Resolución Numérica de Sistemas de Ecuaciones Lineales

Elementos de Programación y Cálculo Numérico - 2025

Material de Estudio - Obligatorio

Introducción General

Los sistemas de ecuaciones lineales constituyen uno de los pilares fundamentales del análisis numérico y la computación científica moderna. Su resolución eficiente y precisa es esencial para abordar una amplia gama de problemas en ingeniería, ciencias físicas y matemáticas aplicadas. La importancia de estos sistemas radica en que muchos fenómenos naturales y procesos industriales pueden modelarse, después de una adecuada discretización, mediante relaciones lineales entre variables desconocidas.

En el contexto de la ingeniería, los sistemas de ecuaciones lineales emergen naturalmente en diversas disciplinas. En ingeniería estructural, el método de elementos finitos transforma las ecuaciones diferenciales que gobiernan el comportamiento mecánico de estructuras en sistemas lineales de gran escala, donde la matriz de rigidez relaciona los desplazamientos nodales con las fuerzas aplicadas. La resolución de estos sistemas permite predecir deformaciones, tensiones y verificar la seguridad estructural de edificios, puentes y componentes mecánicos.

En el ámbito de la ingeniería eléctrica, el análisis de circuitos complejos mediante las leyes de Kirchhoff conduce invariablemente a sistemas lineales donde las incógnitas son las corrientes de malla o los voltajes nodales. La eficiencia en la resolución de estos sistemas es crucial para el diseño de circuitos integrados, redes de distribución eléctrica y sistemas de potencia, donde pueden involucrar miles o millones de variables.

Los problemas de transferencia de calor, fundamentales en ingeniería térmica y mecánica, también se reducen a sistemas lineales tras la discretización espacial de las ecuaciones de difusión. La distribución de temperaturas en componentes industriales, intercambiadores de calor y sistemas de climatización se determina resolviendo estos sistemas, cuya estructura depende de la geometría y las condiciones de contorno del problema.

La dinámica de fluidos computacional, esencial en ingeniería mecánica y ambiental, requiere la solución de sistemas lineales masivos resultantes de la discretización de las ecuaciones de Navier-Stokes. Estos sistemas permiten simular el flujo alrededor y dentro de estructuras, el comportamiento de turbomaquinaria y la dispersión de contaminantes en el ambiente.

El desafío principal radica en que los sistemas reales de ingeniería frecuentemente involucran miles o millones de incógnitas, haciendo impracticable la aplicación de métodos analíticos tradicionales como la regla de Cramer, que requiere el cálculo de determinantes con un costo computacional factorial. Esta limitación motivó el desarrollo de métodos numéricos especializados que explotan las características particulares de cada tipo de

problema: matrices ralas para problemas de elementos finitos, matrices simétricas en problemas de conducción de calor, o matrices bien condicionadas en circuitos pasivos.

La elección del método de resolución adecuado—directo o iterativo—depende críticamente del tamaño del sistema, la estructura de la matriz de coeficientes, los recursos computacionales disponibles y la precisión requerida. Los métodos directos ofrecen soluciones exactas en aritmética infinita pero con un costo cúbico en el número de incógnitas, mientras que los métodos iterativos pueden ser más eficientes para sistemas grandes y estructurados, aunque requieren condiciones de convergencia específicas.

Este documento presenta un análisis integral de los principales métodos numéricos para la resolución de sistemas lineales, proporcionando las herramientas teóricas y algorítmicas necesarias para abordar eficientemente los desafíos computacionales que surgen en la práctica ingenieril moderna.

1. Sistemas Lineales

1.1. Planteamiento del Problema

En este documento estudiaremos la resolución numérica de sistemas de ecuaciones lineales (SEL) de la forma:

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

donde:

- $A \in \mathbb{R}^{n\times n}$ es una matriz cuadrada de coeficientes reales
- $\mathbf{x} \in \mathbb{R}^n$ es el vector incógnita
- $\mathbf{b} \in \mathbb{R}^n$ es el vector de términos independientes

Asumiremos que A es no singular ($\det(A) \neq 0$), garantizando la existencia y unicidad de la solución.

1.2. Relación con Álgebra Lineal

La resolución de SEL se fundamenta en conceptos clave del álgebra lineal:

Definición 1 (Sistema Compatible Determinado)**.** Un sistema $A\mathbf{x} = \mathbf{b}$ es compatible determinado si $\text{rang}(A) = \text{rang}(A|\mathbf{b}) = n$, donde $(A|\mathbf{b})$ es la matriz ampliada.

Definición 2 (Normas Matriciales)**.** Una norma matricial $\|\cdot\|$ es una función que asigna a cada matriz $A \in \mathbb{R}^{n\times n}$ un número real no negativo $\|A\|$ que satisface:

- $\|A\| \geq 0$ y $\|A\| = 0$ si y solo si $A = 0$
- $\|\alpha A\| = |\alpha|\|A\|$ para todo escalar α
- $\|A + B\| \leq \|A\| + \|B\|$ (desigualdad triangular)
- $\|AB\| \leq \|A\|\|B\|$ (submultiplicatividad)

Definición 3 (Normas Matriciales Inducidas)**.** Dada una norma vectorial $\|\cdot\|_p$, la norma matricial inducida se define como: $\|A\|_p = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} = \max_{\|\mathbf{x}\|_p=1} \|A\mathbf{x}\|_p$

Las normas matriciales más utilizadas son:

Norma-1 (norma columna): $\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{i,j}|$ Es la máxima suma absoluta de los elementos por columnas.

