Tarea 05 - Solvers iterativos, raices de funciones y eigensolvers

Isaac Rodríguez Bribiesca

Resumen Para esta tarea se implementaron dos solvers iterativos, Jacobi y Gauss-Seidel. También se implementaron dos métodos para encontrar raices de funciones, Newton-Raphson y bisección y su graficación con gnuplot. Finalmente, también se implementó el método de potencia para encontrar el valor y vector propio más grande de una matriz.

1. Solver iterativo por Jacobi

1.1. Descripción

El método de Jacobi es un algoritmo iterativo para encontrar la solución a un sistema lineal de n ecuaciones. Para este método sólo se asumen 2 cosas: (1) Que el sistema tiene una solución única y (2) Que los elementos de A en la diagonal principal no sean igual a cero.

Sea $A * x = b$ un sistema lineal de n ecuaciones con n incógnitas. Es decir:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

EL método de Jacobi itera sobre la ecuación:

$$x^k = D^{-1} * (b - (L + U) * x^{k-1}) \tag{1}$$

Donde D es una matriz diagonal con elementos $d_{ii} = a_{ii}$ con $i = 1, 2, ..., n$. L es una matriz triangular inferior con la diagonal principal igual a cero, es decir, $l_{ij} = a_{ij}$ con $j < i$, $i = 1, 2, ..., n$. U es una matriz triangular superior con la diagonal principal igual a cero, es decir, $u_{ij} = a_{ij}$ con $j > i$, $i = 1, 2, ..., n$. Por lo tanto, L, D y U están en realidad guardadas en la matriz A, o sea $A = L + D + U$.

Se detiene la iteración en uno de dos casos: (1) Se ha completado el máximo número de iteraciones especificado por el usuario o (2) El error entre la solución x^k y x^{k-1} es menor a una cierta tolerancia definida también por el usuario.

Se inician las iteraciones con:

$$x^0 = \begin{bmatrix} b1 \\ b2 \\ \vdots \\ b_n \end{bmatrix}$$

Si la matriz A es estrictamente diagonalmente dominante, se puede asegurar que el algoritmo va a converger para cualquier vector inicial x^0. Es decir, es una condición suficiente de convergencia.

1.2. Ejemplo de prueba

1.3. Resultados

Para el caso de la matriz de prueba M_BIG.txt se obtuvo la convergencia después de 2353 iteraciones, teniendo como condición que:

$$\frac{\left\|x^k - x^{k-1}\right\|}{\|x^k\|} < e \tag{2}$$

Con $e = 1 * 10^{-9}$

Una ventaja del método de Jacobi, es que se puede paralelizar fácilmente la obtención de x^k ya que el cálculo de la componente x_i^k con $i = 1, 2, ..., n$ no depende de x_j^k con $j \neq i$.

Aunque se tiene la desventaja de que en cada iteración del algoritmo se necesitan 2 vectores, uno para guardar la solución actual x^k y otro para la solución de la iteración anterior x^{k-1}.

2. Solver iterativo por Gauss-Seidel

El método de Gauss-Seidel también es un algoritmo iterativo para encontrar la solución a un sistema lineal de n ecuaciones y que asume los mismos 2 puntos mencionados en el método de Jacobi.

La diferencia es que en Gauss-Seidel se itera sobre la siguiente ecuación:

$$x^k = D^{-1} * (b - L * x^k - U * x^{k-1}) \tag{3}$$

Donde D es una matriz diagonal con elementos $d_{ii} = a_{ii}$ con $i = 1, 2, ..., n$. L es una matriz triangular inferior con la diagonal principal igual a cero, es decir, $l_{ij} = a_{ij}$ con $j < i$, $i = 1, 2, ..., n$. U es una matriz triangular superior con la diagonal principal igual a cero, es decir, $u_{ij} = a_{ij}$ con $j > i$, $i = 1, 2, ..., n$. Por lo tanto, L, D y U están en realidad guardadas en la matriz A, o sea $A = L + D + U$.

Se detiene la iteración en uno de dos casos: (1) Se ha completado el máximo número de iteraciones especificado por el usuario o (2) El error entre la solución x^n y x^{n-1} es menor a una cierta tolerancia definida también por el usuario.

