
**Universidad de la República**

Facultad de Ingeniería

10 de septiembre de 2024

# Álgebra Lineal Numérica

## Notas del curso

Florencia Uslenghi

Pablo Ezzatti

Ernesto Dufrechou

# Índice

**1. Repaso de Matrices** **7**

1.1. Vectores . . . 7

1.1.1. Dependencia Lineal . . . 8

1.1.2. Espacios vectoriales . . . 9

1.2. Matrices . . . 10

1.2.1. Operaciones matriciales . . . 10

1.2.2. Normas . . . 11

1.2.3. Determinante . . . 12

1.2.4. Matrices particulares . . . 13

1.2.5. Valores y Vectores propios . . . 16

1.2.6. Sistemas de ecuaciones . . . 18

**2. Errores y representaciones** **19**

2.1. Representaciones . . . 19

2.1.1. Representación de punto fijo . . . 19

2.1.2. Representación de punto flotante . . . 20

**3. Matrices y Máquinas** **27**

3.1. Órdenes . . . 27

3.2. Evaluación de Desempeño . . . 28

3.3. Máquina de Von Neumann . . . 29

3.4. Pipeline . . . 31

3.5. Máquinas Vectoriales . . . 32

3.6. Encadenamiento . . . 33

3.7. Estrategias de Compilación . . . 33

3.8. Jerarquías de memoria . . . 35

3.8.1. Principio de localidad . . . 36

3.8.2. Memoria Cache . . . 37

3.8.3. Memoria virtual . . . 37

**4. Matrices Dispersas** **39**
4.1. Representación en memoria de una matriz densa . . . 40
4.2. Representación de matrices dispersas . . . 40
4.2.1. Formatos estáticos . . . 41
4.2.2. Formatos dinámicos . . . 47
4.2.3. Otros formatos . . . 48

**5. High Performance Computing (HPC)** **49**
5.1. Arquitecturas paralelas . . . 49
5.1.1. Categorización de Flynn . . . 50
5.1.2. Conectividad entre procesadores y factores determinantes de la eficiencia . . . 53
5.2. Modelo de Programación Paralela . . . 55
5.2.1. Modelo conceptual de paralelismo . . . 55
5.2.2. Problemas con la computación paralela-distribuida . . . 57
5.3. Técnicas de programación paralela . . . 58
5.3.1. Descomposición de dominio . . . 58
5.3.2. Descomposición funcional . . . 59
5.3.3. Modelos híbridos . . . 60
5.3.4. Paralelización de aplicaciones existentes . . . 61
5.4. Medidas de performance . . . 62
5.4.1. Speed-Up . . . 63
5.4.2. Escalabilidad . . . 65
5.4.3. Utilización de Recursos Disponibles . . . 66
5.5. Scheduling o Mapeo . . . 66
5.6. Balance de Cargas . . . 67
5.6.1. Técnicas de balance de cargas . . . 68
5.6.2. Parámetros relevantes para la determinación de la carga . . . 68
5.7. Herramientas . . . 68
5.7.1. Multi-Threading . . . 68
5.8. OpenMP . . . 71
5.9. Message Passing Interface (MPI) . . . 72

**6. Sistemas Lineales** **79**
6.1. Métodos directos . . . 79
6.1.1. Resolución de sistemas particulares . . . 79
6.1.2. Eliminación Gaussiana . . . 80
6.1.3. Factorización LU . . . 81
6.2. Errores . . . 85

**7. Bibliotecas** **89**
7.1. Historia . . . 89
7.2. Basic Linear Algebra Subprograms (BLAS) . . . 90
7.3. Linear Algebra PACKage (LAPACK) . . . 91

**8. Sistemas Lineales Dispersos** **93**
8.1. Factorización de matrices dispersas . . . 93
8.2. Sistemas dispersos y grafos . . . 94
8.2.1. Árbol de eliminación . . . 96
8.3. Etapas de resolución de matrices dispersas . . . 96
8.3.1. Ordenamientos . . . 97
8.3.2. Factorización Simbólica . . . 102
8.4. Factorización numérica . . . 104
8.4.1. Método frontal . . . 104
8.4.2. Método multifrontal . . . 107
8.4.3. Bibliotecas . . . 109