Norma-∞ (norma fila): $\|A\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{i,j}|$ Es la máxima suma absoluta de los elementos por filas.

Norma-2 (norma espectral): $\|A\|_2 = \sqrt{\rho(A^T A)} = \sigma_{\max}(A)$ donde $\rho(\cdot)$ es el radio espectral y $\sigma_{\max}(A)$ es el valor singular máximo de A.

Norma de Frobenius: $\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{i,j}|^2}$

Definición 4 (Condicionamiento)**.** El número de condición de una matriz A en la norma p se define como: $\kappa_p(A) = \|A\|_p \cdot \|A^{-1}\|_p$ y mide la sensibilidad de la solución ante perturbaciones en los datos. El subíndice p indica qué norma matricial se está utilizando.

1.3. Aplicaciones en Ingeniería

Los SEL aparecen frecuentemente en problemas de ingeniería:

Ejemplo 1 (Análisis Estructural)**.** En el método de elementos finitos para análisis de estructuras, la ecuación fundamental es:

$$K\mathbf{u} = \mathbf{f}$$

donde K es la matriz de rigidez, $\mathbf{u}$ son los desplazamientos nodales y $\mathbf{f}$ las fuerzas aplicadas.

Ejemplo 2 (Circuitos Eléctricos)**.** Aplicando las leyes de Kirchhoff a un circuito con n nodos, obtenemos:

$$G\mathbf{v} = \mathbf{i}$$

donde G es la matriz de conductancias, $\mathbf{v}$ los voltajes nodales e $\mathbf{i}$ las corrientes inyectadas.

Ejemplo 3 (Transferencia de Calor)**.** La discretización de la ecuación de difusión del calor lleva a sistemas de la forma:

$$A\mathbf{T} = \mathbf{q}$$

donde $\mathbf{T}$ representa las temperaturas en los nodos de la malla.

2. Clasificación de Métodos

2.1. Métodos Directos

Definición 5 (Método Directo)**.** Un método directo es aquel que obtiene la solución exacta (en aritmética exacta) del sistema en un número finito y conocido a priori de operaciones aritméticas.

Características:

- Ideales para sistemas relativamente"pequeños ($n < 10^4$)
- Eficientes para matrices densamente pobladas
- Costo computacional conocido: $O(n^3)$ operaciones
- No requieren condiciones de convergencia

2.2. Métodos Iterativos

Definición 6 (Método Iterativo)**.** Un método iterativo genera una sucesión $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ que converge a la solución exacta bajo ciertas condiciones.

Características:

- Ideales para sistemas grandes ($n > 10^4$) y matrices ralas
- Costo por iteración: $O(n^2)$ o menos para matrices estructuradas
- Requieren condiciones de convergencia
- Número de iteraciones a priori desconocido

3. Métodos Directos

3.1. Algoritmo de Descenso

Para sistemas con matriz triangular inferior $L\mathbf{x} = \mathbf{b}$:

Algorithm 1 Algoritmo de Descenso

Require: Matriz triangular inferior L, vector $\mathbf{b}$
Ensure: Vector solución $\mathbf{x}$

$x_1 \leftarrow b_1/l_{1,1}$
for $i = 2$ to n **do**
    suma $\leftarrow 0$
    for $j = 1$ to $i - 1$ **do**
        suma $\leftarrow$ suma $+ l_{i,j} \cdot x_j$
    end for
    $x_i \leftarrow (b_i - \text{suma})/l_{i,i}$
end for

Costo computacional: $O(n^2)$ operaciones.

3.2. Algoritmo de Remonte

Para sistemas con matriz triangular superior $U\mathbf{x} = \mathbf{b}$:

Algorithm 2 Algoritmo de Remonte

Require: Matriz triangular superior U, vector $\mathbf{b}$
Ensure: Vector solución $\mathbf{x}$

$x_n \leftarrow b_n/u_{n,n}$
for $i = n - 1$ down to 1 **do**
    suma $\leftarrow 0$
    for $j = i + 1$ to n **do**
        suma $\leftarrow$ suma $+ u_{i,j} \cdot x_j$
    end for
    $x_i \leftarrow (b_i - \text{suma})/u_{i,i}$
end for

3.3. Eliminación de Gauss

La eliminación de Gauss transforma el sistema original en uno triangular superior equivalente.

Algorithm 3 Eliminación de Gauss

Require: Matriz A, vector $\mathbf{b}$
Ensure: Sistema triangular superior

```
Formar matriz aumentada Ã = [A|b]
for k = 1 to n − 1 do                                   ▷ Columnas
    if ã_{k,k} = 0 then
        Error: Pivote nulo
    end if
    for i = k + 1 to n do                               ▷ Filas debajo del pivote
        l_{i,k} ← ã_{i,k}/ã_{k,k}                       ▷ Multiplicador
        for j = k to n + 1 do                           ▷ Toda la fila
            ã_{i,j} ← ã_{i,j} − l_{i,k} · ã_{k,j}
        end for
    end for
end for
Aplicar remonte al sistema resultante
```

Condición de aplicabilidad: Las submatrices principales A_i de orden $i = 1, \ldots, n-1$ deben ser no singulares.

