Por definición de valores y vectores propios tenemos que

$(A - \lambda I)v = \vec{0}$

Dónde A es una matriz $n \times n$, λ es el valor propio, v el vecor propio al que esta ligado el valor porpio, I la matriz identidad y $\vec{0}$ el vector nulo en $\mathbb{R}^n$.

Aqui para que haya soluciones no nulas es necesario que $Det(A - \lambda I) = 0$ pues de lo contrario la matriz será invertible y por tanto las unicas soluciones serán triviales.

Con esto claro tenemos que al expandir $Det(A - \lambda I) = 0$ se obtiene un polinomio de grado n cuya incognita es λ, el cual conocemos como polinomio caracteristico, al hayar los ceros del polinomio estamos hayando posibles valores propios y al reemplazar estos en $(A - \lambda I)v = \vec{0}$ y despejando el vector v podremos hayar el vector propio ligada a este valor propio.

Aunque el polinomio característico ofrece una forma de calcular los valores propios, resolver directamente las raíces del polinomio no es factible para matrices grandes. Por lo tanto, se aplican métodos iterativos como el método de Newton para encontrar aproximaciones a los valores propios.

Haz doble clic (o ingresa) para editar

supuestos para la convergencia con el metodo de Newton

Suposición Inicial:

El método requiere una suposición inicial que esté razonablemente cerca de la raíz real (valor propio). Las malas suposiciones iniciales pueden resultar en una convergencia lenta o divergente.

No Singularidad de la Matriz:

El método de Newton asume que la derivada (p_A'(\lambda)) no es cero en el valor propio. Si el valor propio tiene multiplicidad mayor que 1, el método puede fallar a menos que se modifique para manejar tales casos.

Valores Propios Distintos:

El método de Newton funciona mejor para matrices con valores propios distintos. Si la matriz tiene múltiples valores propios con alta multiplicidad, la convergencia puede ser lenta o inestable sin modificaciones al método.

Polinomios con Raíces Simples:

El método de Newton generalmente asume que las raíces del polinomio característico son simples (no repetidas). Para raíces repetidas, se suelen usar algoritmos especializados como deflación o técnicas shift-invert para mejorar la convergencia.

Con todas estas suposiciones, según "Numerical Analysis" de Burden y Faires pagina 84 podemos afirmar que el polinomio será cuadraticamente convergente, es decirque existe una constante $C > 0$ tal que $|x_{k+1} - x^*| \leq C|x_k - x^*|^2$ para cada iteración del metodo.

Codigo y test

```
import numpy as np

# Function to compute eigenvalues and eigenvectors of a matrix
def compute_eigen(matrix):
    eigenvalues, eigenvectors = np.linalg.eig(matrix)
    # Force eigenvalues and eigenvectors to be real if the imaginary part is negligible
    eigenvalues = eigenvalues.real  # Remove imaginary part completely
    eigenvectors = eigenvectors.real  # Remove imaginary part completely
    return eigenvalues, eigenvectors

# Function to compute the characteristic polynomial of a matrix
def characteristic_polynomial(matrix):
    return np.poly(matrix)
```

```
# Compute the error for eigenvalues and eigenvectors
def compute_error(matrix, eigenvalues, eigenvectors):
    max_error = 0
    for i, eigenvalue in enumerate(eigenvalues):
        vec = eigenvectors[:, i]
        error = np.linalg.norm(matrix @ vec - eigenvalue * vec)
        max_error = max(max_error, error)
    return max_error

# Define a function to display results clearly for a single matrix
def display_results_for_matrix(matrix, matrix_name):
    eigenvalues, eigenvectors = compute_eigen(matrix)
    char_poly = characteristic_polynomial(matrix)
    max_error = compute_error(matrix, eigenvalues, eigenvectors)

    # Format the results nicely
    print(f"\nResults for Matrix {matrix_name}:")
    print(f"Matrix:\n{matrix}\n")
    print("Characteristic Polynomial:")
    print(" + ".join([f"{coeff:.3f}x^{len(char_poly)-i-1}" for i, coeff in enumerate(char_poly) if coeff != 0]).replace("x^0", "")
    print("\nEigenvalues:")
    print(", ".join([f"{val:.3f}" for val in eigenvalues]))

    # Format eigenvectors nicely
    print("\nEigenvectors:")
    for i in range(eigenvectors.shape[1]):
        print(f"Eigenvector {i+1}: [{', '.join([f'{v:.3f}' for v in eigenvectors[:, i]])}]")

    print(f"\nMax Error: {max_error:.2e}")
    print("-" * 50)

# Define matrices as provided
matrices = {
    "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]]),
}

# Display results for each matrix one by one
for name, matrix in matrices.items():
    display_results_for_matrix(matrix, name)
```

```
Results for Matrix A:
Matrix:
[[2 1]
 [3 4]]

Characteristic Polynomial:
1.000x^2 + -6.000x^1 + 5.000

Eigenvalues:
1.000, 5.000

Eigenvectors:
Eigenvector 1: [-0.707, 0.707]
Eigenvector 2: [-0.316, -0.949]

Max Error: 0.00e+00
--------------------------------------------------

Results for Matrix B:
Matrix:
[[3 2]
 [3 4]]

Characteristic Polynomial:
1.000x^2 + -7.000x^1 + 6.000

Eigenvalues:
1.000, 6.000

Eigenvectors:
Eigenvector 1: [-0.707, 0.707]
Eigenvector 2: [-0.555, -0.832]

Max Error: 0.00e+00
--------------------------------------------------

Results for Matrix C:
```

```
Matrix:
[[2 3]
 [1 4]]

Characteristic Polynomial:
1.000x^2 + -6.000x^1 + 5.000

Eigenvalues:
1.000, 5.000

Eigenvectors:
Eigenvector 1: [-0.949, 0.316]
Eigenvector 2: [-0.707, -0.707]

Max Error: 5.55e-17
--------------------------------------------------

Results for Matrix D:
Matrix:
[[1 1 2]
```

