Ayudantía
SCT - Semana - 12

Repaso Certamen 2

1. Recordemos a los ilustres profesores Jørgen Pedersen Gram y Erhard Schmidt, los cuales desarrollaron el proceso de Gram-Schmidt para ortonormalizar vectores en un espacio vectorial equipado con un producto interno. Este proceso genera 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 $\widetilde{Q} \in \mathbb{R}^{m\times n}$, i.e. sus columnas son ortonormales, y la matriz $\widetilde{R} \in \mathbb{R}^{n\times n}$ es triangular superior, como se muestra a continuación:

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

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 descomposición QR. Esto genera el inicio de la revolución científica liderada por los estudiantes de INF/ILI-285, donde su más fuerte arma de combate es la construcción de algoritmos sofisticados que velen por el continuo avance de la Ciencia y la Ingeniería, considerando la restricción de no usar matrices triangulares superiores. Para resolver este problema, se propone construir la siguiente descomposición matricial:

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

Construya e implemente, en Python con NumPy, un algoritmo que determine la descomposición TU propuesta, i.e. el input del algoritmo es la matriz A y retorna la matriz $\widetilde{T}$ donde sus columnas son ortonormales y la matriz $\widetilde{U}$ que es triangular inferior.

El proceso de Gram-Schmidt toma, en orden, los vectores $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$ y genera vectores ortonormales $\mathbf{q}_1$, $\mathbf{q}_2$, ..., $\mathbf{q}_n$:

- $\mathbf{a}_1$ se normaliza para obtener $\mathbf{q}_1$.
- $\mathbf{a}_2$ se hace ortogonal a $\mathbf{q}_1$ y se normaliza el resultado para obtener $\mathbf{q}_2$.
- $\mathbf{a}_3$ se hace ortogonal a $\mathbf{q}_1$ y $\mathbf{q}_2$ y se normaliza el resultado para obtener $\mathbf{q}_3$.
- ...
- **$\mathbf{a}_i$ se hace ortogonal a los vectores anteriormente generados $\mathbf{q}_1$, ..., $\mathbf{q}_{i-1}$ y se normaliza el resultado para obtener $\mathbf{q}_i$.**
- ...
- $\mathbf{a}_n$ se hace ortogonal a $\mathbf{q}_1$, ..., $\mathbf{q}_{n-1}$ y se normaliza el resultado para obtener $\mathbf{q}_n$.

Al hacer esto, los vectores $\mathbf{a}_i$ se pueden expresar de la siguiente manera:

$$\begin{aligned} \mathbf{a}_1 &= r_{11}\mathbf{q}_1 \\ \mathbf{a}_2 &= r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 \\ \mathbf{a}_3 &= r_{13}\mathbf{q}_1 + r_{23}\mathbf{q}_2 + r_{33}\mathbf{q}_3 \\ &\vdots \\ \mathbf{a}_n &= r_{1n}\mathbf{q}_1 + r_{2n}\mathbf{q}_2 + r_{3n}\mathbf{q}_3 + ... + r_{nn}\mathbf{q}_n, \end{aligned}$$

lo cual genera, naturalmente, la factorización QR.

En cambio, la factorización TU genera las siguientes expresiones, si uno se da el trabajo de calcular el producto $\widetilde{T}\widetilde{U}$:

$$\begin{aligned}
\mathbf{a}_1 &= u_{11}\mathbf{t}_1 + u_{21}\mathbf{t}_2 + u_{31}\mathbf{t}_3 + ... + u_{n1}\mathbf{t}_n \\
\mathbf{a}_2 &= \qquad u_{22}\mathbf{t}_2 + u_{32}\mathbf{t}_3 + ... + u_{n2}\mathbf{t}_n \\
\mathbf{a}_3 &= \qquad\qquad u_{33}\mathbf{t}_3 + ... + u_{n3}\mathbf{t}_n \\
&\vdots \\
\mathbf{a}_n &= \qquad\qquad\qquad u_{nn}\mathbf{t}_n
\end{aligned}$$

