos para aproximar los valores de $f(x)$ en los n puntos equiespaciados x_j (notar que estos puntos equiespaciados son distintos a los puntos equiespaciados x_k^e). El problema de construir el vector $\mathbf{w}$ se hubiera resuelto solo si es que el conjunto de datos S_1 tuviera n puntos, pero lamentablemente solo tiene m, donde $m < n$. En resumen, McFly y el Doc tienen que resolver 2 problemas computacionales para poder volver a 1985: Construir $\mathbf{w}$ y luego obtener $\mathbf{v}$.

(a) Construya un algoritmo que permita al Doc encontrar una estimación del vector condensador $\mathbf{v}$ para la ecuación $T_n\,\mathbf{v} = \mathbf{w}$, para cualquier n, **tomando ventaja** de la estructura de T_n presentada y construyendo una aproximación del vector $\mathbf{w}$. Debe explicar cada elemento a considerar en la construcción, tanto para obtener $\mathbf{w}$ como el vector condensador $\mathbf{v}$. Notar que usted dispone como "input" para obtener $\mathbf{w}$ los vectores $\mathbf{x}^e$ e $\mathbf{y}^e$, y $\mathbf{x}^c$ e $\mathbf{y}^c$ de los conjuntos de datos S_1 y S_2 respectivamente. Asegúrese de indicar las razones de la elección de puntos a utilizar.

(b) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) su 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`.
- `pV=Vandermonde(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` e y_i en el vector `yi`, y retorna la función *callable* `pV` que se construyó utilizando la interpolación con la matriz de Vandermonde.
- `pL=Lagrange(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` e y_i en el vector `yi`, y retorna la función *callable* `pL` que se construyó utilizando la interpolación de Lagrange.
- `pB=BarycentricInterpolation(xi,yi)`: Esta función recibe los puntos de interpolación x_i en el vector `xi` e y_i en el vector `yi`, y retorna la función *callable* `pB` que se construyó utilizando la interpolación Baricéntrica.

Notar que las funciones `pV`, `pL`, y `pB` están vectorizadas, por lo que pueden recibir un vector `x` y retorna un vector `y` donde se evaluó el polinomio interpolador para cada elemento de `x`. Recuerde que al momento de implementar usted debe decidir qué componentes se deben vectorizar y que componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input:
n : (integer) dimension of the matrix T_n.
output:
v : (ndarray) array of dimension n that stores the estimation of capacitor vector "v".
'''
def capacitor_vector(n):
    % Loading m-dimensional vector data $\mathbf{x}^e$ and $\mathbf{y}^e$
    xe = np.load('/ScientificComputing/DrEmmettData_xe.npz')
    ye = np.load('/ScientificComputing/DrEmmettData_ye.npz')
    % Loading m-dimensional vector data $\mathbf{x}^c$ and $\mathbf{y}^c$
    xc = np.load('/ScientificComputing/DrEmmettData_xc.npz')
    yc = np.load('/ScientificComputing/DrEmmettData_yc.npz')
    % Hint: You should only use one pair of vectors
    # Your own code.
    return v
```

2.23. Ambigüedad espacial

En aplicaciones para sistemas de transportes inteligentes, los vehículos cuentan con mediciones del sistema de posicionamiento global (GPS) que son integradas a un sistema de información geográfico (GIS). Estos datos se utilizan para analizar el trayecto de los vehículos de transporte y tomar decisiones operativas y de logística.

Desafortunadamente las mediciones de GPS no son exactas y contienen errores, por lo que el análisis de rutas de los vehículos de transporte no puede realizarse directamente sobre el conjunto de mediciones. En la Figura 1 se muestra la trayectoria de un vehículo de transporte donde la mayoría de sus mediciones están asociadas a alguna calle. Por ejemplo, el conjunto de cruces $\times$ representa la trayectoria del vehículo por la calle C_1 y se observa que las mediciones GPS no están exactamente sobre la calle C_1. Por otro lado, el punto $\mathbf{p}_1$ no está asociado a ninguna calle, por lo que se necesita determinar a qué calle corresponde, si a la calle C_1 o calle C_2. Este problema se conoce como ambigüedad espacial (o en inglés *map-matching problem*), el cual se genera por la inconsistencia en la asociación de las mediciones GPS con la calle por la cual el vehículo ha realizado un trayecto.

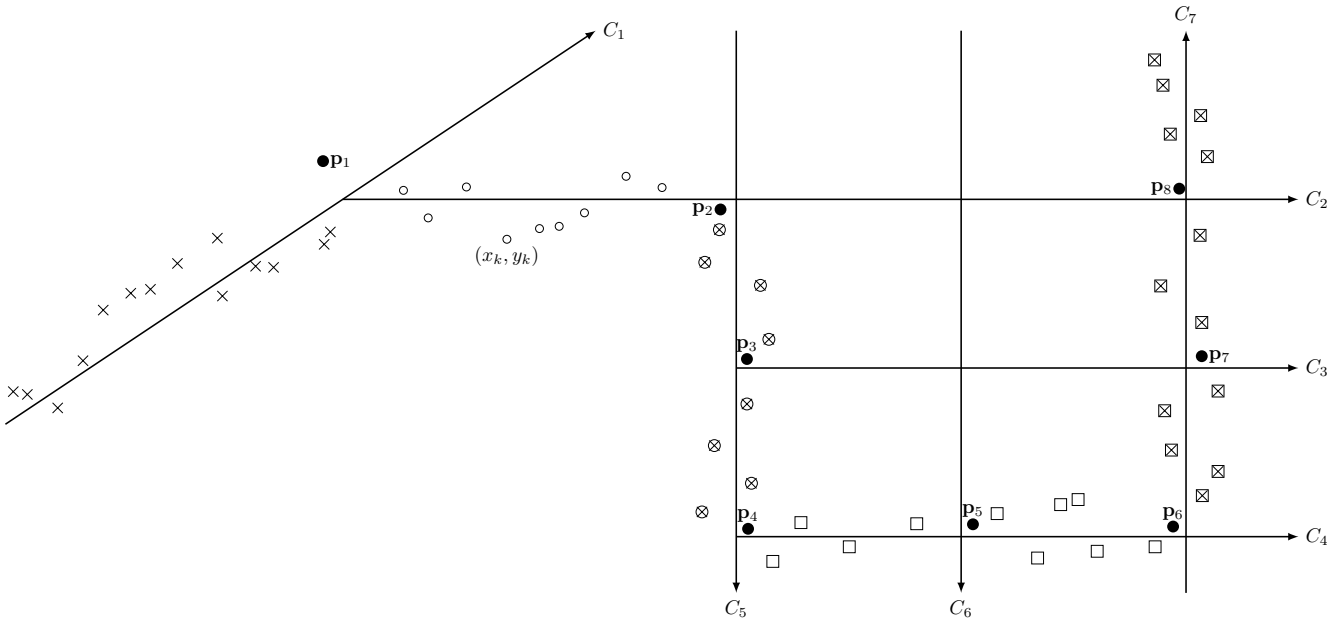

Figura 1: Problema de ambigüedad espacial (map-matching problem). Cada punto (x_k, y_k) está asociado a una única calle C_j, por ejemplo las mediciones marcadas con $(\times)$ están asociadas a C_1, las mediciones marcadas con $(\circ)$ están asociadas a C_2 y así sucesivamente. La dirección de cada calle C_j está indicado por la punta de flecha de cada segmento. Cada punto $\mathbf{p}_i$ no está asociado a alguna calle C_j.

En la Figura 1 se puede observar a simple vista que el vehículo realizó una trayectoria que contempla las calles C_1, C_2, C_5, C_4, y C_7. Sin embargo, quedaron algunas mediciones sin la asociación a una calle, esto corresponde a los puntos $\mathbf{p}_i$. Una

alternativa sería asociar el punto $\mathbf{p}_i$ a la misma calle que tiene su punto más cercano, desafortunadamente esto puede producir algunas inconsistencias, por ejemplo el punto $\mathbf{p}_2$, bajo este algoritmo, se asociaría a la calle C_5, sin embargo debería ser asociado a la calle C_2 porque está más cercano a ella. Para corregir este problema se propone implementar la siguiente versión intermedia del algoritmo *map-matching*, la cual construye una aproximación numérica de la calle a partir de un conjunto de mediciones de GPS y luego determina la mínima distancia entre el punto y la representación numérica obtenida. Notar que se omitirá la dependencia de los sub-índices j e i en lo siguiente para simplificar la notación.

Considere que tiene las mediciones asociadas a la calle C están agrupadas en un conjunto de datos $G = \{(x_1, y_1, t_1), \ldots, (x_k, y_k, t_k), \ldots, (x_n, y_n, t_n)\}$, donde el par (x_k, y_k) representa la coordenada en el mapa y t_k representa en qué tiempo fue obtenida la medición. Por simplicidad, se considera que las calles se representarán como una **recta**, ya sea vertical, horizontal o en otra dirección.

Para poder representar todas las alternativas posibles en cómo puede estar una calle, se considerará una representación paramétrica de la misma, es decir la función paramétrica $\mathbf{r}(t) = \langle f_1(t), f_2(t)\rangle$ representa la calle C, donde $f_1(t) = a_1 + b_1\, t$ y $f_2(t) = a_2 + b_2\, t$. Esto significa que la primera parte del problema es obtener los parámetros a_1, b_1, a_2 y b_2 a partir de las mediciones obtenidas por GPS, es decir, debe ajustar convenientemente los datos con las aproximaciones propuestas. *Hint: This looks like a least-square problem, actually, two least-square problem!* La segunda parte del problema corresponde a determinar la distancia mínima de un punto $\mathbf{p}$ a la recta paramétrica $\mathbf{r}(t)$ obtenida.

Para obtener la distancia mínima entre el punto $\mathbf{p}$ y la función $\mathbf{r}(t)$ se puede resolver mediante la minimización de la siguiente función,

$$\begin{aligned} g(t) &= \|\mathbf{p} - \mathbf{r}(t)\|_2^2 = \|\langle p_x, p_y\rangle - \langle f_1(t), f_2(t)\rangle\|_2^2 = \|\langle p_x, p_y\rangle - \langle a_1 + b_1\, t, a_2 + b_2\, t\rangle\|_2^2. \\ &= (p_x - a_1 - b_1\, t)^2 + (p_y - a_2 - b_2\, t)^2, \end{aligned}$$

donde $g(t)$ corresponde a la norma Euclidiana al cuadrado y es la función que se debe minimizar, es decir se debe encontrar el valor de $\tilde{t}$ tal que $g\left(\tilde{t}\right)$ sea mínimo. Luego se puede obtener la distancia Euclidiana mínima entre el punto $\mathbf{p}$ y el punto $\mathbf{r}\left(\tilde{t}\right)$ solo calculando la norma 2 entre ambos puntos. En resumen, usted debe primero construir la aproximación paramétrica $\mathbf{r}(t)$ y luego obtener la distancia mínima de $\mathbf{p}$ a $\mathbf{r}(\tilde{t})$.

(a) Construya un algoritmo que permita, dado un punto $\mathbf{p}$ y un conjunto de mediciones G, representado por los vectores $\mathbf{x}$, $\mathbf{y}$, y $\mathbf{t}$, obtener la distancia mínima desde el punto $\mathbf{p}$ a la aproximación *paramétrica* $\mathbf{r}(t)$ de la calle C construida con las mediciones de GPS correspondientes. Considere que $\texttt{xk} = [x_1, x_2, \ldots, x_n]^T$, $\texttt{yk} = [y_1, y_2, \ldots, y_n]^T$, y $\texttt{tk} = [t_1, t_2, \ldots, t_n]^T$.

(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.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 $\mathbf{x}$.
- `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`.
- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.
- `np.sqrt(x)`: Entrega la raíz cuadrada de `x` para $\texttt{x} \geq 0$.

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