Método de las Potencias

Introducción

El método de las potencias es una técnica numérica utilizada principalmente para encontrar el valor propio dominante de una matriz, es decir, aquel con el mayor valor absoluto. Además, este método puede extenderse para determinar otros valores propios mediante modificaciones específicas en su algoritmo, como se detalla en *Numerical Analysis* (Burden y Faires, 9ª edición, 2011).

En esencia, el método consiste en realizar iteraciones sucesivas de productos entre una matriz y un vector inicial (normalizado), generando una secuencia de vectores unitarios que, en cada iteración, convergen hacia el vector propio asociado al valor propio dominante. Cabe destacar que la convergencia del método depende de que el valor propio dominante sea único y que su módulo sea significativamente mayor que el de los demás valores propios.

Este método tiene numerosas aplicaciones, entre las cuales destaca su uso en algoritmos de recomendación y clasificación en la web. Un ejemplo notable es el algoritmo PageRank de Google, que emplea el método de las potencias para calcular la importancia relativa de páginas web en función de su estructura de enlaces (Ipsen, I. (n.d.). Analysis and computation of Google's PageRank).

Convergencia

Para garantizar la convergencia del método, es necesario asumir las siguientes condiciones:

Dada una matriz $A \in \mathbb{R}^{n\times n}$, esta debe cumplir que:

- Existe un conjunto de n vectores propios linealmente independientes (Kincaid y Cheney, 2002, p. 257) que forman una base del espacio: $\{v_1, v_2, \ldots, v_n\}$.
- Existe un valor propio cuyo valor absoluto es mayor que los valores absolutos del resto. Es decir, si λ_j es el valor propio asociado al vector propio v_j, se cumple que:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|.$$

Es importante resaltar que, según Burden y Faires (2011), el primer criterio no necesariamente requiere n vectores propios linealmente independientes.

Para iniciar el análisis del método, consideremos que para cualquier vector $x \in \mathbb{R}^n$, se tiene:

$$x = \sum_{j=1}^{n} \beta_j v_j,$$

donde $\beta_j \in \mathbb{R}$. Así, es posible observar que:

$$Ax = \sum_{j=1}^{n} \beta_j A v_j = \sum_{j=1}^{n} \beta_j \lambda_j v_j, \quad A^2 x = \sum_{j=1}^{n} \beta_j \lambda_j^2 v_j, \quad \ldots, \quad A^k x = \sum_{j=1}^{n} \beta_j \lambda_j^k v_j,$$

lo cual se deduce de la definición de vector propio.

A continuación, se elige un vector unitario arbitrario x_0 según la norma y el espacio en el que se desee trabajar. De acuerdo con *Matrix Computations* (Golub y Van Loan, 4ª edición, 2013, p. 391), en $\mathbb{C}$ es posible deducir el método utilizando la norma 2 ($||\cdot||_2$). Sin embargo, Burden y Faires (2011) desarrollan el método en $\mathbb{R}$ empleando la norma infinita ($||\cdot||_\infty$). En esta deducción se utilizará la norma infinita.

Definimos dos secuencias y_k y x_k tales que:

$$y_k = Ax_{k-1},$$
$$x_k = \frac{y_k}{||y_k||_\infty}.$$

Con estas, se define una secuencia μ_k como:

$$\mu_k = \frac{||y_k||_\infty}{||x_{k-1}||_\infty} = \frac{||Ax_{k-1}||_\infty}{||x_{k-1}||_\infty} = \frac{||\sum_{j=1}^{n} \beta_j \lambda_j^k v_j||_\infty}{||\sum_{j=1}^{n} \beta_j \lambda_j^{k-1} v_j||_\infty}.$$

Dado que la norma infinita no es un operador lineal, este paso de simplificación es válido debido a que el valor propio dominante escala el vector propio asociado (v_1) de manera que:

$$||y_k||_\infty = |\beta_1 \lambda_1^k| ||v_1||_\infty.$$

Esto implica que:

$$\mu_k = \lambda_1 \left(\frac{\sum_{j=1}^{n} |\beta_j| \left(\frac{|\lambda_j|}{|\lambda_1|}\right)^k ||v_j||_\infty}{\sum_{j=1}^{n} |\beta_j| \left(\frac{|\lambda_j|}{|\lambda_1|}\right)^{k-1} ||v_j||_\infty} \right).$$

Gracias a la segunda hipótesis, dado que $|\lambda_1| > |\lambda_j|$ para todo $j \in \{2, \ldots, n\}$, se tiene que $\left(\frac{|\lambda_j|}{|\lambda_1|}\right)^k \to 0$ cuando $k \to \infty$. Por lo tanto, podemos afirmar que:

$$\lim_{k\to\infty} \mu_k = \lim_{k\to\infty} \lambda_1 \left(\frac{|\beta_1| ||v_1||_\infty + \sum_{j=2}^{n} |\beta_j| \left(\frac{|\lambda_j|}{|\lambda_1|}\right)^k ||v_j||_\infty}{|\beta_1| ||v_1||_\infty + \sum_{j=2}^{n} |\beta_j| \left(\frac{|\lambda_j|}{|\lambda_1|}\right)^{k-1} ||v_j||_\infty} \right) = \lambda_1.$$

Esto demuestra que, bajo las hipótesis asumidas, el método de las potencias converge al valor propio dominante con suficientes iteraciones.