Esto sugiere hacer un proceso similar al de Gram-Schmidt, pero al revés: tomar los vectores $\mathbf{a}_n$, $\mathbf{a}_{n-1}$, ..., $\mathbf{a}_1$ y generar vectores ortonormales $\mathbf{t}_n$, $\mathbf{t}_{n-1}$, ..., $\mathbf{t}_1$, en ese orden. Es decir:

- $\mathbf{a}_n$ se normaliza para obtener $\mathbf{t}_n$.
- $\mathbf{a}_{n-1}$ se hace ortogonal a $\mathbf{t}_n$ y se normaliza el resultado para obtener $\mathbf{t}_{n-1}$.
- $\mathbf{a}_{n-2}$ se hace ortogonal a $\mathbf{t}_n$ y $\mathbf{t}_{n-1}$ y se normaliza el resultado para obtener $\mathbf{t}_{n-2}$.
- ...
- **$\mathbf{a}_i$ se hace ortogonal a los vectores anteriormente generados $\mathbf{t}_{i+1}$, ..., $\mathbf{t}_n$ y se normaliza el resultado para obtener $\mathbf{t}_i$.**
- ...
- $\mathbf{a}_1$ se hace ortogonal a $\mathbf{t}_2$, ..., $\mathbf{t}_n$ y se normaliza el resultado para obtener $\mathbf{t}_1$.

Por lo tanto, el algoritmo que se debe implementar en Python es casi igual al de la factorización QR descrita en los notebooks, pero iterando en el orden opuesto:

```python
def get_TU(A):
    m, n = A.shape
    T = np.zeros((m, n))
    T[:] = A  # Se copia la matriz A en T para trabajar sobre las columnas de T
    U = np.zeros((n, n))

    for i in range(n-1, 0, -1):
        # Iteramos sobre las columnas de T en orden inverso.
        # w es el vector acumulador que, finalmente, sera el vector ortonormal ti.
        w = T[:, i]

        for j in range(i+1, n):
            # Iteramos sobre los vectores ortonormales tj ya calculados (columnas de T)
            # para ortogonalizar w respecto de estos vectores.
            tj = T[:, j]
            U[i, j] = np.dot(tj, w)
            w -= U[i, j] * tj

        # Normalizamos el resultado.
        U[i, i] = np.linalg.norm(w)
        w /= U[i, i]

        # Se guarda en la matriz T.
        T[:, i] = w

    # Listo
    return T, U
```

2. 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:

$$f(x_1, a, b, \omega) = a\sin(x_1) + b\cos(\omega x_1) = y_1 \quad (1)$$
$$f(x_2, a, b, \omega) = a\sin(x_2) + b\cos(\omega x_2) = y_2 \quad (2)$$

Tenemos 3 coeficientes desconocidos a, b y ω, pero solo tenemos 2 ecuaciones. Para resolver este problema, considere la información adicional que se entregará en cada pregunta.

a) Considere que conocemos el valor de b, el cual llamamos $\hat{b}$, Entregue todas las componentes necesarias para utilizar el método de Newton en $\mathbb{R}^2$, es decir, debe indicar explícitamente la función $F(\cdot) : \mathbb{R}^2 \mapsto \mathbb{R}^2$ 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. Note que no se solicita la inversa de la matriz asociada, sino solo la descripción explícita de la matriz. Justifique su resultado.