Costo computacional: $\frac{2n^3}{3} + O(n^2)$ operaciones.

3.4. Pivoteo

Cuando $\tilde{a}_{k,k}$ es pequeño o nulo, es necesario realizar pivoteo:

Definición 7 (Pivoteo Parcial)**.** Intercambiar la fila k con la fila p tal que:

$$p = \arg \max_{k \leq i \leq n} |\tilde{a}_{i,k}|$$

El sistema resultante es $PA\mathbf{x} = P\mathbf{b}$, donde P es la matriz de permutaciones.

3.5. Factorización LU

Definición 8 (Factorización LU)**.** Descomponer la matriz A como el producto de una matriz triangular inferior L y una triangular superior U:

$$A = LU$$

donde L tiene unos en la diagonal principal.

Algorithm 4 Factorización LU

Require: Matriz A
Ensure: Matrices L y U

```
Aplicar eliminación de Gauss solo a la matriz A
U ← matriz triangular superior resultante
L ← matriz con multiplicadores l_{i,k} en el triángulo inferior y unos en la diagonal
```

Resolución del sistema:

1. Resolver $L\mathbf{y} = \mathbf{b}$ (descenso)
2. Resolver $U\mathbf{x} = \mathbf{y}$ (remonte)

Ventajas:

- Eficiente para múltiples sistemas con la misma matriz A
- Cálculo directo del determinante: $\det(A) = \prod_{i=1}^{n} u_{i,i}$

3.6. Factorización de Cholesky

Para matrices simétricas y definidas positivas:

Definición 9 (Factorización de Cholesky)**.** Si A es simétrica y definida positiva, entonces existe una única matriz triangular inferior L con elementos diagonales positivos tal que:

$$A = LL^T$$

Algorithm 5 Factorización de Cholesky

Require: Matriz simétrica definida positiva A
Ensure: Matriz triangular inferior L

$l_{1,1} \leftarrow \sqrt{a_{1,1}}$
for $i = 2$ to n **do**
  $l_{i,1} \leftarrow a_{i,1}/l_{1,1}$
end for
for $j = 2$ to $n-1$ **do**
  $l_{j,j} \leftarrow \sqrt{a_{j,j} - \sum_{k=1}^{j-1} l_{j,k}^2}$
  for $i = j+1$ to n **do**
    $l_{i,j} \leftarrow \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} l_{j,k}\right) / l_{j,j}$
  end for
end for
$l_{n,n} \leftarrow \sqrt{a_{n,n} - \sum_{k=1}^{n-1} l_{n,k}^2}$

Ventajas:

- Costo: $\frac{n^3}{3}$ operaciones (la mitad que LU)
- Ahorro de memoria: solo se almacena L
- Estabilidad numérica garantizada

4. Condicionamiento

Definición 10 (Número de Condición)**.** El número de condición de una matriz A en la norma p se define como:

$$\kappa_p(A) = \|A\|_p \cdot \|A^{-1}\|_p$$

Para matrices simétricas definidas positivas: $\kappa_2(A) = \frac{\lambda_{\text{máx}}}{\lambda_{\text{mín}}}$ donde $\lambda_{\text{máx}}$ y $\lambda_{\text{mín}}$ son los autovalores máximo y mínimo de A, respectivamente.

Ejemplo 4 (Cálculo de Normas Matriciales)**.** Para la matriz $A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$:

Norma-1: $\|A\|_1 = \text{máx}\{|2| + |-1|, |-1| + |2|\} = \text{máx}\{3, 3\} = 3$
Norma-∞: $\|A\|_\infty = \text{máx}\{|2| + |-1|, |-1| + |2|\} = \text{máx}\{3, 3\} = 3$
Norma de Frobenius: $\|A\|_F = \sqrt{2^2 + (-1)^2 + (-1)^2 + 2^2} = \sqrt{10}$
Norma-2: Como A es simétrica, los autovalores son $\lambda_1 = 3$, $\lambda_2 = 1$, entonces: $\|A\|_2 = \lambda_{\text{máx}} = 3$

Ejemplo 5 (Interpretación del Número de Condición)**.** Consideremos el sistema $A\mathbf{x} = \mathbf{b}$ con: $A = \begin{pmatrix} 1 & 1 \\ 1 & 1{,}0001 \end{pmatrix}$, $\mathbf{b} = \begin{pmatrix} 2 \\ 2{,}0001 \end{pmatrix}$

La solución exacta es $\mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Si perturbamos $\mathbf{b}$ ligeramente a $\mathbf{b}' = \begin{pmatrix} 2 \\ 2{,}0002 \end{pmatrix}$, la nueva solución es $\mathbf{x}' = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$.

El número de condición $\kappa_2(A) \approx 40000$ explica por qué una perturbación relativa de $\frac{0{,}0001}{2{,}0001} \approx 5 \times 10^{-5}$ en $\mathbf{b}$ produce un error relativo de $\frac{\|\mathbf{x}' - \mathbf{x}\|}{\|\mathbf{x}\|} \approx 1$ en la solución.