**9. Métodos Iterativos Básicos** **111**

**10. Métodos Estacionarios** **113**
10.1. Método de Jacobi . . . 114
10.2. Método de Gauss-Seidel . . . 115
10.3. Comparación de Jacobi - Gauss-Seidel . . . 115
10.4. Método SOR (Succesive Over-Relaxation) . . . 115

**11. Métodos No Estacionarios** **117**
11.1. Métodos Basados en Subespacios de Krylov . . . 117
11.1.1. Propiedades de los subespacios de Krylov . . . 117
11.1.2. Métodos de descenso . . . 119
11.1.3. Método del máximo descenso o de descenso por gradiente . . . 120
11.1.4. Método del gradiente conjugado . . . 121
11.1.5. Generalized Minimal Residual (GMRES) . . . 124
11.1.6. Método del Gradiente Bi-Conjugado (BiCG) . . . 124
11.1.7. Método del Gradiente Conjugado Cuadrático . . . 125
11.2. Método del Gradiente Bi-Conjugado Estabilizado (BiCGSTAB) . . . 126

**12. Precondicionadores** **127**
12.1. Precondicionadores implícitos . . . 127
12.1.1. Método de Richardson Precondicionado . . . 127

12.1.2. Precondicionador de Jacobi . . . 128
12.1.3. Método de Gauss-Seidel y SOR . . . 128
12.1.4. Método del Gradiente jugado Precondicionado . . . 128
12.1.5. Factorizaciones Incompletas . . . 129
12.2. Precondicionadores Explícitos . . . 132
12.2.1. Precondicionadores Polinomiales . . . 132
12.3. Inversas aproximadas . . . 133

**13. Valores y Vectores propios** **135**
13.1. Resolución de la ecuación característica . . . 135
13.1.1. Método de las potencias . . . 135
13.1.2. Iteración inversa . . . 137
13.1.3. Deflación . . . 138
13.2. Transformaciones de semejanza . . . 138
13.3. Iteración QR . . . 139
13.3.1. Iteración QR con desplazamiento . . . 140
13.3.2. Extensión del método de las potencias . . . 140
13.4. Métodos basados en subespacios de Krylov . . . 141
13.4.1. Iteración de Arnoldi . . . 141
13.4.2. Iteración de Lanczos . . . 142
13.4.3. Reinicio Implícito . . . 142
13.5. Método Strum . . . 143

**14. Descomposición SVD** **145**
14.1. Propiedades de los valores singulares . . . 146
14.2. Aproximación de $A$ por $A_k$ . . . 147
14.3. Métodos para calcular la descomposición SVD en matrices densas . . . 147
14.3.1. Método de Golub-Kahan-Reinsch . . . 147
14.3.2. Otros métodos . . . 148
14.4. Métodos para calcular la descomposición SVD en matrices dispersas . . . 148
14.5. Aplicaciones de la descomposición SVD . . . 149
14.5.1. Técnicas de regularización . . . 149
14.5.2. Cálculo de inversa . . . 149
14.5.3. Mínimos cuadrados . . . 149
14.5.4. Compresión de imágenes . . . 150

# Capítulo 1

# Repaso de Matrices

Este capítulo es un repaso de conceptos que ya han sido abordados a lo largo de la carrera pero que será útil tenerlos presentes durante el resto del curso. El abordaje de los conceptos en estas notas será superficial y no del todo riguroso, priorizando el sentido práctico de cada tema, sin detenerse en la demostración de propiedades matemáticas o detalles que se pueden abordar con más profundidad en un curso básico de álgebra lineal.