Para usar el método de Newton en $\mathbb{R}^2$, se debe definir una función $F(a, \omega)$ a la cual buscarle su raíz $\begin{pmatrix}0\\0\end{pmatrix}$. Para ello, en las ecuaciones anteriores, se puede restar y_1 o y_2, según corresponda, para que el lado derecho sea 0:

$$\begin{cases} a\sin(x_1) + \widehat{b}\cos(\omega x_1) - y_1 &= 0 \\ a\sin(x_2) + \widehat{b}\cos(\omega x_2) - y_2 &= 0 \end{cases}$$

A partir de esto, se puede definir la función $F(a, \omega)$:

$$F(a, \omega) = \begin{pmatrix} a\sin(x_1) + \widehat{b}\cos(\omega x_1) - y_1 \\ a\sin(x_2) + \widehat{b}\cos(\omega x_2) - y_2 \end{pmatrix}$$

También se requiere el jacobiano de F, el cual es:

$$J(a, \omega) = \begin{pmatrix} \dfrac{\partial F}{\partial a} & \dfrac{\partial F}{\partial \omega} \end{pmatrix} = \begin{pmatrix} \sin(x_1) & -\widehat{b}x_1\sin(\omega x_1) \\ \sin(x_2) & -\widehat{b}x_2\sin(\omega x_2) \end{pmatrix}$$

Finalmente, el método de Newton se construye como la siguiente IPF:

$$\begin{aligned} \begin{pmatrix} a_{k+1} \\ \omega_{k+1} \end{pmatrix} &= \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} - J^{-1}(a_k, \omega_k)F(a_k, \omega_k) \\ &= \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} - \begin{pmatrix} \sin(x_1) & -\widehat{b}x_1\sin(\omega_k x_1) \\ \sin(x_2) & -\widehat{b}x_2\sin(\omega_k x_2) \end{pmatrix}^{-1} \begin{pmatrix} a_k\sin(x_1) + \widehat{b}\cos(\omega_k x_1) - y_1 \\ a_k\sin(x_2) + \widehat{b}\cos(\omega_k x_2) - y_2 \end{pmatrix} \end{aligned}$$

partiendo del *initial guess* $\begin{pmatrix} a_0 \\ \omega_0 \end{pmatrix}$.

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

Al tener un par ordenado adicional (x_3, y_3), se agrega la condición de que $f(x_3, a, b, \hat{\omega}) = y_3$. Ahora tenemos 3 ecuaciones lineales con respecto a 2 incógnitas a y b:

$$\begin{cases} a\sin(x_1) + b\cos(\hat{\omega}x_1) = y_1 \\ a\sin(x_2) + b\cos(\hat{\omega}x_2) = y_2 \\ a\sin(x_3) + b\cos(\hat{\omega}x_3) = y_3 \end{cases} \implies \underbrace{\begin{pmatrix} \sin(x_1) & \cos(\hat{\omega}x_1) \\ \sin(x_2) & \cos(\hat{\omega}x_2) \\ \sin(x_3) & \cos(\hat{\omega}x_3) \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\mathbf{k}} = \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}}_{\mathbf{y}}$$

Como este sistema está sobredeterminado, es altamente probable que no exista una solución. En su lugar, se nos pide encontrar los valores $\bar{a}$ y $\bar{b}$ tal que $\bar{a}\sin(x_i) + \bar{b}\cos(\hat{\omega}x_i)$ mejor aproxime a y_i, minimizando el error cuadrático total. Para encontrarlos, hay que plantear las ecuaciones normales:

$$A^T A\bar{\mathbf{k}} = A^T\mathbf{y}$$

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

$$\begin{pmatrix} \sum_{i=1}^{3} \sin^2(x_i) & \sum_{i=1}^{3} \sin(x_i)\cos(\hat{\omega}x_i) \\ \sum_{i=1}^{3} \sin(x_i)\cos(\hat{\omega}x_i) & \sum_{i=1}^{3} \cos^2(\hat{\omega}x_i) \end{pmatrix} \begin{pmatrix} \bar{a} \\ \bar{b} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{3} y_i \sin(x_i) \\ \sum_{i=1}^{3} y_i \cos(\hat{\omega}x_i) \end{pmatrix}$$

Resolver este nuevo sistema de 2×2 entregará los valores $\bar{a}$ y $\bar{b}$ buscados.

