Modelación Numérica Examen Final

Iván Vega Gutiérrez

2 de diciembre de 2021

1 Ejercicio 1

Consideremos la siguiente matriz pentadiagonal $A \in M_{10}$:

$$A = \begin{pmatrix} 10 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 10 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 10 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 10 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 10 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 10 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 10 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 10 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 10 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 10 \end{pmatrix}$$

Y sean $D, M, N \in M_{10}$, las siguientes matrices.

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \end{pmatrix} N = \begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$D = \begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Notemos que la matriz A se puede descomponer como $A = D + M + N$.

Consideremos el sistema **Ax = b** y los siguientes métodos iterativos.

1. $(M + D)x^{(k+1)} = -Nx^k + b$,
2. $Dx^{(k+1)} = -(M + N)x^k + b$

3. $(M+N)x^{(k+1)} = -Dx^k + b$

Notemos que los métodos iterativos 1), 2) y 3), se pueden escribir como las siguientes ecuaciones, respectivamente:

$$x^{k+1} = -(M+D)^{-1}Nx^k + (M+D)^{-1}b \tag{1}$$

$$x^{k+1} = -D^{-1}(M+N)x^k + D^{-1}b \tag{2}$$

$$x^{k+1} = -(M+N)^{-1}Dx^k + (M+N)^{-1}b \tag{3}$$

Luego, las matrices de iteración para (1), (2) y (3) son:

$$M_1 = (M+D)^{-1}N$$

$$M_2 = D^{-1}(M+N)$$

$$M_3 = (M+N)^{-1}D$$

Por un resultado, sabemos que un método iterativo converge si y solo si el radio espectral de la matriz de iteración M satisface que $\rho(M) < 1$.

Analizando los métodos iterativos 1, 2 y 3, tenemos los siguientes resultados

[1]:
```python
import numpy as np

def RadioEspectral(A):
    """
    Esta función calcula el radio espectral de una matriz A
    """
    # Hallamos los valores propios
    valores, vectores = np.linalg.eig(A)
    # Hallamos el valor propio más grande
    rho = max(abs(valores))
    return rho
```

[2]:
```python
import numpy as np

# Ejercicio 1.

# Creamos la matriz D
D = 8*np.identity(10)
# Creamos la matriz M
M = np.identity(10)
for i in range(len(M)-2):
    for j in range(i,i+1):
        M[i,j+1] = -1
        M[i,j+2] = -1
M[len(M)-2, len(M)-1] = -1
# Creamos la matriz N
N = np.transpose(M)
# Definimos las matrices de iteración para cada método iterativo
M1 = np.dot(np.linalg.inv(M+D), N)
```

```
M2 = np.dot(np.linalg.inv(D), M+N)
M3 = np.dot(np.linalg.inv(M+N), D)
# Hallamos el radio espectral de cada matriz de iteración
rho1 = RadioEspectral(M1)
rho2 = RadioEspectral(M2)
rho3 = RadioEspectral(M3)
#Impresión de resultados
print("El radio espectral para el primer método iterativo es {}".format(rho1))
print("El radio espectral para el segundo método iterativo es {}".format(rho2))
print("El radio espectral para el tercer método iterativo es {}".format(rho3))
```

```
El radio espectral para el primer método iterativo es 0.14503954977535374
El radio espectral para el segundo método iterativo es 0.5000000000000001
El radio espectral para el tercer método iterativo es 12.287023474955681
```

De lo anterior, se concluye que los métodos 1 y 2 convergen, mientras que el método 3 no converge ya que su radio espectral es muy grande.

2 Ejercicio 2

Consideremos el sistema lineal **Ax = b**, donde

$$A = \begin{pmatrix} 62 & 24 & 1 & 8 & 15 \\ 23 & 50 & 7 & 14 & 16 \\ 4 & 6 & 58 & 20 & 22 \\ 10 & 12 & 19 & 66 & 3 \\ 11 & 18 & 25 & 2 & 54 \end{pmatrix} \qquad \text{y} \qquad b = \begin{pmatrix} 110 \\ 110 \\ 110 \\ 110 \\ 110 \end{pmatrix}$$

Resolvamos el sistema mediante los métodos Jacobi y Gauss-Seidel. Asimismo verifiquemos si el método estacionario de Richardson se puede aplicar cuando se tienen los precondicionadores $P = I$ y $P = D$, donde D es la parte diagonal de la matriz A.