Análisis de Error

Según Burden y Faires (2011, p. 579) y Golub y Van Loan (2013, p. 366), el error asociado al método depende principalmente del cociente $\frac{\lambda_2}{\lambda_1}$, y existe una constante r para k suficientemente grande tal que:

$$|\mu_k - \lambda_1| \approx r\left|\frac{\lambda_2}{\lambda_1}\right|^k.$$

De esta forma, se puede expresar explícitamente la tasa de convergencia mediante el cociente de dos errores consecutivos:

$$\frac{|\mu_k - \lambda_1|}{|\mu_{k-1} - \lambda_1|} = \left|\frac{\lambda_2}{\lambda_1}\right|.$$

Tomando el límite cuando $k \to \infty$, se obtiene:

$$\lim_{k\to\infty} \frac{|\mu_k - \lambda_1|}{|\mu_{k-1} - \lambda_1|} = \left|\frac{\lambda_2}{\lambda_1}\right|.$$

Así, se concluye que el algoritmo tiene una tasa de convergencia lineal, expresada como:

$$\mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right).$$

Nota:

El código anterior puede contener errores debido a la conversión del documento .tex al formato aceptado por Colab. Por esta razón, dejo a continuación el enlace donde se puede encontrar el archivo original: Tarea_Analisis_Numerico_Power_method.pdf y al codigo en Overleaf: https://www.overleaf.com/read/hztqtypcdvyf#a0f348 (En caso de tener problemas al dar click, copie y pegue en el buscador el link)

Codigo

```
import numpy as np

def PowerMethod(A, x0 , TOL, N): #A es la matriz, x0 el vector inicial, TOL es la tole
  k = 1  # Contador de iteraciones
  x = x0 / np.linalg.norm(x0, ord=np.inf)  # Normalizar el vector inicial utilizando l
  mu = 0  # Inicializar el valor propio


  while k <= N:
    y = A @ x  # Calcular el producto matriz-vector
    mu_n = np.linalg.norm(y, ord = np.inf) / np.linalg.norm(x, ord = np.inf)  # Calcul

    if np.linalg.norm(y, ord=np.inf) == 0:  # Si la norma de y es cero, la matriz tier
      print(f'Por favor ingrese otro vector x, pues con {x0} la matriz A tiene un valo
      return None

    x_n = y / np.linalg.norm(y, ord=np.inf)  # Normalizar y para obtener la siguiente
    ERR = abs(mu_n - mu)  # Calcular el error entre los vectores sucesivos

    if ERR < TOL:
```

```
      return mu_n, x_n  # Retornar el valor propio dominante y el vector propio corres

    # Actualizar x y el valor propio para la siguiente iteración
    x = x_n
    mu = mu_n
    k += 1  # Incrementar el contador de iteración

  print('El número máximo de iteraciones ha sido alcanzado y no se logró llegar a la t
  return None
```

Tests y comparación

Notas:

- Los vectores iniciales x_0 se generan de forma aleatoria, lo cual puede ocasionar que, en raras ocasiones, el método tarde más de lo esperado en converger o incluso no converja. Sin embargo, se recomienda volver a ejecutar el código para intentar obtener una o varias convergencias exitosas.
- Observe a continuación que los vectores propios normalizados por el verificador, en la mayoría de los casos, corresponden al vector obtenido por el método de potencias programado, multiplicado por -1. Esto tiene sentido, ya que todo múltiplo escalar (diferente de 0) de un vector propio también es un vector propio.

Verificador de valor y vector propio:

```
import numpy as np

def verificador(A):
  eigenvalues, eigenvectors = np.linalg.eig(A)
  for i in range(len(eigenvalues)):
    print(f"Valor propio {i + 1}: {eigenvalues[i]}")
    print(f"Vector propio {i + 1}: {eigenvectors[:, i] / np.linalg.norm(eigenvectors[
```

1.

```
A = np.array([[2,1],
              [3,4]])

x0 = np.random.rand(2)
TOL = 1e-10
N = 100

Resultado = PowerMethod(A,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
```

```
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(A)
```

```
Valor propio: 5.00000000000644, Vector propio: [0.33333333 1.        ]


 Valor propio 1: 1.0
 Vector propio 1: [-1.  1.]

 Valor propio 2: 5.0
 Vector propio 2: [-0.33333333 -1.        ]
```

2.

```
B = np.array([[3, 2],
              [3, 4]])

x0 = x0 = np.random.rand(2)
TOL = 1e-10
N = 100

Resultado = PowerMethod(B,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(B)
```

```
Valor propio: 6.000000000009539, Vector propio: [0.66666667 1.        ]


 Valor propio 1: 1.0
 Vector propio 1: [-1.  1.]

 Valor propio 2: 6.0
 Vector propio 2: [-0.66666667 -1.        ]
```

3.

```
C = np.array([[2, 3],
              [1, 4]])

x0 = np.random.rand(2)
TOL = 1e-10
N = 100

Resultado = PowerMethod(C,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
```

```
    print('\n')
    verificador(C)
```

```
Valor propio: 4.999999999993047, Vector propio: [1. 1.]


 Valor propio 1: 1.0
 Vector propio 1: [-1.         0.33333333]

 Valor propio 2: 5.0
 Vector propio 2: [-1. -1.]
```

