

## Resumen

En este trabajo se realizaron distintos métodos para la solución de sistemas de ecuaciones y para encontrar los valores y vectores propios.

## 1. Introducción

El método de iteración en subespacio, es un método eficiente para encontrar los  $m$  valores propios más grandes o más chicos, además, permite disminuir la dificultad del problema y así también mejorar los errores numéricos, este método utiliza ideas de métodos anteriores, como el método de Jacobi y los métodos de la potencia y potencia inversa. Así, además de interesarse en los eigenvalores y eigenvectores, los métodos de resolución de sistemas de ecuaciones siguen mostrando un avance en cuanto a tiempos de ejecución y errores numéricos, lo que se mostrará más adelante.

## 2. Metodología

### 2.1. Iteración en subespacio

El método de iteración en subespacios, es un método que combina el método de Jacobi para encontrar eigenvalores y la idea del método de la potencia o potencia inversa. El objetivo de este método, es encontrar  $m$  valores propios en una matriz de dimensión  $n \times n$ , donde  $m \ll n$ . Por ejemplo, si se tiene una matriz de dimensión  $1000 \times 1000$ , y sólo se buscan los cinco valores propios más pequeños (o más grandes), la dimensión del problema se reduce a aplicar el método de Jacobi a una matriz de  $5 \times 5$ . Esto se calcula siguiendo la siguiente metodología.

Primero, se utiliza idea del método de la potencia, no el método en sí, es decir, este método itera utilizando un vector normalizado inicial y a cada iteración, el vector se acerca al vector propio más grande, en este caso, se aplica el método de la potencia haciendo una multiplicación matricial  $AX_0$ , donde  $X_0$  se inicializa con una matriz identidad no cuadrada. En términos de ecuaciones y dimensiones de matriz:

$$Z_{0 \times m} = A_{n \times n} X_{0 \times m}, \quad (1)$$

donde  $A$  es la matriz de interés, y  $X_0$  es una matriz con unos en  $(0,0), (1,1), (2,2), \dots, (m,m)$  y los términos restantes son iguales a cero. Segundo, se ortonormaliza la matriz  $Z$  con el método de Gram-Schmidt o con la factorización QR. Después, se aplica una rotación a la matriz  $A$  y es donde se baja por completo la dimensión de la matriz, la ventaja de esta reducción es que el método de Jacobi no tendrá que hacer tantas multiplicaciones, ya que para encontrar los eigenvectores, este método hace multiplicaciones matriciales consecutivas. Así, sea  $B$  la matriz a la que se le aplicará Jacobi:

$$B_{m \times m} = Z_{m \times n}^T A_{n \times n} Z_{n \times m}. \quad (2)$$

Finalmente, para iterar de nuevo y buscar que la matriz  $B$  se diagonalice, se rota la matriz de eigenvectores obtenida en Jacobi:

$$X_{n \times m} = Z_{n \times m} (\text{matriz eigenvectores}_{m \times m}), \quad (3)$$

y se itera hasta que el máximo de  $B$  (entre los términos distintos de la diagonal) sea menor a una tolerancia. En caso de buscar los  $m$  valores propios más pequeños, se utiliza la idea del método de la potencia inversa, y cambia sólo el primer paso, en lugar de hacer una multiplicación matricial, se resuelve sistemas de ecuaciones, donde la primera columna de  $X_0$  se utiliza como el término  $b$  para la solución  $Ax = b$  y la primera columna de  $Z_0$  sería la solución de  $x$  (así para todo  $m$ ), en este caso, se utilizó la factorización LU al inicio del programa para mejorar los tiempos de ejecución.

---

**Algorithm 1** Iteración en subespacio - método potencia

---

**Entrada:** Matriz A, Matriz X, y matrices auxiliares: Z, ZT, B, Matriz Jacobi, double maximo, int cantidad de eigenvalores, n cantidad de iteraciones, N dimensión de matriz A.

**Salida:** Los  $m$  eigenvalores más grandes.

- 1: **Función:** multiplicación matrices (matmul)
  - 2: **Función:** Gram-Schmidt ortonormalización(gram – schmidt)
  - 3: **Función:** Máximo(max)
  - 4: **Método de Jacobi para eigenvectores** (metodo – jacobi)
  - 5: **Función:** Matriz transpuesta (matriz – transpuesta)
  - 6: **for**  $i = 0; i < \text{iteraciones}, i++$  **do**
  - 7:      $Z = \text{matmul}(A, X)$
  - 8:      $Z = \text{gram – schmidt}(Z)$
  - 9:      $ZT = \text{matriz – transpuesta}(Z)$
  - 10:     $B = \text{matmul}(ZT, \text{matmul}(A, Z))$
  - 11:     $\text{maximo} = \max(B) \rightarrow$  condición de paro
  - 12:    metodo – jacobi(B)
  - 13:    Reemplazo:  $X = \text{matmul}(Z, \text{matriz – jacobi})$
  - 14: **retorna** Matriz B diagonalizada
- 

---

**Algorithm 2** Iteración en subespacio - método potencia inversa

---

**Entrada:** Matriz A, Matriz X, y matrices auxiliares: L, U, Z, ZT, B, Matriz Jacobi, double maximo, int cantidad de eigenvalores, n cantidad de iteraciones, N dimensión de matriz A.

