

Practica 1

González Borja, Miguel
Illescas Arizti, Rodrigo
Meyer Mañón, Juan Carlos
Rodríguez Orozco, Alejandro

11 / Marzo / 2018

Introducción

A continuación los ejercicios de la práctica. Todos fueron realizados en Python 3.6 utilizando los paquetes *numpy* y *scipy.linalg*. También se incluye un cuaderno de IPython para cada ejercicio dentro de la carpeta Ejercicios, junto con los scripts *.py*. Cada uno de estos utiliza los siguientes imports:

```
In [1]: import sys
        sys.path.append('../')
        from IPython.display import Latex
        import latexStrings as ls
        import numpy as np
        import scipy.linalg as linear
        import eigenvalues as ev
```

Se puede encontrar un repositorio del proyecto [aquí](#).

Ejercicio 1

Tenemos la matriz A y el vector q_0 definidos como:

```
In [1]: A = np.array([[1,1,2],[-1,9,3],[0,-1,3]])
        q = np.array([1,1,1])
```

$$A = \begin{pmatrix} 1 & 1 & 2 \\ -1 & 9 & 3 \\ 0 & -1 & 3 \end{pmatrix} \quad \vec{q}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Queremos calcular 10 iteraciones del metodo de la potencia. Esto nos da como resultado:

```
In [2]: [q10, l10 ,iterations]=ev.powerMethod(A,q,1e-6,10)
```

Numero de Iteraciones: 10

$$\vec{q}_{10} = \begin{pmatrix} 0.083519 \\ 0.979588 \\ -0.182845 \end{pmatrix} \quad \lambda = 8.35525106702442$$

Despues comparemos los resultados con los valores 'exactos' calculados por el paquete *scipy.linalg*:

```
In [3]: [L,V] = linear.eig(A)
```

$$\vec{\lambda} = \begin{pmatrix} 8.354545 \\ 1.224672 \\ 3.420784 \end{pmatrix} \quad V = \begin{pmatrix} 0.083444 & -0.992728 & 0.515311 \\ 0.979576 & -0.104882 & -0.332386 \\ -0.182943 & -0.059078 & 0.789921 \end{pmatrix}$$

Vemos que en efecto el método de la potencia calculo el eigenvector dominante de la matriz. Esto se debe a que se cumplen las condiciones del método, es decir:

1. A tiene un eigenvalor dominante (8.3545)
2. Nuestro vector inicial q_0 puede ser escrito como combinación lineal de los eigenvectores de A con coeficientes todos distintos de 0

Ahora, tomemos el eigenvector asociado al eigenvalos dominante, dado por:

```
In [4]: v=V[:,0]
```

$$\vec{v} = \begin{pmatrix} 0.083444 \\ 0.979576 \\ -0.182943 \end{pmatrix}$$

Y con esto queremos calcular las razones de convergencia en cada paso de la iteración, dadas por:

$$\tilde{r} = \{r_i\}, \quad r_i = \frac{\|q_i - v\|_\infty}{\|q_{i-1} - v\|_\infty}, \quad i \in \{1, 2, \dots, 10\}$$

```
In [5]: ratios=[]
prevq=q
for i in range(1,11):
    [currentq,_,_] = ev.powerMethod(A,q,1e-6,i)
    ratio = linear.norm(currentq-v,np.inf)/linear.norm(prevq-v,np.inf)
    ratios.append(ratio)
    prevq = currentq
```

$$\tilde{r} = \{0.297033, 0.382353, 0.390998, 0.400885, 0.405792, 0.407929, 0.408825, 0.409194, 0.409346, 0.409409\}$$

Observemos que en efecto, las razones de cada iteracion rapidamente al valor teorico, dado por:

$$r = \left| \frac{\lambda_2}{\lambda_1} \right| = 0.40945181373495726$$

Ejercicio 2.1

Utilizando la matriz A y el vector q_0 del ejercicio anterior definidos como sigue:

```
In [1]: A = np.array([[1,1,2],[-1,9,3],[0,-1,3]])  
        q = np.array([1,1,1])
```

$$A = \begin{pmatrix} 1 & 1 & 2 \\ -1 & 9 & 3 \\ 0 & -1 & 3 \end{pmatrix} \quad \vec{q}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Se calcularán 10 iteraciones del método de la potencia con shift ρ_1 y ρ_2 y donde \vec{q}_{10} es el vector que se aproxima al eigenvector y σ_{10} el eigenvalor aproximado después de 10 iteraciones. Para $\rho_1 = 0$, que es simplemente aplicar el método de la potencia inversa, se obtiene:

```
In [2]: [q10, l10, iterations]=ev.inversePowerShift(A,q,0,1e-6,10)
```

