

# 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:

$$r = \|Ax - \alpha x\| \quad (1)$$

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

$$\alpha = \frac{x^t Ax}{x^t x} \quad (2)$$

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:

$$x_{k+1} = Ax_k \quad (3)$$

## Algoritmo

```
1  n -> Tamaño de la matriz
2  a[n][n] -> Matriz a
3  xprev -> Vector de tamaño n con nuestra aproximación al vector propio
4  xnext -> Vector de tamaño n
5  MAX_ITER -> Número máximo de iteraciones
6  lambda -> Valor propio
7  last_lambda -> Ultimo Valor propio calculado
8  epsilon -> Valor mínimo para considerar un cero
9
10 while MAX_ITER --:
11     normalizar(xprev)
12     xnext = A * xprev
13     lambda = xprev * A * xprev
14     xprev = xnext
15     if (abs(last_lambda) - abs(lambda)) <= epsilon:
16         break
17     last_lambda = lambda
18
19 xnext <- Contiene el vector propio sin normalizar
```

## Ejemplo de prueba

### Entrada

```

1 4 4
2 6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
3 -1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
4 -1.0000000000 2.0000000000 8.0000000000 -1.0000000000
5 4.0000000000 -1.0000000000 -1.0000000000 -5.0000000000
6
```

```

1 4 1
2 17
3 -7
4 19
5 -14
```

### Salida

```

1 Eigen Value: -10.371044
2 Eigen vector:
3 -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.

## 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  $n \times n$ , luego:

$$A_0 = A \quad (4)$$

$$A_0 = Q_0 R_0 \quad (5)$$

$$A_{k+1} = R_k Q_k \quad (6)$$

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

La matriz de vectores propios  $\Phi$  se obtienen mediante:

$$\Phi = I Q_0 Q_1 \dots Q_k \quad (7)$$

donde  $I$  es la matriz Identidad.

## Proceso de factorización $QR$

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

$$A = (a_1 \quad a_2 \quad \dots \quad a_n) \quad (8)$$

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

$$Q = (q_1 \quad q_2 \quad \dots \quad q_n) \quad (9)$$

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

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,n} \\ 0 & r_{2,2} & \dots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{n,n} \end{pmatrix} \quad (10)$$

Obtenemos las siguientes expresiones:

$$a_1 = r_{1,1} q_1 \quad (11)$$

$$q_1 = \frac{a_1}{r_{1,1}} \quad (12)$$

lo que implica que

$$r_{1,1} = \|a_1\| \quad (13)$$

puesto que todos los  $q_j$  son unitarios.

Luego

$$a_2 = r_{1,2} q_1 + r_{2,2} q_2 \quad (14)$$

$$q_2 = \frac{a_2 - r_{1,2} q_1}{r_{2,2}} \quad (15)$$

sea

$$a_2^* = a_2 - r_{1,2}q_1 \quad (16)$$

$$q_2 = \frac{a_2^*}{r_{2,2}} \quad (17)$$

lo que implica que

$$r_{2,2} = \|a_2^*\| \quad (18)$$

además multiplicando (13) por  $q_1^t$

$$q_1^t a_2 = q_1^t r_{1,2}q_1 + q_1^t r_{2,2}q_2 \quad (19)$$

$$r_{1,2} = q_1^t a_2 \quad (20)$$

puesto que todos los  $q_j$  son ortogonales.

Luego

$$a_3 = r_{1,3}q_1 + r_{2,3}q_2 + r_{3,3}q_3 \quad (21)$$

$$q_3 = \frac{a_3 - r_{1,3}q_1 - r_{2,3}q_2}{r_{3,3}} \quad (22)$$

sea

$$a_3^* = a_3 - r_{1,3}q_1 - r_{2,3}q_2 \quad (23)$$

$$q_3 = \frac{a_3^*}{r_{3,3}} \quad (24)$$

lo que implica que

$$r_{3,3} = \|a_3^*\| \quad (25)$$

además multiplicando (20) por  $q_1^t$

$$q_1^t a_3 = q_1^t r_{1,3}q_1 + q_1^t r_{2,3}q_2 + q_1^t r_{3,3}q_3 \quad (26)$$

$$r_{1,3} = q_1^t a_3 \quad (27)$$

y multiplicando (20) por  $q_2^t$

$$q_2^t a_3 = q_2^t r_{1,3}q_1 + q_2^t r_{2,3}q_2 + q_2^t r_{3,3}q_3 \quad (28)$$

$$r_{2,3} = q_2^t a_3 \quad (29)$$

Generalizando:

$$r_{i,j} = q_i^t a_j, \quad i < j, \quad i, j \quad (30)$$

$$a_j^* = a_j - \sum_{k=1}^{j-1} r_{k,j}q_k \quad (31)$$

$$r_{j,j} = \|a_j^*\| \quad (32)$$

$$q_j = \frac{a_j^*}{\|a_j^*\|} \quad (33)$$

## Algoritmo de Factorización $QR$