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

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

con B una matriz que cumple $\|B\|_\infty = \epsilon$, y D_1, D_2 y D_3 matrices diagonales. Además, se sabe que:

$$|(D_1)_{i,i}| > \sum_{j=1}^{n} |C_{i,j}| + 10^{-2},$$
$$|(D_2)_{i,i}| > \sum_{j=1}^{n} |C_{i,j}| + 10^{-2},$$
$$|(D_3)_{i,i}| > \sum_{j=1}^{n} |C_{i,j}| + 10^{-2},$$

Sin embargo, no se tiene acceso explícito a la matriz A, sino que solamente a D_1, D_2, D_3, B y C, que provienen de un conjunto de mediciones y cálculos de otros programas. Para tener acceso a estas matrices, 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? Recuerda que la norma-∞ de una matriz es el máximo de las sumas absolutas de sus filas.

Una forma de asegurar la convergencia del método de Jacobi es asegurando que A sea una matriz diagonal-dominante.
Para explicar la solución más intuitivamente, se dará un ejemplo con $n = 2$ para ayudar a visualizarlo mejor, pero una solución real debería pasar al caso general con n cualquier natural:

$$D_1 = \begin{pmatrix} d_1 & 0 \\ 0 & d_2 \end{pmatrix}$$
$$D_2 = \begin{pmatrix} d_3 & 0 \\ 0 & d_4 \end{pmatrix}$$
$$D_3 = \begin{pmatrix} d_6 & 0 \\ 0 & d_6 \end{pmatrix}$$
$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$
$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

y por lo tanto

$$A = \left(\begin{array}{cc|cc|cc} d_1 & 0 & b_{11} & b_{12} & c_{11} & c_{12} \\ 0 & d_2 & b_{21} & b_{22} & c_{21} & c_{22} \\ \hline b_{11} & b_{12} & d_3 & 0 & b_{11} & b_{12} \\ b_{21} & b_{22} & 0 & d_4 & b_{21} & b_{22} \\ \hline -c_{11} & -c_{12} & b_{11} & b_{12} & d_5 & 0 \\ -c_{21} & -c_{22} & b_{21} & b_{22} & 0 & d_6 \end{array}\right)$$

Para que A sea diagonal-dominante, cada elemento de su diagonal $d_1, d_2, d_3, d_4, d_5, d_6$ debe tener una magnitud mayor a la suma de las magnitudes del resto de su fila. Por ejemplo, para la fila 1, se debe cumplir que

$$|d_1| > |0| + |b_{11}| + |b_{12}| + |c_{11}| + |c_{12}|$$

Para ello se usa la información adicional entregada:

- $\|B\|_\infty = \epsilon$. La norma-∞ de una matriz es el máximo de las sumas absolutas de sus filas. En este ejemplo, la suma absoluta de la fila 1 de B es $|b_{11}| + |b_{12}|$ y la de la fila 2 es $|b_{21}| + |b_{22}|$. Dado eso, la condición anterior dice que

$$\text{máx}(|b_{11}| + |b_{12}|, \quad |b_{21}| + |b_{22}|) = \epsilon$$

pero como en nuestra desigualdad solo están $|b_{11}|$ y $|b_{12}|$ para la 1° fila de A, nos basta con plantear que

$$|b_{11}| + |b_{12}| \leq \epsilon$$

- $|(D_1)_{i,i}| > \sum_{j=1}^{n} |C_{i,j}| + 10^{-2}$, que en palabras significa que el elemento i-ésimo de la diagonal de D_1 tiene una magnitud mayor a la suma absoluta de la fila i-ésima de C, más 10^{-2}. En este ejemplo, esto nos da información sobre d_1 y d_2. Ahora mismo nos interesa solamente d_1:

$$|d_1| > |c_{11}| + |c_{12}| + 10^{-2}$$

En la última desigualdad, si imponemos que $10^{-2} \geq \epsilon$, entonces obtendremos

$$\begin{aligned} |d_1| &> |c_{11}| + |c_{12}| + 10^{-2} \\ &\geq |c_{11}| + |c_{12}| + \epsilon \\ &\geq |c_{11}| + |c_{12}| + |b_{11}| + |b_{12}| \end{aligned}$$