4.

```
D = np.array([[1, 1, 2],
              [2, 1, 1],
              [1, 1, 3]])

x0 = np.random.rand(3)
TOL = 1e-10
N = 100

Resultado = PowerMethod(D,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(D)
```

```
Valor propio: 4.507018644097645, Vector propio: [0.77812384 0.72889481 1.        ]


 Valor propio 1: 4.507018644092977
 Vector propio 1: [-0.77812384 -0.72889481 -1.        ]

 Valor propio 2: -0.285142481829786
 Vector propio 2: [-0.57840419  1.         -0.1283341 ]

 Valor propio 3: 0.7781238377368094
 Vector propio 3: [-0.14722855 -1.          0.51633325]
```

5.

```
E = np.array([[1, 1, 2],
              [2, 1, 3],
              [1, 1, 1]])

x0 = np.random.rand(3)
TOL = 1e-10
N = 100
```

```
Resultado = PowerMethod(E,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(E)
```

```
Valor propio: 4.048917339519715, Vector propio: [0.69202147 1.         0.55495813]


 Valor propio 1: 4.0489173395223075
 Vector propio 1: [-0.69202147 -1.         -0.55495813]

 Valor propio 2: -0.3568958678922092
 Vector propio 2: [-1.          0.2469796   0.55495813]

 Valor propio 3: -0.6920214716300966
 Vector propio 3: [-0.35689587 -1.          0.80193774]
```

6.

```
F = np.array([[2, 1, 2],
              [1, 1, 3],
              [1, 1, 1]])

x0 = np.random.rand(3)
TOL = 1e-10
N = 100

Resultado = PowerMethod(F,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(F)
```

```
Valor propio: 4.1248854198028155, Vector propio: [1.         0.90539067 0.6097473]


 Valor propio 1: 4.1248854197645715
 Vector propio 1: [-1.         -0.90539067 -0.60974737]

 Valor propio 2: 0.6366717620673152
 Vector propio 2: [-1.          0.91934399  0.22199212]

 Valor propio 3: -0.7615571818318907
 Vector propio 3: [-0.08323655 -1.          0.61493124]
```

7.

```
G = np.array([[1, 1, 1,2],
              [2, 1, 1, 1],
```

```
              [3, 2, 1, 2],
              [2, 1, 1, 4]])

x0 = np.random.rand(4)
TOL = 1e-10
N = 100

Resultado = PowerMethod(G,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(G)
```

```
Valor propio: 6.634534463651611, Vector propio: [0.60704873 0.54782057 0.87261643


 Valor propio 1: 6.634534463633592
 Vector propio 1: [0.60704873 0.54782057 0.87261643 1.        ]

 Valor propio 2: 1.5085633449433251
 Vector propio 2: [-0.17633369 -0.88988542 -1.          0.90010428]

 Valor propio 3: -0.7356415384387976
 Vector propio 3: [ 0.95088434 -0.46661713 -1.         -0.09188862]

 Valor propio 4: -0.4074562701381244
 Vector propio 4: [ 0.48583491 -1.          0.5553643  -0.11957784]
```

8.

```
H = np.array([[1, 2, 1,2],
              [2, 1, 1, 1],
              [3, 2, 1, 2],
              [2, 1, 1, 4]])

x0 = np.random.rand(4)
TOL = 1e-10
N = 100

Resultado = PowerMethod(H,x0,TOL,N)
if Resultado is not None:
    mu, x = Resultado
    print(f'Valor propio: {mu}, Vector propio: {x}')
    print('\n')
    verificador(H)
```

```
Valor propio: 6.827262250123926, Vector propio: [0.6883438  0.56058521 0.88998943


 Valor propio 1: 6.82726225010404
 Vector propio 1: [-0.6883438  -0.56058521 -0.88998943 -1.        ]

 Valor propio 2: 1.7281159082896402
```

```
Vector propio 2: [-0.36944133 -0.73599467 -0.79700677  1.        ]

Valor propio 3: -1.087934923662562
Vector propio 3: [-1.          0.49346694  0.83834525  0.1313279 ]

Valor propio 4: -0.467443234731121
Vector propio 4: [-0.30500876 -0.24339641  1.         -0.03281207]
```

Conclusión

El método presentado es especialmente útil cuando se trabaja con matrices grandes y no negativas. Además, su estructura permite realizar múltiples modificaciones que mejoran la velocidad y/o precisión de la convergencia. De hecho, Burden y Faires (2011) establecen modificaciones útiles a las sucesiones obtenidas originalmente, logrando una mejor aproximación al valor propio.

Es importante resaltar que, además de ser un método matemáticamente sólido, presenta la ventaja de ser computacionalmente eficiente.

Aunque el método funciona para los casos ilustrados anteriormente, puede tener problemas para converger, o no lo hará en alguno de los siguientes casos:

- El valor propio dominante no es único.
- El vector propio asociado al valor propio dominante es ortogonal al vector inicial proporcionado.
- La matriz no es diagonalizable.
- El valor propio dominante es igual a cero.
- Los valores propios son muy cercanos al dominante.

```
import numpy as np
np.set_printoptions(precision=3)
np.set_printoptions(suppress=True)
```

Algoritmo QR

Sea A una matriz $n \times n$. Supongamos que A es no singular, y que sus autovalores cumplen que $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$. Nuestro objetivo es encontrar los autovalores de A. Notemos que si encontramos una matriz $\tilde{A}$ triangular superior, tal que $\tilde{A}$ es similar a A, entonces hemos acabado, pues $\tilde{A}$ tiene los mismos autovalores de A, y como $\tilde{A}$ es triangular superior, sus autovalores aparecen en la diagonal, ya que el polínomio característico es $p(x) = (x - a_1)(x - a_2)\ldots(x - a_n)$, con los a_i los elementos de la diagonal. Con base en lo anterior, enfoquemos nuestros esfuerzos en encontrar una aproximación a una matriz diagonal $\tilde{A}$ similar a A.

