Tarea analisis numerico1

December 6, 2024

1 Método directo

El **método directo** (o **método del polinomio característico**) realmente no se basa en un método específico. Distintos autores (como los incluídos en la bibliografía) utilizan distintos métodos (como el método de Muller o el método de Durand–Kerner). Nosotros vamos a utilizar el **método de Newton**. La estrategia general de este método consiste en lo siguiente:

1. Determinar el polinomio característico de la matriz en cuestión.

2. Determinar la derivada del polinomio característico.

3. Escoger una aproximación inicial.

4. Evaluar el polinomio característico en la aproximación inicial.

5. Evaluar la derivada del polinomio característico en la aproximación inicial.

6. Aplicar el método de Newton para aproximar una raíz.

Para hallar todas las raíces (reales) del polinomio característico (y por tanto los valores propios de la matriz), simplemente repetimos el procedimiento desde el paso 3, partiendo de una aproximación inicial diferente. También podemos llevar a cabo una **reducción** del polinomio característico. Dicha reducción se describe a continuación, para un polinomio de grado n cualquiera:

$$P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 ... + a_n \cdot x^n$$

Supongamos por un momento que conocemos una raíz α_1 de P. Entonces, por el algoritmo de la división para polinomios:

$$P(x) = (x - \alpha_1) \cdot Q(x)$$

donde Q es un polinomio de grado $n - 1$. Para hallar otra raíz de P, hallamos una raíz α_2 de Q. De este modo, aplicando nuevamente el algoritmo de la división para polinomios:

$$Q(x) = (x - \alpha_2) \cdot K(x)$$

donde K es un polinomio de grado $n - 2$. Luego:

$$P(x) = (x - \alpha_1) \cdot (x - \alpha_2) \cdot K(x)$$

Podemos seguir de este modo hasta hallar todas las raíces de P (técnicamente, es suficiente seguir hasta obtener un cociente polinomial cuyo grado sea igual a 2, pues a partir de aquí podemos usar la fórmula general cuadrática para hallar las raíces que faltan).

En relación al método, aún hay varios aspectos que debemos discutir:

– **Naturaleza de las raíces:** En el caso de las raíces simples (de multiplicidad 1) sabemos que la convergencia del método de Newton es cuadrática. En el caso de las raíces múltiples (de multiplicidad mayor que 1) la cuestión es un tanto más delicada. De acuerdo con Hoffman, una función no lineal tiende a cero más rápido que su derivada. Por lo tanto, en este caso también es posible usar el método de Newton, teniendo cuidado de interrumpir las iteraciones cuando la derivada del polinomio característico se aproxime a cero. Hay un problema, no obstante, y es que el método pasa de converger cuadráticamente a converger linealmente. Podemos solventar este problema de varias formas. Una de ellas consiste en modificar el método de Newton para recuperar la convergencia cuadrática.

La variante que vamos a considerar se conoce como **método de Newton modificado**. Dicha variante trata simplemente de aplicar el método de Newton a la función:

$$u(x) = \frac{f(x)}{f'(x)}$$

donde f es el polinomio de característico de una matriz A cualquiera. Supongamos que f tiene una raíz α de multiplicidad m. Entonces:

$$f(x) = (x - \alpha)^m \cdot h(x)$$

donde h es un polinomio de grado $n - m$ tal que $h(\alpha) \neq 0$. Luego:

$$u(x) = \frac{(x - \alpha)^m \cdot h(x)}{m \cdot (x - \alpha)^{m-1} \cdot h(x) + (x - \alpha)^m \cdot h'(x)}$$

Es decir:

$$u(x) = \frac{(x - \alpha) \cdot h(x)}{m \cdot h(x) + (x - \alpha) \cdot h'(x)}$$

Como podemos ver, u tiene una raíz simple en $x = \alpha$. Con lo cual, es posible aplicar el método de Newton, con convergencia cuadrática, a la función u para hallar la raíz α de f.