[3]:
```
def MatrizJacobi(A):
    """
    Esta función calcula la matriz de iteración del método
    de Jacobi, definido por la descomposición M=D y N=D-A
    donde D=(a11, . . ., ann)
    """
    n = len(A)
    D = np.zeros((n,n))
    for i in range(n):
        D[i][i] = A[i][i]
    M = D
    N = D - A
    J = np.dot(np.linalg.inv(M),N)
    return J
```

[4]:
```
def MatrizGaussSeidel(A):
    """
```

```
    Esta función calcula la matriz de iteración del método
    de Gauss-Seidel, definido por la descomposición M=D-E
    y N=F
    """
    n = len(A)
    D = np.zeros((n,n))
    E = np.zeros((n,n))
    F = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if i == j:
                D[i][j] = A[i][j]
            elif i > j :
                E[i][j] = - A[i][j]
            else:
                F[i][j] = - A[i][j]
    M = D - E
    N = np.copy(F)
    G = np.dot(np.linalg.inv(M), N)
    return G
```

[5]:
```
def MatrizRichardson(A, P, alpha, optimo = True):
    """
    Esta función calcula la matriz de iteración del método de
    Richardson:  R = I - P^(-1) A

    Argumentos:
    A: matriz del sistema Ax = b que se desea resolver
    P: matriz no singular, precondicionador de A
    alpha: número real positivo
    optimo: valor booleano  para usar el alpha óptimo

    Salida:
    alpha: valor óptimo de alpha si óptimo es verdero
    R: Matriz de iteración
    """
    n = len(A)
    # Hallamoas los valores porpios de la matriz P^(-1)A
    invPA = np.dot(np.linalg.inv(P), A)
    valores, vectores = np.linalg.eig(invPA)
    lambdamin = min(valores)
    lambdamax = max(valores)
    if optimo == True:
        alphaopt = 2/(lambdamin + lambdamax)
        alpha = alphaopt
    I = np.identity(n)
    R = I - alpha * np.dot(np.linalg.inv(P), A)
```

```
    return (alpha, R)
```

[6]:
```
def Jacobi(A, b, x0, tol, iterMax):
    """
    Esta función caula la solución al sistema Ax=b
    mediante el método de Jacobi.

    Agumentos:
    A: (matriz cuadrada): Matriz cuadrada invertible
    b: vector unidimensional de valores independientes
    x0: Solución inicial
    tol: Tolerancia
    iterMax: Número máximo de iteraciones

    Salida
    (Iter, x ): Tupla que contiene el número de iteraciones
                y la solución aproximada  del sistema
    """
    # Creamos el vector solución x y definimos las iteraciones Iter
    n = len(A)
    Iter = 1
    x = np.empty(n)
    # El siguiente ciclo termina si se supera el número máximo de iteraciones
    while Iter <= iterMax:
        for i in range(n):
            suma = 0
            for j in range(n):
                if j != i:
                    suma += A[i][j] * x0[j]
            # Calculamos la solución en la iteración Iter
            x[i] = (b[i] - suma)/ A[i][i]
        # Calculamos el error cometido en la iteración
        error = np.linalg.norm(b - np.dot(A,x)) / np.linalg.norm(b)
        # Verificamos si el error es menor que la tolerancia dada
        if error < tol:
            return(Iter,x)
        # Incrementamos el número de iteraciones
        Iter += 1
        # Actualizamos la solución
        x0 = x.copy()
    return (Iter, x)
```