```
'''
input:
p  : (ndarray) 2-dimensional point p.
xk : (ndarray) array of dimension "n" that stores the "x_k" measurements.
yk : (ndarray) array of dimension "n" that stores the "y_k" measurements.
tk : (ndarray) array of dimension "n" that stores the "t_k" measurements.
n  : (integer) Number of measurements.

output:
d  : (float) minimal distance from point p to the parametric representation of street C.
'''
def map_matching(p,xk,yk,tk,n):
    # Your own code.
    return d
```

2.24. Transformada discreta de coseno

El Teorema de interpolación de la transformada discreta de coseno indica lo siguiente:

Thm 1. *Sea* $\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}]^T$ *un vector con* n *coeficientes reales. Se define* $\mathbf{y} = [y_0, y_1, \ldots, y_{n-1}]^T = C\,\mathbf{x}$*, donde* C *corresponde a la matriz asociada a la Transformada Discreta de Coseno (DCT del inglés Discrete Cosine Transform) de orden* n*. Entonces la función:*

$$P_n(t) = \frac{1}{\sqrt{n}}\, y_0 + \sqrt{\frac{2}{n}} \sum_{k=1}^{n-1} y_k \, \cos\left(\frac{k\,(2\,t+1)\,\pi}{2\,n}\right),$$

satisface que $P_n(j) = x_j$ *para todo* $j \in \{0, 1, \ldots\, n-1\}$ *y los coeficientes de la matriz* $C \in \mathbb{R}^{n\times n}$ *se define de la siguiente forma:*

$$C_{i,j} = \sqrt{\frac{2}{n}}\, a_i \, \cos\left(\frac{i\,(2\,j+1)\,\pi}{2\,n}\right),$$

para $i \in \{0, 1, \ldots\, n-1\}$ *y* $j \in \{0, 1, \ldots\, n-1\}$*, donde*

$$a_i = \begin{cases} \dfrac{1}{\sqrt{2}}, & \text{si } i = 0, \\ 1, & \text{si } i \in \{1, 2, \ldots, n-1\}. \end{cases}$$

Por simplicidad se incluye de forma explícita la matriz C,

$$C = \sqrt{\frac{2}{n}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\left(\frac{\pi}{2\,n}\right) & \cos\left(\frac{3\,\pi}{2\,n}\right) & \cdots & \cos\left(\frac{(2\,n-1)\,\pi}{2\,n}\right) \\ \cos\left(\frac{2\,\pi}{2\,n}\right) & \cos\left(\frac{6\,\pi}{2\,n}\right) & \cdots & \cos\left(\frac{2\,(2\,n-1)\,\pi}{2\,n}\right) \\ \vdots & \vdots & \vdots & \vdots \\ \cos\left(\frac{(n-1)\,\pi}{2\,n}\right) & \cos\left(\frac{(n-1)\,3\,\pi}{2\,n}\right) & \cdots & \cos\left(\frac{(n-1)\,(2\,n-1)\,\pi}{2\,n}\right) \end{bmatrix}$$

En resumen, podemos concluir que necesitamos evaluar recurrentemente la función $f(x) = \cos(\pi\,x)$ al utilizar la DCT. Más aún, es posible notar que para construir la transformada discreta de coseno para $n = 100$ necesitamos evaluar el coeficiente $C_{n-1,n-1} = \cos\left(\frac{(n-1)\,(2\,n-1)\,\pi}{2\,n}\right)$, lo que es $\cos\left(\frac{19701}{200}\,\pi\right) \approx \cos(309.462584341862582954) \approx -0.015707317311820$. Entonces, si consideramos que trabajaremos con vectores de dimensión menor o igual a 100, significa que necesitamos evaluar la función $\cos(\cdot)$ para valores en el intervalo $[0, 100\,\pi]$.

Adicionalmente sabemos que,

$$\left|\frac{\mathrm{d}^m f(x)}{\mathrm{d}x^m}\right| = \begin{cases} \pi^m\,|\sin(\pi\,x)|, & \text{si } m \text{ es impar}, \\ \pi^m\,|\cos(\pi\,x)|, & \text{si } m \text{ es par}. \end{cases}$$

Preguntas:

(a) Proponga un algoritmo basado en interpolación polinomial que permita evaluar la función $f(x)$ para $x \in [0, 100]$ que asegure lo siguientes requerimientos:

Req. 1 Un error de interpolación menor o igual a $\varepsilon > 0$. Usted debe explicar que se debe hacer para asegurar esto y cómo planea determinar lo que necesite para cumplir con lo solicitado.

Req. 2 Que el interpolador construido tenga una cota superior de error lo más pequeña posible.

Su respuesta debe:

- Explicar claramente como **construirá** su interpolador polinomial.
- Explicar claramente como se debe **utilizar** su interpolador polinomial.

Recuerde que cada algoritmo tiene distintos requerimientos, por lo que verifique que usted efectivamente describe explícitamente cada componente del algoritmo elegido para que funcione completamente. En caso de no cumplir con alguno de los 2 requerimientos completamente, se entregarán puntos parciales.

(b) Explique cómo puede utilizar la interpolación polinomial anterior para evaluar $\sin(\pi\,x)$ sin construir un nuevo interpolador.

- Hint 1: It is important to recall that the cosine function is periodic, so, what can you do if you want to evaluate it for values greater that $2\,\pi$? The same applies for negative values, but they are not needed for this problem.
- Hint 2: You may find useful to consider the modulus operator, which returns the reminder of the operation. For instance in NumPy we can performe the following computation `np.mod(6.2,2.5)` which returns `1.2`.
- Hint 3: The lowest upper bound should contain the coefficient π^n but not $\pi^{2\,n}$ nor other additional coefficients that makes it larger. In this case n corresponds to the number of points used in the interpolation.

2.25. Interpolación de Lagrange Matricial

2.25.1. Contexto

Considere que usted tiene acceso a una secuencia de imágenes en escala de grises denotadas por $X_i \in \mathbb{R}^{n\times n}$, para $i \in \{1, 2, \ldots, m\}$. Las imágenes representan la "data" original obtenida al medir el flujo sanguíneo del anverso de una mano (parte opuesta a la palma). Particularmente se conoce que cada imagen X_i ha sido adquirida en un tiempo t_i, donde $t_i \in [0, T]$. Para hacer un uso médico de las imágenes, se necesita que sean pre-procesadas. Sin embargo, el costo del pre-procesamiento para todas las imágenes no fue considerado en el presupuesto, por lo cual hay que buscar otra alternativa.