A pesar de que recuperamos la convergencia cuadrática, siguen existiendo algunas desventajas. Para ver mejor ésto, apliquemos el método de Newton con la función u:

$$x_{i+1} = x_i - \frac{u(x_i)}{u'(x_i)}$$

Es decir:

$$x_{i+1} = x_i - \frac{f(x_i) \cdot f'(x_i)}{[f'(x_i)]^2 - f(x_i) \cdot f''(x_i)}$$

Como podemos ver, en esta versión es necesario calcular f''. Adicionalmente, es necesario llevar a cabo más cálculos, lo que reduce la eficiencia y aumenta los errores de redondeo.

Nos falta mencionar el caso de las raíces complejas en pares conjugados. Sin embargo, ya que sólo estamos interesados en valores propios reales, no será necesario abordar esta cuestión.

– **Eficiencia de los cálculos:** El método directo requiere evaluar varias veces el polinomio característico y su derivada en ciertos puntos. Para llevar a cabo dichas operaciones eficientemente, es común implementar dentro del método el **algoritmo de Horner**.

– **Problemas asociados a las raíces:** Unos de los problemas más serios del método tiene que ver con la ausencia de una buena aproximación. Otros problemas asociados a las raíces son: determinación de la existencia de raíces múltiples; raíces muy próximas; raíces en puntos críticos o puntos de inflexión; convergencia a una raíz inesperada o incorrecta. Anteriormente analizamos una variante del método de Newton para abordar éste problema. En general, para abordar los problemas en cuestión, podemos utilizar otras herramientas tecnológicas, como un graficador de funciones, o llevar a cabo lo que Hoffman en su libro llama **incremental search method**.

–**Reducción polinomial:** Anteriormente describimos el proceso de reducción de un polinomio para hallar todas las raíces del mismo. Ésto es posible teóricamente, pero en la práctica puede ser problemático, pues estamos trabajando de hecho con raíces aproximadas y no necesariamente exactas. Consideremos nuevamente al polinomio P de antes:

$$P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 ... + a_n \cdot x^n$$

Sabíamos que:

$$P(x) = (x - \alpha_1) \cdot Q(x)$$

Al obtener una raíz aproximada de Q, no necesariamente obtendremos una raíz aproximada de P. El **polinomio de Wilkinson** muestra a la perfección que las raíces de un polinomio pueden ser muy sensibles a pequeñas perturbaciones de sus coeficientes. Una forma de tratar de solventar el problema es la siguiente: una vez obtenemos una raíz aproximada α_2 de Q, aplicamos nuevamente el método de Newton a P con aproximación inicial α_2 (en lugar de considerar directamente a α_2 como una raíz aproximada de P). Sin embargo, no hay garantía de que esto vaya a funcionar en todos los casos. La mejor forma de abordar este problema es utilizar métodos que aproximen simultáneamente todas las raíces del polinomio, como el **método de Alberth**.