Teorema 1 (Cota de Error)**.** Si $A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$, entonces:

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(A) \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

Un sistema se considera:

- **Bien condicionado** si $\kappa(A) \approx 1$
- **Mal condicionado** si $\kappa(A) \gg 1$

5. Métodos Iterativos

5.1. Formulación General

Definición 11 (Método de Punto Fijo)**.** Reescribir el sistema $A\mathbf{x} = \mathbf{b}$ en la forma:

$$\mathbf{x} = G\mathbf{x} + \mathbf{c}$$

donde G es la matriz de iteración y $\mathbf{c}$ un vector constante.

Descomposición fundamental: $A = M - N$ donde M es fácilmente invertible.

Entonces:

$$A\mathbf{x} = \mathbf{b} \tag{2}$$
$$(M - N)\mathbf{x} = \mathbf{b} \tag{3}$$
$$M\mathbf{x} = N\mathbf{x} + \mathbf{b} \tag{4}$$
$$\mathbf{x} = M^{-1}N\mathbf{x} + M^{-1}\mathbf{b} \tag{5}$$

Por lo tanto: $G = M^{-1}N$ y $\mathbf{c} = M^{-1}\mathbf{b}$.

5.2. Formulación Eficiente

Como $N = M - A$:

$$\mathbf{x}^{(k+1)} = M^{-1}(M - A)\mathbf{x}^{(k)} + M^{-1}\mathbf{b} \tag{6}$$
$$= \mathbf{x}^{(k)} - M^{-1}A\mathbf{x}^{(k)} + M^{-1}\mathbf{b} \tag{7}$$
$$= \mathbf{x}^{(k)} + M^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \tag{8}$$
$$= \mathbf{x}^{(k)} + M^{-1}\mathbf{r}^{(k)} \tag{9}$$

donde $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$ es el residuo en la iteración k.

Algorithm 6 Esquema Iterativo General

Require: Matriz A, vector $\mathbf{b}$, aproximación inicial $\mathbf{x}^{(0)}$, tolerancia ε, máximo de iteraciones $N_{\text{máx}}$
Ensure: Vector solución aproximada $\mathbf{x}$

```
k ← 0
r^(0) ← b − Ax^(0)
while k < N_máx and ‖r^(k)‖ > ε‖b‖ do
    Resolver Mz = r^(k)
    x^(k+1) ← x^(k) + z
    r^(k+1) ← b − Ax^(k+1)
    k ← k + 1
end while
if k = N_máx then
    Error: No se alcanzó convergencia
end if
```

5.3. Criterios de Parada

Definición 12 (Error Absoluto)**.**

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \varepsilon$$

Definición 13 (Error Relativo)**.**

$$\frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}^{(k)}\|} < \varepsilon$$

Definición 14 (Criterio del Residuo)**.**

$$\frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} < \varepsilon$$

donde $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$.

5.4. Método de Jacobi

Descomposición: $A = D - L - U$ donde:

- D: matriz diagonal de A
- L: parte triangular inferior estricta de A (cambiada de signo)
- U: parte triangular superior estricta de A (cambiada de signo)

Elección: $M = D$, $N = L + U$
Matriz de iteración: $G_J = D^{-1}(L + U)$

Algorithm 7 Método de Jacobi

Require: Matriz A, vector $\mathbf{b}$, aproximación inicial $\mathbf{x}^{(0)}$
Ensure: Sucesión convergente $\{\mathbf{x}^{(k)}\}$

while no converge **do**
  for $i = 1$ to n **do**
    suma $\leftarrow 0$
    for $j = 1$ to n, $j \neq i$ **do**
      suma $\leftarrow$ suma $+ a_{i,j} \cdot x_j^{(k)}$
    end for
    $x_i^{(k+1)} \leftarrow (b_i - \text{suma})/a_{i,i}$
  end for
  $k \leftarrow k + 1$
end while

Formulación por componentes:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right)$$

5.5. Método de Gauss-Seidel

Elección: $M = D - L$, $N = U$
Matriz de iteración: $G_{GS} = (D - L)^{-1}U$
La diferencia clave con Jacobi es que se utilizan los valores ya actualizados en la misma iteración.

Algorithm 8 Método de Gauss-Seidel

Require: Matriz A, vector $\mathbf{b}$, aproximación inicial $\mathbf{x}^{(0)}$
Ensure: Sucesión convergente $\{\mathbf{x}^{(k)}\}$

while no converge **do**
  for $i = 1$ to n **do**
    suma1 $\leftarrow 0$
    for $j = 1$ to $i - 1$ **do**
      suma1 $\leftarrow$ suma1 $+ a_{i,j} \cdot x_j^{(k+1)}$ ▷ Valores nuevos
    end for
    suma2 $\leftarrow 0$
    for $j = i + 1$ to n **do**
      suma2 $\leftarrow$ suma2 $+ a_{i,j} \cdot x_j^{(k)}$ ▷ Valores viejos
    end for
    $x_i^{(k+1)} \leftarrow (b_i - \text{suma1} - \text{suma2})/a_{i,i}$
  end for
  $k \leftarrow k + 1$
end while

Formulación por componentes:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right)$$

5.6. Método de Relajación (SOR)

Definición 15 (Método SOR)**.** El método de Sobre-Relajación Sucesiva introduce un parámetro de relajación $\omega > 0$:

Elección: $M = \frac{1}{\omega} D - L$, $N = \frac{1-\omega}{\omega} D + U$
Matriz de iteración:

$$G_\omega = (D - \omega L)^{-1}[(1 - \omega)D + \omega U]$$

Formulación eficiente:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega(D - \omega L)^{-1}\mathbf{r}^{(k)}$$

Observación 1. Para $\omega = 1$, el método SOR se reduce al método de Gauss-Seidel.