Se inician las iteraciones con:

$$x^0 = \begin{bmatrix} b1 \\ b2 \\ \vdots \\ b_n \end{bmatrix}$$

Si la matriz A es estrictamente diagonalmente dominante o simétrica positiva definida, se puede asegurar que el algoritmo va a converger para cualquier vector inicial x^0. Es decir, son condiciones suficientes de convergencia.

A diferencia del método de Jacobi, donde el cálculo de x^k se hace usando sólamente las entradas de x^{k-1}, en Gauss-Seidel también usamos las entradas de x^k tan pronto como se calculan cada una de las x_i^k con $i = 2, 3, ..., n$. Por ejemplo para el cálculo de la entrada x_2^k se utiliza x_1^k en vez de x_1^{k-1}. La única excepción es para x_1^k donde forzosamente se tienen que usar x_2^{k-1}, ..., x_n^{k-1}.

2.1. Ejemplo de prueba

```
isaac  master U:1 ?:4  (c ) base  ~/Documents/CIMAT/Métodos Numéricos/programs/iterative_solvers/gaussSeidelSolver  61  make && ./runTest ../M_BIG.txt ../V_BIG.txt
gcc -c gaussSeidel_solver_test.c
gcc -c ../solve_iterative.c
gcc -c  ../matrix_struct.c
gcc -o runTest gaussSeidel_solver_test.o solve_iterative.o matrix_struct.o -lm

Read matrix A
Succesfully read: ../M_BIG.txt

Read vector b
Succesfully read: ../V_BIG.txt

**********

Convergence reached at iteration 1230

**********

Comparing A*x_solve with b...

Test PASSED. A * x_solve = b for given A and b.
```

2.2. Resultados

Para el caso de la matriz de prueba M_BIG.txt, se pudo observar que el número de iteraciones para encontrar la solución x fue menor usando el método Gauss-Seidel (1230 iteraciones) que usando el método de Jacobi (2353 iteraciones), lo cual es razonable ya que el método Gauss-Seidel incorpora las entradas x_i^k para el cálculo de las x_j^k con $i < j$, $j = 2, 3, ..., n$, por lo que al final de la iteración k esperamos que nuestra aproximación x^k sea mejor y la convergencia se obtenga en un menor número de iteraciones.

También se tiene la ventaja de no necesitar dos vectores de solución, x^k y x^{k-1}, ya que el segundo se puede ir guardando en el primero. Al inicio se de cada iteración x^k tiene la solución de la iteración anterior, mientras que al final la iteración, x^k tendrá la solución actual.

Sin embargo, el uso de las compnentes de x^k para encontrar la solución actual, hace que no sea posible paralelizar el cálculo de la solución.

3. Método de Bisección para raices de funciones

El método de bisección es un algoritmo para encontrar las raices de una función $f(x)$ de variable real x. Donde f es una función continua en el intervalo cerrado $[a, b]$ con $a < b$ y donde $f(a)$ y $f(b)$ tienen signos opuestos. Entonces f tendrá al menos una raíz en el intervalo cerrado $[a, b]$, por el teorema de Bolzano.

En cada iteración del algoritmo se calcula c, el punto intermedio del intervalo cerrado $[a, b]$, esto es:

$$c = \frac{a + b}{2} \tag{4}$$

Posteriormente se actualiza $a = c$ si se cumple que $f(c) * f(b) < 0$, de lo contrario, se actualiza $b = c$ si se cumple que $f(a) * f(c) < 0$.

El algoritmo se detiene si se cumple alguna de las siguientes condiciones:

1. $a - b < \epsilon$ con $\epsilon = 1 * 10^{-9}$ o algún valor definido por el usuario. En cuyo caso se regresa el valor $c = \frac{a+b}{2}$ como resultado.
2. $f(a) = 0$, $f(b) = 0$ o $f(c) = 0$, en cuyo caso se regresa a si $f(a) = 0$, b si $f(b) = 0$ y c si $f(c) = 0$
3. Se alcanza el límite de iteraciones definido por el usuario. En este caso, se imprime en pantalla que no se ha alcanzado la convergencia y se regresa el último valor c calculado.

Para el algoritmo implementado, antes de empezar las iteraciones se verifica que $f(a) * f(b) \leq 0$ y que $a < b$. Se acepta la igualdad $f(a) * f(b) = 0$ en la validación ya que esto implica que $f(a) = 0$ o $f(b) = b$ y por lo tanto $x = a$ o $x = b$ es una raíz de f en el intervalo cerrado $[a, b]$, lo cual se detecta dentro de la iteración.