Para proponer una solución a esta problemática, se gestiona **solo** el pre-procesamiento de las imágenes X_1 y X_2, lo cual genera las imágenes Y_1 e Y_2, respectivamente. El acceso a estas dos imágenes pre-procesadas, Y_1 e Y_2, abre la posibilidad de pre-procesar, a bajo costo, las imágenes faltantes, es decir desde X_3 hasta X_m.

Una estrategia consiste en realizar una interpolación lineal matricial utilizando las imágenes X_1 y X_2, es decir, construir un polinomio matricial de la forma $Y = P(X) = A\,X + B$, donde A y B pertenecen a $\mathbb{R}^{n\times n}$.

Recordando la versión unidimensional del método de Interpolación de Lagrange para dos puntos,

$$p(x) = y_1\,\frac{(x - x_2)}{(x_1 - x_2)} + y_2\,\frac{(x - x_1)}{(x_2 - x_1)}, \tag{8}$$

nos da el *insight* para extenderlo de forma matricial. La extensión matricial debe asegurar que $P(X_1) = Y_1$ y $P(X_2) = Y_2$, para lo cual se propone la siguiente expresión,

$$P(X) = Y_1\,(X_1 - X_2)^{-1}\,(X - X_2) + Y_2\,(X_2 - X_1)^{-1}\,(X - X_1).$$

La cual al evaluarla en $X = X_1$ y $X = X_2$ nos entrega Y_1 e Y_2 respectivamente. Adicionalmente, se pueden simplificar las expresiones para encontrar las matrices A y B, de la siguiente forma,

$$\begin{aligned} P(X) &= Y_1\,(X_1 - X_2)^{-1}\,X - Y_1\,(X_1 - X_2)^{-1}\,X_2 + Y_2\,(X_2 - X_1)^{-1}\,X - Y_2\,(X_2 - X_1)^{-1}\,X_1 \\ &= \left(Y_1\,(X_1 - X_2)^{-1} - Y_2\,(X_1 - X_2)^{-1}\right)\,X - Y_1\,(X_1 - X_2)^{-1}\,X_2 - Y_2\,(X_2 - X_1)^{-1}\,X_1 \\ &= \underbrace{\left((Y_1 - Y_2)\,(X_1 - X_2)^{-1}\right)}_{A}\,X + \underbrace{Y_2\,(X_1 - X_2)^{-1}\,X_1 - Y_1\,(X_1 - X_2)^{-1}\,X_2}_{B} \\ &= A\,X + B \end{aligned} \tag{9}$$

2.25.2. Pregunta 1

(a) Considere, solo para este ítem, que tiene a su disposición las factorizaciones SVD de las matrices X_1 y X_2, es decir, $X_1 = U\,\Sigma_1\,V^T$ y $X_2 = U\,\Sigma_2\,V^T$. Notar que la matriz de los vectores singulares izquierdos U es la misma para ambas imágenes. De forma análoga, la matriz de los vectores singulares derechos V, también es la misma para ambas imágenes. Determine cuáles son las condiciones que se deben cumplir para que exista $(X_1 - X_2)^{-1}$. Su respuesta debe basarse en las SVD's entregadas.

(b) Proponga un algoritmo que permita obtener la matriz A considerando que la matriz $(X_1 - X_2)$ es no singular. Recuerde que no está permitido (¡ni es recomendado!) la posibilidad de obtener la inversa de una matriz de forma explícita. *Hint: recall that* $(C_1\,C_2)^T = C_2^T\,C_1^T$

(c) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo para obtener las matrices A y B. 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.sqrt(x)`: Entrega la evaluación de la raíz cuadrada no negativa de un vector o escalar `x`.
- `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 $PA = LU$.

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:
X_1  : (ndarray) Image X_1.
X_2  : (ndarray) Image X_2.
Y_1  : (ndarray) Image Y_1.
Y_2  : (ndarray) Image Y_2.
n    : (int)     Each image has size n x n.

output:
A   : (ndarray) Matrix A.
B   : (ndarray) Matrix B.
'''
def compute_AB(X_1,X_2,Y_1,Y_2,n):
    # Your own code.
    return A,B
```

2.25.3. Pregunta 2

Suponga que ahora se ha realizado un nuevo experimento, en el cual el equipamiento para obtener las imágenes Y se ha descalibrado y por lo tanto, ahora se obtienen imágenes perturbadas, es decir, $\widehat{Y}$. Particularmente se han obtenido tres imágenes: $(X_1, \widehat{Y}_1), (X_2, \widehat{Y}_2)$ y $(X_3, \widehat{Y}_3)$. Una primera idea evaluada fue la realización de una interpolación matricial del tipo de Lagrange con 3 matrices, la cual generó la siguiente expresión:

$$\begin{aligned} P(X) = &\widehat{Y}_1 (X_1 - X_2)^{-1} (X - X_2) (X_1 - X_3)^{-1} (X - X_3) \\ &+ \widehat{Y}_2 (X_2 - X_1)^{-1} (X - X_1) (X_2 - X_3)^{-1} (X - X_3) \\ &+ \widehat{Y}_3 (X_3 - X_1)^{-1} (X - X_1) (X_3 - X_2)^{-1} (X - X_2). \end{aligned}$$

Sin embargo, esta alternativa fue descartada dado que las imágenes pre-procesadas ahora están perturbadas. Por lo tanto se ha sugerido la utilización de una aproximación lineal en donde las matrices incógnitas ahora se deberán obtener al minimizar el error cuadrático. Entonces, las aproximaciones obtenidas son las siguientes:

$$\begin{aligned} A X_1 + B &\approx \widehat{Y}_1, \\ A X_2 + B &\approx \widehat{Y}_2, \\ A X_3 + B &\approx \widehat{Y}_3. \end{aligned}$$

Las cuales se puede reescribir *convenientemente* de la siguiente forma:

$$\underbrace{\begin{bmatrix} I_n & X_1 \\ I_n & X_2 \\ I_n & X_3 \end{bmatrix}}_{W} \begin{bmatrix} B \\ A \end{bmatrix} \approx \begin{bmatrix} \widehat{Y}_1 \\ \widehat{Y}_2 \\ \widehat{Y}_3 \end{bmatrix} \tag{10}$$

donde $I_n \in \mathbb{R}^{n \times n}$ es la matriz identidad y $W \in \mathbb{R}^{3\,n \times 2\,n}$. Entonces, estamos interesados en obtener la factorización QR reducida de W, es decir la matrices $\widehat{Q} \in \mathbb{R}^{3\,n \times 2\,n}$ y $\widehat{R} \in \mathbb{R}^{2\,n \times 2\,n}$.