5.7. Velocidad de Convergencia

La velocidad de convergencia es un aspecto crucial en el análisis de métodos iterativos, ya que determina la eficiencia práctica del algoritmo.

Definición 16 (Convergencia Lineal)**.** Una sucesión $\{\mathbf{x}^{(k)}\}$ converge linealmente a $\mathbf{x}^*$ si existe una constante $0 < C < 1$ tal que: $\lim_{k\to\infty} \frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|}{\|\mathbf{x}^{(k)} - \mathbf{x}^*\|} = C$ La constante C se denomina factor de convergencia.

Teorema 2 (Velocidad de Convergencia y Radio Espectral)**.** Para un método iterativo lineal $\mathbf{x}^{(k+1)} = G\mathbf{x}^{(k)} + \mathbf{c}$, si $\rho(G) < 1$, entonces: $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \|G\|^k \|\mathbf{x}^{(0)} - \mathbf{x}^*\|$ y la velocidad de convergencia está determinada por $\rho(G)$.

Interpretación práctica:

- Cuanto menor sea $\rho(G)$, más rápida será la convergencia
- El número de iteraciones para reducir el error inicial en un factor ϵ es aproximadamente: $k \approx \frac{\ln(\epsilon)}{\ln(\rho(G))}$

Ejemplo 6 (Comparación de Velocidades)**.** Para un sistema con las siguientes características:

- Método de Jacobi: $\rho(G_J) = 0{,}9$
- Método de Gauss-Seidel: $\rho(G_{GS}) = 0{,}81$
- Método SOR óptimo: $\rho(G_\omega) = 0{,}5$

Para reducir el error inicial en un factor de 10^{-6}:

- Jacobi: $k \approx \frac{\ln(10^{-6})}{\ln(0{,}9)} \approx 131$ iteraciones
- Gauss-Seidel: $k \approx \frac{\ln(10^{-6})}{\ln(0{,}81)} \approx 66$ iteraciones
- SOR óptimo: $k \approx \frac{\ln(10^{-6})}{\ln(0{,}5)} \approx 20$ iteraciones

Observación 2 (Relación entre Jacobi y Gauss-Seidel)**.** Para matrices simétricas y definidas positivas, se cumple la relación: $\rho(G_{GS}) = [\rho(G_J)]^2$ Esto explica por qué Gauss-Seidel generalmente converge más rápido que Jacobi.

Definición 17 (Convergencia Superlineal y Cuadrática)**.**

- **Convergencia superlineal:** $\lim_{k\to\infty} \frac{\|\mathbf{x}^{(k+1)}-\mathbf{x}^*\|}{\|\mathbf{x}^{(k)}-\mathbf{x}^*\|} = 0$
- **Convergencia cuadrática:** $\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C\|\mathbf{x}^{(k)} - \mathbf{x}^*\|^2$

Los métodos de Jacobi, Gauss-Seidel y SOR presentan convergencia lineal. Los métodos más avanzados como gradiente conjugado o GMRES pueden alcanzar convergencia superlineal bajo ciertas condiciones.

6. Convergencia de Métodos Iterativos

Teorema 3 (Condición de Convergencia)**.** Un método iterativo lineal $\mathbf{x}^{(k+1)} = G\mathbf{x}^{(k)} + \mathbf{c}$ es convergente si y solo si:

$$\rho(G) < 1$$

donde $\rho(G) = \max_i |\lambda_i|$ es el radio espectral de G.

6.1. Condiciones Suficientes de Convergencia

Teorema 4 (Diagonal Dominante Estricta)**.** Si A es estrictamente diagonal dominante por filas:

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|, \quad \forall i = 1, \ldots, n$$

entonces los métodos de Jacobi y Gauss-Seidel convergen.

Teorema 5 (Matrices Simétricas Definidas Positivas)**.** Si A es simétrica y definida positiva, entonces:

- Los métodos de Jacobi y Gauss-Seidel convergen
- El método SOR converge para $\omega \in (0, 2)$

7. Ejemplos Numéricos

Ejemplo 7 (Sistema 3×3)**.** Resolver el sistema:

$$\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix}$$

Verificación de convergencia: La matriz es simétrica, definida positiva y diagonal dominante, por lo que todos los métodos iterativos convergen.