Para apreciar visualmente el resultado, se muestra una gráfica de la función en el intervalo definido, y un punto con la ubicación de la raíz obtenida.

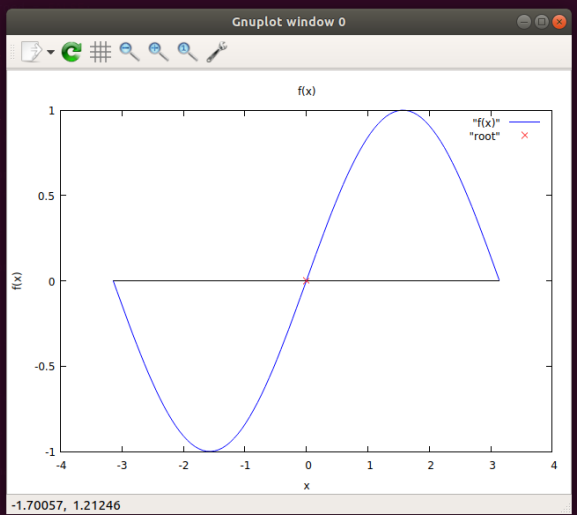
3.1. Ejemplo de prueba

Prueba con función $f(x) = x^2 - 2$ en intervalo $[-10, 10]$

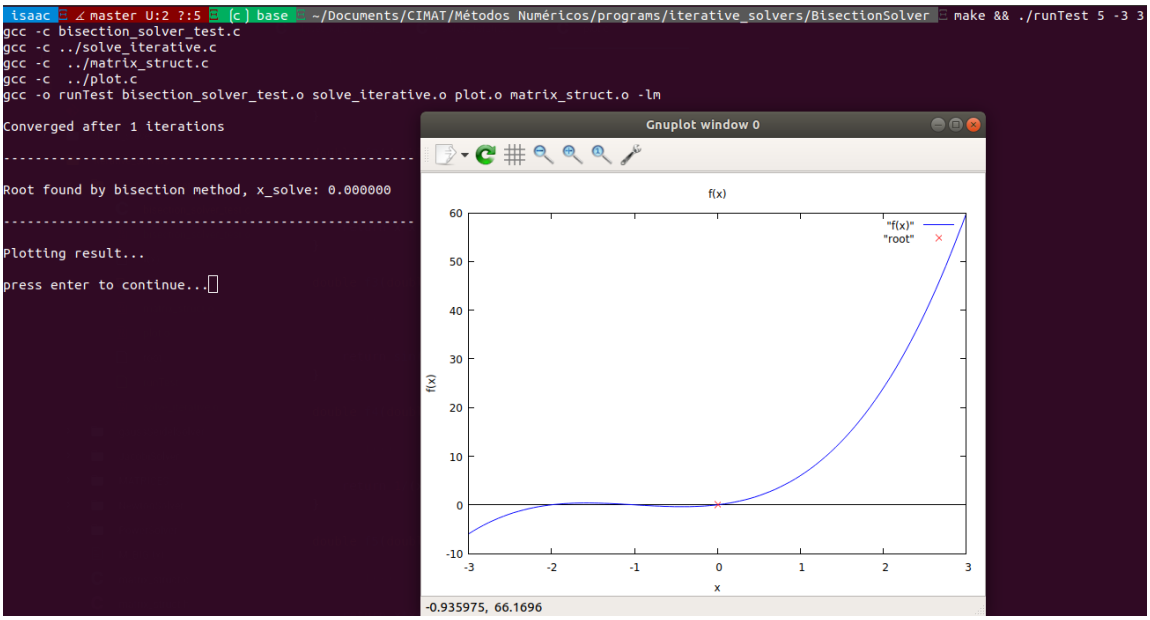
```
isaac  master U:2 ?:5  [c ] base  ~/Documents/CIMAT/Métodos Numéricos/programs/iterative_solvers/BisectionSolver  255  make && ./runTest 2 -10 10
gcc -c bisection_solver_test.c
gcc -c ../solve_iterative.c
gcc -c  ../matrix_struct.c
gcc -c  ../plot.c
gcc -o runTest bisection_solver_test.o solve_iterative.o plot.o matrix_struct.o -lm

Input interval not valid for Bisection algorithm
```