1.1 Implementación del método directo

[ ]:
```
import math
import numpy as np
from sympy import Matrix, symbols, diff

def f(A):
```

```
    # Convertimos la lista a un array de NumPy
    A = np.array(A)

    # Validamos si la matriz es cuadrada
    if A.shape[0] != A.shape[1]:
        raise ValueError("La matriz debe ser cuadrada para calcular el␣
 ↪polinomio característico.")

    # Convertimos el array a una matriz de SymPy para usar métodos algebraicos
    A_sympy = Matrix(A)

    # Declaramos el símbolo lambda para el polinomio característico
    lambd = symbols(' ')

    # Calculamos el polinomio característico
    polinomio = A_sympy.charpoly(lambd).as_expr()

    return polinomio

def fdx(polinomio):
    # Declaramos el símbolo lambda para el polinomio característico
    lambd = symbols(' ')
    # Calculamos la derivada del polinomio
    derivada = diff(polinomio, lambd)

    return derivada


# función para evaluar un punto en el polinomio característico
def f_evaluada(polinomio, punto):
  # Declaramos el símbolo lambda para el polinomio característico
    lambd = symbols(' ')
  # Se retorna el valor del polinomio
    return polinomio.subs(lambd, punto)

# función para evaluar un punto en la derivada del polinomio característico
def fdx_evaluada(polinomio, punto):
  # Declaramos el símbolo lambda para el polinomio característico
    lambd = symbols(' ')
  # Se retorna el valor de la derivada del polinomio
    return fdx(polinomio).subs(lambd, punto)

# función que aproximará el valor propio, dado un un punto inicial
def Newton_method(polinomio, p_0: float, tolerancia: float = 0.001,␣
 ↪max_number_iter:int = 10):
    i : int = 1 # Se inicia un contador para saber el número de iteraciones
```

```
  while i <= max_number_iter: # Se hacen las N iteraciones que se hayan␣
↪declarado

      p = float(p_0 - f_evaluada(polinomio,p_0)/fdx_evaluada(polinomio,p_0))␣
↪# Se aproxima el valor de la nueva iteración
      if abs(p-p_0)<tolerancia: # Se verifica si la diferencia entre la nueva␣
↪iteriacion y la anterior es menor a la tolerancia declarada
          return p # Si esto ocurre, se retorna el valor de la iteracion

      i += 1  #Se incrementa el contador por cada iteración
      p_0 = p # Se actualiza el valor de la iteración anterior, con el valor␣
↪de la nueva iteración

  # Si el error no es menor que la tolerancia al cabo de las N, se retorna lo␣
↪siguiente.
  return f"El método falló después de las {max_number_iter}"
```

2 Método de las potencias

En adelante, cuando hablemos del **valor propio dominante** de una matriz, nos referimos al mayor valor propio en valor absoluto. El **método de las potencias** determina el valor propio dominante junto con su respectivo vector propio. Dicho método se basa en la multiplicación de un vector aproximado $x^(0)$ por una matriz A, con un "escalamiento" del vector resultante y de tal modo que el factor de escalamiento se aproxima al valor propio dominante, mientras que el vector y se aproxima al respectivo vector propio. La estrategia general de este método consiste en lo siguiente:

1. Considere una aproximación inicial $x^{(0)}$ para el vector propio x asociado al valor propio dominante. Una de las componentes de $x^{(0)}$, que llamaremos **componente unidad**, debe ser igual a 1.

2. Se lleva a cabo la operación $Ax^{(0)}$, y hacemos $y^{(1)} = Ax^{(0)}$.

3. "Escalamos" el vector $y^{(1)}$ para que la componente en la posición de la componente unidad siga siendo igual a 1. Obtenemos entonces que $y^{(1)} = \lambda^{(1)}x^{(1)}$.

Lo que sigue es repetir el procedimiento antes dado, a partir del paso 2, reemplazando el papel de $x^{(0)}$ con el de $x^{(1)}$, y continuar con las iteraciones. De este modo, tendremos en general que:

$$Ax^{(k)} = y^{(k+1)} = \lambda^{(k+1)}x^{(k+1)}$$

Si la componente unidad es cero en alguna iteración, se debe cambiar la componente unidad de la aproximación inicial.

Profundicemos en algunos detalles: para aplicar el método de las potencias, supongamos que la matriz cuadrada A tiene n valores propios λ_1,..., λ_n, con un conjunto asociado de vectores propios linealmente independientes $\{v_1, ..., v_n\}$. Si los vectores de dicho conjunto son linealmente dependientes, el método aún podría ser exitoso, pero ya no podemos asegurar que efectivamente lo es (de acuerdo con Burden). Adicionalmente, supongamos que:

$$|\lambda_n| \leq ... \leq |\lambda_2| < |\lambda_1|$$

En palabras, estamos suponiendo que A tiene exactamente un valor propio dominante λ_1. Según Hoffman, la convergencia del método es lenta si hay presencia de valores propios cuyo valor absoluto es cercano al valor absoluto del valor propio dominante. Si no hay un único valor propio dominante, el método directamente falla.

Sea x un vector de $\mathbb{R}^n$ cualquiera. Como los vectores v_1,...,v_n son linealmente independientes, podemos expresar a x como combinación lineal de dichos vectores:

$$x = \sum_{i=1}^{n} C_i v_i$$

Luego:

–$Ax = A(\sum_{i=1}^{n} C_i v_i) = \sum_{i=1}^{n} C_i(Av_i) = \sum_{i=1}^{n} C_i(\lambda_i v_i)$.