Numero de Iteraciones: 10

$$\vec{q}_{10} = \begin{pmatrix} 0.992719 \\ 0.104174 \\ 0.060466 \end{pmatrix} \quad \sigma_{10} = 1.2267894261411831$$

Comparemos los resultados anteriores con los valores "exactos" calculados por el paquete *scipy.linalg*:

```
In [3]: [L,V] = linear.eig(A)  
        [L,V] = ev.pairSort(L,V)
```

$$\vec{\lambda} = \begin{pmatrix} 8.354545 \\ 3.420784 \\ 1.224672 \end{pmatrix} \quad V = \begin{pmatrix} 0.083444 & 0.515311 & -0.992728 \\ 0.979576 & -0.332386 & -0.104882 \\ -0.182943 & 0.789921 & -0.059078 \end{pmatrix}$$

$\vec{\lambda}$ es el vector que contiene a los tres eigenvalores "exactos" de la matriz A ordenados por magnitud y las columnas de la matriz V son los eigenvectores "exactos" de A .

Se observa que $\sigma_{10} = 1.22678942614$ tiene dos decimales iguales a $\lambda_3 = 1.224672$, la tercera entrada de $\vec{\lambda}$. El método de la potencia inversa converge teóricamente al menor eigenpar de A bajo las siguientes condiciones:

1. A tiene un menor eigenvalor, es decir $|\lambda_1| > |\lambda_2| > |\lambda_3|$ (En este caso 1.2246)
2. \vec{q}_0 puede ser escrito como combinación lineal de los eigenvectores de A con los coeficientes de \vec{v}_2 y \vec{v}_3 distintos de 0. En este caso, \vec{q}_0 claramente no es ortogonal a los vectores de la matriz V , por lo tanto ninguno de los coeficientes de la combinación lineal será 0.

Ahora comparemos \vec{v}_3 , el eigenvector asignado a λ_3 , con \vec{q}_{10} .

```
In [4]: v=V[:,2]
```

$$\vec{v}_3 = \begin{pmatrix} -0.992728 \\ -0.104882 \\ -0.059078 \end{pmatrix} \quad q_{10} = \begin{pmatrix} 0.992719 \\ 0.104174 \\ 0.060466 \end{pmatrix}$$

Observamos que \vec{q}_{10} es aproximadamente $-\vec{v}_3$. Esto se debe a que el método de la potencia inversa puede converger a $\pm \vec{v}_3$. Entonces de ahora en adelante nos referiremos a \vec{v}_3 por $-\vec{v}_3$:

In [5]: `v=-v`

$$\vec{v}_3 = \begin{pmatrix} 0.992728 \\ 0.104882 \\ 0.059078 \end{pmatrix}$$

Calculamos las razones de convergencia en cada paso de la iteración, dadas por:

$$\tilde{r} = \{r_i\}, \quad r_i = \frac{\|q_i - v\|_\infty}{\|q_{i-1} - v\|_\infty}, \quad i \in \{1, 2, \dots, 10\}$$

```
In [6]: ratios=[]
        prevq=q
        for i in range(1,10):
            [currentq,_,_] = ev.inversePowerShift(A,q,0,1e-6,i)
            ratio = linear.norm(currentq-v,np.inf)/linear.norm(prevq-v,np.inf)
            ratios.append(ratio)
            prevq = currentq
```

$$\tilde{r} = \{0.752001, 0.966084, 0.853759, 0.678343, 0.495284, 0.404372, 0.373393, 0.363308, 0.359880\}$$

Ahora veamos la razón de convergencia teórica, dada por:

$$r = \left| \frac{\lambda_3}{\lambda_2} \right| = 0.35800909831776$$

Se observa que, en efecto, $r_{10} = 0.359880$ es aproximadamente la razón teórica $r = 0.35800909831776$

Ejercicio 2.2

Para la misma matriz A y vector q_0 definidos anteriormente, se realizarán las mismas pruebas para un shift $\rho_2 = 3.3$. Se encontrarán nuevos: vector \vec{q}_{10} , la aproximación de eigenvector, y σ_{10} la aproximación del eigenvalor.

```
In [1]: [q10, l10 ,iterations]=ev.inversePowerShift(A,q,3.3,1e-6,10)
```

Numero de Iteraciones: 4

$$\vec{q}_{10} = \begin{pmatrix} 0.515311 \\ -0.332385 \\ 0.789921 \end{pmatrix} \quad \sigma_{10} = 3.420782623785237$$

La primera observación es que el proceso sólo hizo 4 iteraciones, esto quiere decir que el método alcanzó el criterio de error relativo sin necesidad de hacer las 10 iteraciones. Comparemos los resultados anteriores con los valores "exactos" calculados por el paquete *scipy.linalg* calculados anteriormente

$$\vec{\lambda} = \begin{pmatrix} 8.354545 \\ 3.420784 \\ 1.224672 \end{pmatrix} \quad V = \begin{pmatrix} 0.083444 & 0.515311 & -0.992728 \\ 0.979576 & -0.332386 & -0.104882 \\ -0.182943 & 0.789921 & -0.059078 \end{pmatrix}$$

El vector \vec{q}_{10} es aproximadamente al eigenvector "exacto" \vec{v}_2 , segunda columna de la matriz V . Además, el valor σ_{10} tiene 5 cifras decimales iguales al eigenvalor "exacto" λ_2 .