## 1.1. Vectores

**Definición 1.1.0.1** (Vector)**.** *Un* ***vector fila*** $x = [x_1, \cdots, x_n]$ *de dimensión* $n$ *es un conjunto ordenado de* $n$ *escalares para el cual están definidas ciertas operaciones (suma, resta, módulo, producto escalare, producto vectorial, etc.)*

Se pueden utilizar para representar diversas cosas:

- Valores de una magnitud física en distintas componentes del espacio
- Variables de estado o salida de un sistema

#### Construcciones

**Producto interno:** También es llamado *producto escalar* ya que devuelve un escalar producto de multiplicar componente a componente los vectores de entrada y sumar el resultado.

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i' = \alpha \tag{1.1}$$

En general, en álgebra se suele utilizar la convención de que todos los vectores son columnas, por lo que el producto escalar se puede denotar como $x^T y$ (un vector fila por un vector columna). El producto escalar posee un significado geométrico relacionado con el ángulo entre los dos vectores:

$$\langle x, y \rangle = \|x\| \cdot \|y\| \cdot \cos\theta \tag{1.2}$$

En particular, $\langle x, y\rangle y$ es la proyección del vector $x$ sobre la dirección $y$, o la componente de $x$ en la dirección de $y$. Esto también relaciona el producto escalar con el concepto de **ortogonalidad**.

**Definición 1.1.0.2** (Vectores ortogonales)**.** *Dos vectores $v_1$ y $v_1$ se dicen **ortogonales** si $\langle v_1, v_2\rangle = 0$*

En relación con la interpretación anterior, si dos vectores son ortogonales, la proyección de uno sobre la dirección del otro es el vector nulo.

**Definición 1.1.0.3** (Conjuntos ortogonales)**.** *Un conjunto $V = \{v_1, v_2, \cdots, v_n\}$ se dice **ortogonal** si $\langle v_i, v_j\rangle = 0$ si $i \neq j$*

**Producto externo:** El producto externo de dos vectores $xy^T$ es una matriz que representa los productos de cada coordenada de $x$ por una coordenada de $y$:

$$xy^T = \begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{pmatrix} \tag{1.3}$$

Puede verse como el producto de una matriz con una sola columna, por otra matriz con una sola fila. No hay que confundir este producto con el producto vectorial (también llamado *cross product* y representado por el operador $\times$), que produce como resultado un vector y tiene otro significado. Sin ser muy riguroso, se puede decir que este tipo de matrices está relacionado con las factorizaciones matriciales. Suelen aparecer cuando se tiene una matriz o una factorización, e iterativamente se le realizan pequeños cambios para llegar a otra matriz final.

Es sencillo ver que cada columna o fila de la matriz $xy^T$ es un múltiplo de las demás. Esto se relaciona con el siguiente concepto.

### 1.1.1. Dependencia Lineal

Los siguientes conceptos están estrechamente relacionados:

**Definición 1.1.1.1** (Combinación lineal)**.** *Dado un conjunto de vectores $V = \{v_1, v_2, \cdots, v_n\}$, una combinación lineal de estos vectores es otro vector $u$ tal que para ciertos escalares $\alpha_i$ con $i = \{1, 2, \cdots, n\}$:*

$$u = \sum_{i=1}^{n} \alpha_i v_i \tag{1.4}$$

**Definición 1.1.1.2** (Dependencia lineal)**.** *Se dice que el conjunto de vectores $V = \{v_1, v_2, \cdots, v_n\}$ es **linealmente dependiente** si y solo si existen $a_1, a_2, \cdots, a_n \in \mathbb{C}$ no todos nulos ($a_i \neq 0$ para algún $i$) tales que:*

$$a_1v_1 + a_2v_2 + \cdots + a_nv_n = 0 \tag{1.5}$$

Esto equivale a decir que cada vector del conjunto puede escribirse como una combinación lineal del resto.