–$A^2x = A(Ax) = A(\sum_{i=1}^{n} C_i(\lambda_i v_i)) = \sum_{i=1}^{n}(C_i\lambda_i)(Av_i) = \sum_{i=1}^{n}(C_i\lambda_i)(\lambda_i v_i) = \sum_{i=1}^{n} C_i(\lambda_i^2 v_i)$.

–$A^3x = A(A^2x) = A(\sum_{i=1}^{n} C_i(\lambda_i^2 v_i)) = \sum_{i=1}^{n}(C_i\lambda_i^2)(Av_i) = \sum_{i=1}^{n}(C_i\lambda_i^2)(\lambda_i v_i) = \sum_{i=1}^{n} C_i(\lambda_i^3 v_i)$.

En general:

$$A^k x = \sum_{i=1}^{n} C_i(\lambda_i^k v_i) = \lambda_1^k \sum_{i=1}^{n}(\frac{\lambda_i}{\lambda_1})^k v_i$$

Como λ_1 es el valor propio dominante, $|\lambda_i/\lambda_1| < 1$, para todo i=2,...,n. Luego $(\lambda_i/\lambda_1)^k$ tiende a cero a medida que k tiende a infinito. Por lo tanto:

$$\lim_{k\to\infty} A^k x = \lim_{k\to\infty} \lambda_1^k(C_1 v_1)$$

El límite anterior tiende a cero si $|\lambda_1| < 1$, y diverge si $1 < |\lambda_1|$. Por ésta razón es que debemos llevar a cabo el "escalamiento" entre iteraciones. Supongamos que partimos del vector inicial $x^{(0)}$ cuya componente unidad está en la primera coordenada x_1. Reemplzanado $x^{(0)}$ por x en el procedimiento anterior, llegamos a que $A^k x^{(0)} = \lambda_1^k C_1 v_1 = y^{(k)}$. Luego $y_1^{(k)} = \lambda_1^k C_1$ ($y_1^{(k)}$ es la primera coordenada de $y^{(k)}$). Aplicando el procedimiento una vez más, llegamos a que $y_1^{(k+1)} = \lambda_1^{k+1} C_1$. De modo pues que:

$$\frac{y_1^{(k+1)}}{y_1^{(k)}} = \frac{\lambda_1^{k+1} C_1}{\lambda_1^k C_1} = \lambda_1$$

Así pues, si $y_1^{(k)} = 1$, entonces $y_1^{(k+1)} = \lambda_1$. Si escalamos $y_1^{(k+1)}$ por λ_1 de tal modo que $y_1^{(k+1)} = 1$, tendremos que $y_1^{(k+2)} = \lambda_1$, y así sucesivamente. De esta forma, estamos "factorizando" λ_1 entre

iteraciones, logrando así que el límite que establecimos antes converja. En el límite, el factor de escalamiento se aproxima a λ_1, mientras que el vector escalado se aproxima a su respectivo vector propio.

La siguiente relación establece una forma sencilla en la que el "escalamiento" entre iteraciones se puede llevar a cabo:

$$x^{(k+1)} = \frac{Ax^{(k)}}{||Ax^{(k)}||_\infty}$$

Algunas variaciones del método de las potencias permiten encontrar otros valores propios. Por ejemplo, el **método de las potencias inversas** encuentra el menor valor propio (en valor absoluto). Las **técnicas de deflación** permiten obtener aproximaciones para los otros valores propios de la matriz. La intuición de dichas técnicas es la siguiente: se forma una nueva matriz B, cuyos valores propios son iguales a los de A, a excepción del valor propio dominante de A, que es reemplazado por 0 en B.

