Tarea 07 - Eigensolvers y solvers iterativos

Isaac Rodríguez Bribiesca

Resumen Para esta tarea se implementaron los algoritmos: Rayleigh, iteración en subespacio y QR para valores y vectores propios. Y gradiente conjugado para solución de sistemas de ecuaciones.

1. Algoritmo de Rayleigh

1.1. Introducción

El método de Rayleigh es bastante parecido al de iteración de potencia inversa, donde la idea es obtener el valor propio λ_n con valor absoluto más pequeño o más cercano a cero y su respectivo vector propio v_n para una matriz A de nxn para la cual existe su matriz inversa A^{-1}.

En el caso de potencia inversa se itera sobre la ecuación:

$$A * v^{k+1} = v^k \tag{1}$$

La cual se resuelve para v^{k+1}.

La diferencia con el método de Rayleigh es que se itera sobre la ecuación:

$$(A - \sigma * I) * v^{k+1} = v^k \tag{2}$$

Donde también se resuelve para v^{k+1} y donde σ será la aproximación a algún valor propio que queremos obtener. Dicho algoritmo convergerá al valor propio λ_0 más cercano a σ.

La ecuación para obtener λ_0 es:

$$\lambda^{k+1} = (v^k)^T * A * v^k \tag{3}$$

1.2. Metodología

Para la implementación del algoritmo, como ya se comentó, se hace uso del método ya implementado en tareas anteriores, el método de potencia inversa.

La única modifcación que se realiza al algoritmo es iterar sobre la matriz $(A-\sigma*I)$ la cual se guarda en otra matriz ya que se necesita conservar A para actualizar λ^{k+1}. El parámetro σ se matiene constante durante toda la ejecución del algoritmo.

El resultado final será la última λ^{k+1} y el último v^{k+1} normalizado. Deteniendo las iteraciones cuando se alcanza el límite o se llega a la convergencia de λ^{k+1}.

1.3. Ejemplo de prueba

Ejemplo con matriz simétrica de 3x3. Se prueba el algortimo con 2 valores iniciales $\sigma_1 = -3{,}0$ y $\sigma_1 = -6{,}5$

```
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/Numerical-Methods/iterative_solvers3/RayleighSolver master ● make && ./runTest ../M_SIM2.txt
gcc -c rayleigh_solver_test.c
gcc -c ../solve_iterative2.c
gcc -c ../solve_iterative3.c
gcc -c  ../matrix_struct.c
gcc -c ../solve_matrix_direct.c
gcc -o runTest rayleigh_solver_test.o solve_iterative3.o solve_iterative2.o matrix_struct.o solve_matrix_direct.o -lm

Succesfully read: ../M_SIM2.txt

Valor propio inicial: -3.000000

Converged after 42 iterations


Valor propio aproximado: -6.892043

isaac@irb ~/Documents/CIMAT/Métodos Numéricos/Numerical-Methods/iterative_solvers3/RayleighSolver master ● make && ./runTest ../M_SIM2.txt
gcc -c rayleigh_solver_test.c
gcc -c ../solve_iterative2.c
gcc -c ../solve_iterative3.c
gcc -c  ../matrix_struct.c
gcc -c ../solve_matrix_direct.c
gcc -o runTest rayleigh_solver_test.o solve_iterative3.o solve_iterative2.o matrix_struct.o solve_matrix_direct.o -lm

Succesfully read: ../M_SIM2.txt

Valor propio inicial: -6.500000

Converged after 6 iterations


Valor propio aproximado: -6.892025
```