Por lo tanto, cualquier $\epsilon \leq 10^{-2}$ asegura que $|d_1| > |0| + |b_{11}| + |b_{12}| + |c_{11}| + |c_{12}|$.
Para la fila 2, es exactamente lo mismo: $\epsilon \leq 10^{-2}$.
Para las filas 5 y 6, también pasa lo mismo, pues están involucradas las matrices D_3, B y C de una manera muy similar: $\epsilon \leq 10^{-2}$.
Sin embargo, para las filas 3 y 4, la situación es algo diferente, pues, en vez de estar involucradas las matrices B y C, se repite dos veces la matriz B. Por ejemplo, en la fila 3:

$$|d_3| > |b_{11}| + |b_{12}| + 0 + |b_{11}| + |b_{12}|$$

es decir:

$$|d_3| > 2|b_{11}| + 2|b_{12}|$$

Podemos reutilizar la misma información de antes sobre las filas de B, es decir, $|b_{11}| + |b_{12}| \leq \epsilon$. También hay que agregar que $|d_3| > |c_{11}| + |c_{12}| + 10^{-2}$.
Esta vez debemos imponer que $2\epsilon \leq 10^{-2}$, para asegurar que

$$\begin{aligned} |d_3| &> |c_{11}| + |c_{12}| + 10^{-2} \\ &\geq 10^{-2} \\ &\geq 2\epsilon \\ &> 2(|b_{11}| + |b_{12}|) \end{aligned}$$

obteniendo, así, que cualquier $\epsilon \leq \frac{10^{-2}}{2}$ asegura que $|d_3| > 2|b_{11}| + 2|b_{12}|$. Lo mismo ocurre en la fila 4.
Por lo tanto, en este ejemplo, cualquier $\epsilon \leq \frac{10^{-2}}{2}$ asegura que A es estrictamente diagonal-dominante, y por lo tanto asegura que el método de Jacobi converge.
Esta lógica se extiende para cualquier n natural.

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.

El método de Jacobi propone descomponer $A = L + D + U$ de la siguiente forma (se usará el mismo ejemplo

de antes, con $n = 2$):

$$L = \left(\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline b_{11} & b_{12} & 0 & 0 & 0 & 0 \\ b_{21} & c_{22} & 0 & 0 & 0 & 0 \\ \hline -c_{11} & -c_{12} & b_{11} & b_{12} & 0 & 0 \\ -c_{21} & -c_{22} & b_{21} & b_{22} & 0 & 0 \end{array}\right) = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & \mathbf{0} \\ -C & B & \mathbf{0} \end{pmatrix}$$

$$D = \left(\begin{array}{cc|cc|cc} d_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & d_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_4 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & d_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & d_6 \end{array}\right) = \begin{pmatrix} D_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & D_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & D_3 \end{pmatrix}$$

$$U = \left(\begin{array}{cc|cc|cc} 0 & 0 & b_{11} & b_{12} & c_{11} & c_{12} \\ 0 & 0 & b_{21} & b_{22} & c_{21} & c_{22} \\ \hline 0 & 0 & 0 & 0 & b_{11} & b_{12} \\ 0 & 0 & 0 & 0 & b_{21} & b_{22} \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \begin{pmatrix} \mathbf{0} & B & C \\ \mathbf{0} & \mathbf{0} & B \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

Para el caso n general, simplemente $L = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & \mathbf{0} \\ -C & B & \mathbf{0} \end{pmatrix}$, $D = \begin{pmatrix} D_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & D_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & D_3 \end{pmatrix}$ y $U = \begin{pmatrix} \mathbf{0} & B & C \\ \mathbf{0} & \mathbf{0} & B \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$.