[7]:
```
def GaussSeidel(A, b, x0, tol, iterMax):
    """
    Esta función calcula la solución al sistema Ax = b mediante
    el método de Gauss-Seidel
```

```
    Argumentos:
    A : Matriz cuadrada del sistema
    b : vector unidimensional de valores independientes
    x0 : Solución inicial
    tol: Tolerancia
    iterMax: Número máximo de iteraciones

    Salida:
    (Iter, x ): Tupla que contiene el número de iteraciones
                y la solución aproximada  del sistema
    """
    # Creamos el vector solución x
    n = len(A)
    x = np.empty(n)
    # Inicializamos el número de iteraciones
    Iter = 1
    # Realizamos las iteraciones necesarias para
    while Iter <= iterMax:
        for i in range(n):
            suma1, suma2 = 0,0
            for j in range(i):
                suma1 += A[i][j] * x[j]
            for j in range(i + 1, n):
                suma2 += A[i][j] * x[j]
            # Calculamos la solución en la iteración Iter
            x[i] = (b[i] - suma1 - suma2)/A[i][i]
        # Calculamos el error cometido en la iteración
        error = np.linalg.norm(b - np.dot(A,x)) / np.linalg.norm(b)
        #Verificamos si el error es menor que la tolerancia dada
        if error < tol:
            return(Iter, x)
        # Actualizamos el número de iteraciones
        Iter += 1
        # Actualizamos la solución
        x0 = x.copy()
    return(Iter, x)
```

[8]:
```
def Richardson(A, b, P, x0, tol, iterMax, alpha, optimo = True):
    """
    Esta función calcula la solución al sistema Ax = b mediante
    el método de Richardson
    """
    # Inicializamos el contador de iteraciones k
    k = 0
    n = len(A)
    normab = np.linalg.norm(b)
    # Creamos el vector solución
```

```
    x = np.empty(n)
    # Hallamos el valor alpha óptimo
    if optimo:
        alpha = MatrizRichardson(A, P, alpha, optimo = True)[0]
    # Calculamos el residual inicial
    r = b - np.dot(A,x0)
    # Hallamos la inversa del precondicionador
    invP = np.linalg.inv(P)
    while k <= iterMax:
        y = np.dot(invP, r)
        # Calculamos el valor de la solución
        x = x0 + alpha * y
        # Actualizamos el valor del residual de la késima iteración
        r = r - alpha * np.dot(A,y)
        # Calculamos el error
        error = np.linalg.norm(b - np.dot(A,x))/normab
        if error < tol:
            return (k,x)
        # Actualizamos la solución
        k += 1
        x0 = x.copy()
    return (k, x)
```

[9]:
```
import numpy as np

# Ejercicio 2.

# Definimos la matriz A
A = np.array([
    [62,24,1,8,15],
    [23,50,7,14,16],
    [4,6,58,20,22],
    [10,12,19,66,3],
    [11,18,25,2,54]
    ])
# Definimos el vector b
b = np.array([110,110,110,110,110])

# Inciso (1)

# Definimos la solución inicial, tolerancia y número máximo de iteraciones
x0 = np.zeros(len(A))
tol = 1e-15
iterMax = 100
# Calculamos el radio espectral de cada método
J = MatrizJacobi(A)
G = MatrizGaussSeidel(A)
```

```
rhoj = RadioEspectral(J)
rhogs = RadioEspectral(G)
# Hallamos la solución mediante el método de Jacobi y Gauss-Seidel
(Iter1, SolJacobi) = Jacobi(A, b, x0, tol, iterMax)
(Iter2, SolGS) = GaussSeidel(A, b, x0, tol, iterMax)
# Impresión de resultados
print("Para el método de Jacobi se tienen los siguientes resultados:")
print("Radio espectral = {}".format(rhoj))
print("Número de iteraciones = {}".format(Iter1))
print("Solución = {}".format(SolJacobi))
print("\n")
print("Para el método de Gauss-Seidel se tienen los siguientes resultados:")
print("Radio espectral = {}".format(rhogs))
print("Número de iteraciones = {}".format(Iter2))
print("Solución = {}".format(SolGS))
print("\n")

# Inciso (2)
print("Resultados cuando se usa el método de Richardson \n")

n = len(A)

# Si P = I
P = np.identity(n)
# Calculamos el radio espectral asociado al alpha óptimo
alphaopt1, R1 = MatrizRichardson(A, P, 1, optimo = True)
rho1 = RadioEspectral(R1)
# Hallamos la solución del sistema
iter1, sol1 =  Richardson(A, b, P, x0, tol, iterMax, 1, optimo = True)
# Impresión de resultados
print("Si P = I se tiene que:")
print("El alpha óptimo es {}".format(alphaopt1))
print("El radio espectral mínimo es {}".format(rho1))
print("La solución es {} en {} iteraciones".format(sol1, iter1))
print("\n")

# Si P = D
D = np.zeros((n,n))
for i in range(n):
    D[i][i] = A[i][i]
P = D
# Calculamos el radio espectral asociado al alpha óptimo
alphaopt2, R2 = MatrizRichardson(A, P, 1, optimo = True)
rho2 = RadioEspectral(R2)
# Hallamos la solución del sistema
iter2, sol2 =  Richardson(A, b, P, x0, tol, iterMax, 1, optimo = True)
# Impresión de resultados
```

```
print("Si P = D se tiene que:")
print("El alpha óptimo es {}".format(alphaopt2))
print("El radio espectral mínimo es {}".format(rho2))
print("La solución es {} en {} iteraciones".format(sol1, iter2))
print("\n")
```

```
Para el método de Jacobi se tienen los siguientes resultados:
Radio espectral = 0.927984216764368
Número de iteraciones = 101
Solución = [0.99947837 0.99931314 0.99946658 0.99958324 0.99935602]


Para el método de Gauss-Seidel se tienen los siguientes resultados:
Radio espectral = 0.30657833775344734
Número de iteraciones = 23
Solución = [1. 1. 1. 1. 1.]


Resultados cuando se usa el método de Richardson

Si P = I se tiene que:
El alpha óptimo es 0.01495626401089898
El radio espectral mínimo es 0.6451890411988882
La solución es [1. 1. 1. 1. 1.] en 78 iteraciones


Si P = D se tiene que:
El alpha óptimo es 0.8509810205973402
El radio espectral mínimo es 0.6406779764777074
La solución es [1. 1. 1. 1. 1.] en 77 iteraciones
```