**Definición 1.1.1.3** (Independencia lineal)**.** *Se dice que el conjunto de vectores* $V = \{v_1, v_2, \cdots, v_n\}$ *es linealmente independiente si no es* ***linealmente independiente*** *o en otras palabras si dados* $a_1, a_2, \cdots, a_n \in \mathbb{C}$ *que cumplan:*

$$a_1 v_1 + a_2 v_2 + \cdots + a_n v_n = 0 \tag{1.6}$$

*entonces* $a_1 = a_2 = \cdots = a_n = 0$.

### 1.1.2. Espacios vectoriales

Si variamos los coeficientes de una combinación lineal de vectores para obtener todas las (infinitas) combinaciones posibles, obtenemos un **espacio vectorial**.

**Definición 1.1.2.1** (Espacio vectorial)**.** *Se dice que* $\{V, \mathbb{K}, +, \cdot\}$*, donde* $V \neq \emptyset$ *es un conjunto de vectores y* $\mathbb{K}$ *es un conjunto de escalares, es un* ***espacio vectorial*** *si están definidas las operaciones de suma y producto por un escalar* $\forall u, v, w \in V$ *tal que se cumplan las siguientes propiedades para la suma:*

- *Asociativa:* $(u + v) + w = u + (v + w)$
- *Conmutativa:* $u + v = v + u$
- *Neutro* $\exists \mathcal{O}, \mathcal{O} + u = u, \forall u \in V$ *y además se cumple que este neutro es único.*
- *Opuesto:* $\forall u \in V \exists - u \in V, u + (-u) = \mathcal{O}$

*y siendo* $\alpha, \beta \in \mathbb{K}$ *se cumple para el producto por un escalar:*

- $\alpha \cdot (\beta \cdot u) = (\alpha \cdot \beta) \cdot u$
- $\exists \infty \in \mathbb{K}, \infty u = u, \forall u \in V$

**Definición 1.1.2.2** (Subespacio vectorial)**.** *Sea* $V$ *espacio vectorial y* $\mathbb{K}$ *el cuerpo de escalares. Se dice que* $S$ *es un* ***subespacio vectorial*** *de* $V$ *si cumple las siguientes propiedades:*

- $S \neq \emptyset$
- *Si* $\forall u, v \in S \implies (u + v) \in S$
- *Si* $\forall \alpha \in \mathbb{K}$ *y* $\forall u \in S \implies \alpha \cdot u \in S$

*Es decir, que el conjunto sea no vacío y cerrado bajo suma y producto.*

**Definición 1.1.2.3** (Base y dimensión)**.** *Se dice que un conjunto* $\{v_1, \cdots, v_n\}$ *es* ***base*** *del espacio vectorial* $V$ *si se cumple:*

- *Los vectores $v_1, \cdots, v_n$ son linealmente independientes.*
- *Todo elemento de $V$ es una combinación lineal de elementos del conjunto.*

  *La cantidad de elementos necesarios para formar una base se denomina **dimensión** del espacio vectorial. Si el subespacio al que nos referimos es el determinado por las filas o columnas de una matriz, la dimensión del subespacio coincide con el **rango** de la matriz.*

## 1.2. Matrices

Las matrices ya se han mencionado varias veces. Además de definirlas formalmente, en esta sección nos detenemos en un detalle fundamental de estas construcciones. Primero una definición:

**Definición 1.2.0.1** (Matriz)**.** *Una **matriz** de dimensiones $m \times n$ es un conjunto de escalares ordenados en una grilla de $m$ filas y $n$ columnas:*

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

*con $a_{ij}$ nos referimos al elemento de la fila $i$ y columna $j$ de la matriz $A$.*

Las matrices representan transformaciones lineales de un espacio vectorial en otro. Por ejemplo, si tomamos la combinación lineal de la definición 1.1.1.1, y colocamos los vectores en $V$ como las columnas de una matriz, y organizamos los coeficientes $\alpha_i$ en un vector $\mathbf{a}$, podemos escribir la combinación lineal anterior como $u = V\mathbf{a}$. De esta forma una matriz $A$ transforma todos los vectores de entrada $x$ pertenecientes a un subespacio, en vectores de salida $y$ pertenecientes a otro. Estos subespacios no tienen por qué tener la misma dimensión (la matriz puede no ser cuadrada o tener algunas columnas linealmente dependientes).