**Salida:** Los  $m$  eigenvalores más pequeños.

- 1: **Función:** multiplicación matrices (matmul)
  - 2: **Función:** Gram-Schmidt ortonormalización(gram – schmidt)
  - 3: **Función:** Máximo(max)
  - 4: **Método de Jacobi para eigenvectores** (metodo – jacobi)
  - 5: **Función:** Matriz transpuesta (matriz – transpuesta)
  - 6: **Función:** Factorización LU (LU)
  - 7: **Función:** triangular superior (tri-sup)
  - 8: **Función:** triangular inferior (tri-inf)
  - 9: Factorizar  $LU(A)$ , se obtiene L, U.
  - 10: **for**  $i = 0; i < \text{iteraciones}, i++$  **do**
  - 11:      $Z = \text{tri-sup}(U, \text{tri-inf}(L, X))$
  - 12:      $Z = \text{gram – schmidt}(Z)$
  - 13:      $ZT = \text{matriz – transpuesta}(Z)$
  - 14:      $B = \text{matmul}(ZT, \text{matmul}(A, Z))$
  - 15:      $\text{maximo} = \max(B) \rightarrow$  condición de paro
  - 16:     metodo – jacobi(B)
  - 17:     Reemplazo:  $X = \text{matmul}(Z, \text{matriz – jacobi})$
  - 18: **retorna** Matriz B diagonalizada
- 

## 2.2. Cociente de Rayleigh

Este método consiste en obtener un valor propio (el más cercano al vector inicial dado) es un método iterativo y tiene una convergencia rápida. Para su deducción se parte de:

$$(A - I\sigma)\Phi = 0, \quad (4)$$

y si expresamos la ec.4, en otros términos:

$$(A - I\sigma_0)V_1 = V_0. \quad (5)$$

Además, como todos los vectores  $V_i$  son ortonormales entre sí, al multiplicar la ec.5 por la transpuesta del vector  $V_1$ , se obtiene una fórmula iterativa para mejorar la aproximación del valor propio:

$$\sigma_1 = \sigma_0 + \frac{V_1^T V_0}{V_1^T V_1}. \quad (6)$$

---

**Algorithm 3** Cociente de Rayleigh

---

**Entrada:** Vector inicial  $b$ , matriz  $A$ , double error, double TOL, matrices auxiliares  $L$ ,  $U$ .

**Salida:** Solución aproximada a un eigenvalor.

```
1: Función: matriz vector (matvec)
2: Función: Factorización LU (LU)
3: Función: triangular superior (tri-sup)
4: Función: triangular inferior (tri-inf)
5: while error > TOL do
6:    $x = \text{solve}(A, b)$  // Factorizando con LU, y usando tri-sup y tri-inf.
7:   Cálculo de sigma:  $x^T Ax / x * x$ 
8:    $x = \text{sigma} * x$ 
9:   Calcular el error
10:  Reemplazar la primera propuesta  $b$  por  $x$ .
11: retorna: Eigenvalor asociado
```

---

### 2.3. Factorización QR

Este es un método de factorización de matrices, donde la matriz  $Q$  es una matriz ortogonal, y la matriz  $R$  es una matriz triangular superior. Una ventaja de este método es que la matriz  $A$  no necesita ser cuadrada para funcionar. Esta es útil para resolver sistemas de ecuaciones lineales y se utiliza el proceso de ortogonalización de Gram-Schmidt. Además, este método es distinto porque el algoritmo utiliza más vectores columna que vectores filas, por ello es que aumenta su complejidad. También, una ventaja de este método, es que el costo computacional se reduce al resolver sistemas de ecuaciones, ya que la matriz  $Q^T$  es igual que la matriz  $Q^{-1}$ , y su solución es más directa.

