Eigensolvers y Solver Iterativos

Descripción

Se realizó la implementación de los siguientes algoritmos:

1. Algoritmo de Rayleigh para calcular valores y vectores propios.
2. Algoritmo de Iteración en el Subespacio para calcular valores y vectores propios.
3. Algoritmo QR para calcular valores y vectores propios.
4. Algoritmo de Gradiente Conjugado para solucionar sistemas de ecuaciones.

Algoritmo de Rayleigh para calcular valores y vectores propios.

El Algoritmo de Rayleigh, se basa en una modificación al algoritmo de Potencia al usar el cociente de Rayleigh para reducir el residuo de una aproximación a un valor y vector propio, donde el residuo r está dado por:

$$r = \|Ax - \alpha x\| \tag{1}$$

A es una matriz cuadrada simétrica, x es es un vector propio y λ es un valor propio. Se define el Cociente de Rayleigh de la forma

$$\alpha = \frac{x^t Ax}{x^t x} \tag{2}$$

Notemos que al usar vectores propios unitarios el cociente de (2) es igual a 1. Así, sea x_0 la primer aproximación a nuestro vector propio luego:

$$x_{k+1} = Ax_k \tag{3}$$

Algoritmo

```
n -> Tamaño de la matriz
a[n][n] -> Matriz a
xprev -> Vector de tamaño n con nuestra aproximación al vector propio
xnext -> Vector de tamaño n
MAX_ITER -> Número máximo de iteraciones
lambda -> Valor propio
last_lambda -> Ultimo Valor propio calculado
epsilon -> Valor mínimo para considerar un cero

while MAX_ITER --:
    normalizar(xprev)
    xnext = A * xprev
    lambda = xprev * A * xprev
    xprev = xnext
    if (abs(last_lambda) - abs(lambda)) <= epsilon:
        break
    last_lambda = lambda

xnext <- Contiene el vector propio sin normalizar
```

```
20 lambda <- Contiene valor propio
```

Ejemplo de prueba

Entrada

```
4 4
6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
-1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
-1.0000000000 2.0000000000 8.0000000000 -1.0000000000
4.0000000000 -1.0000000000 -1.0000000000 -5.0000000000

```

```
4 1
17
-7
19
-14
```

Salida

```
Eigen Value: -10.371044
Eigen vector:
-0.016879 -0.983335 0.097845 -0.152294
```

Observaciones y mejoras

Los algoritmos de potencia con deflación implementados previamente mientras más valores y vectores propios se obtenían más era el error acumulado y los resultados ya no eran correctos, se procedió a tomar unos de los resultados arrojados por el Algoritmo de Potencia con deflación usando la matriz $M\_BIG.txt$ y se usaron como entrada para este algoritmo, a pesar de no obtener el valor y vector propio que se esperaba (dado el orden de mayor a menor), se obtuvieron valores y vectores propios cercanos (cercanos en sentido de orden de mayor a menor), esto dado que los valores son tan pequeños y a los errores de aproximación del primer algoritmo es difícil encontrar una entrada que sea lo mayor similar al valor y vector propio esperado.

Algoritmo de Iteración en el Subespacio para calcular valores y vectores propios.

Este algoritmo nos proporciona los valores y vectores propios de mayor módulo iterando en un espacio más pequeño que contiene dichos vectores.

La idea comienza en iniciar con $\{q_k\}$ vectores linealmente independientes con k igual al número de vectores de interés y en cada paso se mejoran los vectores $\{q_k\}$ tal que se aproximen a los vectores propios necesitados.

Sea A una matriz de tamaño nxn y k vectores y valores propios necesitados. Proponemos una matriz P de tamaño nxk dónde cada columna es una propuesta a cada vector característico. Luego:

$$P_0 = Q_0 R_0 \tag{4}$$

donde Q_0 y R_0 son de tamaño nxk y kxk respectivamente

$$\Lambda_1 = Q_0^t A Q_0 \tag{5}$$

$$P_1 = A Q_0 \tag{6}$$

Sucesivamente:

$$\Lambda_{k+1} = Q_k^t A Q_k \tag{7}$$

$$P_{k+1} = A Q_k \tag{8}$$

Converge cuando la matriz Λ de valores propios se vuelve diagonal.