Observamos lo siguiente:

$$|\lambda_2 - \rho_2| < |\lambda_3 - \rho_2| < |\lambda_1 - \rho_2| \Rightarrow |\lambda_2 - \rho_2|^{-1} > |\lambda_3 - \rho_2|^{-1} > |\lambda_1 - \rho_2|^{-1}$$

Con un razonamiento similar al ejercicio anterior, al tener la desigualdad anterior, el método de la potencia inversa con shift $\rho_2 = 3.3$ cumple una de las hipótesis y el converge al valor $(\lambda_2 - \rho_2)^{-1}$ y el eigenvector asignado \vec{v}_2 . Luego con un simple despeje obtenemos λ_2 y con ello el eigenpar de A (λ_2, \vec{v}_2) .

Nuevamente, el vector inicial \vec{q}_0 no es ortogonal a ninguno de los vectores de la matriz V , por lo tanto \vec{q}_0 se puede escribir como combinación lineal de los vectores de V con coeficientes todos distintos de 0, cumpliendo la segunda hipótesis del método, justificando así la convergencia.

Calculamos las razones de convergencia en cada paso de la iteración, dadas por:

$$\tilde{r} = \{r_i\}, \quad r_i = \frac{\|q_i - v\|_\infty}{\|q_{i-1} - v\|_\infty}, \quad i \in \{1, 2, \dots, 10\}$$

```
In [2]: ratios=[]
        v=V[:,1]
        prevq=q
        for i in range(1,11):
            [currentq,_,_] = ev.inversePowerShift(A,q,3.3,0,i)
            ratio = linear.norm(currentq-v,np.inf)/linear.norm(prevq-v,np.inf)
            ratios.append(ratio)
            prevq = currentq
```

$$\tilde{r} = \{0.014461, 0.024879, 0.020834, 0.032431, 0.021112, \\ 0.085590, 0.050552, 0.061824, 0.056628, 0.062165\}$$

Ahora veamos la razón de convergencia teórica, dada por:

$$r = \left| \frac{\lambda_2 - \rho_2}{\lambda_3 - \rho_2} \right| = 0.05819972269618957$$

Observamos que las razones r_i comienzan a oscilar alrededor de $r = 0.05819972269618957$ (razón teórica) desde $i = 7$. Esto se debe a que como el método converge rápidamente y con más iteraciones la aproximación se vuelve prácticamente igual al eigenpar buscado. De hecho, si aumentamos las razones calculadas, veremos que:

```
In [3]: for i in range(11,21):
        [currentq,_,_] = ev.inversePowerShift(A,q,3.3,0,i)
        ratio = linear.norm(currentq-v,np.inf)/linear.norm(prevq-v,np.inf)
        ratios.append(ratio)
        prevq = currentq
```

$$\tilde{r}_{10-20} = \{0.062165, 0.172414, 1.100000, 1.045455, 0.913043, 1.095238, \\ 0.913043, 1.095238, 0.913043, 1.095238, 0.913043\}$$

La razones de la práctica empiezan a acercarse a 1. Esto se debe a que $\vec{q}_i - \vec{v}$ es casi igual a $\vec{q}_{i-1} - \vec{v}$ por lo que la razón se acerca a 1.

Ejercicio 3

Tomando A y q_0 como los definimos anteriormente:

```
In [1]: A = np.array([[1,1,2],[-1,9,3],[0,-1,3]])
        q = np.array([1,1,1])
```

$$A = \begin{pmatrix} 1 & 1 & 2 \\ -1 & 9 & 3 \\ 0 & -1 & 3 \end{pmatrix} \quad \vec{q}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Se calcularán 10 iteraciones del método de la potencia con shift $\rho = 3.6$, donde q_{10} es el vector que se aproxima al eigenvector y σ_{10} el eigenvalor aproximado después de 10 iteraciones

```
In [2]: [q10, l10, iterations]=ev.inversePowerShift(A,q,3.6,1e-15)
```

Numero de Iteraciones: 12

$$\vec{q}_{10} = \begin{pmatrix} 0.5153107267324357 \\ -0.33238561122815846 \\ 0.7899206671324485 \end{pmatrix} \quad \sigma_{10} = 3.4207835356869145$$