Nota: al igual que en el ejercicio 2, esta es una coincidencia dada solo porque D_1, D_2 y D_3 son, en sí mismas, matrices diagonales.

Al igual que en el ejercicio 2, se plantea el método de Jacobi así:

$$\mathbf{x}^{(k+1)} = D^{-1}\left(\mathbf{b} - (L+U)\mathbf{x}^{(k)}\right)$$

y, por la forma de L, D y U, se puede expandir así:

$$\mathbf{x}^{(k+1)} = \begin{pmatrix} D_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & D_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & D_3 \end{pmatrix}^{-1} \left(\mathbf{b} - \begin{pmatrix} \mathbf{0} & B & C \\ B & \mathbf{0} & B \\ -C & B & \mathbf{0} \end{pmatrix} \mathbf{x}^{(k)}\right)$$

Para poder sacarle el provecho a la estructura de matrices, descomponemos los vectores $\mathbf{x}$ y $\mathbf{b}$ en 3 partes:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}$$

para poder simplificar de esta manera:

$$\begin{pmatrix}\mathbf{x}_1^{(k+1)}\\ \mathbf{x}_2^{(k+1)}\\ \mathbf{x}_3^{(k+1)}\end{pmatrix} = \begin{pmatrix}D_1 & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & D_2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & D_3\end{pmatrix}^{-1}\left(\begin{pmatrix}\mathbf{b}_1\\ \mathbf{b}_2\\ \mathbf{b}_3\end{pmatrix} - \begin{pmatrix}\mathbf{0} & B & C\\ B & \mathbf{0} & B\\ -C & B & \mathbf{0}\end{pmatrix}\begin{pmatrix}\mathbf{x}_1^{(k)}\\ \mathbf{x}_2^{(k)}\\ \mathbf{x}_3^{(k)}\end{pmatrix}\right)$$

$$\begin{pmatrix}\mathbf{x}_1^{(k+1)}\\ \mathbf{x}_2^{(k+1)}\\ \mathbf{x}_3^{(k+1)}\end{pmatrix} = \begin{pmatrix}D_1^{-1} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & D_2^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & D_3^{-1}\end{pmatrix}\left(\begin{pmatrix}\mathbf{b}_1\\ \mathbf{b}_2\\ \mathbf{b}_3\end{pmatrix} - \begin{pmatrix}B\mathbf{x}_2^{(k)} + C\mathbf{x}_3^{(k)}\\ B\mathbf{x}_1^{(k)} + B\mathbf{x}_3^{(k)}\\ -C\mathbf{x}_1^{(k)} + B\mathbf{x}_2^{(k)}\end{pmatrix}\right)$$

$$\begin{pmatrix}\mathbf{x}_1^{(k+1)}\\ \mathbf{x}_2^{(k+1)}\\ \mathbf{x}_3^{(k+1)}\end{pmatrix} = \begin{pmatrix}D_1^{-1} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & D_2^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & D_3^{-1}\end{pmatrix}\begin{pmatrix}\mathbf{b}_1 - B\mathbf{x}_2^{(k)} - C\mathbf{x}_3^{(k)}\\ \mathbf{b}_2 - B\mathbf{x}_1^{(k)} - B\mathbf{x}_3^{(k)}\\ \mathbf{b}_3 + C\mathbf{x}_1^{(k)} - B\mathbf{x}_2^{(k)}\end{pmatrix}$$

$$\begin{pmatrix}\mathbf{x}_1^{(k+1)}\\ \mathbf{x}_2^{(k+1)}\\ \mathbf{x}_3^{(k+1)}\end{pmatrix} = \begin{pmatrix}D_1^{-1}\left(\mathbf{b}_1 - B\mathbf{x}_2^{(k)} - C\mathbf{x}_3^{(k)}\right)\\ D_2^{-1}\left(\mathbf{b}_2 - B\mathbf{x}_1^{(k)} - B\mathbf{x}_3^{(k)}\right)\\ D_3^{-1}\left(\mathbf{b}_3 + C\mathbf{x}_1^{(k)} - B\mathbf{x}_2^{(k)}\right)\end{pmatrix}$$