De lo anterior cabe resaltar que tanto el método de Jacobi como el método de Gauss-Seidel convergen, sin embargo, es necesario hacer notar la importancia que tiene el radio espectral sobre la convergencia de un método iterativo. A pesar de que en ambos métodos el radio espectral es menor a 1, el radio espectral del método de Jacobi es tres veces mayor al radio espectral del método de Gauss-Seidel, lo que que se traduce en un mayor esfuerzo computacional, lo cual se ve reflejado en el número de iteraciones que realiza el método. Además, la solución que arroja el método de Jacobi es muy cercana a la solución real, sin embargo, el método de Gauss-Seidel da la solución exacta.

Por otro lado, cuando se aplica el método de Richardson y se utilizan los precondicionadores P = I y P = D, los radios espectrales son menores a uno y son muy parecidos, asimismo, el número de iteraciones diferen en una unidad, esto reafirma la relación directa entre el radio espectral sobre la convergencia de los métodos iterativos.

3 Ejercicio 3

Sea A una matriz con valores propios $\lambda_1, \cdots, \lambda_n$ no no necesariamente distintos que satisfacen

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

El valor propio con magnitud más grande es llamado el valor propio dominante de la matriz A.

Supongamos que los vectores propios asociados $v_1, \cdots, v_n$ son linealmente independientes y forman una base para $\mathbb{R}^n$.

Sea $x^0 \neq 0 \in \mathbb{R}^n$. Pongamos el vector x^0 como combinación lineal de los vectores propios de A,

$$x^0 = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n \tag{4}$$

Veamos que si $\alpha_1 = 0$ y $|\lambda_2| > |\lambda_3| \geq \cdots \geq |\lambda_n|$, entonces el método de la potencia converge a (λ_2, v_2).

En efecto. Sea x^n la sucesión de vectores definidos por las siguientes iteraciones

$$\begin{aligned}
x^1 = Ax^0 &= \alpha_1 Av_1 + \alpha_2 Av_2 + \cdots + \alpha_n v_n \\
&= \alpha_2 Av_2 + \alpha_3 Av_3 + \cdots + \alpha_n Av_n \\
&= \alpha_2 \lambda_2 v_1 + \alpha_3 \lambda_3 v_3 + \cdots + \alpha_n \lambda_n v_n \\
x^2 = Ax^1 &= A^2 x^0 \\
&= \alpha_2 A^2 v_2 + \alpha_3 A^3 v_3 + \cdots + \alpha_n A^2 v_n \\
&= \alpha_2 \lambda_2^2 v_2 + \alpha_3 \lambda_3^2 v_3 + \cdots + \alpha_n \lambda_n^2 v_n \\
&\vdots \\
x^m = Ax^{m-1} &= A^m x^0 \\
&= \alpha_2 A^m v_2 + \alpha_3 A^m v_3 + \cdots + \alpha_n A^m v_n \\
&= \alpha_2 \lambda_2^m v_2 + \alpha_3 \lambda_3^m v_3 + \cdots + \alpha_n \lambda_n^m v_n
\end{aligned}$$