(a) Obtenga las primeras n columnas de las matrices $\widehat{Q}$ y $\widehat{R}$ de la factorización QR reducida de W. *Hint: It may be really useful to look for a pattern.*

(b) Proponga un algoritmo para obtener las últimas n columnas de las matrices $\widehat{Q}$ y $\widehat{R}$ de la factorización QR reducida de W. *Hint: A "modified" algorithm may be useful.*

(c) Implemente en Python utilizando adecuadamente la librería NumPy (en especial su capacidad de vectorización) el algoritmo para obtener las matrices $\widehat{Q}$ y $\widehat{R}$ de la factorización QR reducida de W. 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.sqrt(x)`: Entrega la evaluación de la raíz cuadrada no negativa de un vector o escalar `x`.
- `np.eye(n)`: Entrega un `ndarray` de dimensión `n`×`n` donde cada coeficiente de la diagonal es igual a `1` y el resto de los coeficientes son iguales a `0`, es decir la matriz identidad I_n de dimensión $n \times n$.
- `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]])
```

- `np.linalg.norm(x)`: Entrega la norma Euclidiana del vector `x`.

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 (ver siguiente página).

Considere la siguiente firma:

```
'''
input:
X_1  : (ndarray) Image X_1.
X_2  : (ndarray) Image X_2.
X_3  : (ndarray) Image X_3.
n    : (int) Size.

output:
Qh   : (ndarray) Matrix \widehat{Q}.
Rh   : (ndarray) Matrix \widehat{R}.
'''
def QR_Image(X_1,X_2,X_3,n):
    # Your own code.
    return Qh,Rh
```

3. Desarrollos de referencia

3.1. Desarrollo Pregunta "Back to the Future"

Este desarrollo corresponde a la pregunta en Apartado 2.22.

(a)
- Primero se debe construir el lado derecho de la ecuación, es decir, el vector $\mathbf{w}$. Para asegurar una mejor aproximación, es decir, con la menor cota superior del error en el intervalo, se utilizará el conjunto S_2 de los puntos de Chebyshev: $\mathbf{x}^c$ e $\mathbf{y}^c$. Entonces,
  - (I) se construye un polinomio interpolador $p_{m-1}(x)$ utilizando Interpolación Baricéntrica para reducir la cantidad de operaciones elementales en su evaluación. La interpolación polinomial se realizará con $\mathbf{x}^c$ e $\mathbf{y}^c$, es decir con los puntos de Chebyshev disponibles. Esta elección se hace para reducir el fenómeno de Runge. La interpolación polinomial asegura que $p_{m-1}(x_k^c) = y_k^c$, para así posteriormente aproximar $f(x_j)$ con $p_{m-1}(x_j)$, es decir,
  - (II) se evalúan los n puntos $x_j = \frac{2}{n-1}(j-1) - 1$ en $p_{m-1}(x_j)$ para $j = \{1, \ldots, n\}$. Notar que el vector $\mathbf{w}$ se define como $[f(x_1), f(x_2), \ldots, f(x_j), \ldots, f(x_n)]^T$ sin embargo se aproximará con $[p_{m-1}(x_1), p_{m-1}(x_2), \ldots, p_{m-1}(x_j), \ldots, p_{m-1}(x_n)]^T$

- Segundo, se debe resolver la ecuación, $T_n\,\mathbf{v} = \mathbf{w}$, para encontrar el vector condensador. Tomamos ventaja de la forma de la matriz T_n, ya que está factorizada en dos matrices triangulares, es decir, conocemos la factorización LU de T_n. Entonces realmente tenemos $L_n\,U_n\,\mathbf{v} = \mathbf{w}$, (i) definimos el vector auxiliar $\mathbf{z} = U_n\,\mathbf{v}$, lo cual genera $L_n\,U_n\,\mathbf{v} = L_n\,\mathbf{z} = \mathbf{w}$. Es decir debemos resolver el sistema de ecuaciones lineales $L_n\,\mathbf{z} = \mathbf{w}$ y dado que tiene una estructura bidiagonal-"triangular inferior", podemos obtener $\mathbf{z}$ utilizando una versión reducida de *forward substitution* para tomar ventaja de la estructura presentada y (ii) luego de obtener $\mathbf{z}$, podemos obtener $\mathbf{v}$ resolviendo el sistema de ecuaciones lineales bidiagonal-"triangular superior" $U_n\,\mathbf{v} = \mathbf{z}$ utilizando una versión reducida de *backward substitution* para tomar ventaja de la estructura presentada.

  Para resolver $L_n\,\mathbf{z} = \mathbf{w}$, mediante una versión reducida de *forward substitution*, debemos realizar los siguientes pasos para tomar ventaja de la estructura de L_n y obtener $\mathbf{z}$:

  - *1ra ecuación*: $z_1 = w_1$, la cual entrega el valor de z_1 directamente, es decir $z_1 = w_1$.
  - *2da ecuación*: $-\dfrac{1}{2}\,z_1 + z_2 = w_2$, despejando z_2 obtenemos $z_2 = w_2 + \dfrac{1}{2}\,z_1$. Recordar que en este paso ya conocemos el valor de z_1.
  - *i-ésima ecuación*: $-\dfrac{(i-1)}{i}\,z_{i-1} + z_i = w_i$, despejando z_i se obtiene $z_i = w_i + \dfrac{(i-1)}{i}\,z_{i-1}$ para $i = \{3, \ldots, n-1\}$.
  - *n-ésima ecuación*: $-\dfrac{(n-1)}{n}\,z_{n-1} + z_n = w_n$, finalmente despejando z_n se obtiene $z_n = w_n + \dfrac{(n-1)}{n}\,z_{n-1}$.

- Por otro lado, para resolver $U_n\,\mathbf{v} = \mathbf{z}$, mediante una versión reducida de *backward substitution*, debemos realizar los siguientes pasos para tomar ventaja de la estructura de U_n y obtener $\mathbf{v}$:

  - *n-ésima ecuación*: $\dfrac{n+1}{n}\,v_n = z_n$, por lo tanto despejando v_n obtenemos $v_n = \dfrac{n}{n+1}\,z_n$.
  - *$(n-1)$-ésima ecuación*: $\dfrac{n}{n-1}\,v_{n-1} - v_n = z_n$, despejando v_{n-1} obtenemos $v_{n-1} = \dfrac{n-1}{n}\,(z_n + v_n)$.
  - *i-ésima ecuación*: $\dfrac{i+1}{i}\,v_i - v_{i+1} = z_i$, despejando v_i obtenemos $v_i = \dfrac{i}{i+1}\,(z_i + v_{i+1})$ para $i = \{2, \ldots, n-2\}$.
  - *1ra ecuación*: $2\,v_1 - v_2 = z_1$, despejando v_1 obtenemos $v_1 = \dfrac{1}{2}\,(z_1 + v_2)$.

  Por lo tanto se puede obtener $\mathbf{v}$ primero interpolando la *data* disponibles en los puntos de Chebyshev con interpolación Baricéntrica y luego resolviendo la secuencia de sistemas de ecuaciones lineales generados por L_n y U_n respectivamente, tomando ventaja de las definiciones de L_n y U_n entregadas.

(b)
```
'''
input:
n : (integer) dimension of the matrix T_n.
output:
v : (ndarray) array of dimension n that stores the estimation of capacitor vector "v".
'''
def capacitor_vector(n):
    % Loading m-dimensional vector data $\mathbf{x}^e$ and $\mathbf{y}^e$
    xe = np.load('/ScientificComputing/DrEmmettData_xe.npz')
    ye = np.load('/ScientificComputing/DrEmmettData_ye.npz')
    % Loading m-dimensional vector data $\mathbf{x}^c$ and $\mathbf{y}^c$
    xc = np.load('/ScientificComputing/DrEmmettData_xc.npz')
    yc = np.load('/ScientificComputing/DrEmmettData_yc.npz')
    % Hint: You should only use one pair of vectors
```

- Obtener el vector $\mathbf{w}$ mediante interpolación Baricéntrica

```
j = np.arange(1,n + 1)
xj = (2*(j - 1))/(n - 1) - 1.
pB = BarycentricInterpolation(xc,yc)
w = pB(xj)
```

- Resolver el sistema de ecuaciones $L_n\mathbf{z} = \mathbf{w}$

```
z = np.ones((n,1)) # or better "z = np.ones(n)"
z[0] = w[0]
for i in np.arange(2,n + 1):
  z[i - 1] = w[i - 1] + (i - 1)*z[i - 2]/(i)
```

- Resolver el sistema de ecuaciones $U_n\mathbf{v} = \mathbf{z}$