Sean $v_1, \ldots, v_n$ los autovectores de A ordenados con respecto al orden decreciente de sus autovalores, y sea $T_k = gen(v_1, \ldots v_k)$. Supongamos que $Q = [Q_1 Q_2]$ es una matriz ortogonal cuyas primeras k columnas (Q_1) forman una base ortogonal para el subespacio T_k. Entonces

$$Q^T AQ = \begin{bmatrix} Q_1^T AQ_1 & Q_1^T AQ_2 \\ Q_2^T AQ_1 & Q_2^T AQ_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}, \quad (1)$$

donde $Q_2^T AQ_1 = 0$ pues T es invariante. La matriz anterior es una matriz triangular superior por bloques. Notemos que si $k = 1$, entonces el bloque A_{11} tiene dimensión 1×1. Se sigue que la primera columna tiene ceros excepto en la primera entrada de $Q^T AQ$, la cual es un autovalor de $Q^T AQ$, y por similaridad es un autovalor de A. Si se cumple simultáneamente que la primera columna forma una base ortogonal para $gen(v_1)$, y que las dos primeras columnas forman una base ortogonal para $gen(v_1, v_2)$, (esto es, $k = 2$), entonces la primera columna tiene ceros excepto en la primera entrada, y la segunda columna tiene ceros excepto en la primera y segunda entrada, pues se sigue respetando la forma (1), donde el bloque A_{11} tiene tamaño 2×2, y por lo tanto el bloque inferior es de ceros. Si se cumplieran simultáneamente los casos $k = 1$, $k = 2$, ... $k = n$, entonces la matriz $Q^T AQ$ sería una matriz triangular superior, pues la forma (1) se está cumpliendo simultáneamente para cada k. Por lo tanto, hemos reducido el problema de encontrar una matriz triangular superior $\tilde{A}$ similar a A, a encontrar una matriz ortogonal Q tal que las primeras k columnas formen una base para $gen(v_1, \ldots, v_k)$, $1 \leq k \leq n$.

Recordemos que el método de potencias consiste en escoger un vector v y aplicarle A repetidamente para formar la sucesión

$$v, Av, A^2v, A^3v, \ldots$$

Asumiendo que se reescala adecuada el vector v después de cada iteración, la sucesión usualmente convergerá a un autovector de A. Veamos rapidamente por qué. Si A tiene autovalores λ_1, λ_2, ..., λ_n tales que $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$ y n autovectores linealmente independientes $v_1, v_2, \ldots, v_n$ correspondientes a los autovalores anteriores, entonces podemos expresar v como

$$v = c_1 v_1 + c_2 v_2 + \ldots + c_n v_n.$$

Luego, al aplicar A repetidamente obtenemos

$$A^m v = c_1 \lambda_1^m v_1 + c_2 \lambda_2^m v_2 + \ldots + c_n \lambda_n^m v_n.$$

Como λ_1 domina a los demás autovalores, la componente en la dirección de v_1 eventualmente domina a las demás componentes, y por lo tanto $A^m v$ converge a un elemento del espacio $< v >$ cuando $m \to \infty$.

Notemos que podemos ver el proceso anterior como como una iteración sobre *subespacios*: empezamos con un subespacio inicial $S =< v >$. Entonces iteramos

$$S, AS, A^2S, A^3S, \ldots$$

y la sucesión anterior "converge" al autoespacio $T = gen(v_1)$. Para volver rigurosa la noción anterior, definamos una métrica en el conjunto de supespacios k - dimensionales de $\mathbb{R}^n$:

$$d(S, T) = sup_{s \in S} inf_{t \in T} ||s - t||_2,$$

donde $||\cdot||_2$ es la norma Euclideana. El resultado principal de la convergencia de la iteración de subespacios es el siguiente:

Teorema 1 Sea v_1, ..., v_n una base de autovectores de A. Sean $T =< v_1, \ldots, v_k >, U =< v_{k+1}, \ldots, v_n >$. Los espacios T y U son llamados *dominante* y *co-dominante*, respectivamente. Sea S un espacio k-dimensional de $\mathbb{R}^n$ tal que $S \cap U = (0)$. Entonces existe una constante C tal que

$$d(A^m S, T) \leq C|\lambda_{k+1}/\lambda_k|^m$$

para todo m. Por lo tanto $A^m S \to T$ linealmente con tasa $|\lambda_{k+1}/\lambda_k|$.

Sea $q_1^0, \ldots, q_n^0$ una base para S. Entonces $A^m(q_1^0), \ldots, A^m(q_n^0)$ es una base para $A^m S$, ya que como A^m es invertible, en cada iteración seguimos obteniendo un conjunto generador y linealmente independiente. Por el teorema anterior, podemos iterar la base $q_1^0, \ldots, q_n^0$ con A para obtener una base para T. Si después de cada iteración ortogonalizamos la base obtenida de izquierda a derecha, entonces la matriz Q_m cuyas columnas son $A^m(q_1^0), \ldots, A^m(q_n^0)$ sería una matriz ortogonal cuyas columnas forman una base para un espacio muy cercano a T. Más aún, la ortogonalización preserva la propiedad de que los primeros k vectores forman una base para $gen(A^m(q_1^0), \ldots, A^m(q_k^0)$ para cada k, y por lo tanto Q_m converge a una matriz Q ortogonal cuyas k primeras columnas forman una base de $T_k = gen(v_1, \ldots, v_k)$ para cada k, que es precisamente lo que estamos buscando.