El método requiere 12 iteraciones para cumplir con el criterio de error relativo igual a 10^{-15} , es decir

$$\frac{\|A\vec{q}_{10} - \sigma_{10}\vec{q}_{10}\|}{\|A\vec{q}_{10}\|} \leq 10^{-15}$$

Comparemos esto con los valores exactos, dados por:

```
In [3]: [L,V] = linear.eig(A)
```

$$\vec{v} = \begin{pmatrix} 0.515310726732435 \\ -0.3323856112281598 \\ 0.7899206671324484 \end{pmatrix} \quad \lambda = 3.420783535686916$$

En efecto, vemos que los valores tienen alrededor de 15 cifras correctas con respecto a los calculados por *scipy*

¿Por qué converge tan rápido el metodo? Veamos todos los eigenvalores de la matriz con el shift $\rho = 3.6$:

```
In [4]: [L,V] = linear.eig(A-3.6*np.identity(len(A)))
```

$$\vec{\lambda} = \begin{pmatrix} 4.754545 \\ -2.375328 \\ -0.179216 \end{pmatrix} \quad V = \begin{pmatrix} 0.083444 & -0.992728 & 0.515311 \\ 0.979576 & -0.104882 & -0.332386 \\ -0.182943 & -0.059078 & 0.789921 \end{pmatrix}$$

El método converge "rápido" porque $|\lambda_3 - \rho_3| = |\lambda_3 - 3.6| = 0.179216464313$ es cercano a 0 y la razón de convergencia teórica r es tal que:

$$r = \left| \frac{\lambda_3 - \rho_3}{\lambda_2 - \rho_2} \right| = 0.07544913221790368$$

Ejercicio 4

Definimos la matriz A como en el ejercicio 1:

```
In [1]: A = np.array([[1,1,2],[-1,9,3],[0,-1,3]])
```

$$A = \begin{pmatrix} 1 & 1 & 2 \\ -1 & 9 & 3 \\ 0 & -1 & 3 \end{pmatrix}$$

Los eigenvalores y eigenvectores exactos a los que queremos llegar con nuestro método son los siguientes:

```
In [2]: [eigenvalues,eigenvectors]=linear.eig(A)
[eigenvalues,eigenvectors]=ev.pairSort(eigenvalues,eigenvectors)
```

$$\vec{\lambda} = \begin{pmatrix} 8.35454483516155 \\ 3.420783535686916 \\ 1.2246716291515263 \end{pmatrix} \quad V = \begin{pmatrix} 0.083444 & 0.515311 & -0.992728 \\ 0.979576 & -0.332386 & -0.104882 \\ -0.182943 & 0.789921 & -0.059078 \end{pmatrix}$$

El método aplicado a A con vector inicial q_0 devuelve la aproximación de un eigenpar (λ_j, v_j) con $j \in \{1,2,3\}$ en donde λ_j es el eigenvalor de la matriz A más cercano al primer shift de Rayleigh, definido como:

$$\rho_j = \frac{q_j^H A q_j}{||q_j||^2}$$

Pues, en esencia, el metodo de la potencia inversa con shift de Rayleigh es una mejora al metodo con shift estatico, tomando como primer shift el shit de Rayleigh y mejorandolo en cada iteracion. Esto lleva a que la convergencia se da hacia:

$$\{\lambda_j, \vec{v}_j\}, \quad \left| \frac{1}{\lambda_j - \rho_0} \right| > \left| \frac{1}{\lambda_i - \rho_0} \right| \quad \forall i \neq j \quad i, j \in \{1,2,3\}$$

Dado que la convergencia del metodo ahora depende del vector inicial, tratemos de buscar vectores distintos que nos lleven a todos los eigenvalores. Tomemos la base canónica como nuestros primeros 3 vectores iniciales:

```
In [3]: q1 = np.array([0,1,0])
        q2 = np.array([0,0,1])
        q3 = np.array([1,0,0])
```

$$\vec{q}_0^1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{q}_0^2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \vec{q}_0^3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Aplicamos el método de la potencia inversa con shift de Rayleigh a la matriz A tomando a cada uno de los 3 vectores previamente definidos como vector inicial, obteniendo:

```
In [4]: [w1,l1,i1] = ev.inversePowerRayleigh(A,q1)
```

$$\vec{w}_1 = \begin{pmatrix} 0.083444 \\ 0.979576 \\ -0.182943 \end{pmatrix} \quad \sigma_1 = 8.354544835161578$$

```
In [5]: [w2,l2,i2] = ev.inversePowerRayleigh(A,q2)
```

$$\vec{w}_2 = \begin{pmatrix} 0.515311 \\ -0.332386 \\ 0.789921 \end{pmatrix} \quad \sigma_2 = 3.4207835406851625$$

```
In [6]: [w3,l3,i3] = ev.inversePowerRayleigh(A,q3)
```

$$\vec{w}_3 = \begin{pmatrix} 0.992728 \\ 0.104882 \\ 0.059078 \end{pmatrix} \quad \sigma_3 = 1.2246716248594658$$

En efecto, vemos que los vectores tomados convergen a todos los distintos eigenvalores de A. A que se debe esto?