2.1 Implementación del método de las potencias

[ ]:
```
#Importamos las librerías necesarias.
import numpy as np

#Declaramos la función que aproximará el valor propio dominante y su correspondiente vector propio.

def Power_method(matriz:np.array, vector:np.array, tolerance:float, max_number_iter:int):
    #La función tomará como parámetros una matriz nxn, un vector n, una tolerancia y un número máximo de iteraciones.
    k : int = 1 #Iniciamos un contador para saber el número de iteraciones que van hasta ese momento.
    p : int = np.linalg.norm(vector, np.inf) #Calculamos la norma infinito del vector.
    vector : np.array = vector / p  #Se divide cada valor del vector por la norma p y se toman como los nuevos valores del vector.
    while k <= max_number_iter: #Se hacen las N iteraciones que se hayan declarado.
        y : np.array = matriz.dot(vector) #Se multiplica la matriz por el vector correspondiente a esa iteración y se asigna ese
                                          #nuevo vector a la variable y.
        u : np.array = np.linalg.norm(y, np.inf) #A continuación, calculamos la norma infinito de ese nuevo vector para cada iteración.
        if u == 0:
            #Si la norma infinito de ese vector es cero, se retorna lo siguiente:
            return (f"Vector propio {vector}"
```

```
                    f" La matriz {matriz} tiene valor propio 0, seleccione un␣
↪nuevo vector propio vector x y reinicie la ejecución ")

        new_vector : np.array = vector - (y/u)
        #Declaramos un nuevo vector correspondiente al vector del error entre␣
↪la iteración anterior y la que se está ejecutando.
        Err : np.array = np.linalg.norm(new_vector, np.inf) #Calculamos la␣
↪norma infinito de este nuevo vector.
        vector : np.array = y / u #Reemplazamos el valor de la iteración␣
↪anterior por los valores de la nueva iteración.
        if Err < tolerance: #Si el error es menor a la tolerancia, retornamos␣
↪el valor y vector propio dominantes.
            return u, vector
        k += 1 #Se incrementa el contador por cada iteración.
    #Si el error no es menor que la tolerancia al cabo de las N iteraciones, se␣
↪retorna lo siguiente:
    return f"El número máximo de iteraciones se ha excedido"
```

3 Algoritmo QR

El método directo y el método de potencia permiten determinar un valor propio a la vez. El **algoritmo QR** o **método de descomposición QR**, en cambio, nos permite determinar todos los valores propios de una matriz.

La parte esencial del algoritmo QR es la **descomposición QR** de una matriz. Existen varias formas de determinar dicha descomposición. Algunas de ellas son: el **proceso de ortogonalización de Gram-Schmidt**, mediante **rotaciones de Givens** y mediante **reflexiones de Householder**. En la implementación del algoritmo vamos a utilizar la función **linalg.qr** de Numpy para determinar la descomposición QR de una matriz A. Dicha función hace uso de las reflexiones de Householder para hallar la descomposición en cuestión. La intuición detrás del proceso es la siguiente: las reflexiones de Householder se aplican secuencialmente para "eliminar" los elementos por debajo de la diagonal de A, convirtiéndola en una matriz triangular superior R. Al mismo tiempo, las transformaciones acumuladas forman la matriz ortogonal Q. El costo computacional de éste método es similar al del proceso de ortogonalización de Gram-Schmidt (aunque es importante aclarar que se suele preferir el primero, pues el segundo suele "sufrir" más por los errores de redondeo). Nosotros, por simplicidad (para no perdernos en detalles ajenos al método como tal), vamos a utilizar la función mencionada para llevar a cabo la parte de la descomposición QR, sin profundizar en las reflexiones de Householder. Sin embargo, de ser absolutamente necesario, podríamos implementar fácilmente el conocido proceso de ortogonalización de Gram-Schmidt para llevar a cabo esta tarea.

La estrategia general del algoritmo QR consiste en lo siguiente:

1. Consideramos $A_0 = A$.
2. Hallamos la descomposición QR de A_0, para obtener $A_0 = Q_0 R_0$.
3. Consideramos $A_1 = R_0 Q_0$.

Lo que sigue es repetir el procedimiento antes dado, reemplazando el papel de A_0 con el de A_1, y

continuar con las iteraciones. De este modo, tendremos en general que:

– Si n es par, $A_n = Q_n R_n$. Ésto es: determinamos la descomposición QR de A_n.