Por lo tanto, ahora tenemos una idea para obtener el Q buscado: empezamos con una base ortogonal $e_1, \ldots, e_n$ del espacio, y formamos una matriz Q_0 cuyas columnas son los elementos de esta base. A continuación calculamos AQ_0, y buscamos una nueva matriz Q_1 cuyas columnas sean la ortogonalización de las matrices de A. Esto es, buscamos una matriz Q_1 y una matriz R_1 tal que $A = Q_1 R_1$. A continuación hacemos AQ_1, y buscamos matrices $Q_2 R_2$ tales que $AQ_1 = Q_2 R_2$. En general, definimos

$$D_m = AQ_{m-1}$$

,

y buscamos matrices Q_m, R_m tal que $D_m = Q_{m+1} R_{m+1}$. Verificamos la convergencia calculando $Q_m^T A Q_m$, y verificando que los elementos de la triangular inferior estén cerca de 0. Una vez nos hemos acercado lo suficiente a una matriz triangular superior, recolectamos los valores en la diagonal, que corresponden a los autovalores de A. Este es el algoritmo QR.

Podríamos ortogonalizar usando el proceso de Graham-Schmidt. Sin embargo, en la práctica el algoritmo no es estable numericamente. Por lo tanto usaremos matrices de Householder.

Descomposición QR

Sea A una matriz real cuadrada. Entonces A puede ser factorizada como

$$A = QR,$$

donde Q es una matriz ortogonal y R es una matriz triangular superior. A continuación, describiremos un método para hallar la descomposición QR de cualquier matriz cuadrada A. Sea $v \in \mathbb{R}^n$ un vector no nulo. Sea H una matriz $n \times n$ de la forma

$$H = I - 2vv^T.$$

Entonces H es llamada una *matriz de Householder*. Llamamos a v un vector de Householder. Si aplicamos H a un vector x, este es reflejado en el hiperplano $gen(v)^{\perp}$. Es directo verificar que las matrices de Householder son simétricas y ortogonales. La utilidad de las matrices de Householder para la descomposición QR yace en el siguiente hecho: sea $x \in R^n$, y sea $u = x - s||x||e_1$. Si tomamos $v = u/||u||$ la normalización de u, entonces

$$Hx = (I - 2\frac{uu^T}{u^T u})x = ||x||e_1$$

.

Esto es, aplicar H a x lo convierte en un múltiplo de e_1, con la misma magnitud de x. Otra forma de verlo es que hemos dejado en 0 todas las componentes de x excepto por la primera, en la que ha quedado su magnitud. Se sigue que si tomamos una sucesión adecuada de matrices de Householder $H_1, H_2, \ldots, H_{n-1}$ podemos convertir una matriz A en una matriz triangular superior, ya que podemos ir convirtiendo en 0 los elementos debajo de la diagonal. Veamos como podemos hacerlo con un ejemplo. Definamos una matriz A de tamaño 4×4.

```
A = np.array([[1, 0, 0, 0],
              [3, 2, 0, 0],
              [1, 1, 4, 0],
              [1, 2, 1, 7]])
```

Ahora, definamos x como la primera columna de la matriz

```
x = A[:, 0]
x
```

```
array([1, 3, 1, 1])
```

La norma de x es:

```
np.linalg.norm(x)
```

```
3.4641016151377544
```

Calculamos u, v y H como lo describimos anteriormente:

```
u = x - np.linalg.norm(x) * np.eye(1, 4, 0)
v = (1/np.linalg.norm(u)) *  u
H1 = np.eye(4) - 2*v*v.T
H1
```

```
array([[ 0.289,  0.866,  0.289,  0.289],
       [ 0.866, -0.054, -0.351, -0.351],
       [ 0.289, -0.351,  0.883, -0.117],
       [ 0.289, -0.351, -0.117,  0.883]])
```

La matriz anterior es la matriz de Householder H_1. Notemos que si aplicamos H_1 a x obtenemos el siguiente arreglo:

```
H1.dot(x)
```

```
array([ 3.464, -0.   ,  0.   ,  0.   ])
```

Lo que nos indica que la matriz H efectivamente hace lo que queremos: dejar en 0 todas las componentes de x excepto por la primera, en la que queda su magnitud. Además, el producto HA es

```
H1.dot(A)
```

```
array([[ 3.464, -1.732,  1.443,  2.021],
       [-0.   , -0.891, -1.757, -2.46 ],
       [ 0.   ,  1.703,  3.414, -0.82 ],
       [-0.   ,  2.703,  0.414,  6.18 ]])
```