Prueba con función $f(x) = sin(x)$ en intervalo $[-\pi, \pi]$

```
isaac  master U:2 ?:5  [c ] base  ~/Documents/CIMAT/Métodos Numéricos/programs/iterative_solvers/BisectionSolver  make && ./runTest 3 -3.1415926536 3.1415926536
gcc -c bisection_solver_test.c
gcc -c ../solve_iterative.c
gcc -c  ../matrix_struct.c
gcc -c  ../plot.c
gcc -o runTest bisection_solver_test.o solve_iterative.o plot.o matrix_struct.o -lm

Converged after 1 iterations

-------------------------------------------------

Root found by bisection method, x_solve: 0.000000

-------------------------------------------------

Plotting result...

press enter to continue...
```



Prueba con función $f(x) = x^3 + 3 * x^2 + 2 * x$ en intervalo $[-3, 3]$



3.2. Resultados

La situación ideal para usar el algoritmo de bisección es cuando sabemos de antemano que la función $f(x)$ tiene sólo una raíz en el intervalo cerrado $[a, b]$ con $a < b$ y que $f(a) * f(b) < 0$.

Pero hay ejemplos en los que estas condiciones no se van a cumplir:

1. Que haya al menos una raíz en el intervalo $[a, b]$ pero que $f(a) > 0$ y $f(b) > 0$, en cuyo caso, no se podrá ejecutar el algoritmo.
2. Que haya más de una raíz en el intervalo $[a, b]$ con $f(a) * f(b) < 0$, en cuyo caso, se encontrará alguna de las raices del intervalo.
3. Que haya una asíntota dentro del intervalo $[a, b]$, como en la función $f(x) = \frac{1}{x-2}$ con $a = 0$ y $b = 4$, donde se cumple que $f(a) < 0$ y $f(b) > 0$ sin que haya una raíz en el intervalo $[0, 4]$. En este caso, la función se podría quedar indefinida si el valor medio c cae en la asíntota o que haya un overflow al evaluar f en un valor cercano a la asíntota.

Por lo que al usar este algoritmo, lo más recomendable es asegurarse de dar un intervalo aproximado donde sabemos que sólo hay una raíz.

4. Método de Newton-Raphson para raices de funciones

El método de Newton-Raphson también es un algoritmo para encontrar las raices de una función $f(x)$ de variable real x, donde f es una función diferenciable y cuya derivada se representará como $f'(x)$.

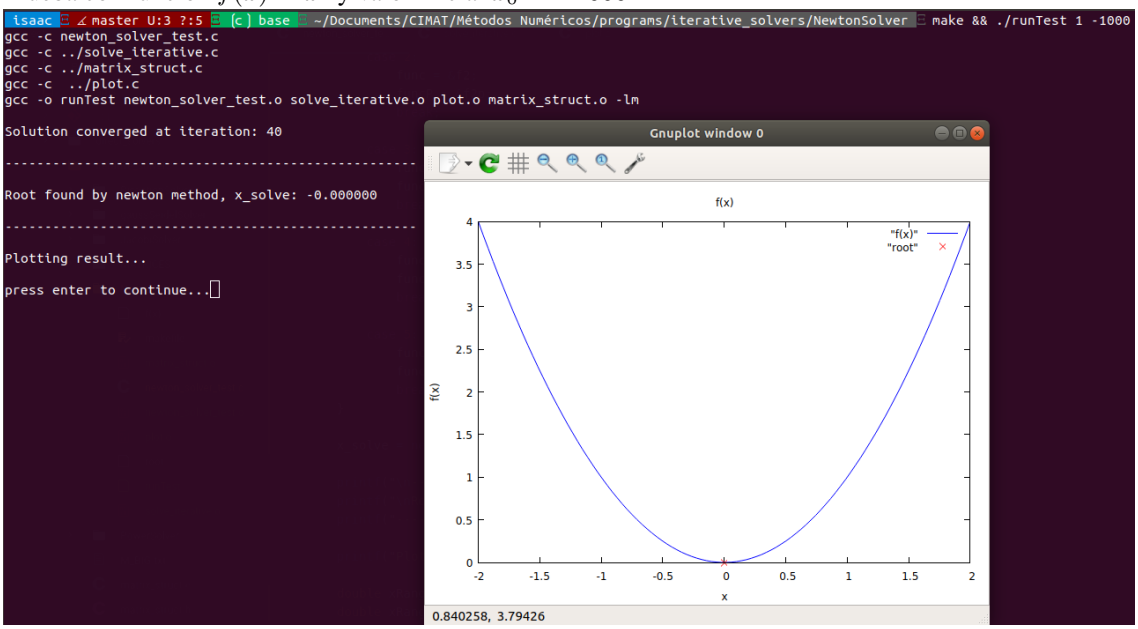
Se realiza una serie de iteraciones sobre la siguiente ecuación, donde n es la iteración actual:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5}$$

