Eigensolvers

Descripción

Se realizó la implementación de los siguientes algoritmos:

1. Algoritmo de Potencia Inversa.
2. Algoritmo de Potencia con deflación
3. Algoritmo de Potencia Inversa con deflación.
4. Algoritmo de Jacobi

Algoritmo Método de Potencia Inversa

El Método de la Potencia Inversa es una modificación al Método de Potencia normal con la que se obtiene una convergencia más rápida. Su usa para calcular el valor propio más pequeño de una matriz. Sea A una matriz de tamaño nxn, luego se toma cualquier vector x_0 y en cada paso k se calcula:

$$x_{k+1} = A^{-1}x_k \tag{1}$$

Dónde x_{k+1} es la convergencia al vector propio, cuyo valor propio está dado por el cociente del producto punto de los vectores.

$$\lambda = \frac{x_{k+1} . x_k}{x_{k+1} . x_{k+1}} \tag{2}$$

Para resolver la ecuación (1), se procede a realizar la factorización LU de A y resolver el sistema resultante $LUx_{k+1} = x_k$.

Algoritmo

```
n -> Tamaño de la matriz
a[n][n] -> Matriz a
xprev -> Vector de tamaño n inicializado con valores aleatoreos
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

LU = factorizar(a)

while MAX_ITER --:
    normalizar(xprev)
    xnext = resolver(LU, xprev)
    lambda = xnext * xprev / xnext * xnext // Producto escalar
    xprev = xnext
    if (abs(last_lambda) - abs(lambda)) <= epsilon:
        break
    last_lambda = lambda

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

```
22 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
3.0000000000 -2.0000000000 8.0000000000 -1.0000000000
1.0000000000 1.0000000000 1.0000000000 -5.0000000000
```

Salida

```
Eigen Value: -5.4922091068
Eigen Vector:
0.3275393946 0.2274122123 -0.1066271118 -0.9108415283
```

Observaciones y mejoras

Durante las pruebas se obtuvieron los resultados esperados incluso con la matriz $M\_BIG.txt$, tanto el vector como el valor propio obtenidos son los mismo que los proporcionados por la solución de otros software como *GNU Octave*.

Durante la implementación se optó por usar el Método de Doolittle para resolver el sistema en cada paso, si de ante mano se conoce la matriz a resolver se podría usar otro método que sea más eficiente dada las características de la matriz A.

Algoritmo de Potencia con deflación

Los métodos de deflación son técnicas usadas para obtener aproximaciones a otros valores y vectores propios de una matriz una vez que se ha calculado una aproximación a los valores y vectores propios más dominantes.

Sea una matriz A simétrica con valores propios $\lambda_1, \lambda_2, \ldots, \lambda_n$ y vectores propios $v_1, v_2, \ldots, v_n$ y λ_1 con multiplicidad 1, también sea x un vector tal que $x^t v_1 = 1$, entonces podemos obtener una matriz B

$$B = A - \lambda_1 v_1 x^t \tag{3}$$

con los valores propios $0, \lambda_2, \ldots, \lambda_n$ y vectores propios $v_1, w_2, \ldots, w_n$, dónde w_i y v_i están relacionados por

$$v_i = (\lambda_i - \lambda_1) w_i + \lambda_1 (x^t w_i) v_1, i \in \{2, \ldots, n\} \tag{4}$$

Si λ_1 era el valor propio dominante, entonces deja de serlo, propiciando así, la convergencia del algoritmo de Potencia a el siguiente valor propio dominante.

El método de deflación usado está dado por siguiente proceso proceso.

Por cada vector propio v_i ya calculado:

$$x_k = x_k - x_k . v_i . v_i \tag{5}$$

Dónde x_k es la k-ésima propuesta de vector propio.

Algoritmo

```
MAX_ITER -> Número máximo de iteraciones
n -> Tamaño de la matriz
a[n][n] -> Matriz a
xprev -> Vector de tamaño n
xnext -> Vector de tamaño n

neigen -> Número de valores y vectores propios a calcular
eigen_vectors[n][n] -> Matriz de vectores propios
eigen_values[n] -> Vector de valores propios

lambda -> Valor propio
last_lambda -> Ultimo Valor propio calculado
epsilon -> Valor mínimo para considerar un cero

for k in [0, neigen]:
    inicializar(xprev) // Con valores random
    for iter in [0, MAX_ITER]:
        for i in [0, k]:
            xprev = xprev - xprev * eigen_vectors[i] * eigen_vectors[i]
        normalizar(xnext)
        xnext = a * nprev
        lambda = xnext * xnext / xnext * xprev // Producto escalar
        xprev = xnext
        if (abs(last_lambda) - abs(lambda)) <= epsilon:
            break
        last_lambda = lambda
    eigen_vectors[k] = normalizar(xnext)
    eigen_values[k] = lambda
```

Ejemplo de prueba

Entrada