Es decir, hemos logrado poner los ceros que queríamos en la primera columna de A. Ahora, queremos poner ceros en la siguiente columna. Esto es, queremos hacerle el proceso anterior al vector $x = \begin{bmatrix} -2.029 \\ -0.343 \\ 0.657 \end{bmatrix}$. Notemos que este vector ahora tiene 3 componentes, pues lo que queremos es tranformar a A en una matriz triangular superior, por lo que no nos interesa lo que le suceda a la primera componente. Luego, debemos operar ahora sobre la submatriz $(n-1) \times (n-1)$ que resulta de omitir la primera fila y la segunda columna. En nuestro ejemplo, ahora queremos operar sobre la siguiente matriz:

```
sub_matrix1 = H1.dot(A)[1:, 1:]
sub_matrix1
```

```
array([[-0.891, -1.757, -2.46 ],
       [ 1.703,  3.414, -0.82 ],
       [ 2.703,  0.414,  6.18 ]])
```

Como habíamos hecho antes, ahora definamos x_2 como la primera columna de la matriz anterior, y calculemos la matriz H_2 correspondiente.

```
x2  = sub_matrix1[:, 0]
u2 = x2 - np.linalg.norm(x2) * np.eye(1, 3, 0)
v2 = (1/np.linalg.norm(u2)) *  u2
H2 = np.eye(3) - 2*v2*v2.T
H2
```

```
array([[-0.269,  0.513,  0.815],
       [ 0.513,  0.792, -0.33 ],
       [ 0.815, -0.33 ,  0.477]])
```

Recordemos que las matrices A y $H_1 A$ son matrices 4×4. Sin embargo nuestra H_2 actual es una matriz 3×3, pues esta en principio solo debía operar en una submatriz de HA. Arreglamos este problema introduciendo un padding a la matriz H_2, de tal manera que la primera fila y la primera columna de H_2 sean el vector e_1. Esto no afecta la ortogonalidad ni la simetría de H_2.

```
e1 = np.array([1, 0, 0, 0])

H_padded = np.zeros((4, 4))

H_padded[0, :] = e1
H_padded[:, 0] = e1

H_padded[1:, 1:] = H2

H2 = H_padded
H2
```

```
array([[ 1.   ,  0.   ,  0.   ,  0.   ],
       [ 0.   , -0.269,  0.513,  0.815],
       [ 0.   ,  0.513,  0.792, -0.33 ],
       [ 0.   ,  0.815, -0.33 ,  0.477]])
```

Ahora calculamos $H_2 H_1 A$:

```
H2.dot(H1).dot(A)
```

```
array([[ 3.464, -1.732,  1.443,  2.021],
       [-0.   ,  3.317,  2.563,  5.276],
       [ 0.   ,  0.   ,  1.666, -3.951],
       [-0.   ,  0.   , -2.361,  1.21 ]])
```

Siguiendo el mismo proceso anterior, obtenemos una matriz H_3

```
sub_matrix2 = H2.dot(H1).dot(A)[2:, 2:]
x3  = sub_matrix2[:, 0]
u3 = x3 - np.linalg.norm(x3) * np.eye(1, 2, 0)
v3 = (1/np.linalg.norm(u3)) *  u3
H3 = np.eye(2) - 2*v3*v3.T

#Le hacemos el padding
e1 = np.array([1, 0, 0, 0])
e2 = np.array([0, 1, 0, 0])

H_padded = np.zeros((4, 4))

H_padded[0, :] = e1
H_padded[:, 0] = e1

H_padded[1, :] = e2
H_padded[:, 1] = e2

H_padded[2:, 2:] = H3

H3 = H_padded
H3
```

```
array([[ 1.   ,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  1.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.577, -0.817],
       [ 0.   ,  0.   , -0.817, -0.577]])
```

Notemos que por construcción $H_3 H_2 H_1 A$ debe ser una matriz triangular superior R. En efecto:

```
R = H3.dot(H2).dot(H1).dot(A)
R
```

```
array([[ 3.464, -1.732,  1.443,  2.021],
       [-0.   ,  3.317,  2.563,  5.276],
       [ 0.   , -0.   ,  2.889, -3.267],
       [ 0.   , -0.   ,  0.   ,  2.53 ]])
```

Como H_1, H_2 y H_3 son ortonogales, si definimos $Q = (H_3 H_2 H_1)^T$, entonces $Q = H_1^T H_2^T H_3^T = H_1^{-1} H_2^{-1} H_3^{-1}$, y por lo tanto $Q(H_3 H_2 H_1 A) = QR = A$. Esto es, hemos encontrado la descomposición QR de A. Lo comprobamos en el ejemplo:

```
Q = (H3.dot(H2).dot(H1)).T
print('Matriz A: ')
print(A)

print('\nMatriz QR: ')
print(Q.dot(R))
```

```
Matriz A:
[[ 1  0  0  0]
 [ 3 -3  0  0]
 [ 1  1  4  0]
 [ 1  2  1  7]]

Matriz QR:
[[ 1.  0.  0.  0.]
 [ 3. -3.  0.  0.]
 [ 1.  1.  4. -0.]
 [ 1.  2.  1.  7.]]
```

El anterior era solo un ejemplo ilustrativo. En la práctica se realizan cosas extras para llegar más eficientemente a la descomposición. Por ejemplo, para almacenar las matrices de Householder H_i no necesitamos guardar toda la matriz, sino solo el vector de Householder v.

Recordemos que nuestro objetivo es encontrar los autovalores de A. A continuación, realizaremos unas observaciones que nos llevaran intuitivamente (pero también rigurosamente) al algoritmo QR.

Análisis del error

Por el teorema de la convergencia de subespacios, $A^m S$ converge a T con tasa $C|\lambda_{k+1}/\lambda_k|$. Se sigue que el algoritmo converge a una matriz triangular superior a esta tasa. Es posible modificar el algoritmo (con shifts) para lograr una convergencia mucho más rápida a los autovalores.