```
v = np.ones((n,1)) # or better "v = np.ones(n)"
v[n-1] = (n*z[n - 1])/(n + 1)
for i in np.arange(n - 1,0,-1):
  v[i-1] = (i*(z[i-1] + v[i]))/(i + 1)
return v
```

3.2. Desarrollo Pregunta "Ambigüedad espacial"

Este desarrollo corresponde a la pregunta en Apartado 2.23.

(a)

- La primera tarea es construir la aproximación paramétrica:

$$\mathbf{r}(t) = \langle f_1(t), f_2(t) \rangle = \langle a_1 + b_1\, t, a_2 + b_2\, t \rangle$$

Esto significa que debemos obtener los valores de a_1, b_1, a_2 y b_2. Para esto, debemos resolver **dos** problemas de mínimos cuadrados: aproximar $f_1(t)$ que tiene relación con las coordenadas x_k de las mediciones y aproximar $f_2(t)$ que tiene relación con las coordenadas y_k de las mediciones. Entonces debemos resolver los siguientes sistemas de ecuaciones $A\,\mathbf{c}_1 = \mathbf{x}_k$ y $A\,\mathbf{c}_2 = \mathbf{y}_k$, donde los lados derechos $\mathbf{x}_k$ y $\mathbf{y}_k$ corresponden a las coordenadas x_k e y_k y los vectores $\mathbf{c}_1$ y $\mathbf{c}_2$ corresponden a los coeficientes $[a_1, b_1]^T$ y $[a_2, b_2]^T$, respectivamente:

$$\begin{array}{rcl} a_1 + b_1\, t_1 & = & x_1 \\ a_1 + b_1\, t_2 & = & x_2 \\ a_1 + b_1\, t_3 & = & x_3 \\ \vdots & \vdots & \vdots \\ a_1 + b_1\, t_{n-2} & = & x_{n-2} \\ a_1 + b_1\, t_{n-1} & = & x_{n-1} \\ a_1 + b_1\, t_n & = & x_n \end{array} \Rightarrow \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ 1 & t_3 \\ \vdots & \vdots \\ 1 & t_{n-2} \\ 1 & t_{n-1} \\ 1 & t_n \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} \Rightarrow A\,\mathbf{c}_1 = \mathbf{x}_k$$

$$\begin{array}{rcl} a_2 + b_2\, t_1 & = & y_1 \\ a_2 + b_2\, t_2 & = & y_2 \\ a_2 + b_2\, t_3 & = & y_3 \\ \vdots & \vdots & \vdots \\ a_2 + b_2\, t_{n-2} & = & y_{n-2} \\ a_2 + b_2\, t_{n-1} & = & y_{n-1} \\ a_2 + b_2\, t_n & = & y_n \end{array} \Rightarrow \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ 1 & t_3 \\ \vdots & \vdots \\ 1 & t_{n-2} \\ 1 & t_{n-1} \\ 1 & t_n \end{bmatrix} \begin{bmatrix} a_2 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \\ y_n \end{bmatrix} \Rightarrow A\,\mathbf{c}_2 = \mathbf{y}_k$$

Notar que la matriz A asociada a ambos sistemas de ecuaciones lineales sobre-determinados es la misma, dado que se tiene los datos en el mismo tiempo.

- Luego de construido el sistema de ecuaciones lineales sobre-determinado, se puede encontrar los coeficientes que minimizan el error cuadrático de la matriz A, es decir $A = \widehat{Q}\,\widehat{R}$, de la siguiente forma:

$$\overline{\mathbf{c}}_1 = \begin{bmatrix} \overline{a}_1 \\ \overline{b}_1 \end{bmatrix} = \widehat{R}^{-1}\,\widehat{Q}^* \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{y} \quad \overline{\mathbf{c}}_2 = \begin{bmatrix} \overline{a}_2 \\ \overline{b}_2 \end{bmatrix} = \widehat{R}^{-1}\,\widehat{Q}^* \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Notar que $\widehat{R}^{-1}$-por-un-vector nos indica que debemos resolver el sistema de ecuaciones lineales asociado, en particular acá se puede utilizar *backward substitution*. También es posible haber usado las ecuaciones normales.

- Una vez construida las parametrizaciones solicitadas podemos minimizar la función $g(t)$, la cual la definimos ahora con los parámetros obtenidos, es decir,

$$g(t) = \left(p_x - \overline{a}_1 - \overline{b}_1\, t\right)^2 + \left(p_y - \overline{a}_2 - \overline{b}_2\, t\right)^2.$$

Para encontrar el mínimo, podemos obtener la derivada,

$$\begin{aligned} g'(t) &= 2\,\left(p_x - \overline{a}_1 - \overline{b}_1\, t\right)\,(-\overline{b}_1) + 2\,\left(p_y - \overline{a}_2 - \overline{b}_2\, t\right)\,(-\overline{b}_2), \\ &= -2\,\left(p_x\,\overline{b}_1 - \overline{a}_1\,\overline{b}_1 - \overline{b}_1^2\, t\right) - 2\,\left(p_y\,\overline{b}_2 - \overline{a}_2\,\overline{b}_2 - \overline{b}_2^2\, t\right) \\ &= -2\,\left(p_x\,\overline{b}_1 - \overline{a}_1\,\overline{b}_1 - \overline{b}_1^2\, t + p_y\,\overline{b}_2 - \overline{a}_2\,\overline{b}_2 - \overline{b}_2^2\, t\right) \end{aligned}$$

simplificando e igualando a 0 obtenemos,

$$\begin{aligned}\left(p_x\,\overline{b}_1 - \overline{a}_1\,\overline{b}_1 - \overline{b}_1^2\,t + p_y\,\overline{b}_2 - \overline{a}_2\,\overline{b}_2 - \overline{b}_2^2\,t\right) &= 0,\\ \left(\overline{b}_1^2 + \overline{b}_2^2\right)\,\tilde{t} &= \left(p_x\,\overline{b}_1 - \overline{a}_1\,\overline{b}_1 + p_y\,\overline{b}_2 - \overline{a}_2\,\overline{b}_2\right)\\ \tilde{t} &= \frac{\left(p_x\,\overline{b}_1 - \overline{a}_1\,\overline{b}_1 + p_y\,\overline{b}_2 - \overline{a}_2\,\overline{b}_2\right)}{\overline{b}_1^2 + \overline{b}_2^2}.\end{aligned}$$

- Por lo tanto la distancia mínima es $\sqrt{g\left(\tilde{t}\right)}$.

(b)
```
'''
input:
p  : (ndarray) 2-dimensional point p.
xk : (ndarray) array of dimension "n" that stores the "x_k" measurements.
yk : (ndarray) array of dimension "n" that stores the "y_k" measurements.
tk : (ndarray) array of dimension "n" that stores the "t_k" measurements.
n  : (integer) Number of measurements.

output:
d  : (float) minimal distance from point p to the parametric representation of street C.
'''
def map_matching(p,xk,yk,tk,n):
```

- construir la matriz A para los problemas de mínimos cuadrados asociado y obtener su factorización QR reducida.

```
A = np.ones((n,2))
A[:,1] = tk
Q,R = np.linalg.qr(A,mode="reduced")
```

- resolver el problema de mínimos cuadrados $A\,\mathbf{c}_1 = \mathbf{x}_k$ y $A\,\mathbf{c}_2 = \mathbf{y}_k$

```
c1 = np.dot(np.transpose(Q),xk)
a1,b1 = np.linalg.solve(R,c1)
c2 = np.dot(np.transpose(Q),yk)
a2,b2  = np.linalg.solve(R,c2)
```

- calcular $\tilde{t}$ donde la distancia es mínima

```
t_tilde = (p[0]*b1 - a1*b1 + p[1]*b2 - a2*b2)/(b1**2 + b2**2)
```

- calcular la distancia mínima