```
4 4

6 -1 -1 4
-1 -10 2 -1
-1 2 8 -1
4 -1 -1 -5
```

Salida

```
1)
Eigen Value: -10.3710438740
Eigen Vector:
0.0168790461 0.9833350586 -0.0978452204 0.1522943639

2)
```

```
Eigen Value: 9.2688664888
Eigen Vector:
0.5486548426 -0.1225958997 -0.7976798019 0.2183003497

3)
Eigen Value: 6.4472017477
Eigen Vector:
0.7466891597 0.0003100016 0.5949464199 0.2974793439

4)
Eigen Value: -6.3450243625
Eigen Vector:
0.2452965220 0.1322953225 -0.1126272429 -0.9537518902
```

Observaciones y mejoras

Debido a que en cada obtención de un valor y vector propio hay un error de aproximación, estos errores se acumulan mientras más iteraciones se hagan provocado que los cálculos posteriores de vectores y valores propios sean más inexactos. Usando la matriz $M\_Big.txt$ se observó solo la correcta convergencia para los primeros 17 valores y vectores propios, los siguientes a pesar de ser muy parecidos se encontraban inconsistencias en los valores puesto que cada valor propio obtenido debe ser menor al anterior y en algunos casos no sucedía así.

Algoritmo de Potencia Inversa con deflación

El método de deflación usado es el mismo al anterior, esta vez en combinación con el método de Potencia Inversa.

Algoritmo

```
MAX_ITER -> Número máximo de iteraciones
n -> Tamaño de la matriz
a[n][n] -> Matriz a
xprev -> Vector de tamaño n
xnext -> Vector de tamaño n

neigen -> Número de valores y vectores propios a calcular
eigen_vectors[n][n] -> Matriz de vectores propios
eigen_values[n] -> Vector de valores propios

lambda -> Valor propio
last_lambda -> Ultimo Valor propio calculado
epsilon -> Valor mínimo para considerar un cero

LU = factorizar(a)

for k in [0, neigen]:
    inicializar(xprev) // Con valores random
    for iter in [0, MAX_ITER]:
        for i in [0, k]:
            xnprev = xprev - xprev * eigen_vectors[i] * eigen_vectors[i]
        normalizar(xnext)
        xnext = resolver(LU, xprev)
        lambda = xnext * xprev / xnext * xnext // Producto escalar
        xprev = xnext
```

```
        if (abs(last_lambda) - abs(lambda)) <= epsilon:
            break
        last_lambda = lambda
    eigen_vectors[k] = normalizar(xnext)
    eigen_values[k] = lambda
```

Ejemplo de prueba

Entrada

```
4 4
6 -1 -1 9
-1 10 2 -1
-1 2 8 -1
9 -1 -1 -3
```

Salida

```
1)
Eigen Value: 6.715623e+00
Eigen Vector:
9.718471e-02 -4.837968e-01 8.683033e-01 5.045080e-02

2)
Eigen Value: -8.573234e+00
Eigen Vector:
5.237401e-01 -1.559617e-02 -1.783302e-02 -8.515486e-01

3)
Eigen Value: 9.536074e+00
Eigen Vector:
5.761235e-01 6.837826e-01 2.970335e-01 3.351031e-01