---

**Algorithm 4** Factorización QR

---

**Entrada:** Matriz  $A$ , int filas, int columnas, matriz vacia  $Q$ , matriz vacia  $R$

**Salida:** Factorización de la matriz  $A$

```
1: Función: Copiar vectores columnas entre matrices.
2: Función: Multiplicar dos columnas de distintas matrices.
3: Función: Normalizar columna de matriz.
4: for  $i = 0; i < \text{cols}; i++$  do
5:   Copiar la columna  $A_i$  a la columna  $Q_i$ 
6:   for  $j = 0; j < i; j++$  do
7:      $R[j][i] = \text{Multiplicar las columnas (como producto escalar) de la columna } Q_j \text{ con la columna } A_i$ 
8:     Reemplazar las columnas de  $Q$ 
9:      $Q_i = Q_i - R[j][i] * Q_j$ 
10:   $R[i][i] = \text{Norma de la columna } Q_i$ 
11:  Modificar  $Q_i = Q_i / R[i][i]$ 
12: retorna:  $Q, R$ .
```

---

### 2.4. Método de gradiente conjugado

Este método es para resolver matrices simétricas definidas positivas, es un método iterativo, cuya idea se basa en reducir al mínimo la diferencia entre el resultado iterativo y la respuesta exacta. El algoritmo parte de:

$$x^{k+1} = x^k + \alpha^k p^k, \quad (7)$$

y el residuo que buscamos minimizar:

$$r^k = b - Ax^k \quad (8)$$

$$r^{k+1} = b - Ax^{k+1} \quad (9)$$

$$r^{k+1} - r^k = -\alpha^k Ap^k \quad (10)$$

después de desarrollar la última ecuación, y hacer que  $r^{k+1}$  se vaya a cero, se encuentra que el término  $\alpha$ , debería ser:

$$\alpha^k = \frac{p^{kT} r^k}{p^{kT} A p^k}. \quad (11)$$

---

**Algorithm 5** Gradiente conjugado

---

**Entrada:** Vectores auxiliares  $x_k$ ,  $x_0$ ,  $r$ ,  $\alpha$ , matriz  $A$

**Salida:** Solución aproximada de  $x$

- 1: **Función:** Vector por vector ( $vv$ ).
  - 2: **Función:** Multiplicación por vector ( $matvec$ ).
  - 3: **while** error  $\geq TOL$  **do**
  - 4:    $r = b - matvec(A, x_0)$
  - 5:    $\alpha = vv(r, r) / vv(r, matvec(A, x_0))$
  - 6:   Reemplazar  $x_0 = x_0 + \alpha * r$
  - 7:   Calcular el error
  - 8:   Aumentar la iteración
  - 9: **retorna:** vector  $x$
- 

## 2.5. Método de gradiente conjugado preconditionado

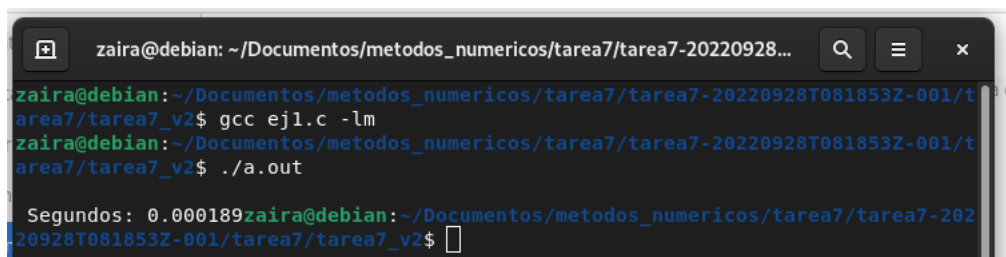
Este método es el gradiente conjugado, pero añadiendo la condición de preconditionado. En este caso, se mejora la convergencia del método. En lugar de escoger un vector inicial al azar, se toma como vector inicial los elementos diagonales de  $A$ , y se ingresa un nuevo parámetro  $y$  que depende de la inversa de la matriz. Así, al algoritmo, se añade:

$$y^{k+1} = M^{-1}y^k \quad (12)$$

que cambiará al parámetro  $\alpha$ .

## 3. Resultados

### 3.1. Subespacio - método potencia



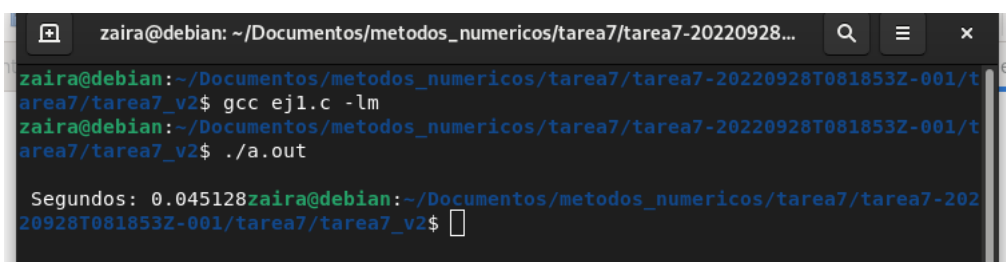
```
zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc ej1.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.000189zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 1: Para la matriz de 3x3 buscando dos eigenvalores (eigen\_3x3\_potencia.txt) .

Valor calculado en Python	Valor obtenido con subespacio y Jacobi
10.03479626	10.034794
6.97386041	6.973850

Cuadro 1: Comparación de resultados para la matriz 3x3.