Aplicación de Jacobi:

$$x_1^{(k+1)} = \frac{3 + x_2^{(k)}}{4}$$

$$x_2^{(k+1)} = \frac{2 + x_1^{(k)} + x_3^{(k)}}{4}$$

$$x_3^{(k+1)} = \frac{3 + x_2^{(k)}}{4}$$

Ejemplo 8 (Factorización LU)**.** Para la matriz:

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix}$$

Aplicando eliminación de Gauss:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

8. Consideraciones Computacionales

8.1. Complejidad Algorítmica

Método	Factorización	Resolución
Eliminación de Gauss	$\frac{2n^3}{3}$	n^2
Factorización LU	$\frac{2n^3}{3}$	$2n^2$
Factorización de Cholesky	$\frac{n^3}{3}$	$2n^2$
Jacobi (por iteración)	-	$2n^2$
Gauss-Seidel (por iteración)	-	$2n^2$

Cuadro 1: Complejidad computacional de los métodos

8.2. Elección del Método

Métodos Directos: Preferibles cuando:

- $n < 10^4$ (sistemas pequeños a medianos)
- Matrices densas
- Se requiere solución exacta
- Se resuelven pocos sistemas

Métodos Iterativos: Preferibles cuando:

- $n > 10^4$ (sistemas grandes)
- Matrices ralas con estructura conocida
- Memoria limitada
- Se acepta solución aproximada

9. Algoritmos Especializados

9.1. Algoritmo de Thomas

Para matrices tridiagonales de la forma:

$$\begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ c_2 & a_2 & b_2 & \cdots & 0 \\ 0 & c_3 & a_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & c_n & a_n \end{pmatrix}$$

Algorithm 9 Algoritmo de Thomas

Require: Vectores **a**, **b**, **c**, **d** de la matriz tridiagonal
Ensure: Vector solución **x**

Fase de eliminación:
$\alpha_1 \leftarrow a_1$, $\beta_1 \leftarrow b_1$
for $i = 2$ to n **do**
    $\gamma_i \leftarrow c_i/\alpha_{i-1}$
    $\alpha_i \leftarrow a_i - \gamma_i\beta_{i-1}$
    if $i < n$ **then**
        $\beta_i \leftarrow b_i$
    end if
end for
Resolución hacia adelante:
$y_1 \leftarrow d_1$
for $i = 2$ to n **do**
    $y_i \leftarrow d_i - \gamma_i y_{i-1}$
end for
Resolución hacia atrás:
$x_n \leftarrow y_n/\alpha_n$
for $i = n - 1$ down to 1 **do**
    $x_i \leftarrow (y_i - \beta_i x_{i+1})/\alpha_i$
end for

Costo computacional: $O(n)$ operaciones, muy eficiente para matrices tridiagonales.

10. Implementación y Consideraciones Prácticas

10.1. Estabilidad Numérica

Definición 18 (Estabilidad Numérica)**.** Un algoritmo es numéricamente estable si pequeñas perturbaciones en los datos de entrada producen pequeñas perturbaciones en la solución.

Consideraciones importantes:

- La eliminación de Gauss sin pivoteo puede ser inestable
- La factorización de Cholesky es inherentemente estable
- Los métodos iterativos son generalmente más estables ante errores de redondeo

10.2. Criterios de Implementación

Observación 3 (Pivoteo)**.** En la implementación práctica, siempre debe considerarse el pivoteo parcial para garantizar estabilidad numérica, especialmente cuando:

- Los elementos diagonales son pequeños
- La matriz no cumple condiciones específicas de convergencia
- Se requiere máxima precisión

10.3. Optimizaciones para Matrices Estructuradas

Matrices banda: Para matrices con ancho de banda p, la complejidad se reduce a $O(np^2)$.

Matrices simétricas: Solo se almacena y opera con la mitad de los elementos.

11. Implementación en Python

Esta sección presenta los comandos y bibliotecas de Python más relevantes para trabajar con sistemas de ecuaciones lineales, desde la generación de matrices hasta la resolución de sistemas utilizando diferentes métodos.

11.1. Bibliotecas Necesarias

```
import numpy as np
import scipy.linalg as la
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt
```

11.2. Generación de Matrices

11.2.1. Matrices Generales

```
# Matriz aleatoria
A = np.random.rand(n, n)  # Elementos entre [0,1)
A = np.random.randn(n, n) # Distribución normal estándar

# Matriz desde lista/array
A = np.array([[2, -1, 0],
              [-1, 2, -1],
              [0, -1, 2]])

# Matriz de ceros y unos
A = np.zeros((n, n))      # Matriz de ceros
A = np.ones((n, n))       # Matriz de unos
```

11.2.2. Matrices Diagonales

```
# Matriz identidad
I = np.eye(n)             # Matriz identidad n×n

# Matriz diagonal con elementos específicos
d = [1, 2, 3, 4]
D = np.diag(d)            # Matriz diagonal con d en la diagonal

# Extraer diagonal de una matriz existente
diagonal = np.diag(A)     # Extrae elementos diagonales de A
```

```
# Matriz diagonal dominante (útil para convergencia)
n = 5
A = np.random.rand(n, n)
A = A + np.diag(np.sum(np.abs(A), axis=1))  # Hace diagonal dominante
```