Algoritmo

```
n -> Tamaño de la matriz
neigen -> Número de eigen vetores y valores a calcular
a[n][n] -> Matriz a
Lambda[neigen][neigen] -> Matrix de eigen valores
p[n][neigen] -> Matriz de eigen vectores iniciada en ramdom
q[n][p] -> Matriz con para la factorización QR
r[n][p] -> Matriz con para la factorización QR

while(true):
    [q,r] = qr_decomposition(p);
    lambda = transpose(q) * a * q
    p = a * q
    if (is_diagonal(a)) break

normalize_by_colums(p);

lambda <- Contiene los eigen valores
p <- Contiene los eigen vectores normalizados
```

Ejemplo de prueba

Entrada

```
4 4
6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
-1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
-1.0000000000 2.0000000000 8.0000000000 -1.0000000000
4.0000000000 -1.0000000000 -1.0000000000 -5.0000000000
```

```
Quantity of eigen values to compute [1 to 4]: 4
```

Salida

```
Eigen Values
-10.3710438740 -0.0000000000 0.0000000000 0.0000000000
-0.0000000000 9.2688664888 -0.0000000000 -0.0000000000
0.0000000000 -0.0000000000 6.3568139827 0.0000000000
0.0000000000 -0.0000000000 0.0000000000 -6.2546365975

Eigen Vectors
0.0168782711 -0.5486543461 0.7754166702 -0.3121258080
0.9833352317 0.1225974752 0.0115027924 -0.1337511379
-0.0978440936 0.7976799725 0.5917778752 0.0627067875
0.1522940555 -0.2183000895 0.2199900445 0.9384859998
```

Observaciones y mejoras

El criterio de parada para el algoritmo es que la matriz Λ converja a una matriz diagonal, sin embargo por definición cuando solo se calcula un valor y vector propio dicha matriz ya es diagonal y el algoritmo terminaría en la primer iteración. La solución propuesta es agregar un mínimo de iteraciones que puede ser observado en la implementación.

Durante las pruebas con la matriz $M\_BIG.txt$ se observó que para aquellos valores propios que tiene multiplicidad mayor a 1 los vectores propios obtenidos no coincidían. Así que solo se obtuvieron los vectores propios correctos para valores propios con multiplicidad 1.

Algoritmo QR para calcular valores y vectores propios.

El algoritmo QR nos ayuda a calcular los valores y vectores propios de una matriz al descomponer una matriz en su factorización QR (Q es una matriz ortogonal y R es una matriz triangular superior) y crear matrices A_k ortogonalmente similares a A bajo el siguiente procedimiento.

Sea una matriz cuadrada real A de tamaño nxn, luego:

$$A_0 = A \tag{9}$$

$$A_0 = Q_0 R_0 \tag{10}$$

$$A_{k+1} = R_k Q_k \tag{11}$$

Como cada matriz A_k es ortogonalmente similar a A y tienen los mismos valores propios, puesto que $A_{k+1} = Q_k^t .. Q_1^t A_0 Q_1 \dots Q_k$, entonces A_{k+1} converge a una matriz diagonal con los valores propios de A.

La matriz de vectores propios Φ se obtienen mediante:

$$\Phi = I Q_0 Q_1 \dots Q_k \tag{12}$$

donde I es la matriz Identidad.

Proceso de factorización QR

Expresando las matrices A , Q y R de la forma siguiente:

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \tag{13}$$

donde a_j es la j-ésima columna de A con $j \in \{1, n\}$

$$Q = \begin{pmatrix} q_1 & q_2 & \dots & q_n \end{pmatrix} \tag{14}$$

donde q_j es la j-ésima columna de Q con $j \in \{1, n\}$

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,n} \\ 0 & r_{2,2} & \dots & r_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & r_{n,n} \end{pmatrix} \tag{15}$$

Obtenemos las siguientes expresiones:

$$a_1 = r_{1,1} q_1 \tag{16}$$

$$q_1 = \frac{a_1}{r_{1,1}} \tag{17}$$

lo que implica que

$$r_{1,1} = \|a_1\| \tag{18}$$

puesto que todos los q_j son unitarios.