```
xt_tilde = a1 + b1*t_tilde
yt_tilde = a2 + b2*t_tilde
d = np.linalg.norm(p - np.array([xt_tilde,yt_tilde]))
return d
```

3.3. Desarrollo Pregunta "¿Lineal o no-lineal?"

Este desarrollo corresponde a la pregunta en Apartado 2.5.

(a)
- Definir $\mathbf{F}\left(\begin{bmatrix}a\\ \omega\end{bmatrix}\right) = \begin{bmatrix}a\,\sin(x_1) + b\,\cos(\omega\,x_1) - y_1\\ a\,\sin(x_2) + b\,\cos(\omega\,x_2) - y_2\end{bmatrix}$.
- Obtener matriz Jacobiana J, es decir:

$$J\left(\begin{bmatrix}a\\ \omega\end{bmatrix}\right) = \begin{bmatrix}\sin(x_1) & -b\,x_1\,\sin(\omega\,x_1)\\ \sin(x_2) & -b\,x_2\,\sin(\omega\,x_2)\end{bmatrix}$$

- Iteración de Newton:

$$\begin{bmatrix}\sin(x_1) & -b\,x_1\,\sin(\omega_i\,x_1)\\ \sin(x_2) & -b\,x_2\,\sin(\omega_i\,x_2)\end{bmatrix}\begin{bmatrix}\Delta a_i\\ \Delta\omega_i\end{bmatrix} = -\begin{bmatrix}a_i\,\sin(x_1) + b\,\cos(\omega_i\,x_1) - y_1\\ a_i\,\sin(x_2) + b\,\cos(\omega_i\,x_2) - y_2\end{bmatrix},$$

$$\begin{bmatrix}a_{i+1}\\ \omega_{i+1}\end{bmatrix} = \begin{bmatrix}a_i\\ \omega_i\end{bmatrix} + \begin{bmatrix}\Delta a_i\\ \Delta\omega_i\end{bmatrix}$$

(b)

- Se propone el sistema de ecuaciones lineales sobre-determinado:

$$\begin{bmatrix} \sin(x_1) & \cos(\omega\, x_1) \\ \sin(x_2) & \cos(\omega\, x_2) \\ \sin(x_3) & \cos(\omega\, x_3) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

- El sistema de ecuaciones lineales cuadrado que minimiza el error cuadrático se puede obtener con las ecuaciones normales[3]:

$$\begin{bmatrix} \sin(x_1) & \sin(x_2) & \sin(x_3) \\ \cos(\omega\, x_1) & \cos(\omega\, x_2) & \cos(\omega\, x_3) \end{bmatrix} \begin{bmatrix} \sin(x_1) & \cos(\omega\, x_1) \\ \sin(x_2) & \cos(\omega\, x_2) \\ \sin(x_3) & \cos(\omega\, x_3) \end{bmatrix} \begin{bmatrix} \overline{a} \\ \overline{b} \end{bmatrix} = \begin{bmatrix} \sin(x_1) & \sin(x_2) & \sin(x_3) \\ \cos(\omega\, x_1) & \cos(\omega\, x_2) & \cos(\omega\, x_3) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$\begin{bmatrix} \displaystyle\sum_{i=1}^{3} \sin^2(x_i) & \displaystyle\sum_{i=1}^{3} \sin(x_i)\, \cos(\omega\, x_i) \\ \displaystyle\sum_{i=1}^{3} \sin(x_i)\, \cos(\omega\, x_i) & \displaystyle\sum_{i=1}^{3} \cos^2(\omega\, x_i) \end{bmatrix} = \begin{bmatrix} \displaystyle\sum_{i=1}^{3} \sin(x_i)\, y_i \\ \displaystyle\sum_{i=1}^{3} \cos(\omega\, x_i)\, y_i \end{bmatrix}$$

3.4. Desarrollo Pregunta "Transformada discreta de coseno"

Este desarrollo corresponde a la pregunta en Apartado 2.24.

(a)

- En esta pregunta se solicita interpolar la función $f(x) = \cos(\pi\, x)$ para $x \in [0, 100]$, lo cual es equivalente a interpolar la función $\cos(x)$ en el intervalo $[0, 100\,\pi]$, más aún sabemos que la función $\cos(x)$ es periódica con periodo $2\,\pi$, por lo que sería suficiente interpolar $\cos(x)$ en $[0, 2\,\pi]$ y utilizar esta propiedad para evaluarla en $[0, 100\,\pi]$. Esto se puede lograr utilizando la función módulo, en particular en NumPy se debería utilizar `x=np.mod(y,2*np.pi)` donde y correspondería a un valor en el intervalo $[0, 100\,\pi]$ y entrega $x \in [0, 2\,\pi]$, el cual se puede evaluar en el interpolador que construiremos.
- Para asegurar un error de interpolación de error menor o igual $\varepsilon > 0$ se utilizarán puntos de Chebyshev en el intervalo $[0, 2\,\pi]$. En este caso la cota del error para la función $\cos(x)$ con puntos de Chebyshev es la siguiente:

$$|\cos(x) - p(x)| \leq \frac{|(x - \widehat{x}_1)(x - \widehat{x}_2) \ldots (x - \widehat{x}_n)|}{n!} \left| \frac{\mathrm{d}^n\,(\cos(x))}{\mathrm{d}x^n} \right|,$$

donde

$$\widehat{x}_i = \pi\, \cos\left(\frac{(2\,i - 1)\, \pi}{2\,n} \right) + \pi, \quad i \in \{1, 2, \ldots, n\},$$

$$|(x - \widehat{x}_1)(x - \widehat{x}_2) \ldots (x - \widehat{x}_n)| \leq \frac{\left(\dfrac{2\,\pi - 0}{2} \right)^n}{2^{n-1}} = \frac{\pi^n}{2^{n-1}},$$

$$\left| \frac{\mathrm{d}^n\,(\cos(x))}{\mathrm{d}x^n} \right| \leq 1.$$

Notar que las derivadas de la función $\cos(x)$ oscilan entre $\{\pm\cos(x), \pm\sin x\}$, por lo que es directo obtener que el valor absoluto de todas ellas está acotado por 1. Por lo tanto la cota superior al interpolar con n puntos de Chebyshev en el intervalo $[0, 2\,\pi]$ es la siguiente:

$$|\cos(x) - p(x)| \leq \frac{\pi^n}{2^{n-1}\, n!}.$$

Para lograr asegurar que el error sea menor o igual a ε se debe encontrar n tal que cumpla la siguiente ecuación:

$$\frac{\pi^n}{2^{n-1}\, n!} \leq \varepsilon.$$

La cota superior obtenida es lo más pequeño posible considerando los puntos de Chebyshev y que $\cos(x)$ es periódica, haber utilizado el intervalo $[0, 100\,\pi]$ implicaría que la cota superior sea mucho mayor a la obtenida y solicitada.

[3] Esto también se puede obtener por medio de la minimización del error cuadrático calculando el gradiente respectivo e igualándolo a $\mathbf{0}$

- La construcción del polinomio se puede hacer directamente con la interpolación Baricéntrica de la siguiente forma:

$$p(x) = \frac{\sum_{i=1}^{n} \cos(\widehat{x}_i) \frac{w_i}{(x - \widehat{x}_i)}}{\sum_{i=1}^{n} \frac{w_i}{(x - \widehat{x}_i)}},$$
$$w_i = \frac{1}{l_i(\widehat{x}_i)},$$
$$l_i(\widehat{x}_i) = \prod_{k=1, i \neq k}^{n} (\widehat{x}_i - \widehat{x}_k)$$

- La evaluación de $f(x)$ se puede hacer directamente con $p(\pi\, x)$ si $x \in [0, 2]$, sin embargo, si se necesita evaluar para valores de x mayores a $2\,\pi$ se debe utilizar la función módulo antes descrita, es decir se debe utilizar la siguiente expresión `p(np.mod(np.pi*x,2*np.pi))`, la cual puede evaluarse para $x \in [0, 100]$, que fue justamente lo solicitado.

(b) En este caso podemos considerar la identidad $\cos\left(\pi\, x + \frac{\pi}{2}\right) = -\sin(\pi\, x)$, por lo que para evaluar $\sin(\pi\, x)$ se puede obtener como $-\cos\left(\pi\, x + \frac{\pi}{2}\right)$, es decir, `-p(np.mod(np.pi*x+np.pi/2,2*np.pi))`.

3.5. Desarrollo Pregunta "Interpolación de Lagrange Matricial"

Este desarrollo corresponde a la pregunta en Apartado 2.25.

3.5.1. Pregunta 1

(a) Se tiene que,

$$(X_1 - X_2)^{-1} = \left(U\, \Sigma_1\, V^T - U\, \Sigma_2\, V^T\right)^{-1} = \left(U\, (\Sigma_1 - \Sigma_2)\, V^T\right)^{-1} = (V^T)^{-1}\, (\Sigma_1 - \Sigma_2)^{-1}\, U^{-1}$$

Sabemos que las matrices U y V son unitarias, por lo tanto sus inversas existen y son $V^{-1} = V^T$ y $U^{-1} = U^T$. Entonces,

$$(X_1 - X_2)^{-1} = (V^T)^{-1}\, (\Sigma_1 - \Sigma_2)^{-1}\, U^{-1} = V\, (\Sigma_1 - \Sigma_2)^{-1}\, U^T$$

Por lo tanto, para que $(X_1 - X_2)^{-1}$ exista, debe existir la inversa de la matriz $(\Sigma_1 - \Sigma_2)$. Sea $(\sigma_i)_1$ y $(\sigma_i)_2$ los elementos de la diagonal de las matrices Σ_1 y Σ_2, respectivamente, entonces

$$(\Sigma_1 - \Sigma_2)^{-1} = \begin{bmatrix} \frac{1}{(\sigma_1)_1 - (\sigma_1)_2} & & & & \\ & \frac{1}{(\sigma_2)_1 - (\sigma_2)_2} & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \frac{1}{(\sigma_n)_1 - (\sigma_n)_2} \end{bmatrix},$$

por lo tanto, para que $(X_1 - X_2)^{-1}$ exista, se debe cumplir que $(\sigma_i)_1 \neq (\sigma_i)_2$ para todo $i \in \{1, \ldots, n\}$.

(b) Sea $Y_{12} = (Y_1 - Y_2)$ y $X_{12} = (X_1 - X_2)$, entonces

$$A = Y_{12}\, X_{12}^{-1} \Rightarrow A\, X_{12} = Y_{12}.$$

Si aplicamos la transpuesta a cada lado de la ecuación, se tiene que,

$$(A\, X_{12})^T = Y_{12}^T \Rightarrow X_{12}^T\, A^T = Y_{12}^T.$$

Matricialmente se obtiene,

$$X_{12}^T \left[\begin{array}{c|c|c|c|c} \mathbf{a}_1^T & \mathbf{a}_2^T & \cdots & \mathbf{a}_{n-1}^T & \mathbf{a}_n^T \end{array}\right] = \left[\begin{array}{c|c|c|c|c} \mathbf{y}_1^T & \mathbf{y}_2^T & \cdots & \mathbf{y}_{n-1}^T & \mathbf{y}_n^T \end{array}\right].$$

Donde $\mathbf{a}_i^T$ es la i-ésima columna de la matriz A^T e $\mathbf{y}_i^T$ es la i-ésima columna de la matriz Y_{12}^T. Luego, se tiene que resolver $X_{12}^T\, \mathbf{a}_i^T = \mathbf{y}_i^T$ para $i = \{1, \ldots, n\}$. El algoritmo propuesto viene dado por:

- Calcular factorización $PA = LU$ para la matrix X_{12}^T
- Para cada $i = \{1, \ldots, n\}$:
  - Calcular $\mathbf{b}_i = P\,\mathbf{y}_i^T$.
  - Resolver $L\,\mathbf{z}_i = \mathbf{b}_i$ para $\mathbf{z}_i$ con *Forward substitution.*
  - Resolver $U\,\mathbf{a}_i^T = \mathbf{z}_i$ para $\mathbf{a}_i^T$ con *Backward substitution.*

(c) Para calcular A se utiliza el algoritmo propuesto anteriormente.