En nuestro ejemplo podemos ver que al tomar la base canónica como nuestros vectores iniciales estamos tomando a los elementos de la diagonal como nuestros shifts de Rayleigh iniciales pues $\vec{e}_j^t A \vec{e}_j = a_{jj}$. Observemos mas detalladamente esto:

Tomando $\vec{q}_0^1 = (0, 1, 0)$ como vector inicial, nuestro primer shift es $\rho_0^1 = a_{22} = 9$, y se cumple que:

$$\left| \frac{1}{8.354544... - 9} \right| > \left| \frac{1}{1.224671... - 9} \right| > \left| \frac{1}{3.420783... - 9} \right| \Rightarrow$$

$$\left| \frac{1}{\lambda_1 - 9} \right| > \left| \frac{1}{\lambda_3 - 9} \right| > \left| \frac{1}{\lambda_2 - 9} \right| \Rightarrow \left| \frac{1}{\lambda_1 - \rho_1} \right| > \left| \frac{1}{\lambda_i - \rho_1} \right| \quad \forall i \neq 1$$

por lo que nuestro método de la potencia inversa con shift de Raleigh tomando tomando \vec{q}_0^1 como vector inicial converge a:

$$\vec{w}_1 = \begin{pmatrix} 0.083444 \\ 0.979576 \\ -0.182943 \end{pmatrix} \quad \sigma_1 = 8.354544835161578$$

Tomando $q_0^2 = (0, 0, 1)$ como vector inicial, nuestro shift inicial es $\rho_0^2 = a_{33} = 3$, y se cumple que

$$\left| \frac{1}{3.420783... - 3} \right| > \left| \frac{1}{1.224671... - 3} \right| > \left| \frac{1}{8.354544..... - 3} \right| \Rightarrow$$

$$\left| \frac{1}{\lambda_2 - 3} \right| > \left| \frac{1}{\lambda_3 - 3} \right| > \left| \frac{1}{\lambda_1 - 3} \right| \Rightarrow \left| \frac{1}{\lambda_2 - \rho_2} \right| > \left| \frac{1}{\lambda_i - \rho_2} \right| \quad \forall i \neq 2$$

por lo que nuestro método de la potencia inversa con shift de Raleigh tomando tomando q_0^2 como vector inicial converge a:

$$\vec{w}_2 = \begin{pmatrix} 0.515311 \\ -0.332386 \\ 0.789921 \end{pmatrix} \quad \sigma_2 = 3.4207835406851625$$

Tomando $q_0^3 = (1, 0, 0)$ como vector inicial, nuestro shift inicial es $\rho_0^3 = a_{11} = 1$, y se cumple que:

$$\left| \frac{1}{1.224671... - 1} \right| > \left| \frac{1}{3.420783... - 1} \right| > \left| \frac{1}{8.354544..... - 1} \right| \Rightarrow$$

$$\left| \frac{1}{\lambda_3 - 1} \right| > \left| \frac{1}{\lambda_2 - 1} \right| > \left| \frac{1}{\lambda_1 - 1} \right| \Rightarrow \left| \frac{1}{\lambda_3 - \rho_3} \right| > \left| \frac{1}{\lambda_i - \rho_3} \right| \quad \forall i \neq 3$$

por lo que nuestro método de la potencia inversa con shift de Raleigh tomando tomando q_0^3 converge a:

$$\vec{w}_3 = \begin{pmatrix} 0.992728 \\ 0.104882 \\ 0.059078 \end{pmatrix} \quad \sigma_3 = 1.2246716248594658$$

Ahora observemos que la convergencia del método de la potencia inversa con shift de Rayleigh es cuadrática. Primero veamos la aproximación de λ_j en cada iteración del método para cada q_0^i , y luego restando el eigenvalor correspondiente para ver la diferencia:

```
In [7]: aprox=[]
        for i in range(1,7):
            [_,sigmai,_] = ev.inversePowerRayleigh(A,q1,1e-14,i)
            aprox.append(sigmai)
```

$$\tilde{\sigma}_1 = \{8.436241610738254, 8.355776480878252, 8.35454512345497,$$

$$8.354544835161578, 8.354544835161558, 8.354544835161558\}$$

$$|\tilde{\sigma}_1 - \lambda_1| = \{0.08169677557670418, 0.00123164571670209, 2.882934211356769e - 07,$$