11.2.3. Matrices n-diagonales

```
# Matriz tridiagonal
def tridiagonal_matrix(n, a, b, c):
    """
    Genera matriz tridiagonal n×n
    a: diagonal inferior, b: diagonal principal, c: diagonal superior
    """
    A = np.zeros((n, n))
    A += np.diag(a * np.ones(n-1), -1)  # Diagonal inferior
    A += np.diag(b * np.ones(n), 0)     # Diagonal principal
    A += np.diag(c * np.ones(n-1), 1)   # Diagonal superior
    return A

# Ejemplo: matriz tridiagonal 5×5
A_tri = tridiagonal_matrix(5, -1, 2, -1)

# Matrices pentadiagonales (5 diagonales)
def pentadiagonal_matrix(n, a, b, c, d, e):
    """
    a: diagonal -2, b: diagonal -1, c: diagonal 0
    d: diagonal +1, e: diagonal +2
    """
    A = np.zeros((n, n))
    A += np.diag(a * np.ones(n-2), -2)
    A += np.diag(b * np.ones(n-1), -1)
    A += np.diag(c * np.ones(n), 0)
    A += np.diag(d * np.ones(n-1), 1)
    A += np.diag(e * np.ones(n-2), 2)
    return A

# Matriz n-diagonal general
def n_diagonal_matrix(n, diagonals, offsets):
    """
    n: tamaño de la matriz
    diagonals: lista de valores para cada diagonal
    offsets: lista de posiciones de las diagonales
    """
    A = np.zeros((n, n))
    for diag, offset in zip(diagonals, offsets):
        if offset >= 0:
            A += np.diag(diag * np.ones(n-offset), offset)
        else:
            A += np.diag(diag * np.ones(n+offset), offset)
```

```
    return A

# Ejemplo: matriz 7-diagonal
diags = [-1, -0.5, 2, 4, 2, -0.5, -1]
offsets = [-3, -2, -1, 0, 1, 2, 3]
A_7diag = n_diagonal_matrix(10, diags, offsets)
```

11.3. Resolución de Sistemas Lineales

11.3.1. Métodos Generales

```
# Método más simple y eficiente (recomendado)
x = np.linalg.solve(A, b)

# Usando scipy (más opciones de control)
x = la.solve(A, b)

# Método manual usando factorización LU
LU, piv = la.lu_factor(A)
x = la.lu_solve((LU, piv), b)

# Verificación de la solución
residual = np.linalg.norm(A @ x - b)
print(f"Residual: {residual}")
```

11.3.2. Métodos Específicos

```
# Para matrices simétricas definidas positivas (Cholesky)
def solve_cholesky(A, b):
    L = la.cholesky(A, lower=True)
    y = la.solve_triangular(L, b, lower=True)
    x = la.solve_triangular(L.T, y, lower=False)
    return x

# Para matrices triangulares
x = la.solve_triangular(L, b, lower=True)   # Matriz triangular inferior
x = la.solve_triangular(U, b, lower=False)  # Matriz triangular superior

# Para matrices tridiagonales (algoritmo de Thomas)
def thomas_algorithm(a, b, c, d):
    """
    Resuelve sistema tridiagonal usando algoritmo de Thomas
    a: diagonal inferior, b: diagonal principal, c: diagonal superior
    d: vector de términos independientes
    """
    n = len(d)
    # Eliminación hacia adelante
    for i in range(1, n):
        factor = a[i-1] / b[i-1]
```

```
        b[i] = b[i] - factor * c[i-1]
        d[i] = d[i] - factor * d[i-1]

    # Sustitución hacia atrás
    x = np.zeros(n)
    x[n-1] = d[n-1] / b[n-1]
    for i in range(n-2, -1, -1):
        x[i] = (d[i] - c[i] * x[i+1]) / b[i]

    return x
```

11.3.3. Matrices Ralas (Sparse)

```
# Creación de matrices ralas
from scipy.sparse import diags, csr_matrix

# Matriz tridiagonal rala
diagonals = [np.ones(n-1), -2*np.ones(n), np.ones(n-1)]
A_sparse = diags(diagonals, [-1, 0, 1], shape=(n, n), format='csr')

# Resolución de sistemas ralos
x = spsolve(A_sparse, b)

# Para matrices simétricas definidas positivas ralas
from scipy.sparse.linalg import spsolve
from scikits.sparse.cholmod import cholesky

# Factorización de Cholesky para matrices ralas
factor = cholesky(A_sparse)
x = factor.solve_A(b)
```

11.4. Análisis de Condicionamiento

```
# Número de condición
cond_num = np.linalg.cond(A)          # Norma-2
cond_num_1 = np.linalg.cond(A, 1)     # Norma-1
cond_num_inf = np.linalg.cond(A, np.inf)  # Norma-infinito

# Normas matriciales
norm_1 = np.linalg.norm(A, 1)         # Norma-1
norm_2 = np.linalg.norm(A, 2)         # Norma-2
norm_inf = np.linalg.norm(A, np.inf)  # Norma-infinito
norm_fro = np.linalg.norm(A, 'fro')   # Norma de Frobenius

# Autovalores y valores singulares
eigenvals = np.linalg.eigvals(A)
singular_vals = np.linalg.svd(A, compute_uv=False)

# Radio espectral
```

```
spectral_radius = max(abs(eigenvals))
```