**Definición 1.2.0.2** (Matriz transpuesta y transpuesta conjugada)**.** *Se define la **matriz transpuesta** de $A$ y se nota como $A^T$: $a^T_{ij} = a_{ji}$.*

*Es decir, se intercambian filas y columnas de la matriz $A$.*

*A su vez se define la **matriz transpuesta conjugada** y se denota por $A^*$ o $A^H$: $a^H_{ij} = \bar{a_{ji}}$*

### 1.2.1. Operaciones matriciales

**Definición 1.2.1.1** (Suma y multiplicación de matrices)**.** *Se define la **suma** de las matrices $((a_{ij}))$ y $((b_{ij}))$ como:*

$$s_{ij} = a_{ij} + b_{ij}$$

*Mientras que la **multiplicación** se define como:*

$$m_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Otra forma de ver el producto de una matriz $A$ con columnas $a_1, \cdots, a_p$ por una matriz $B$ con filas $b_1^T, \cdots, b_p^T$ es usando el producto externo definido anteriormente:

$$AB = \sum_{i=1}^{p} a_i b_i^T \tag{1.7}$$

Es fácil observar que se trata de los mismos productos y sumas, realizados en distinto orden. Sin embargo ver el producto de esta manera ilustra lo que se comentaba anteriormente sobre el producto externo (sobre que es útil para construir matrices iterativamente). En este caso se construye el producto $AB$ en $p$ iteraciones (sumando $p$ matrices).

**Propiedades de operaciones matriciales**

- $A \cdot (B + C) = AB + AC$
- $A + B = B + A$
- $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- $\alpha A \cdot B = A \cdot (\alpha B) = A \cdot B\alpha$
- $(AB)^T = B^T A^T$

### 1.2.2. Normas

Si tenemos dos vectores, por ejemplo $u = (21 - 3)^T$ y $v = (3 - 24)^T$: ¿cuál de ellos es "mayor¿. Es difícil definir un criterio satisfactorio para comparar los dos vectores directamente. Podríamos compararlos componente a componente, pero nos encontraríamos con el problema de que la primer componente de $u$ es menor que la primera de $v$ mientras que en la segunda componente ocurre lo inverso. Comparar la magnitud de vectores y matrices es la principal utilidad de las normas.

**Definición 1.2.2.1** (Norma)**.** *Se define la **norma** como una función $\|\cdot\| : V \times V \to \mathbb{K}$, es decir cuyo dominio es un espacio de vectores, matrices u otros y cuyo codominio es un cuerpo de escalares, que cumple las siguientes condiciones:*

- $\|x\| \geq 0 \ \forall x \in V$
- $\|x\| = 0 \iff x = 0$
- $\|\alpha x\| = |\alpha| \cdot \|x\|$ *con* $x \in V, \ \alpha \in \mathbb{K}$
- $\|x + y\| \geq \|x\| + \|y\|$ *con* $x, y \in V$

Podemos definir infinitas funciones que cumplan las propiedades de una norma, pero hay algunas normas particularmente útiles en ciertos contextos.

**Normas-p**

La norma "p."está dada por la ecuación

$$\|x\|_p = \left( \sum_{j=1}^{n} |x_j|^p \right)^{\frac{1}{p}}$$

Sustituyendo $p$ por distintos valores obtenemos normas que son muy utilizadas en distintos contextos. Por ejemplo, con $p = 2$ obtenemos la norma Euclídea. Si definimos $p = 1$, la norma pasa a ser simplemente la suma de los componentes del vector. Esta norma es conocida como "norma Manhattan."o "norma del taxista", ya que nos dice la distancia que recorre un taxi para ir desde el punto 0 hasta el definido por el vector, yendo por una cuadrícula que representa las calles. Por último, otra norma importante (por ejemplo en aplicaciones de procesamiento de señales y sistemas dinámicos) es la que surge de definir $p = \infty$, es decir $\|x\|_\infty = \text{máx}\{(x_j) : j = 1, \cdots, n\}$.