Implementación

A continuación presentamos implementaciones para calcular la descomposición QR, y otro para calcular la matriz triangular superior $\tilde{A}$ que permite recolectar los autovalores de A.

householder_qr()

Es una implementación generalizada del proceso anterior para encontrar la descomposición QR a través de matrices de Householder.

```
def householder_qr(A):
    """

    Parametros
    A: Matriz a descomponer

    Returna:
    tupla: (Q, R) donde Q es matriz ortogonal y R es matriz triangular superior
    """

    A = np.array(A, dtype=float)
    m, n = A.shape
    Q = np.eye(m)
    R = A.copy()

    # Itera cada columna
    for j in range(min(m-1, n)):
        # Extraemos la columna en la que estamos trbajando
        x = R[j:, j]

        # Calculamos el vector de householder
        e1 = np.zeros_like(x)
        e1[0] = 1

        alpha = -np.sign(x[0]) * np.linalg.norm(x) # Multiplicamos por el signo para estabilidad
                                                   # numérica
        u = x - alpha * e1
        v = u / np.linalg.norm(u)
        H = np.eye(m)
        H[j:, j:] -= 2.0 * np.outer(v, v)
        R = H @ R
        Q = Q @ H.T # Vamos acumulando Q

    # Ponemos ceros en la triangular inferior para asegurar que es triangular superior
    for i in range(m):
        for j in range(i):
            if j < n:
                R[i, j] = 0.0

    return Q, R
```

```
A
```

```
array([[ 1,  0,  0,  0],
       [ 3, -3,  0,  0],
       [ 1,  1,  4,  0],
       [ 1,  2,  1,  7]])
```

```
Q, R = householder_qr(A)
Q @ R
```

```
array([[ 1., -0., -0., -0.],
       [ 3., -3.,  0.,  0.],
       [ 1.,  1.,  4., -0.],
       [ 1.,  2.,  1.,  7.]])
```

qr_eigenvalue_algorithm()

El siguiente código calcula el algoritmo QR, mediante el procedimiento descrito anteriormente. Además, se hace uso de la siguiente observación:

Para $A_1 = A_1 I$, esta tiene una descomposición

$$A_1 = Q_2' R_2'.$$

Luego $Q_2'^T A_1 = R_2'$. Por definición, $A_1 = Q_1^T A Q_1$, $D_1 = A Q_1$, y D_1 tiene una descomposición $Q_2 R_2$. Se sigue que

$$Q_2 R_2 = D_1 = A Q_1 = Q_1 A_1 Q_1^T Q_1 = Q_1 A_1 = Q_1 Q_2' R_2'.$$

Ahora, es un hecho que la descomposición QR es única si A se escoge invertible, y R con diagonal positiva. Por lo tanto $Q_2 = Q_1 Q_2', R_2 = R_2'$ y así

$$A_2 = Q_2^T A Q_2 = Q_2'^T Q_1^T A Q_1 Q_2' = Q_2^T A_1 Q_2' = R_2' Q_2'.$$

Ahora para calcular A_3, basta calcular la descomposición $Q_3 R_3$ de A_2, y hacer $A_3 = R_3 Q_3$. La siguiente implementación usa esta idea para calcular matrices triangulares A_m.

```
def qr_eigenvalue_algorithm(A, max_iter=1000, tolerance=1e-10):

    """
    Computa autovalores usando el algoritmo QR

    Parameters:
    A (list): Matriz a la que se hallarán los autovalores
    max_iter (int): Número máximo de iteraciones
    tolerance (float): Tolerancia de la convergencia

    Returna:
    np.array: eigenvalues
    """

    A = np.array(A, dtype=float)
    n = A.shape[0]
    V = np.eye(n)
    H = A.copy()

    for _ in range(max_iter):

        Q, R = householder_qr(H)
        H = R @ Q
        V = V @ Q

        off_diag_norm = np.sum(np.abs(np.tril(H, -1)))

        if off_diag_norm < tolerance:
            break

    eigenvalues = np.diag(H)

    return eigenvalues
```

```
qr_eigenvalue_algorithm(A)
```

```
array([ 7.,  4., -3.,  1.])
```

Ejemplo

Calculamos los autovalores de las siguientes matrices dadas en la tarea:

```
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(f'Los autovalores de A son: {qr_eigenvalue_algorithm(A)}\n')
print(f'Los autovalores de B son: {qr_eigenvalue_algorithm(B)}\n')
print(f'Los autovalores de C son: {qr_eigenvalue_algorithm(C)}\n')
print(f'Los autovalores de D son: {qr_eigenvalue_algorithm(D)}\n')
print(f'Los autovalores de E son: {qr_eigenvalue_algorithm(E)}\n')
print(f'Los autovalores de F son: {qr_eigenvalue_algorithm(F)}\n')
print(f'Los autovalores de G son: {qr_eigenvalue_algorithm(G)}\n')
print(f'Los autovalores de H son: {qr_eigenvalue_algorithm(H)}\n')
```

```
Los autovalores de A son: [5. 1.]

Los autovalores de B son: [6. 1.]

Los autovalores de C son: [5. 1.]

Los autovalores de D son: [ 4.507  0.778 -0.285]

Los autovalores de E son: [ 4.049 -0.692 -0.357]

Los autovalores de F son: [ 4.125 -0.762  0.637]

Los autovalores de G son: [ 6.635  1.509 -0.736 -0.407]

Los autovalores de H son: [ 6.827  1.728 -1.088 -0.467]
```