Luego

$$a_2 = r_{1,2} q_1 + r_{2,2} q_2 \tag{19}$$

$$q_2 = \frac{a_2 - r_{1,2} q_1}{r_{2,2}} \tag{20}$$

sea

$$a_2^* = a_2 - r_{1,2} q_1 \tag{21}$$

$$q_2 = \frac{a_2^*}{r_{2,2}} \tag{22}$$

lo que implica que

$$r_{2,2} = \|a_2^*\| \tag{23}$$

además multiplicando (13) por q_1^t

$$q_1^t a_2 = q_1^t r_{1,2} q_1 + q_1^t r_{2,2} q_2 \tag{24}$$

$$r_{1,2} = q_1^t a_2 \tag{25}$$

puesto que todos los q_j son ortogonales.

Luego

$$a_3 = r_{1,3} q_1 + r_{2,3} q_2 + r_{3,3} q_3 \tag{26}$$

$$q_3 = \frac{a_3 - r_{1,3} q_1 - r_{2,3} q_2}{r_{3,3}} \tag{27}$$

sea

$$a_3^* = a_3 - r_{1,3} q_1 - r_{2,3} q_2 \tag{28}$$

$$q_3 = \frac{a_3^*}{r_{3,3}} \tag{29}$$

lo que implica que

$$r_{3,3} = \|a_3^*\| \tag{30}$$

además multiplicando (20) por q_1^t

$$q_1^t a_3 = q_1^t r_{1,3} q_1 + q_1^t r_{2,3} q_2 + q_1^t r_{3,3} q_3 \tag{31}$$

$$r_{1,3} = q_1^t a_3 \tag{32}$$

y multiplicando (20) por q_2^t

$$q_2^t a_3 = q_2^t r_{1,3} q_1 + q_2^t r_{2,3} q_2 + q_2^t r_{3,3} q_3 \tag{33}$$

$$r_{2,3} = q_2^t a_3 \tag{34}$$

Generalizando:

$$r_{i,j} = q_i^t a_j,\ i < j,\ i, j \tag{35}$$

$$a_j^* = a_j - \sum_{k=1}^{j-1} r_{k,j} q_k \tag{36}$$

$$r_{j,j} = \|a_j^*\| \tag{37}$$

$$q_j = \frac{a_j^*}{\|a_j^*\|} \tag{38}$$

Algoritmo de Factorización QR

```
n -> Tamaño de la matriz
a[n][n] -> Matriz a // Al final "a" contendrá los valores de "q"
r[n][n] -> Matriz a
ap[n] -> Vector de tamaño n que representa "a*"

for j = 0 to n - 1:
    // Compute r_ij
    for i = 0 to j - 1:
        r[i][j] = r[j][i] = 0
        for k = 0 to n - 1:
            r[i][j] += q[k][i] * a[k][i]
    // Compute ap_j
    norm = 0
    for i = 0 to n -1:
        ap[i] = a[i][j]
        for k = 0 to n - 1:
            ap[i] -= r[k][j] * q[i][k]
        norm += ap[i] * ap[i]
    // Compute r_jj
    r[j][j] = sqrt(norm)
    // Compute q_j
    for i to n - 1:
        q[i][j] = ap[i] / norm
```

Algoritmo QR

```
n -> Tamaño de la matriz
A[n][n] -> Matriz A
Q[n][n] -> Matriz Q
R[n][n] -> Matriz R
Phi[n][n] -> Inicializada como una matriz Identidad

while is_not_diagonal(A):
    [Q, R] = qr_descomposition(A)
    A = R * Q
    Phi = Phi * Q
```

Ejemplo de entrada

Entrada

```
4 4
6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
-1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
-1.0000000000 2.0000000000 8.0000000000 -1.0000000000
4.0000000000 -1.0000000000 -1.0000000000 -5.0000000000
```

Salida

```
Eigen Values
-10.3710438740 0.0000000000 0.0000000000 0.0000000000
-0.0000000000 9.2688664888 -0.0000000000 -0.0000000000
-0.0000000000 0.0000000000 6.3568139827 0.0000000000
0.0000000000 -0.0000000000 0.0000000000 -6.2546365975

Eigen Vectors
-0.0168782711 -0.5486543461 0.7754166702 0.3121258080
-0.9833352317 0.1225974752 0.0115027924 0.1337511379
0.0978440936 0.7976799725 0.5917778752 -0.0627067875
-0.1522940555 -0.2183000895 0.2199900445 -0.9384859998
```

Observaciones y mejoras

Es mucho menos eficiente (en tiempo) que otros algoritmos, puesto que en cada paso hay que factorizar la matriz A en sus factores QR y luego multiplicar RQ, lo cual es una tarea costosa muy costosa.