Ejemplo con matriz de la tarea "M_BIG.txtz valor inicial $\sigma_1 = 0{,}8$

```
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/Numerical-Methods/iterative_solvers3/RayleighSolver master ● make && ./runTest ../M_BIG.txt
gcc -c rayleigh_solver_test.c
gcc -c ../solve_iterative2.c
gcc -c ../solve_iterative3.c
gcc -c  ../matrix_struct.c
gcc -c ../solve_matrix_direct.c
gcc -o runTest rayleigh_solver_test.o solve_iterative3.o solve_iterative2.o matrix_struct.o solve_matrix_direct.o -lm

Succesfully read: ../M_BIG.txt

Valor propio inicial: 0.800000

Converged after 2 iterations


Valor propio aproximado: 1.000000
```

Ejemplo con matriz de la tarea "M_BIG.txtz valor inicial $\sigma_1 = 0{,}000081$

```
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/Numerical-Methods/iterative_solvers3/RayleighSolver master ● make && ./runTest ../M_BIG.txt
gcc -c rayleigh_solver_test.c
gcc -c ../solve_iterative2.c
gcc -c ../solve_iterative3.c
gcc -c  ../matrix_struct.c
gcc -c ../solve_matrix_direct.c
gcc -o runTest rayleigh_solver_test.o solve_iterative3.o solve_iterative2.o matrix_struct.o solve_matrix_direct.o -lm

Succesfully read: ../M_BIG.txt

Valor propio inicial: 8.100000e-05

Converged after 14 iterations


Valor propio aproximado: 8.443715e-05
```

1.4. Resultados

Para el caso de la prueba con la matriz "M_BIG.txtçon valor inicial $\sigma_1 = 0{,}8$ vemos que se converge de manera muy rápida a uno de los 12 valores propios con valor 1 que tiene la matriz. La desventaja de esto es que al haber muchos valores propio repetidos, no podemos identificar cual se ha obtenido. A diferencia del método de potencia inversa con deflación, el cual nos podría regresar los 12 valores propios con el mismo valor y sus respectivos vectores propios de manera ordenada.

Para el caso de la prueba con la matriz "M_BIG.txtçon valor inicial $\sigma_1 = 0{,}000081$ vemos que se ha llegado a la convergencia a 0.00008443715, mientras que el valor propio correcto es 0.00008446244. Este último valor se ha obtenido también con potencia inversa con deflación, y como se puede observar hay un pequeño error en el cálculo.

Esto se compensa ya que con Rayleigh el número de iteraciones para llegar a la convergencia fue de 14 iteraciones, mientras que con potencia inversa con deflación se necesitaron 117 iteraciones.

Finalmente se puede observar que es un método muy sensible al valor inicial σ_1, por lo que es un método factible de usar cuando ya tenemos una aproximación al valor propio que deseamos obtener, dándonos la ventaja de que si la aproximación que tenemos es buena, la convergencia se conseguirá de manera más rápida.

2. Iteración en subespacio

2.1. Introducción

El método en subespacio es un algoritmo que permite obtener k valores y vectores propios dominantes, de una matriz A diagonalizable de nxn, de manera simultanea. Esto se puede ver como una generalización del método de potencia, donde a partir de un vector inicial q, se genera la secuencia $\{q, Aq, A^2q, ...\}$, la cual, con un reescalamiento adecuado, convergerá al vector propio asociado al valor propio dominante λ_1.

De manera más general si S es un subespacio de k dimensiones, podemos analizar la secuencia $\{S, AS, A^2S, ...\}$, esta podría converger al subespacio invariante generado por los vectores $q_1, q_2, ..., q_k$, que son los vectores propios asociados a los valores propios dominantes $\lambda_1, ..., \lambda_k$ de A. Esto tomando en cuenta que se cumpla $|\lambda_k| > |\lambda_{k+1}|$.

Por lo tanto la idea de iteración en subespacio, es iniciar con un subespacio de k dimensiones, representado por una matriz FI_0 de nxk, sobre la cual se iterarán las siguientes operaciones: 1. Factorizar FI_i en $FI_i = QR$, 2. obtener matriz $\Lambda_i = Q^T AQ$, cuya diagonal principal convergerá a los k valores propios dominantes de A, y 3. Actualizar la matriz $FI_{i+1} = AQ$, donde FI_{i+1} convergerá a los k vectores propios asociados a los k valores propios dominantes.