– Si n es impar, $A_n = R_{n-1} Q_{n-1}$. Ésto es: consideramos la matriz a la cual le hallaremos la descomposición QR en el siguiente paso.

Las matrices A_n y A_{n+1} son **semejantes**, lo que nos asegura que tienen los mismos valores propios (pero no los mismos vectores propios). Ésto nos asegura, por así decirlo, que durante las iteraciones no perdemos de vista los valores propios de la matriz A. Además, la matriz A_n se aproxima a una matriz triangular superior, en la cual podemos encontrar los valores propios en la diagonal principal. Ésta aproximación de A_n a una matriz triangular superior no se justifica en ningún libro de la bibliografía. Está demostrada, por supuesto, pero la cuestión parece ser un poco difícil, por lo que tampoco será abordada aquí. Bastará decir que, intuitivamente, el algoritmo QR "reorganiza" progresivamente la matriz A para reflejar mejor su estructura espectral, reduciendo las entradas fuera de la diagonal.

De acuerdo con Hoffman, la convergencia del algoritmo está asegurada (por lo general), aunque podría ser algo lenta. Para acelerar la convergencia, se suelen llevar a cabo dos modificaciones:

– **Transformar la matriz de partida:** Generalmente lo que se hace es transformar la matriz A en otra matriz "casi" triangular. Formalmente, se utilizan **transformaciones de Householder** para convertir la matriz A en una **matriz de Hessenberg**.

– **Shifting:** Mediante lo que Hoffman llama "shifting" de los valores propios de la matriz de Hessenberg de la cual se parte, a medida que se ejecuta el algoritmo, se puede mejorar significativamente la convergencia del método. Hay varias formas de llevar a cabo dicho "shifting", de acuerdo a ciertas consideraciones.

3.1 Implementación del algoritmo QR

[ ]:
```
import numpy as np #importamos la librería numpy que utilizaremos para trabajar␣
 ↪con matrices

#La siguiente función toma como parámetros una Lista de tamaño nxn, un número␣
 ↪máximo de iteraciones opcional,
#que por defecto será 100, y una tolerancia numérica, también opcional
def QRDecomposition(A, maxIteraciones = 1000, tol = 1e-10):
    A_n = np.array(A) #Convertimos la lista en un array de numpy para operar␣
 ↪con el fácilmente

    for i in range(maxIteraciones): #Comenzamos a iterar el método

        #Cuando i es par:
        if i%2 == 0:
            Q_n,R_n = np.linalg.qr(A_n) #Descomposición QR de la matriz usando␣
 ↪la función integrada de numpy

        #Cuando i es impar
        if i%2 == 1:
```

```
            A_n = R_n@Q_n

    """Sabemos que en este método los valores que convergen a los valores␣
 ↪propios están ubicados
       sobre la diagonal de la matriz A_n, entonces vamos a recorrer esa diagonal␣
 ↪y guardarlos en una lista"""

    valoresPropios = [] #Creamos la lista
    for i in range(A_n.shape[0]): #.shape[0] es para contar el número de filas,␣
 ↪el cual será igual al numero de valores en la diagonal
                                  #dado que la matriz es cuadrada
        valoresPropios.append(A_n[i,i]) #Ingresamos los valores con índice␣
 ↪[i,i] en la diagonal

    return valoresPropios
```

Casos de prueba:

[ ]:
```
import numpy as np

# Declaramos los vectores correspondientes para los casos de prueba.
vector = np.array([1, 1]) #Dimensión 2.
vector1 = np.array([1, 2, 0]) #Dimensión 3.
vector2 = np.array([1, 1, 1, 1]) #Dimensión 4.

#Casos de prueba: declaramos un diccionario con las matrices de prueba.

test = {"A": np.array([[2, 1], [3, 4]]),
        "B": np.array([[3, 2], [3, 4]]),
        "C": np.array([[2, 3], [1, 4]]),
        "D": np.array([[1, 1, 2], [2, 1, 1],[1, 1, 3]]),
        "E": np.array([[1, 1, 2], [2, 1, 3],[1, 1, 1]]),
        "F": np.array([[2, 1, 2], [1, 1, 3],[1, 1, 1]]),
        "G": np.array([[1, 1, 1, 2], [2, 1, 1, 1],[3, 2, 1, 2],[2, 1, 1, 4]]),
         "H": np.array([[1, 2, 1, 2], [2, 1, 1, 1],[3, 2, 1, 2],[2, 1, 1, 4]])}

print("----------------------------------------Método␣
 ↪directo----------------------------------------")
valor_inicial = 0
for name, matriz in test.items(): # iteramos sobre el diccionario con su␣
 ↪respectiva clave y valor.
    #Utilizamos el metodo de Newton e imprimimos cada valor propio,␣
 ↪correspondiente al valor inicial que se la haya pasado como parámetro.
    print(f"La matriz {name} tiene el siguiente valor propio:␣
 ↪{Newton_method(f(matriz),valor_inicial)} correspondiente al valor inicial␣
 ↪{valor_inicial} ")
    valor_inicial += 1
```

```
print("\n")
print("----------------------------------------Método de las␣
 ↪potencias----------------------------------------")
for name, matriz in test.items(): # Iteramos sobre el diccionario con su␣
 ↪respectiva clave y valor.
    dimension : int = matriz.shape[0] # Obtenemos la dimensión de cada matriz␣
 ↪del diccionario.
    if dimension == 2: # Matrices de dimensión 2.
        print(# Retornamos el correspondiente valor dominante a cada matriz,␣
 ↪con su vector propio.
            f"El valor propio dominante asociado a la matriz {name} es:␣
 ↪{Power_method(matriz, vector, tolerance=1e-5, max_number_iter=10)[0]}"
             f" y su vector propio asociado es: {Power_method(matriz, vector,␣
 ↪tolerance=1e-5, max_number_iter=10)[-1]}")
    if dimension == 3: # Matrices de dimensión 3.
        print( # Retornamos el correspondiente valor dominante a cada matriz,␣
 ↪con su vector propio.
            f"El valor propio dominante asociado a la matriz {name} es:␣
 ↪{Power_method(matriz, vector1, tolerance=1e-12, max_number_iter=20)[0]}"
            f" y su vector propio asociado es: {Power_method(matriz, vector1,␣
 ↪tolerance=1e-10, max_number_iter=20)[-1]}")
    if dimension == 4: # Matrices de dimensión 4.
        print( # Retornamos el correspondiente valor dominante a cada matriz,␣
 ↪con su vector propio.
            f"El valor propio dominante asociado a la matriz {name} es:␣
 ↪{Power_method(matriz,vector2 , tolerance=1e-5, max_number_iter=10)[0]}"
            f" y su vector propio asociado es: {Power_method(matriz, vector2,␣
 ↪tolerance=1e-5, max_number_iter=10)[-1]}")

print("\n")
print("----------------------------------------Algoritmo␣
 ↪QR----------------------------------------")
for nombre, matriz in test.items(): #Creamos un ciclo que recorre el␣
 ↪diccionario y va probando cada caso
  print(f"Los valores propios de la matriz {nombre} son:")
  print(QRDecomposition(matriz)) #Utilizamos la funcion QRDecomposition e␣
 ↪imprimimos cada valor propio
  print()
```

```
----------------------------------------Método
directo----------------------------------------
La matriz A tiene el siguiente valor propio: 0.9999999999737855 correspondiente
al valor inicial 0
La matriz B tiene el siguiente valor propio: 1.0 correspondiente al valor
inicial 1
```

```
La matriz C tiene el siguiente valor propio: 0.9999999070777055 correspondiente
al valor inicial 2
La matriz D tiene el siguiente valor propio: -0.2851435433398339 correspondiente
al valor inicial 3
La matriz E tiene el siguiente valor propio: 4.048917339522421 correspondiente
al valor inicial 4
La matriz F tiene el siguiente valor propio: 4.124885445932234 correspondiente
al valor inicial 5
La matriz G tiene el siguiente valor propio: 6.634534720780106 correspondiente
al valor inicial 6
La matriz H tiene el siguiente valor propio: 6.827262252354639 correspondiente
al valor inicial 7


----------------------------------------Método de las
potencias----------------------------------------
El valor propio dominante asociado a la matriz A es: 5.000017066739485 y su
vector propio asociado es: [0.33333447 1.        ]
El valor propio dominante asociado a la matriz B es: 6.000017861289228 y su
vector propio asociado es: [0.66666766 1.        ]
El valor propio dominante asociado a la matriz C es: 5.0 y su vector propio
asociado es: [1. 1.]
El valor propio dominante asociado a la matriz D es: 4.5070186440933915 y su
vector propio asociado es: [0.77812384 0.72889481 1.        ]
El valor propio dominante asociado a la matriz E es: 4.048917339520433 y su
vector propio asociado es: [0.69202147 1.         0.55495813]
El valor propio dominante asociado a la matriz F es: 4.124885419764718 y su
vector propio asociado es: [1.         0.90539067 0.60974737]
El valor propio dominante asociado a la matriz G es: 6.634567462477316 y su
vector propio asociado es: [0.60704991 0.54782282 0.87261933 1.        ]
El valor propio dominante asociado a la matriz H es: 6.8272801484610675 y su
vector propio asociado es: [0.68834475 0.56058636 0.88999093 1.        ]


----------------------------------------Algoritmo
QR----------------------------------------
Los valores propios de la matriz A son:
[4.999999999999999, 0.9999999999999997]

Los valores propios de la matriz B son:
[5.999999999999998, 1.0000000000000002]

Los valores propios de la matriz C son:
[5.0, 1.0]

Los valores propios de la matriz D son:
[4.507018644092973, 0.7781238377368096, -0.28514248182978574]
```

```
Los valores propios de la matriz E son:
[4.048917339522307, -0.692021471630096, -0.3568958678922091]

Los valores propios de la matriz F son:
[4.124885419764577, -0.7615571818318913, 0.6366717620673171]

Los valores propios de la matriz G son:
[6.634534463633592, 1.5085633449433231, -0.7356415384387974,
-0.4074562701381241]

Los valores propios de la matriz H son:
[6.827262250104039, 1.7281159082896407, -1.0879349236625606,
-0.4674432347311208]
```