Dado que sabemos que R es una matriz triangular podemos modificar el algoritmo de multiplicación de matrices para que sea más eficiente y evite calcular los productos denotados por los ceros debajo de la diagonal.

Algoritmo de Gradiente Conjugado para solucionar sistemas de ecuaciones

El método de Gradiente Conjugado es usado para resolver sistemas de la forma $Ax = b$, al tratar de minimizar el residuo generado por

$$r_0 = b - Ax_0 \tag{39}$$

Dónde x_0 es una primer aproximación a la solución y r_0 el residuo generado.

Una forma de mejorar la aproximación de x es calcular recursivamente un x^{k+1} mediante:

$$x_{k+1} = x_k + \alpha_k P_k \tag{40}$$

Dónde los vectores $\{p_k\}$ son llamados vectores de dirección y α_k es un escalar elegido para minimizar la expresión previa. α_k está dado por:

$$\alpha_k = \frac{P_k^t(r_k)}{P_k^t A P_k} \tag{41}$$

Los vectores dirección $\{P_k\}$ están dados por

$$P_{k+1} = r_{k+1} + B_k P_k \tag{42}$$

$$B_k = -\frac{P_k^t r_{k+1}}{P_k^t P_k} \tag{43}$$

Sea $r^k = b - Ax^k$ y $r^{k+1} = b - Ax^{k+1}$ dos residuos para dos aproximaciones x^k y x^{k+1}, sumando ambas expresiones:

$$r_{k+1} + r_k = -A(x_{k+1} - x^k) \tag{44}$$

luego

$$r^{k+1} = r_k - \alpha_k A P_k \tag{45}$$

Algoritmo

```
n -> Tamaño de la matriz
a[n][n] -> Matriz a
b[n] -> Vector b

r[n] -> Vector residuo
p[n] -> Vector dirección
x[n] -> Vector solución

MAX_ITER -> Número máximo de iteraciones
epsilon -> Valor mínimo para ser considerado como cero

r = b
p = r
x = {0}

while MAX_ITER:
    MAX_ITER -= 1
    w = A*p
    alpha = (p * r) / (p * w)
    x = x + alpha * p
    r = r - alpha * w
    error = norm(r)
    if error < epsilon:
```

```
        break
    beta = (p * r) / (p*p)
    p = r + beta * p

```

Ejemplo de entrada

Entrada

```
3 3
4.000000 -1.000000 0.000000
-1.000000 4.000000 -1.000000
0.000000 -1.000000 4.000000
```

```
3 1
2.000000
6.000000
2.000000
```

Salida

```
error: 2.052057e+00
error: 3.627558e-01
error: 1.122219e-01
error: 1.983821e-02
error: 6.137133e-03
error: 1.084902e-03
error: 3.356245e-04
error: 5.933058e-05
error: 1.835446e-05
error: 3.244641e-06
error: 1.003760e-06
error: 1.774413e-07
error: 5.489311e-08
error: 9.703822e-09
error: 3.001967e-09
error: 5.306778e-10
error: 1.641701e-10
error: 2.902144e-11
error: 8.978050e-12
error: 1.587110e-12

x_0: 1.0000000000
x_1: 2.0000000000
x_2: 1.0000000000
```

Observaciones y mejoras

Como podemos ver en la salida de ejemplo, por cada iteración el error se reduce al buscar el mínimo en la función implícita de error a minimizar. Durante las pruebas se observó que el antes de converger el error oscila demasiado y el decremento del error se vuelve cada vez más lento.

Con la matriz de entrada $M\_BIG.txt$ la convergencia a la solución fue bastante tardada lo cual lo hace no tan eficiente para matrices con las características de esta.