Esta simplificación permite diseñar un algoritmo donde se tenga solamente una matriz de entre D_1, D_2, D_3, B y C en un tiempo dado:

ALGORITMO

Se recibe como entrada un *initial guess* $\mathbf{x}^{(0)} = \begin{pmatrix}\mathbf{x}_1^{(0)}\\ \mathbf{x}_2^{(0)}\\ \mathbf{x}_3^{(0)}\end{pmatrix}$, un vector $\mathbf{b} = \begin{pmatrix}\mathbf{b}_1\\ \mathbf{b}_2\\ \mathbf{b}_3\end{pmatrix}$ y un número de iteraciones K.

En la iteración $k+1$, con $k \in \{0, 1, ..., K-1\}$, donde se tiene el vector $\mathbf{x}^{(k)} = \begin{pmatrix}\mathbf{x}_1^{(k)}\\ \mathbf{x}_2^{(k)}\\ \mathbf{x}_3^{(k)}\end{pmatrix}$:

1) Obtener B llamando a `getElementFromA(0, 1)`.
2) Calcular los vectores $B\mathbf{x}_1^{(k)}$, $B\mathbf{x}_2^{(k)}$ y $B\mathbf{x}_3^{(k)}$, y almacenarlos.
3) Descartar B, y obtener C llamando a `getElementFromA(0, 2)`.
4) Calcular los vectores $C\mathbf{x}_1^{(k)}$ y $C\mathbf{x}_3^{(k)}$, y almacenarlos.
5) Calcular los 3 vectores $\mathbf{b}_1 - B\mathbf{x}_2^{(k)} - C\mathbf{x}_3^{(k)}$, $\mathbf{b}_2 - B\mathbf{x}_1^{(k)} - B\mathbf{x}_3^{(k)}$ y $\mathbf{b}_3 + C\mathbf{x}_1^{(k)} - B\mathbf{x}_2^{(k)}$, y almacenarlos. Se puede descartar los 5 vectores anteriores para liberar memoria.
6) Descartar C, y obtener D_1 llamando a `getElementFromA(0, 0)`.
7) Calcular $\mathbf{x}_1^{(k+1)} = D_1^{-1}\left(\mathbf{b}_1 - B\mathbf{x}_2^{(k)} - C\mathbf{x}_3^{(k)}\right)$.
8) Descartar D_1, y obtener D_2 llamando a `getElementFromA(1, 1)`.
9) Calcular $\mathbf{x}_2^{(k+1)} = D_2^{-1}\left(\mathbf{b}_1 - B\mathbf{x}_1^{(k)} - B\mathbf{x}_3^{(k)}\right)$.
10) Descartar D_2, y obtener D_3 llamando a `getElementFromA(2, 2)`.
11) Calcular $\mathbf{x}_3^{(k+1)} = D_3^{-1}\left(\mathbf{b}_1 + C\mathbf{x}_1^{(k)} - B\mathbf{x}_2^{(k)}\right)$.
12) Descartar D_3 para la próxima iteración.

para finalmente obtener el vector $\mathbf{x}^{(k+1)} = \begin{pmatrix}\mathbf{x}_1^{(k+1)}\\ \mathbf{x}_2^{(k+1)}\\ \mathbf{x}_3^{(k+1)}\end{pmatrix}$ en esta iteración.

Se realiza K iteraciones, partiendo del *initial guess* $\mathbf{x}^{(0)}$, hasta obtener un resultado $\mathbf{x}^{(K)}$, el cual es retornado por el algoritmo.