**Normas matriciales**

Una vez más podemos definir diversas funciones que transformen una matriz en un escalar y que cumpla las propiedades de una norma. Sin embargo, para tomar la norma de una matriz de una forma útil, conviene recordar de lo que se mencionaba acerca de que una matriz es una función, una transformación lineal de un espacio en otro. Esto resulta en algunas normas particularmente útiles relacionadas con con la norma "p"de los vectores:

- $\|A\|_{pq} = \text{máx}_{x \in \mathbb{R}^n,\ x \neq 0} \frac{\|Ax\|_p}{\|x\|_q}$
- $\|A\| = \text{máx}_{\|x\|=1} \|Ax\|$ (Norma matricial natural).
- $\|A\|_1 = \text{máx}\{\sum_{i=1}^{n} |a_{ij}|, j = 1, \cdots, n\}$
- $\|A\|_\infty = \text{máx}\left\{\sum_{j=1}^{n} |a_{ij}|, i = 1, \cdots, n\right\}$

En estos casos, podemos interpretar la norma de una matriz como la la norma del vector "más grande"que puedo generar a partir de multiplicar un vector de norma 1 por la matriz.

Una vez más, sustituyendo $p$ también se obtienen expresiones para la norma 2, 1 y la norma $\infty$.

### 1.2.3. Determinante

El determinante de una matriz es otra función que nos devuelve un escalar a partir de una matriz.

**Definición 1.2.3.1** (Matriz adjunta)**.** *Sea $A = ((a_{ij}))$ una matriz $n \times n$, se define* ***la matriz adjunta*** *del elemento $a_{ij}$ como la submatriz $A_{ij}$ de $A$ que se obtiene eliminando la fila $i$ y la columna $j$ de la matriz $A$.*

*Por lo tanto la matriz adjunta será de tamaño $(n-1) \times (n-1)$.*

**Definición 1.2.3.2** (Determinante)**.** *Se define el* ***determinante*** *de una matriz cuadrada como una función que le asocia a dicha matriz un único escalar de la siguiente forma:*

- *El determinante de una matriz* $1 \times 1$ *es el propio número.*
- *el determinante de una matriz* $2 \times 2$, $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ *es el número* $|A| = ad - bc$
- *El determinante de una matriz* $n \times n$ *se define de forma inductiva como:* $|A| = \sum_{i=1}^{n} (-1)^{i+1} a_{i1} |A_{i1}|$

**Propiedades del determinante**

- $det(AB) = det(BA)$
- $det(AB) = det(A)det(B)$
- $det(A^T) = det(A)$
- $det(\alpha A) = \alpha^n det(A)$
- $det(\overline{A}) = \overline{det(A)}$
- $det(\mathbb{I}) = 1$

### 1.2.4. Matrices particulares

**Matrices hermitiana o hermítica**

$$A = A^H \text{ es decir } a_{ij} = \overline{a_{ji}}$$

Ejemplo: $A = \begin{pmatrix} 3 & 2+i \\ 2-i & 1 \end{pmatrix}$

**Propiedades:**

- Son diagonalizables por una base ortonormal.
- Poseen todos sus valores propios reales.
- Su determinante es un número real.

**Matrices simétricas**

$$A = A^T \text{ es decir } a_{ij} = a_{ji}$$

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

**Propiedades:**

- Si posee entradas reales son un caso particular de matrices hermíticas.

**Matrices no-negativas**

$$a_{ij} \geq 0$$

No confundir con definida positiva.

**Propiedades:**

- Si $A$ es no negativa entonces tiene un valor propio real que además es el valor propio de mayor magnitud. El vector propio asociado a este valor propio es tambien no negativo.