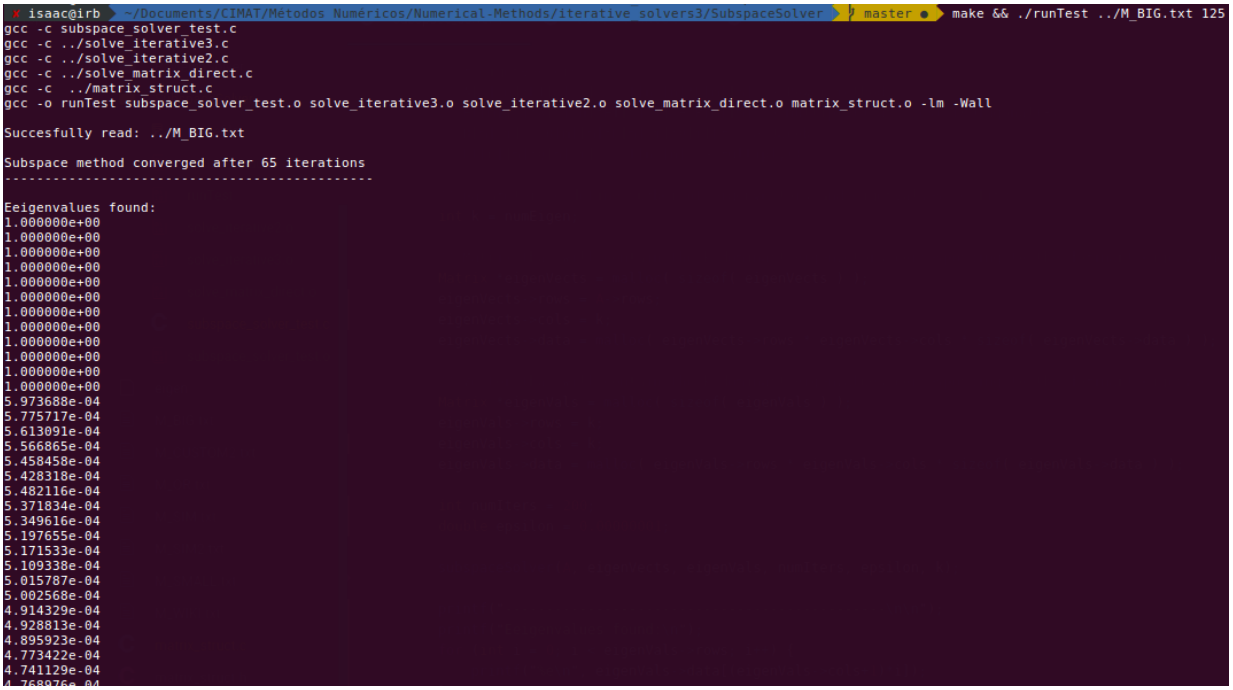
2.2. Metodología

Como se comentó anteriormente, se inicializa una matriz FI_0 de nxk de manera aleatoria.

Para cada iteración i desde 0 hasta m:

1. Factorizar $FI_i = QR$
2. Calcular $\Lambda_i = Q^T AQ$
3. Actualizar $FI_{i+1} = AQ$
4. Si Λ_i es diagonal, se converge y se detienen las iteraciones.

2.3. Ejemplo de prueba



2.4. Resultados

Para el caso de la matriz ”M_BIG.txt”se obtuvo la convergencia después de 65 iteraciones, con $k = 125$, es decir para el caso en que se calcular al mismo tiempo todos los valores y vectores propios de A. Para el caso $k = 10$, la convergencia es mucho más rápido, y se requieren tan solo 2 iteraciones.

Debido a las operaciones que se deben realizar, 3 multiplicaciones entre matrices y la factorización QR, resulta ser un algoritmo costoso.