Luego,

$$x^m = \lambda_2^m \Big(\alpha_2 v_2 + \alpha_3 \Big(\frac{\lambda_3}{\lambda_2} \Big)^m v_3 + \cdots + \alpha_n \Big(\frac{\lambda_n}{\lambda_2} \Big)^m \Big)$$

Por hipótesis tenemos que λ_2 es el valor propio más grande, así, $\left| \frac{\lambda_j}{\lambda_2} \right| < 1$ para cada $j = \{3, \cdots, n\}$, en consecuencia

$$\text{Si } m \to \infty \text{ entonces } \left| \frac{\lambda_j}{\lambda_2} \right| \to 0$$

Entonces,

$$x^m = \lambda_2^m \alpha_2 v2$$

Por lo tanto, se concluye que la solución converge al valor propio y vector propio (λ_2, v_2)

Por otro lado, sea

$$A = \begin{pmatrix} 1 & -1 & 2 \\ -2 & 0 & 5 \\ 6 & -3 & 6 \end{pmatrix}$$

Apliquemos el método de la potencia para hallar el valor propio más grande tomando $q^{(0)} = 1/\sqrt{3}$ y $q^{(0)} = w^{(0)}/||w^{(0)}||_2$, donde $w^{(0)} = (1/3)x_2 - (2/3)x_3$

[10]:
```
def MetPotencia(A, x0, tol, itermax):
    """
    Esta función calcula el valor propio dominante
    (valor propio con magnitud más grande) de la matriz A
    mediante el método de la potencia

    Argumentos:
    A: Matriz cuadrada de la que sea desea hallar el valor propio
    x0: Valor inicial
    tol: Tolerancia
    itermax:  Número máximo de iteraciones

    Salida:
    (Lambda, k): Tupla que contiene el valor proipio buscado
                 y el número de iteraciones realizados
    """
    k = 1
    n = len(A)
    x =  x0 / np.linalg.norm(x0)
    y = np.dot(A,x0)
    while k <= itermax:
        x = y/np.linalg.norm(y)
        y = np.dot(A,x)
        Lambda = np.dot(np.transpose(x), y)
        error = np.linalg.norm(np.dot(A,x) - Lambda*x)
        if error < tol:
            return (Lambda, k)
        k += 1
    return (Lambda, k)
```

[11]:
```
import math

# Ejercicio 3.

# Definimos la matriz A

A = np.array([[1, -1, 2],
              [-2, 0, 5],
              [6, -3, 6]])
n = len(A)
tol = 1e-15
itermax = 100

# Cuando x0 =  1 / sqrt(3)
x0 = (1/math.sqrt(3)) * np.ones((3,))
Lambda, Iter = MetPotencia(A, x0, tol, itermax)
```

```
print("Tomando en cuenta q0 = 1/sqrt(3), se obtiene")
print("El valor propio más grande es {}".format(Lambda))
print("Número de iteraciones {}".format(Iter))
print("\n")

# Si x0 = w0 / norm(w0)
valores, vectores = np.linalg.eig(A)
v2 = vectores[1]
v3 = vectores[2]
w0 = (1/3)*v2 - (2/3)*v3
x0 = w0 / np.linalg.norm(w0)
Lambda, Iter = MetPotencia(A, x0, tol, itermax)
print("Tomando en cuenta q0 = w0/norm(w0), se obtiene")
print("El valor propio más grande es {}".format(Lambda))
print("Número de iteraciones {}".format(Iter))
```

```
Tomando en cuenta q0 = 1/sqrt(3), se obtiene
El valor propio más grande es 5.000000000000002
Número de iteraciones 66


Tomando en cuenta q0 = w0/norm(w0), se obtiene
El valor propio más grande es 5.000000000000001
Número de iteraciones 69
```

Observemos que numéricamente en ambos casos se llega al valor propio $\lambda_1 = 5$, a pesar de que en la segunda parte $q^{(0)}$ se define como una combinación lineal de los vectors x_2 y x_3, análiticamente se debió haber llegado al siguien valor propio más grande.