11.5. Ejemplos Completos

Ejemplo 9 (Sistema 3×3 con Diferentes Métodos). `import numpy as np`
`import scipy.linalg as la`

```
# Definir el sistema
A = np.array([[4, -1, 0],
              [-1, 4, -1],
              [0, -1, 4]], dtype=float)
b = np.array([3, 2, 3], dtype=float)

# Solución exacta
x_exact = np.linalg.solve(A, b)
print(f"Solución exacta: {x_exact}")

# Factorización LU
P, L, U = la.lu(A)
print(f"Matriz L:\n{L}")
print(f"Matriz U:\n{U}")

# Factorización de Cholesky
L_chol = la.cholesky(A, lower=True)
print(f"Cholesky L:\n{L_chol}")

# Verificar: A = L @ L.T
print(f"Verificación Cholesky: {np.allclose(A, L_chol @ L_chol.T)}")

# Métodos iterativos
x0 = np.zeros(3)
x_jacobi, iter_j = jacobi(A, b, x0)
x_gs, iter_gs = gauss_seidel(A, b, x0)

print(f"Jacobi: {x_jacobi} en {iter_j} iteraciones")
print(f"Gauss-Seidel: {x_gs} en {iter_gs} iteraciones")

# Análisis de convergencia
eigenvals_jacobi = np.linalg.eigvals(np.linalg.inv(np.diag(np.diag(A))) @
                                     (np.diag(np.diag(A)) - A))
rho_jacobi = max(abs(eigenvals_jacobi))
print(f"Radio espectral Jacobi: {rho_jacobi}")
```

Ejemplo 10 (Matriz Tridiagonal Grande). `import numpy as np`
`import time`

```
n = 1000
# Crear matriz tridiagonal del operador diferencial -d²/dx²
A = tridiagonal_matrix(n, -1, 2, -1)
```

```
b = np.ones(n)  # Vector de términos independientes

# Método directo
start_time = time.time()
x_direct = np.linalg.solve(A, b)
time_direct = time.time() - start_time

# Algoritmo de Thomas (específico para tridiagonales)
a = -np.ones(n-1)  # Diagonal inferior
c = -np.ones(n-1)  # Diagonal superior
d = 2 * np.ones(n) # Diagonal principal

start_time = time.time()
x_thomas = thomas_algorithm(a, d, c, b)
time_thomas = time.time() - start_time

print(f"Tiempo método directo: {time_direct:.4f}s")
print(f"Tiempo algoritmo Thomas: {time_thomas:.4f}s")
print(f"Error relativo: {np.linalg.norm(x_direct - x_thomas)/np.linalg.norm(x_direct)

# Verificación de la estructura tridiagonal
print(f"Número de elementos no nulos: {np.count_nonzero(A)}")
print(f"Densidad de la matriz: {np.count_nonzero(A)/(n**2)*100:.2f}%")
```

Ejemplo 11 (Comparación de Velocidades de Convergencia)**.** `import matplotlib.pyplot as plt`

```
# Sistema bien condicionado
A = np.array([[3, 1], [1, 2]], dtype=float)
b = np.array([4, 3], dtype=float)
x_exact = np.linalg.solve(A, b)

# Seguimiento de la convergencia
def track_convergence(method, A, b, x0, max_iter=50):
    errors = []
    x = x0.copy()

    for k in range(max_iter):
        if method == 'jacobi':
            x, _ = jacobi(A, b, x, max_iter=1)
        elif method == 'gauss_seidel':
            x, _ = gauss_seidel(A, b, x, max_iter=1)

        error = np.linalg.norm(x - x_exact)
        errors.append(error)

        if error < 1e-10:
            break

    return errors
```

```
x0 = np.array([0., 0.])
errors_j = track_convergence('jacobi', A, b, x0)
errors_gs = track_convergence('gauss_seidel', A, b, x0)

# Gráfico de convergencia
plt.figure(figsize=(10, 6))
plt.semilogy(errors_j, 'b-o', label='Jacobi')
plt.semilogy(errors_gs, 'r-s', label='Gauss-Seidel')
plt.xlabel('Iteración')
plt.ylabel('Error ||x - x*||')
plt.title('Comparación de Velocidades de Convergencia')
plt.legend()
plt.grid(True)
plt.show()

# Radios espectrales
G_j = np.linalg.inv(np.diag(np.diag(A))) @ (np.diag(np.diag(A)) - A)
G_gs = np.linalg.inv(np.diag(np.diag(A)) - np.tril(A, -1)) @ np.triu(A, 1)

rho_j = max(abs(np.linalg.eigvals(G_j)))
rho_gs = max(abs(np.linalg.eigvals(G_gs)))

print(f"Radio espectral Jacobi: {rho_j:.6f}")
print(f"Radio espectral Gauss-Seidel: {rho_gs:.6f}")
print(f"Relación (G_GS) / [(G_J)]²: {rho_gs / rho_j**2:.6f}")
```

12. Conclusiones

La elección del método apropiado para resolver sistemas de ecuaciones lineales depende de múltiples factores:

- **Tamaño del sistema**: Métodos directos para $n < 10^4$, iterativos para sistemas mayores
- **Estructura de la matriz**: Métodos especializados para matrices con estructura especial
- **Condicionamiento**: Sistemas mal condicionados requieren técnicas de regularización
- **Recursos computacionales**: Memoria disponible y capacidad de cómputo paralelo
- **Precisión requerida**: Métodos directos para máxima precisión, iterativos para aproximaciones

Los métodos presentados forman la base fundamental para la resolución numérica de sistemas lineales y constituyen herramientas esenciales en la práctica computacional de la ingeniería y las ciencias aplicadas.