3.2 Conclusiones

–**Ventajas y desventajas del método directo:** Aquí nos referimos al método directo que utiliza a su vez el método de Newton. Entre las ventajas del método directo encontramos: bajo ciertas condiciones, el método es preciso y converge rápidamente. Es una opción razonable para matrices pequeñas. Algunas desventajas: para matrices grandes, el coste computacional aumenta significativamente (por la cantidad de operaciones que es necesario llevar a cabo; entre ellas, por ejemplo, determinar el polinomio característico). Por razones similares al punto anterior, el método puede ser numéricamente inestable. Como vimos, si hay valores propios repetidos, el método converge más lento, y puede incluso fallar.

–**Ventajas y desventajas del método de las potencias:** Entre las ventajas del método de las potencias encontramos: es fácil y eficiente de implementar, dado que solo necesita operaciones básicas y el almacenamiento de pocos elementos (una matriz y un vector en cada iteración). Es especialmente útil para matrices grandes y **dispersas**. El hecho de encontrar el vector propio asociado al valor propio dominante también es una ventaja, pues los otros métodos solamente determinan valores propios. Algunas desventajas: solamente encuentra el valor propio dominante. Requiere de matrices bien condicionadas y con espectros sencillos (se mencionó, por ejemplo, que si el valor propio dominante no está claramente separado de los demás, la convergencia del método podría ser lenta o inestable).

–**Ventajas y desventajas del algoritmo QR:** Entre las ventajas del algoritmo QR encontramos: determina todos los valores propios de la matriz. El método es estable y funciona bien incluso para matrices mal condicionadas. Algunas desventajas: es computacionalmente costoso. Converge lentamente. Las operaciones que se llevan a cabo durante el método (en particular, la factorización QR) requieren un consumo de memoria considerable.