**Matrices unitarias**

$$Q^H Q = QQ^H = \mathbb{I}$$

Su inversa es igual a la matriz transpuesta conjugada.

**Propiedades:**

- Sus columnas/filas son una base ortonormal de $\mathbb{C}^n$
- Preservan la norma y el producto interno:
  - $\langle Ux, Uy \rangle = \langle x, y \rangle$
  - $\|Ux\| = \|x\|$

**Matrices de Permutación**

En cada fila hay solo un elemento igual a 1 mientras que el resto son 0.

Ejemplo: $A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

**Propiedades:**

- Son unitarias.
- Al multiplicarlas por una matriz/vector producen una permutación de las filas/columnas de la misma o de los elementos del vector.

### Matrices triangulares

$$i > j \implies a_{ij} = 0(\text{superior})$$

Ejemplo: $A = \begin{pmatrix} 4 & 1 & -15 & 1 \\ 0 & 2 & 15 & 0 \\ 0 & 0 & -26 & 29 \\ 0 & 0 & 0 & 151 \end{pmatrix}$

**Propiedades:**

- Sus valores propios son los elementos de la diagonal.
- Su determinante es el producto de los elementos de la diagonal.

### Matrices diagonales

Poseen elementos no nulos únicamente en la diagonal.

Ejemplo: $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 24 & 0 & 0 \\ 0 & 0 & 15 & 0 \\ 0 & 0 & 0 & 6 \end{pmatrix}$

**Propiedades:**

- Son simétricas, triangulares (inferiores y superiores).
- Sus valores propios son los elementos de la diagonal.
- Su determinante es el producto de los elementos de la diagonal.
- Son fáciles de almacenar y operar.

### Matriz bi-diagonal

Ejemplo: Inferior: $BL = \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 & 0 \\ 0 & 4 & 2 & 0 & 0 \\ 0 & 0 & 6 & 7 & 0 \\ 0 & 0 & 0 & 8 & 9 \end{pmatrix}$ Superior: $BL = \begin{pmatrix} 4 & 4 & 0 & 0 & 0 \\ 0 & 4 & 3 & 0 & 0 \\ 0 & 0 & 2 & 8 & 0 \\ 0 & 0 & 0 & 7 & 8 \\ 0 & 0 & 0 & 0 & 9 \end{pmatrix}$

### Matrices de banda

$$\exists m_l, m_u, a_{ij} \neq 0 \text{ si } i - m_l \leq j \leq i + m_u$$

$$\text{Ejemplo: } A = \begin{pmatrix} 4 & 6 & 8 & 0 & 0 \\ 5 & 4 & 4 & 1 & 0 \\ 0 & 4 & 7 & 6 & 5 \\ 0 & 0 & 7 & 9 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

**Matriz Hessenberg superior**

$$a_{ij} = 0 \ \forall i, j, i > j + 1$$

$$\text{Ejemplo: } A = \begin{pmatrix} 4 & 6 & 8 & 3 & 1 \\ 5 & 5 & 0 & 7 & 4 \\ 0 & 2 & 5 & 1 & 6 \\ 0 & 0 & 6 & 4 & 4 \\ 0 & 0 & 0 & 9 & 1 \end{pmatrix}$$

### 1.2.5. Valores y Vectores propios

**Definición 1.2.5.1** (Valores y vectores propios)**.** *Si $A$ es una matriz de tamaño $n \times n$ entonces se dice que $v \neq 0$ es **vector propio** de $A$ si:*

$$AV = \lambda v$$

*donde $\lambda$ es el **valor propio** asociado al vector propio $v$.*

Se utilizan en muy diversas aplicaciones en ingeniería desde la estabilidad de sistemas físicos o soluciones de ecuaciones diferenciales hasta en los algoritmos PageRank de Google.