```
'''
input:
X_1  : (ndarray) Image X_1.
X_2  : (ndarray) Image X_2.
Y_1  : (ndarray) Image Y_1.
Y_2  : (ndarray) Image Y_2.
n    : (int)     Each image has size n x n.

output:
A   : (ndarray) Matrix A.
B   : (ndarray) Matrix B.
'''
def compute_AB(X_1,X_2,Y_1,Y_2,n):
    # Your own code.
```

- Calcular PALU de la matriz X_{12}^T

```
P,L,U = lu(X12.T)
```

- Para cada columna de A^T:

```
A = np.zeros((n,n))
for i in np.arange(n):
```

  - Calcular $\mathbf{b}_i = P\,\mathbf{y}_i^T$.

```
b = np.dot(P,Y12[i,:])
```

  - Resolver $L\,\mathbf{z}_i = \mathbf{b}_i$ para $\mathbf{z}_i$ con *Forward substitution.*

```
z = np.zeros(n)
z[0] = b[0]
for j in np.arange(1,n):
    z[j] = b[j] - np.dot(L[j,:j],z[:j])
```

  - Resolver $U\,\mathbf{a}_i^T = \mathbf{z}_i$ para $\mathbf{a}_i^T$ con *Backward substitution.*

```
w = np.zeros(n)
w[-1] = z[-1]/U[-1,-1]
for j in np.arange(n-1,-1,-1):
    w[j] = (z[j] - np.dot(U[j,j+1:],w[j+1:]))/U[j,j]
A[i,:] = w
```

Para calcular $B = Y_2\,(X_1 - X_2)^{-1}\,X_1 - Y_1\,(X_1 - X_2)^{-1}\,X_2$ se propone lo siguiente:

1ª alternativa

- Sea $Z_1 = Y_2\,(X_1 - X_2)^{-1} = Y_2\,X_{12}^{-1}$. Luego se tiene que $Z_1\,X_{12} = Y_2 \Rightarrow X_{12}^T\,Z_1^T = Y_2^T$. Se resuelve con PALU $X_{12}^T\,\mathbf{z}_{i,[1]}^T = \mathbf{y}_{i,[2]}^T$ para $i = \{1, \ldots, n\}$ donde $\mathbf{z}_{i,[1]}^T$ e $\mathbf{y}_{i,[2]}^T$ corresponden a la i-ésima columna de la matriz Z_1^T e Y_2^T respectivamente. Luego se obtiene $Z_1 = (Z_1^T)^T$.
- Sea $Z_2 = Y_1\,(X_1 - X_2)^{-1} = Y_1\,X_{12}^{-1}$. Luego se tiene que $Z_2\,X_{12} = Y_1 \Rightarrow X_{12}^T\,Z_2^T = Y_1^T$. Se resuelve con PALU $X_{12}^T\,\mathbf{z}_{i,[2]}^T = \mathbf{y}_{i,[1]}^T$ para $i = \{1, \ldots, n\}$ donde $\mathbf{z}_{i,[2]}^T$ e $\mathbf{y}_{i,[1]}^T$ corresponden a la i-ésima columna de la matriz Z_2^T e Y_1^T respectivamente. Luego se obtiene $Z_1 = (Z_1^T)^T$.
- Finalmente se calcula $B = Y_2\,Z_1 - Y_1\,Z_2$.

- Ya contamos que la PALU de la matriz X_{12}^T. Entonces para cada columna de Z_1^T y Z_2^T:

```
Z1,Z2 = np.zeros((n,n)),np.zeros((n,n))
for i in np.arange(n):
```

- Calcular $\mathbf{b}_{i,[1]} = P\,\mathbf{y}_{i,[2]}^T$ y $\mathbf{b}_{i,[2]} = P\,\mathbf{y}_{i,[1]}^T$

```
b1,b2 = np.dot(P,Y2[i,:]),np.dot(P,Y1[i,:])
```

- Resolver $L\,\mathbf{w}_{i,[1]} = \mathbf{b}_{i,[1]}$ para $\mathbf{w}_{i,[1]}$ y $L\,\mathbf{w}_{i,[2]} = \mathbf{b}_{i,[2]}$ para $\mathbf{w}_{i,[2]}$ con *Forward substitution.*

```
w1,w2 = np.zeros(n),np.zeros(n)
w1[0],w2[0] = b1[0],b2[0]
for j in np.arange(1,n):
    w1[j] = b1[j] - np.dot(L[j,:j],w1[:j])
    w2[j] = b2[j] - np.dot(L[j,:j],w2[:j])
```

- Resolver $U\,\mathbf{z}_{i,[1]} = \mathbf{w}_{i,[1]}$ para $\mathbf{z}_{i,[1]}$ y $U\,\mathbf{z}_{i,[2]} = \mathbf{w}_{i,[2]}$ para $\mathbf{z}_{i,[2]}$ con *Backward substitution.*

```
z1,z2 = np.zeros(n),np.zeros(n)
z1[-1] = w1[-1]/U[-1,-1]
z2[-1] = w2[-1]/U[-1,-1]
for j in np.arange(n-1,-1,-1):
    z1[j] = (w1[j] - np.dot(U[j,j+1:],z1[j+1:]))/U[j,j]
    z2[j] = (w2[j] - np.dot(U[j,j+1:],z2[j+1:]))/U[j,j]
Z1[i,:],Z2[i,:] = z1,z2
```

- Calcular $B = Z_1\,X_1 - Z_2\,X_2$