4)
Eigen Value: 1.332154e+01
Eigen Vector:
-6.201510e-01 5.460124e-01 3.968602e-01 -3.997315e-01
```

Observaciones y Mejoras

A diferencia con el algoritmo del Método de Potencia con deflación, este algoritmo obtuvo en su mayoría una muy buena aproximación a todos los valores y vectores propios de la matriz $M\_BIG.txt$ excepto para los últimos valores propios con valor 1 y su vectores asociados.

Algoritmo de Jacobi

El algoritmo de Jacobi nos ayuda a encontrar todos los valores y vectores propios de una matriz simétrica. La solución es garantizada con matrices simétricas reales.

Dada una matriz simétrica A de tamaño nxn, entonces existe una matriz ortogonal Q tal que:

$$Q^t AQ = D \tag{6}$$

dónde D es una matriz diagonal con los valores propios de A.

La técnica consiste en encontrar una serie de matrices ortogonales $\{S_k\}$ tal que

$$\lim_{k->\infty} S_1 S_2 S_3 \dots S_k = Q \tag{7}$$

a través de la diagonalización de A con una serie de transformaciones mediante una matriz de rotación

$$S_k = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \cos(\theta)_{ii} & \dots & \sin(\theta)_{ij} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -\sin(\theta)_{ji} & \dots & \cos(\theta)_{jj} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \tag{8}$$

aplicada sobre la máximo elemento en valor absoluto de la matriz A ignorando la diagonal con posición (i, j).

La multiplicación $A_{k+1} = A_k S_k$ está dada por:

$$a_{lk}^{k+1} = a_{lk}^k, \ cuando\ l \neq i\ y\ l \neq j,\ l \in [1, n] \tag{9}$$

$$a_{li}^{k+1} = a_{li}^k \cos(\theta) + a_{lj}^k \sin(\theta),\ l \in [1, n] \tag{10}$$

$$a_{lj}^{k+1} = -a_{li}^k \sin(\theta) + a_{lj}^k \cos(\theta),\ l \in [1, n] \tag{11}$$

$$a_{im}^{k+1} = a_{im}^k \cos(\theta) + a_{jm}^k \sin(\theta),\ m \in [1, n] \tag{12}$$

$$a_{jm}^{k+1} = -a_{im}^k \sin(\theta) + a_{jm}^k \cos(\theta),\ m \in [1, n] \tag{13}$$

Y la matriz Q_{k+1} con $Q_0 = I$, contendrá los vectores propios asociados a los valores propios de la última A_{k+1} y está dada por:

$$q_{lk}^{k+1} = q_{lk}^k, \ cuando\ l \neq i\ y\ l \neq j,\ l \in [1, n] \tag{14}$$

$$q_{li}^{k+1} = q_{li}^k \cos(\theta) + q_{lj}^k \sin(\theta),\ l \in [1, n] \tag{15}$$

$$q_{lj}^{k+1} = -q_{li}^k \sin(\theta) + q_{lj}^k \cos(\theta),\ l \in [1, n] \tag{16}$$

El valor de θ se calcula dado al máximo elemento de la matriz A con posición a_{ij} mediante

$$\theta = \frac{\arctan(\frac{2 * a_{ij}}{a_{ii} - a_{jj}})}{2} \tag{17}$$

Algoritmo

```
MAX_ITER -> Número máximo de iteraciones
n -> Tamaño de la matriz
a[n][n] -> Matriz a
q[n][n] -> Matriz identidad
epsilon -> Valor mínimo para considerar un cero

while MAX_ITER--
    a_ij, i, j = encontrar_max(a)

    if (abs(a_ij) < epsilon) break
```

```
    theta = atan2(2.0 * a_ij, a[i][i] - a[j][j]) / 2.0;
    cos_theta = cos(theta)
    sin_theta = sin(theta)

    for l in [0,n]:
        a_li = a[l][i];
        a[l][i] = a_li * cos_theta + a[l][j] * sin_theta
        a[l][j] = -a_li * sin_theta + a[l][j] * cos_theta

        q_li = q[l][i]
        q[l][i] = q_li * cos_theta + q[l][j] * sin_theta
        q[l][j] = -q_li * sin_theta + q[l][j] * cos_theta

    for m in [0, n]:
        a_im = a[i][m];
        a[i][m] = a_im * cos_theta + a[j][m] * sin_theta
        a[j][m] = -a_im * sin_theta + a[j][m] * cos_theta

a <- Contiene los valores propios en la diagonal
q <- contiene los vectores propios por columna asociados a los valores
propios de a
```

Ejemplo de prueba

Entrada

```
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

```
9.2688664888 -0.0000000000 0.0000000000 0.0000000000
-0.0000000000 6.3568139827 -0.0000000000 0.0000000000
0.0000000000 -0.0000000000 -10.3710438740 -0.0000000000
0.0000000000 0.0000000000 -0.0000000000 -6.2546365975

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

Observaciones y mejoras

Durante las pruebas se observó un comportamiento extraño que al no ser tan cercanos a ceros los elementos fuera de la diagonal de la matriz, así el algoritmo se veía envuelto en un bucle infinito. Se solucionó usando un épsilon más grande y agregando en el algoritmo un número máximo de iteraciones.

Se observó una mucho mejor aproximación a los valores y vectores propios (todos fueron correctos) de la matriz de entrada $M\_BIG.txt$ mediante el algoritmo de Jacobi que con los otros dos algoritmos de Potencia usando deflación.