$$2.842170943040401e - 14, 8.881784197001252e - 15, 8.881784197001252e - 15\}$$

```
In [8]: aprox=[]
        for i in range(1,7):
            [_,sigmai,_] = ev.inversePowerRayleigh(A,q2,1e-15,i)
            aprox.append(sigmai)
```

$$\begin{aligned}\tilde{\sigma}_2 &= \{3.2018348623853212, 3.434406721090111, 3.420894064163872, \\ &\quad 3.4207835406851625, 3.420783535686915, 3.420783535686915\} \\ |\tilde{\sigma}_2 - \lambda_2| &= \{0.21894867330159462, 0.01362318540319496, 0.0001105284769562509, \\ &\quad 4.998246705412157e - 09, 8.881784197001252e - 16, 8.881784197001252e - 16\}\end{aligned}$$

```
In [9]: aprox=[]
        for i in range(1,7):
            [_,sigmai,_] = ev.inversePowerRayleigh(A,q3,1e-14,i)
            aprox.append(sigmai)
```

$$\begin{aligned}\tilde{\sigma}_3 &= \{1.2076502732240437, 1.2245699277493174, 1.2246716248594658, \\ &\quad 1.2246716291515263, 1.2246716291515263, 1.2246716291515263\} \\ |\tilde{\sigma}_3 - \lambda_3| &= \{0.01702135592748255, 0.0001017014022088869, 4.292060484800686e - 09, \\ &\quad 0.0, 0.0, 0.0\}\end{aligned}$$

Podemos ver que en los 3 casos el número de cifras correctas de la aproximación de σ_j con $j \in \{1, 2, 3\}$ se duplica con cada iteración hasta llegar al valor 'exacto' representable en la computadora

Ejercicio 5

Definimos la matriz A como sigue:

```
In [1]: A = np.array([[1, -4, -6], [-12, -8, -6], [11, 10, 10]])
```

$$A = \begin{pmatrix} 1 & -4 & -6 \\ -12 & -8 & -6 \\ 11 & 10 & 10 \end{pmatrix}$$

Observamos que A es invertible pues $\det(A) = -44 \neq 0$ y por lo tanto A^{-1} existe, permitiendo hacer el metodo de la potencia inversa.

Ahora veamos los eigenvalores y eigenvectores de A utilizando *scipy.linalg*:

```
In [2]: [eigenvalues,eigenvectors]=linear.eig(A)
        [eigenvalues,eigenvectors]=ev.pairSort(eigenvalues,eigenvectors)
```

$$\vec{\lambda} = \begin{pmatrix} 2.881554 \\ 2.881554 \\ -2.763109 \end{pmatrix} \quad V = \begin{pmatrix} -0.386685 & -0.386685 & -0.063674 \\ 0.685967 & 0.685967 & 0.811470 \\ -0.470695 & -0.470695 & -0.580915 \end{pmatrix}$$

Si A tiene eigenvalores $\{\lambda_1, \lambda_2, \lambda_3\}$, entonces sabemos que A^{-1} tiene eigenvalores $\{\frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \frac{1}{\lambda_3}\}$. Entonces, observamos que:

$$\left| \frac{1}{-2.763109} \right| > \left| \frac{1}{2.881554} \right| \Rightarrow \left| \frac{1}{\lambda_3} \right| > \left| \frac{1}{\lambda_2} \right| = \left| \frac{1}{\lambda_1} \right| \Rightarrow \left| \frac{1}{\lambda_3} \right| > \left| \frac{1}{\lambda_i} \right| \quad \forall i \neq 3$$

Por lo tanto existe el eigenvalor dominante $\frac{1}{\lambda_3} = \frac{1}{-2.763109}$ de A^{-1} y el método de la potencia inversa debería, con un vector inicial \vec{q}_0 apropiado, converge a:

```
In [3]: v3 = np.array(eigenvectors[:,2])
```

$$\vec{v}_3 = \begin{pmatrix} -0.063674 \\ 0.811470 \\ -0.580915 \end{pmatrix} \quad \lambda_3 = -2.763108507220637$$

Ahora tomemos \vec{q}_0 que cumpla la segunda hipotesis. En este caso vemos que $\vec{q}_0 = (1, 1, 1)$ lo cumple, pues al no ser ortogonal a ninguno de los eigenvectores, los coeficientes de su combinacion lineal son todos distintos de 0. Comprobemos que el metodo sirve corriendo la función `inversePower` con A y \vec{q}_0 .

```
In [4]: q = np.array([1,1,1])
        [w,l,i] = ev.inversePower(A,q)
```

Numero de iteraciones: 45

$$\vec{w}_3 = \begin{pmatrix} 0.063674 \\ -0.811470 \\ 0.580915 \end{pmatrix} \quad \sigma_3 = -2.7631087076453653$$