Debido a esto, hay algunas cosas que se podrían modificar para mejorar el rendimiento, como ir fijar los valores propios que convergen y dejar que el método opere en los que aún no lo hacen. También se pueden aplicar corrimientos similares al usado en el método de Rayleigh, donde se opera con $A - \sigma I$, facilitando y acelerando la convergencia.

3. QR

3.1. Introducción

Este algoritmo permite calcular todos los valores y vectores propios de una matriz A diagonalizable.

El algortimo se basa en la factorización QR que tiene la forma $A = QR$ donde Q es una matriz ortogonal y R una matriz triangular superior.

Y consiste en iterar el producto:

$$A^k = R^k * Q^k \tag{4}$$

Donde R^k y Q^k son las matrices de la factorización de la matriz A^{k-1} y donde A^0 es la matriz inicial de la cual se quieren obtener los valores y vectores propios.

Después de un cierto número de iteraciones la matriz A^k convergerá a una matriz triangular superior la cuál tendrá en la diagonal principal los valores propios de $A^0 = A$.

Esto se logra debido a que si $A = QR$, premultiplicando por Q^T y postmultiplicando por Q ambos lados de la ecuación se obtiene $RQ = Q^T AQ$, donde se observa que $Q^T AQ$ es una transformación de semejanza de A y por lo tanto tiene los mismos eigenvalores que A.

La matriz de vectores propios se podrá obtener mediante el producto:

$$U^k = U^{k-1} * Q^k \tag{5}$$

Con $U^0 = I_n$ donde I_n es la matriz identidad de nxn.

Para obtener dicha factorización $A = QR$, se realiza un proceso de ortogonalización de Gram-Schmidt a partir de la matriz A.

Sean $a_1, a_2, ..., a_n$ las columnas de la matriz A, $q_1, q_2..., q_n$ las columnas de Q y r_{ij} los elementos de R con $j > i$.

Se inicia la factorización con $q_1 = \frac{a_1}{\|a_1\|}$, donde $\|a_1\|$ es la norma del vector a_1.

Los elementos q_k con $k = 2, ..., n$, se obtienen mediante $q_k = \frac{p_k}{\|p_k\|}$.

Donde

$$p_k = a_k - \sum_{j=1}^{k-1} proj_{p_j} a_k \tag{6}$$

Con $proj_{p_j} a_k = \frac{\langle p_j, a_k \rangle}{\langle p_j, p_j \rangle} p_j$. Donde $\langle a, b \rangle$ es el producto punto de 2 vectores a y b.

Para el caso de los elementos de R, estos se pueden calcular mediante:

$$R = \begin{bmatrix} \langle q_1, a_1 \rangle & \langle q_1, a_2 \rangle & \langle q_1, a_3 \rangle & \dots \\ 0 & \langle q_2, a_2 \rangle & \langle q2, a_3 \rangle & \dots \\ 0 & 0 & \langle q3, a_3 \rangle & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

3.2. Metodología

Para el caso de la factorización de QR se realizan los siguientes pasos:

1. Calcula $q_1 = \frac{a_1}{\|a_1\|}$
2. Calcula $r_{11} = \|a_1\|$

Posteriormente para cada itetación de $i = 2, 3, ..., n$ se realiza lo siguiente:

3. Calcula $p_i = a_i - \sum_{j=1}^{i-1} proj_{p_j} a_i$
4. Calcula $q_i = \frac{p_i}{\|p_i\|}$
5. Calcula $r_{ki} = \langle q_k, a_i \rangle$ con $k = 1, ..., n$