```
B = np.dot(Z1,X1) - np.dot(Z2,X2)
return A,B
```

2^{a} alternativa

- Dado que se tiene A entonces se puede calcular B como:

$$P(X_1) = Y_1 = A\,X_1 + B \rightarrow B = Y_1 - A\,X_1$$

- Calcular $B = Y_1 - A\,X_1$

```
B = Y1 - np.dot(A,X1)
return A,B
```

3.5.2. Pregunta 2

(a) Lo primero que notamos es que las primeras n columnas de W son vectores que siguen un patrón muy regular, es decir, es el concatenamiento de 3 matrices identidad. Una característica importante de las matrices identidad es que sus columnas son los vectores canónicos, los cuales son ortonormales. En este caso, no son exactamente los vectores canónicos pero siguen siendo ortonormales, en particular, si definimos la matriz W de la siguiente forma,

$$W = \begin{bmatrix}\mathbf{w}_1 & \mathbf{w}_2 & \dots & \mathbf{w}_n & \mathbf{w}_{n+1} & \dots & \mathbf{w}_{2\,n}\end{bmatrix}.$$

Entonces podemos concluir que $\langle \mathbf{w}_i, \mathbf{w}_j \rangle = 0$ para $i, j \in \{1, 2, \dots, n\}$. Esto implica que si queremos escribir las primeras n columnas de W como una combinación lineal de vectores ortonormales $\mathbf{q}_j$, llegamos a la siguiente identidad,

$$\mathbf{w}_i = \sum_{j=1}^{n} r_{j,i}\,\mathbf{q}_i,$$

para $i \leq n$. Lo que induce las siguiente secuencia de ecuaciones,

$$\begin{aligned}
\mathbf{w}_1 &= r_{1,1}\,\mathbf{q}_1,\\
\mathbf{w}_2 &= r_{1,2}\,\mathbf{q}_1 + r_{2,2}\,\mathbf{q}_1,\\
&\vdots\ \ \vdots\\
\mathbf{w}_n &= \sum_{j=1}^{n} r_{j,n}\,\mathbf{q}_i.
\end{aligned}$$

Al utilizar Gram-Schmidt, obtenemos en el primer caso que $r_{1,1} = \sqrt{3}$ y $\mathbf{q}_1 = \mathbf{w}_1/\sqrt{3}$. Es decir, solo se re-escala $\mathbf{w}_1$ (que es lo tradicional), sin embargo, al considerar la ortogonalidad $\langle \mathbf{w}_i, \mathbf{w}_j \rangle = 0$, llegamos a una versión simplificada de las ecuaciones anteriores,

$$\begin{aligned}
\mathbf{w}_1 &= r_{1,1}\,\mathbf{q}_1,\\
\mathbf{w}_2 &= r_{2,2}\,\mathbf{q}_1,\\
&\vdots\ \ \vdots\\
\mathbf{w}_n &= r_{n,n}\,\mathbf{q}_n.
\end{aligned}$$

Por lo tanto, las primeras n columnas de $\widehat{Q} = W_1/\sqrt{3}$ y las primeras n columnas de $\widehat{R} = \sqrt{3}\, I_n$. Para mayor detalle, podemos escribir las ecuaciones correspondientes,

$$W = \widehat{Q}\,\widehat{R}$$
$$\begin{bmatrix} W_1 & W_2 \end{bmatrix} = \begin{bmatrix} \widehat{Q}_1 & \widehat{Q}_2 \end{bmatrix} \begin{bmatrix} \widehat{R}_0 & \widehat{R}_1 \\ \underline{\underline{0}} & \widehat{R}_2 \end{bmatrix},$$

donde W_1 representa las primeras n columnas de W, W_2 representa las últimas n columnas de W, $\widehat{Q}_1$ representa las primeras n columnas de $\widehat{Q}$, $\widehat{Q}_2$ representa las últimas n columnas de $\widehat{Q}$, y $\widehat{R}$ se descompone en 4 matrices: $\widehat{R}_0 \in \mathbb{R}^{n\times n}$, $\widehat{R}_1 \in \mathbb{R}^{n\times n}$, $\widehat{R}_2 \in \mathbb{R}^{n\times n}$. y $\underline{\underline{0}} \in \mathbb{R}^{n\times n}$ es la matriz nula. Al multiplicar las matrices obtenemos,

$$\begin{bmatrix} W_1 & W_2 \end{bmatrix} = \begin{bmatrix} \widehat{Q}_1\,\widehat{R}_0 & \widehat{Q}_1\,\widehat{R}_1 + \widehat{Q}_2\,\widehat{R}_2 \end{bmatrix},$$

Lo que implica que $\widehat{Q}_1 = W_1/\sqrt{3}$ y $\widehat{R}_0 = \sqrt{3}\, I_n$.

(b) Del desarrollo de la pregunta anterior, podemos notar que la obtención de las últimas n columnas de la factorización QR reducida implica el obtener las matrices $\widehat{R}_1$, $\widehat{R}_2$, y $\widehat{Q}_2$. Lo que implica la siguiente ecuación,

$$W_2 = \widehat{Q}_1\,\widehat{R}_1 + \widehat{Q}_2\,\widehat{R}_2.$$

Ahora, tomando ventaja del patrón de $\widehat{Q}_1$, considerando que *operan* sobre distintos elementos de los vectores en W_2, obtenemos $\widehat{R}_1$ de la siguiente forma,

$$\begin{aligned} \widehat{Q}_1^\top W_2 &= \widehat{Q}_1^\top \widehat{Q}_1\,\widehat{R}_1 + \widehat{Q}_1^\top \widehat{Q}_2\,\widehat{R}_2 \\ &= I_n\,\widehat{R}_1 + \underline{\underline{0}}\,\widehat{R}_2 \\ &= \widehat{R}_1. \end{aligned}$$

Esto no da la oportunidad de obtener, utilizando Gram-Schmidt modificado lo siguiente,

$$\underbrace{W_2 - \widehat{Q}_1\,\widehat{R}_1}_{W_3} = \widehat{Q}_2\,\widehat{R}_2.$$

Esto nos permite aplicar ahora, utilizar Gram-Scmidt modificado directamente sobre la matriz W_3 para obtener $\widehat{Q}_2$ y $\widehat{R}_2$. Por lo tanto, al obtener un procedimiento que obtiene $\widehat{R}_1$, $\widehat{R}_2$, y $\widehat{Q}_2$, se ha logrado obtener las últimas n columnas de $\widehat{Q}$ y $\widehat{R}$.

(c)
```
'''
input:
X_1  : (ndarray) Image X_1.
X_2  : (ndarray) Image X_2.
X_3  : (ndarray) Image X_3.
n    : (int) Size.

output:
Qh   : (ndarray) Matrix \widehat{Q}.
Rh   : (ndarray) Matrix \widehat{R}.
'''
def QR_Image(X_1,X_2,X_3,n):

    # Building initial matrices
    In = np.eye(n)
    W1 = np.concatenate((In,In,In),0)
    W2 = np.concatenate((X_1,X_2,X_3),0)

    # Computing first n-column
    Qh1 = W1/np.sqrt(3)
    Rh0 = In*np.sqrt(3)

    # Computing last n-column
    Rh1 = np.dot(Qh1.T,W2)
    W3  = W2 - np.dot(Qh1,Rh1)
```

```
# Initialization of partial output matrices
Qh2 = np.zeros((3*n,n))
Rh2 = np.zeros((n,n))

# Applying modified Gram-Schimdt to compute Qh2 and Rh2
for k in range(n):
    y = W3[:,k]
    for i in range(k):
        Rh2[i,k] = np.dot(Qh2[:,i],y)
        y=y-Rh2[i,k]*Qh2[:,i]
    Rh2[k,k] = np.linalg.norm(y)
    Qh2[:,k] = y/Rh2[k,k]

# Building final output
Qh = np.concatenate((Qh1,Qh2), axis=1)
Rh_tmp1 = np.concatenate((Rh0,0*In), axis=0)
Rh_tmp2 = np.concatenate((Rh1,Rh2), axis=0)
Rh = np.concatenate((Rh_tmp1,Rh_tmp2), axis=1)

return Qh,Rh
```