Podemos ver que el método si funciona pues aproxima de manera correcta el eigenpar (\vec{v}_3, λ_3) siendo $\lambda_3 = -2.7663109$ el eigenvalor con norma más pequeña y su eigenvector normal asociado

Ejercicio 6

Considerémos la matriz `fiedler(25)`. Le aplicaremos el método de QR simple con un máximo de 2000 iteraciones, y el de QR con shift dinámico con un máximo de 20 iteraciones. En ambos casos la tolerancia será de $1e-10$. Despues compararemos con los eigenvalores calculados por `scipy.linalg`

```
In [1]: QR_eig, QR_iter = simpleQR(fiedler(25), maxIter=2000, tol=1e-10)
        shQR_eig, shQR_iter = shiftQR(fiedler(25), maxIter=20, tol=1e-10)
        actual = linear.eig(fiedler(25))[0].real
        err_abs_QR = abs(QR_eig - actual)
        err_abs_shQR = abs(shQR_eig - actual)
        err_rel_QR = abs((QR_eig - actual) / actual)
        err_rel_shQR = abs((shQR_eig - actual) / actual)
```

Comparemos los errores absolutos y relativos:

Error absoluto:

Error absoluto QR Simple:

```
[2.27373675e-13 4.26325641e-14 1.42108547e-13 5.32907052e-15
 3.37507799e-14 0.00000000e+00 1.15463195e-14 5.32907052e-15
 4.44089210e-15 8.88178420e-16 2.22044605e-15 3.10862447e-15
 0.00000000e+00 5.55111512e-16 1.99840144e-15 2.10942375e-15
 1.88737914e-15 2.10942375e-15 6.66133815e-15 5.32907052e-15
 5.55111512e-15 5.21804822e-15 1.88737914e-15 5.99405459e-03
 5.99405459e-03]
```

Error absoluto QR Shift:

```
[1.37143843e-07 1.37144099e-07 1.20792265e-13 2.88221891e-09
3.78781628e-09 1.13858407e-06 3.78995473e-07 1.09361161e-05
1.06148077e-08 1.72948440e-05 7.22173454e-06 2.95257115e-06
1.31078567e-05 6.32192284e-06 5.27355225e-06 7.93329257e-07
2.22044605e-16 1.11022302e-15 3.33066907e-16 5.55111512e-16
3.33066907e-16 1.11022302e-16 3.33066907e-16 5.99405459e-03
5.99405459e-03]
```

Error relativo:

Error relativo QR Simple:

```
[1.04910088e-15 3.36170502e-16 3.55003718e-15 3.74226059e-16
3.97076283e-15 0.00000000e+00 3.06526811e-15 1.93219313e-15
2.03944571e-15 5.10010441e-16 1.52496224e-15 2.52612633e-15
0.00000000e+00 5.89967253e-16 2.36765448e-15 2.76127153e-15
2.68750931e-15 3.23970951e-15 1.08996871e-14 9.21379574e-15
1.00386608e-14 9.79065872e-15 3.64179474e-15 1.19406927e-02
1.17997949e-02]
```

Error relativo QR Shift:

```
[6.32780931e-10 1.08142218e-09 3.01753160e-15 2.02399540e-10
4.45634742e-10 2.17450208e-07 1.00614117e-07 3.96517336e-06
4.87476920e-09 9.93105756e-06 4.95975685e-06 2.39931449e-06
1.22401079e-05 6.71887966e-06 6.24796866e-06 1.03848148e-06
3.16177566e-16 1.70511027e-15 5.44984353e-16 9.59770390e-16
6.02319648e-16 2.08311888e-16 6.42669660e-16 1.19406927e-02
1.17997949e-02]
```

Para ver de manera mas clara los errores, compararemos con los valores reales utilizando la norma 2 de los vectores de error:

QR simple: $\epsilon_{abs} = 0.00847687330084258$

QR con shift dinámico: $\epsilon_{abs} = 0.008476915844806674$

QR simple: $\epsilon_{rel} = 0.016787355420100413$

QR con shift dinámico: $\epsilon_{rel} = 0.01678736673339094$

Nótese que en ambos casos el método de QR simple logra una mejor aproximación. Pero vemos que...

```
In [2]: print('Iteraciones QR simple: '+str(QR_iter))
        print('Iteraciones QR shift dinamico: '+str(shQR_iter))
```

Iteraciones QR simple: 1571

Iteraciones QR shift dinamico: 20

El método de QR simple hace casi 80 veces el número de iteraciones que el QR con shift! y aún así ambas aproximaciones son bastante buenas.