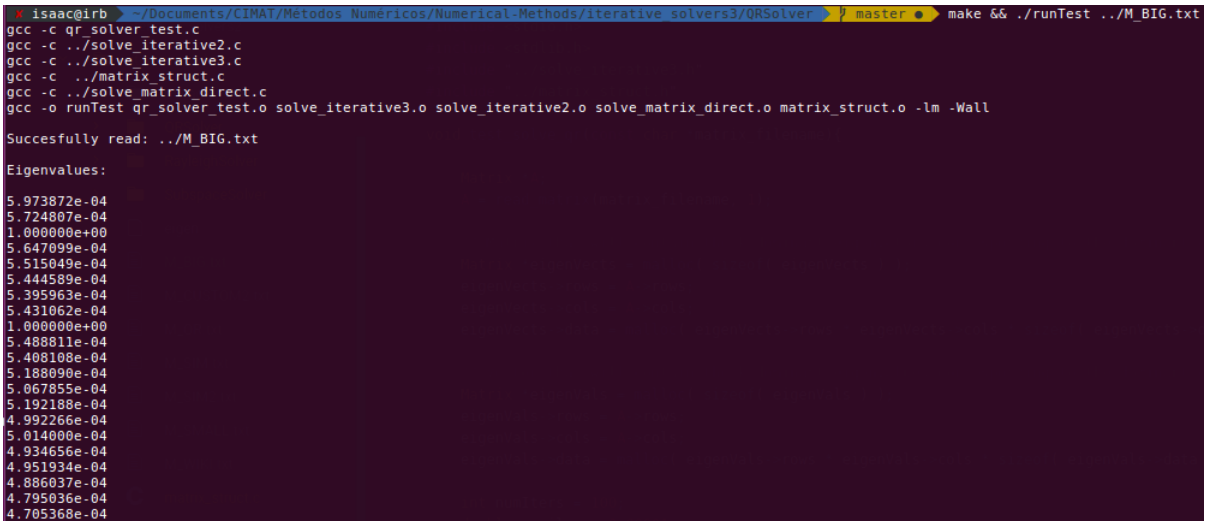
Para el caso del método de QR se realizan los siguientes pasos:

1. Inicializar $U = I_n$

En cada iteración desde 1 hasta m:

2. Factorizar A en Q y R
3. Actualizar $A = R * Q$
4. Actualizar $U = U * Q$

3.3. Ejemplo de prueba



3.4. Resultados

A pesar de que da buenos resultados al obtener todos los valores y vectores propios calculándolos al mismo tiempo, es un algoritmo costoso ya que cada paso es de orden $O(n^3)$, teniendo que realizar la factorización y 2 productos de matrices, $U^k = U^{k-1} * Q^k$ y $A^k = R^k * Q^k$.

Y como en general se necesitan al menos n iteraciones para lograr la convergencia del algoritmo, QR termina siendo de complejidad $O(n^4)$.

Una mejora que se podría implementar al algoritmo es agregar una fase previa, la cual consiste en primero reducir a la matriz A a una matriz H de Hessenberg, cuyos elementos $h_{ij} = 0$ para $i > j$. La matriz H se podrá obtener mediante transformaciones de reflexión de Householder. Dicha transformación tiene un costo de complejidad $O(n^3)$, pero sólo se realiza una vez.

Posterior a la obtención de la matriz H se podrá continuar con el algoritmo ya explicado anteriormente.

Con esto, cada factorización QR se reduce en complejidad a $O(n^2)$, por lo que el algoritmo total se reduce a $O(n^3)$.

4. Método de Gradiente Conjugado

4.1. Introducción

El método de gradiente conjugado es un algoritmo para encontrar soluciones a sistemas de ecuaciones lineales $A * x = b$, donde la matriz A es simétrica y positiva definida.

La idea del algortimo es encontrar el vector solución $x^* = \sum_{i=1}^{n} \alpha_i * p_i$ tal que $A * x^* = b$. Es decir, se expresa la solución como una combinación lineal de vectores conjugados p_i respecto a la matriz A. La tarea del algoritmo será encontrar estas direcciones conjugadas p_i y el tamaño de paso que se debe dar α_i en estas direcciones para llegar a la solución x^* al sistema.