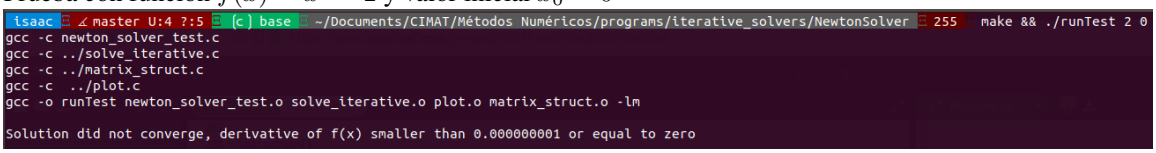
Para la primer iteración $n = 0$ se usa algún x_0, que llamaremos valor inicial, definido por el usuario, aunque de preferencia debería ser un valor cercano al lugar donde se encentra la raíz que buscamos.

4.1. Ejemplo de prueba

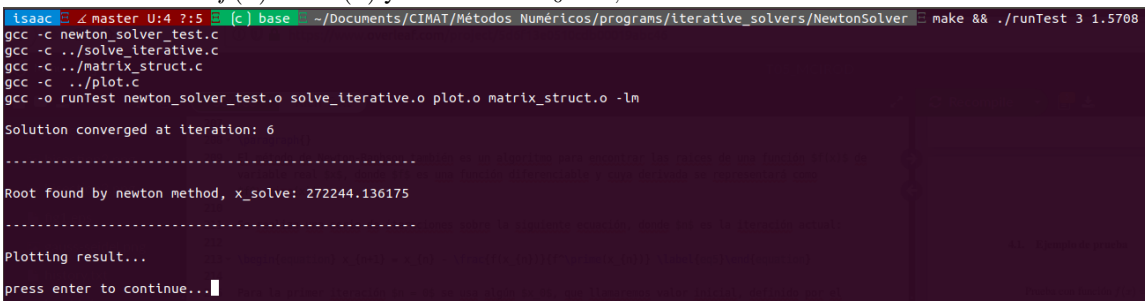
Prueba con función $f(x) = x^2$ y valor inicial $x_0 = -1000$



Prueba con función $f(x) = x^2 - 2$ y valor inicial $x_0 = 0$



Prueba con función $f(x) = sin(x)$ y valor inicial $x_0 = 1{,}5708$



4.2. Resultados

Para el caso del método de Newton-Raphson, la mejor opción de valor inicial x_0 es cuando este está lo más cercano posible a la raíz, con $f'(x_0) \neq 0$. Incluso si se toma un valor inicial tal que $f'(x_0)$ sea cercano a cero, se pueden presentar problemas. Como se observa en los resultados, al iterar sobre la ecuación $f(x) = sin(x)$ con un valor inicial $x_0 \approx \frac{\pi}{2}$ donde $f'(x_0) \approx 0$, la estimación que obtenemos se aleja demasiado del lugar donde empezamos y terminamos estimanto una raíz no deseada, ya que esperaríamos obtener la raíz más cercana a x_0, que sería $x = 0$ o $x = \pi$.

Una segunda desventaja de este metodo es la necesidad de tener que calcular la derivada directamente, ya sea que se tenga la función que implemente la derivada o que se use algún metodo numérico para estimarla.

También se puede correr el riesgo de entrar en un bucle donde se oscile entre un conjunto de puntos x y no se llegue a la convergencia. Un ejemplo de función donde se obtenga tal comportamiento de oscilación es $f(x) = \sqrt{|x|}$.

Una alternativa que se puede implementar para evitar el problema de que no se pueda calcular directamente la derivada, sería estimarla mediante la pendiente de la recta que se forma entre dos puntos de la función cercanos. Lo cual resultaría en el método de la secante.