Veamos que pasa si limitamos el método de QR simplemente a 20 iteraciones:

```
In [3]: QR_eig, QR_iter = simpleQR(fiedler(25), maxIter=20, tol=1e-10)
        err_abs_QR = abs(QR_eig - actual)
        err_rel_QR = abs((QR_eig - actual) / actual)
```

QR simple (20 iteraciones): $\epsilon_{abs} = 0.1402924694363105$

QR con shift dinámico: $\epsilon_{abs} = 0.008476915844806674$

QR simple (20 iteraciones): $\epsilon_{rel} = 0.16155309127644163$

QR con shift dinámico: $\epsilon_{rel} = 0.01678736673339094$

Nótese que el error absoluto ahora es mucho mayor, y el error relativo es 10 mayor! Si no limitamos el número de iteraciones, observamos que:

```
In [4]: QR_eig, QR_iter = simpleQR(fiedler(25), maxIter=10000, tol=1e-10)
        shQR_eig, shQR_iter = shiftQR(fiedler(25), tol=1e-10)
        err_rel_QR = abs((QR_eig - actual) / actual)
        err_rel_shQR = abs((shQR_eig - actual) / actual)
```

QR simple: $\epsilon_{rel} = 0.016787355420100413$

QR con shift dinámico: $\epsilon_{rel} = 0.016787355420088787$

Debido a que ambos métodos cumplen el criterio de tolerancia 10^{-10} , sabemos que ambos errores relativos son prácticamente iguales. Pero observemos el número de iteraciones:

```
In [5]: print('Iteraciones QR simple:'+str(QR_iter))
        print('Iteraciones QR shift dinamico:'+str(shQR_iter))
```

Iteraciones QR simple:1571

Iteraciones QR shift dinamico:38

$$\frac{i_{QRsimple}}{i_{QRshift}} = 41.3421052631579$$

El método de QR simple hace más de 40 veces el número de iteraciones!

Ejercicio 7

Queremos probar el método SVD. Para esto, primero cargamos una imagen original:



Luego separamos los valores RGB en 3 matrices:

```
In [1]: img = img.astype('float')
        red = np.array(img[:, :, 0])
        green = np.array(img[:, :, 1])
        blue = np.array(img[:, :, 2])
```

Hagamos una prueba sin reducir el rango de la matriz:

```
In [2]: tolerance = 0

In [3]: u1, s1, v1 = SVD(red, tolerance)
        u2, s2, v2 = SVD(green, tolerance)
        u3, s3, v3 = SVD(blue, tolerance)
```

En este caso, vemos que el rango original es:

```
In [4]: len(s1), len(s2), len(s3)

Out[4]: (115, 117, 117)
```



En efecto, el proceso no modifiko la imagen.

Ahora apliquemos el SVD economizado cortando los valores singulares menores a 100:

```
In [5]: tolerance = 100

In [6]: u1, s1, v1 = SVD(red, tolerance)
        u2, s2, v2 = SVD(green, tolerance)
        u3, s3, v3 = SVD(blue, tolerance)

In [7]: len(s1), len(s2), len(s3)

Out[7]: (72, 65, 65)
```

Observamos que el rango de la matriz es casi la mitad del original. Pero como afecta esto a la imagen?



No hubo perdida de detalle, salvo el tono blanco del fondo
 Probemos un orden de magnitud mayor en la tolerancia...

```
In [8]: tolerance = 1000
```

```
In [9]: u1, s1, v1 = SVD(red,tolerance)
        u2, s2, v2 = SVD(green,tolerance)
        u3, s3, v3 = SVD(blue,tolerance)
```

```
In [10]: len(s1), len(s2), len(s3)
```

```
Out[10]: (16, 11, 11)
```

Los rangos de las matrices son ahora un decimo de su valor original. Como sigue la imagen?



A pesar de que existe una evidente perdida de detalle, aun es posible reconocer la imagen original

Ahora veamos que pasa cuando reducimos a solo los 3 valores singulares mayores de cada matriz

```
In [11]: n = 3
```

```
In [12]: u1, s1, v1 = SVD(red, n = n)
        u2, s2, v2 = SVD(green, n = n)
        u3, s3, v3 = SVD(blue, n = n)
```

```
In [13]: len(s1), len(s2), len(s3)
```

```
Out[13]: (3, 3, 3)
```



La imagen original esta basicamente perdida, pero aun es posible reconocer la forma.
Que tal si tomamos solo el valor singular dominante?

```
In [14]: n = 1
```

```
In [15]: u1, s1, v1 = SVD(red, n = n)  
         u2, s2, v2 = SVD(green, n = n)  
         u3, s3, v3 = SVD(blue, n = n)
```

```
In [16]: len(s1), len(s2), len(s3)
```

```
Out[16]: (1, 1, 1)
```



La imagen se pierde totalmente, y solo permanecen los colores dominantes en sus posiciones relativas.