Se puede mostrar que la solución x^*, también minimiza la función $f(x) = \frac{1}{2}x^T Ax - x^T b$, ya que la derivada de $f(x)$, es $f'(x) = Ax - b$, y por lo tanto el punto mínimo x de $f(x)$ cumple con $Ax = b$.

Se inicia con un vector $p_1 = b - A * x_1$ que es equivalente a la dirección opuesta a la derivada de $f(x)$. Es decir, $p_1 = -\nabla f(x)$. El resto de direcciones p_i serán construidos de forma que sean conjugados al gradiente p_1.

A continuación se describe el algoritmo implementado para encontrar las direcciones conjugadas p_i y los escalares α_i.

4.2. Metodología

1. Se inicializan vectores $r_1 = p_1 = b - A * x_1$. Donde x_1 puede ser una aproximación a la solución x^* o el vector cero.

Después, para cada iteración $k = 2, ..., n$:

2. Calcular producto $w = Ap_k$
3. Calcular $\alpha_k = \frac{r_k^T r_k}{p_k^T w}$
4. Calcular $x_{k+1} = x_k + \alpha_k p_k$
5. Calcular $r_{k+1} = r_k - \alpha_k w$
6. Si $\|r_{k+1}\| < \epsilon$ se ha llegado a la solución y se termina la iteración. Se fija $\epsilon = 1e^{-9}$.
7. Calcular $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
8. Calcular $p_{k+1} = r_{k+1} + \beta_k p_k$ 9. Repetir pasos 2 a 8.

El resultado habrá quedado en x_{k+1}.

4.3. Ejemplo de prueba

```
isaac@irb ~/Documents/CIMAT/Metodos Numericos/Numerical-Methods/iterative_solvers3/CGradientSolver master make && ./runTest ../M_BIG.txt ../V_BIG.txt
gcc -c cgradient_solver_test.c
gcc -c ../solve_iterative3.c
gcc -c ../solve_iterative2.c
gcc -c  ../matrix_struct.c
gcc -c ../solve_matrix_direct.c
gcc -o runTest cgradient_solver_test.o solve_iterative3.o solve_iterative2.o solve_matrix_direct.o matrix_struct.o -lm -Wall -fopenmp

Succesfully read: ../M_BIG.txt

Succesfully read: ../V_BIG.txt

Succesfully read: ../V_BIG.txt

Algorithm converged after 42 iterations


 0.604900
 0.567970
 0.006738
 0.603159
 0.597789
 0.588702
 0.485865
 0.370397
 0.006738
 0.191088
 0.575848
 0.074430
 0.604508
 0.123127
 0.538496
 0.449283
 0.336415
 0.207982
```

4.4. Resultados

Para el caso de la matriz "M_BIG.txt"se llegó a la convergencia en 42 iteraciones empezando con un vector $x_1 = b$. Para el caso de un vector inicial $x_1 = 0$ el número de iteraciones fue de 53. Se observa que la convergencia se da en un número de iteraciones mucho menor que el tamaño de la matriz A de $125x125$.

Aunque el método de gradiente conjugado es inestable con respecto a pequeñas perturbaciones, al aplicarlo de manera iterativa la solución x_{k+1} va mejorando monotónicamente hacia la solución exacta, la cual se puede alcanzar fijando una tolerancia relativamente pequeña.

En cuanto al costo de las operciones del método, se podría decir que es bajo ya que a excepción del producto matriz-vector Ap_k, de complejidad $O(n^2)$, las demás operaciones son productos punto, de orden $O(n)$.

Finalmente, Una mejora que se puede hacer al algoritmo es añadir una fase de precondicionamiento si el número de condición $\kappa(A)$ de la matriz A es grande. En este precondicionamiento se reemplaza el sistema original $Ax - b = 0$ por $M^{-1}(Ax - b) = 0$ tal que $\kappa(M^{-1}A) < \kappa(A)$.