```
zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc ej1.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.045128zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 2: Para la matriz de 50x50 buscando cinco eigenvalores.

Valor calculado en Python	Valor obtenido con subespacio y Jacobi
500.00154272	500.001543
490.00143117	490.001431
480.00052665	480.000366
470.00105808	470.001057
460.00126902	459.992985

Cuadro 2: Comparación de resultados para la matriz 50x50 (eigen\_50x50\_potencia.txt) .

### 3.2. Subespacio - potencia inversa

```

zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc ej2_v4.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.002884zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$

```

Figura 3: Para la matriz de 50x50 buscando cinco eigenvalores más pequeños.

Figura 4: Comparación de resultados para la matriz 50x50 (potencia inversa).

Valor calculado en Python	Valor obtenido con subespacio y Jacobi
2.99134332	2.991344
6.97386041	6.975432

Cuadro 3: Comparación de resultados para la matriz 50x50 (eigen\_50x50\_potencia\_inversa.txt) .

```

zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc ej2_v4.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.005267zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$

```

Figura 5: Para la matriz de 3x3 buscando dos eigenvalores más pequeños.

Figura 6: Comparación de resultados para la matriz 3x3 (potencia inversa).

Valor calculado en Python	Valor obtenido con subespacio y Jacobi
9.99805017	9.998050
19.99879386	19.998815
29.99893095	29.999350
39.99976329	40.002649
49.99836638	50.009731

Cuadro 4: Comparación de resultados para la matriz 3x3 (eigen\_3x3\_potencia\_inversa.txt) .

### 3.3. Coeficiente de Rayleigh

```
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out
2.991372
Segundos: 0.000079zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 7: Se encontró el eigenvalor más pequeño de la matriz de 3x3.

```
zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc ej3_v1.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out
9.998089
Segundos: 0.001264zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 8: Se encontró el eigenvalor más pequeño de la matriz de 50x50.

### 3.4. Factorización QR

```
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc qr_v2.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out
Segundos: 0.016838zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 9: Resolviendo un sistema de ecuaciones utilizando la factorización QR (x.125x125\_qr.txt).

```
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out
Segundos: 0.000214zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 10: Resultados de resolver un sistema de ecuaciones de 3x3 (x.3x3\_qr.txt).

### 3.5. Método de gradiente conjugado

```
zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc gradiente_conjugado.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out
Segundos: 0.000224zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$
```

Figura 11: Solución del sistema de ecuaciones 3x3 (x\_grad\_3x3r.txt).

```

zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc gradiente_conjugado.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.002858zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$

```

Figura 12: Solución del sistema de 125x125 (x\_grad\_125x125r.txt).

### 3.6. Gradiente conjugado precondicionado

```

zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc gradiente_conjugado_pre_v2.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.000730zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$

```

Figura 13: Solución del sistema de ecuacionex 3x3 (x\_grad\_pre\_125x125r.txt).

```

zaira@debian: ~/Documentos/metodos_numericos/tarea7/tarea7-20220928...
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ gcc gradiente_conjugado_pre_v2.c -lm
zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$ ./a.out

Segundos: 0.014248zaira@debian:~/Documentos/metodos_numericos/tarea7/tarea7-20220928T081853Z-001/tarea7/tarea7_v2$

```

Figura 14: Solución del sistema de 125x125 (x\_grad\_pre\_125x125r.txt).

### 3.7. Comparación de tiempos

Método	Tiempos de ejecución (seg) - dimensión matriz
Iteración en subespacio con método potencia	0.000189 (3x3)
Iteración en subespacio con método potencia inversa	0.045128 (50x50)
Factorización QR	0.002884 (3x3)
Coeficiente de Rayleigh	0.005267 (50x50)
Gradiente Conjugado	0.000214 (3x3)
Gradiente Conjugado Precondicionado	0.016838 (125x125)
	0.000079 (3x3)
	0.001264 (125x125)
	0.000224 (3x3)
	0.002858 (125x125)
	0.000730 (3x3)
	0.014248 (125x125)

Cuadro 5: Tabla de comparación de tiempos de ejecución.

## 4. Discusión y conclusión

Aunque el método de Jacobi sea un método sencillo de desarrollar, las ideas que conllevan son las más complicadas. Por ejemplo, la utilización de los métodos de la potencia sin hacer deflación, para obtener los m valores más grandes o más chicos. No obstante, el método de la potencia con deflación, funciona más rápido y

con poco error numérico al utilizar las mismas matrices. En el caso del método QR, este tuvo su complejidad en las operaciones entre columnas, ya que utilizar la columna incorrecta, y no normalizarla al final, llevaba errores numéricos muy grandes. Asimismo, los tiempos, y los resultados comparados con otros programas, mostraron buenos resultados.