```
1  n -> Tamaño de la matriz
2  a[n][n] -> Matriz a // Al final "a" contendrá los valores de "q"
3  r[n][n] -> Matriz a
4  ap[n] -> Vector de tamaño n que representa "a*"
5
6  for j = 0 to n - 1:
7      // Compute r_ij
8      for i = 0 to j - 1:
9          r[i][j] = r[j][i] = 0
10         for k = 0 to n - 1:
11             r[i][j] += q[k][i] * a[k][j]
12     // Compute ap_j
13     norm = 0
14     for i = 0 to n - 1:
15         ap[i] = a[i][j]
16         for k = 0 to n - 1:
17             ap[i] -= r[k][j] * q[i][k]
18         norm += ap[i] * ap[i]
19     // Compute r_jj
20     r[j][j] = sqrt(norm)
21     // Compute q_j
22     for i to n - 1:
23         q[i][j] = ap[i] / norm
```

## Algoritmo $QR$

```
1  n -> Tamaño de la matriz
2  A[n][n] -> Matriz A
3  Q[n][n] -> Matriz Q
4  R[n][n] -> Matriz R
5  Phi[n][n] -> Inicializada como una matriz Identidad
6
7  while is_not_diagonal(A):
8      [Q, R] = qr_descomposition(A)
9      A = R * Q
10     Phi = Phi * Q
```

## Ejemplo de entrada

### Entrada

```
1  4 4
2  6.0000000000 -1.0000000000 -1.0000000000 4.0000000000
3  -1.0000000000 -10.0000000000 2.0000000000 -1.0000000000
4  -1.0000000000 2.0000000000 8.0000000000 -1.0000000000
5  4.0000000000 -1.0000000000 -1.0000000000 -5.0000000000
```

### Salida

```

1 Eigen Values
2 -10.3710438740 0.0000000000 0.0000000000 0.0000000000
3 -0.0000000000 9.2688664888 -0.0000000000 -0.0000000000
4 -0.0000000000 0.0000000000 6.3568139827 0.0000000000
5 0.0000000000 -0.0000000000 0.0000000000 -6.2546365975
6
7 Eigen Vectors
8 -0.0168782711 -0.5486543461 0.7754166702 0.3121258080
9 -0.9833352317 0.1225974752 0.0115027924 0.1337511379
10 0.0978440936 0.7976799725 0.5917778752 -0.0627067875
11 -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

$$r_0 = b - Ax_0 \quad (34)$$

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:

$$x_{k+1} = x_k + \alpha_k P_k \quad (35)$$

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

$$\alpha_k = \frac{P_k^t(r_k)}{P_k^t A P_k} \quad (36)$$

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

$$P_{k+1} = r_{k+1} + B_k P_k \quad (37)$$

$$B_k = -\frac{P_k^t r_{k+1}}{P_k^t P_k} \quad (38)$$

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:

$$r_{k+1} + r_k = -A(x_{k+1} - x^k) \quad (39)$$

luego

$$r^{k+1} = r_k - \alpha_k A P_k \quad (40)$$

# Algoritmo

```
1  n -> Tamaño de la matriz
2  a[n][n] -> Matriz a
3  b[n] -> Vector b
4
5  r[n] -> Vector residuo
6  p[n] -> Vector dirección
7  x[n] -> Vector solución
8
9  MAX_ITER -> Número máximo de iteraciones
10 epsilon -> Valor mínimo para ser considerado como cero
11
12 r = b
13 p = r
14 x = {0}
15
16 while MAX_ITER:
17     MAX_ITER -= 1
18     w = A*p
19     alpha = (p * r) / (p * w)
20     x = x + alpha * p
21     r = r - alpha * w
22     error = norm(r)
23     if error < epsilon:
24         break
25     beta = (p * r) / (p*p)
26     p = r + beta * p
27
```

## Ejemplo de entrada

### Entrada

```
1  3 3
2  4.000000 -1.000000 0.000000
3  -1.000000 4.000000 -1.000000
4  0.000000 -1.000000 4.000000
```

```
1  3 1
2  2.000000
3  6.000000
4  2.000000
```

### Salida

```
1  error: 2.052057e+00
2  error: 3.627558e-01
3  error: 1.122219e-01
4  error: 1.983821e-02
5  error: 6.137133e-03
6  error: 1.084902e-03
7  error: 3.356245e-04
8  error: 5.933058e-05
```

```
9 error: 1.835446e-05
10 error: 3.244641e-06
11 error: 1.003760e-06
12 error: 1.774413e-07
13 error: 5.489311e-08
14 error: 9.703822e-09
15 error: 3.001967e-09
16 error: 5.306778e-10
17 error: 1.641701e-10
18 error: 2.902144e-11
19 error: 8.978050e-12
20 error: 1.587110e-12
21
22 x_0: 1.0000000000
23 x_1: 2.0000000000
24 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.