5. Método de potencia

El método de iteración de potencia es un algoritmo para obtener el valor propio λ_1 más grande y su respectivo vector propio v_1 para una matriz diagonalizable A de nxn. Esto es, se encuentra λ_1 y v_1 tales que $A * v_1 = \lambda_1 * v_1$ con $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ y donde los vectores propios $v_1, ..., v_n$ son linealmente independientes. A v_1 y a λ_1 se les llama vectores y valores propios dominantes respectivamente.

La ecuación sobre la que se itera para encontrar el vector propio dominante v_1 es:

$$v^{k+1} = \frac{A * v^k}{\|A * v^k\|} \tag{6}$$

Empezando con un vector v^0 que tenga una componente diferente de cero en la dirección del vector propio v_1, lo cual se puede evitar construtendo v^0 de manera aleatoria o con componentes diferentes de cero.

Mientras que para el valor propio dominante λ_1 se itera sobre:

$$\lambda^{k+1} = \frac{(v^{k+1})^T * A * v^{k+1}}{(v^{k+1})^T * v^{k+1}} \tag{7}$$

Como el vector propio v^{k+1} está normalizado, el denominador será siempre igual a 1.

Asumiendo que λ_1 es dominante y que v^0 tiene componente diferente de cero en la dirección del vector propio dominante v_1, la convergencia del algoritmo está determinada por la proporción de $\frac{|\lambda_2|}{|\lambda_1|} < 1$, donde λ_2 es el segundo valor propio dominante de A.

Si $\frac{|\lambda_2|}{|\lambda_1|} \approx 1$ la convergencia será más lenta. Y si tienen la misma magnitud no se puede asegurar que el algoritmo llegue a la convergencia.

5.1. Ejemplo de prueba

```
isaac@irb ~/Documents/CIMAT/Métodos Numéricos/Numerical-Methods/iterative solvers/PowerSolver  master ● make && ./runTest ../M_BIG.txt
gcc -c power_solver_test.c
gcc -c ../solve_iterative.c
gcc -c  ../matrix_struct.c
gcc -c  ../plot.c
gcc -o runTest power_solver_test.o solve_iterative.o matrix_struct.o plot.o -lm

Read matrix A
Succesfully read: ../M_BIG.txt

Converged after 2 iterations

Dominant eigenvalue found: 1.000000

Dominant eigenvector found:

 -0.000000
 -0.000000
 0.288675
 -0.000000
 0.000000
 -0.000000
 -0.000000
 0.000000
 0.288675
 0.000000
 0.000000
 0.000000
 -0.000000
 -0.000000
 0.000000
 -0.000000
 0.000000
 0.000000
 -0.000000
 -0.000000
 0.000000
```

5.2. Resultados

Para verificar el resultado obtenido al iterar la matriz de ejemplo "M_BIG.txtçon el algoritmo implementado, primero se calcularon los valores y vectores propios de la matriz con el eigensolver de la biblioteca de python, numpy. Se pudo observar que dicha matriz tienen 12 valores propios con valor 1 y que estos son los más grandes. Mientras que los 12 respectivos vectores propios resultan ser canónicos, es decir, con sólo una componente igual a 1 y las demás igual a 0.

Al tratar de calcular el valor propio dominante de la matriz con el algoritmo implementado para esta tarea, vemos que si se obtiene el valor propio $\lambda_1 = 1$. Sin embargo, el vector propio asociado no logra converger, lo cual, como ya se comentó, no se puede asegurar cuando $\frac{|\lambda_2|}{|\lambda_1|} = 1$, como en este caso.

Para este algoritmo la operación más costosa que se realiza es la multiplicación de matriz por vector la cuál es de orden $O(n^2)$, por lo que podría ser de mayor utilidad en matrices ralas, ya que se puede implementar la muliplicación matriz-vector de manera más eficiente. Un ejemplo de esto es el algoritmo Page Rank de Google, donde se manejan matrices Markov con muchos ceros y de dimensiones del orden de miles de millones.

Finalmente, a pesar de que este algortimo sólo se puede usar para calcular el valor y vector propio dominante de una matriz, se puede extender y servir como base para otros algortimos más complejos como el método de potencia inversa para encontrar el valor y vector propio más pequeño, el método de Arnoldi para encontrar los k valores y vectores propios más grandes o el método QR para encontrar todos los eigenvectores.