Los **valores propios** son las soluciones de la ecuación característica:

$$det(A - \lambda \mathbb{I}) = 0$$

Los **vectores propios** asociados a $\lambda$ son los $x$ solución de:

$$(A - \lambda \mathbb{I})x = 0$$

**Definición 1.2.5.2** (Espectro de una matriz)**.** *Se define el espectro de una matriz $A$ como $\sigma(A)$, siendo este el conjunto de valores propios de $A$:*

$$\sigma(A) = \{\lambda_j : Av_j = \lambda v_j\}$$

**Definición 1.2.5.3** (Radio espectral)**.** *Se define el radio espectral de la matriz $A$, $\rho(A)$, como el mayor de los módulos de los valores propios de la matriz:*

$$\rho(A) = \max |\lambda_j|$$

**Propiedades de los valores y vectores propios**

- La suma de los valores propios de una matriz es igual a su traza:

$$\lambda_1 + \lambda_2 + \cdots + \lambda_n = \sum_{i=1}^{n} a_{ii} \tag{1.8}$$

- El producto de los valores propios es igual al determinante de la matriz:

$$det(A) = \lambda_1 \cdot \lambda_2 \cdot \cdots \cdot \lambda_n \tag{1.9}$$

- Los valores propios de una matriz simétrica son reales.
- Una matriz definida positiva tiene valores propios positivos.
- Los valores propios de una matriz triangular son los elementos de la diagonal.
- $Av = \lambda v \implies A^k v = \lambda^k v$
- $Av = \lambda v \implies (Ac + \mathbb{I})\, v = (\lambda + c)v$

**Definición 1.2.5.4** (Matrices semejantes)**.** *Dos matrices $A$ y $B$ se dicen semejantes si $\exists P, PBP^{-}$.*

Las dos matrices poseen el mismo polinomio característico por lo que tienen los mismos valores propios y sus vectores propios cumplen $v_A = Pv_B$. Además si la matriz es simétrica, la matriz $P$ es ortogonal.

**Definición 1.2.5.5** (Matriz diagonalizable)**.** *Una matriz es **diagonalizable** si es semejante a una matriz diagonal. Esta matriz diagonal, denominada $D$, está formada por los valores propios. Mientras que las matriz $P$ está formada por los vectores propios como columnas.*

**Definición 1.2.5.6** (Matriz ortogonal)**.** *Una matriz $A \in \mathbb{R}^{m \times m}$ es **ortogonal** si sus columnas forman una base ortonormal de vectores de $\mathbb{R}^m$.*

- *La inversa de una matriz unitaria es igual a su transpuesta.*
- *Las tra4nsformaciones por matrices unitarias conservan la norma y el producto escalar.*

**Definición 1.2.5.7** (Matriz ortonormal)**.** *Una **matriz ortonormal** cumple que:*

- *Las columnas son un conjunto ortonormal.*
- *El producto por $Q$ mantiene la norma $\|Qx\| = \|x\|$*
- *El producto por $Q$ mantiene el producto escalar: $\langle Qx, Qy\rangle = \langle x, y\rangle$*

**Teorema 1.2.5.1** (Teorema de Gershgorin)**.** *Dada una matriz $A$, se denomina $R_i$ a la región circular del plano complejo con centro $a_{ii}$ y radio $r_i = \sum_{j=1, j\neq 1}^{n} |a_{ii}|$. Entonces:*

- *Los valores propios de $A$ están contenidos en $R = \bigcup_{i=1}^{n} R_i$*
- *La unión de cualesquiera $k$ de estos círculos que sea disjunta con los $n - k$ restantes debe contener exactamente $k$ valores propios.*

**Ejemplo 1.2.5.1.** *Dada la matriz* $A = \begin{pmatrix} 4 & 1 & 1 \\ 0 & 2 & 1 \\ -2 & 0 & 9 \end{pmatrix}$ *posee 3 valores propios que son:* $\lambda_1 = 8.4853$, $\lambda_2 = 4.6318$ $\lambda_3 = 1.8828$ *y según los círculos de Gershgorin pueden visualizarse en la Figura 1.1.*

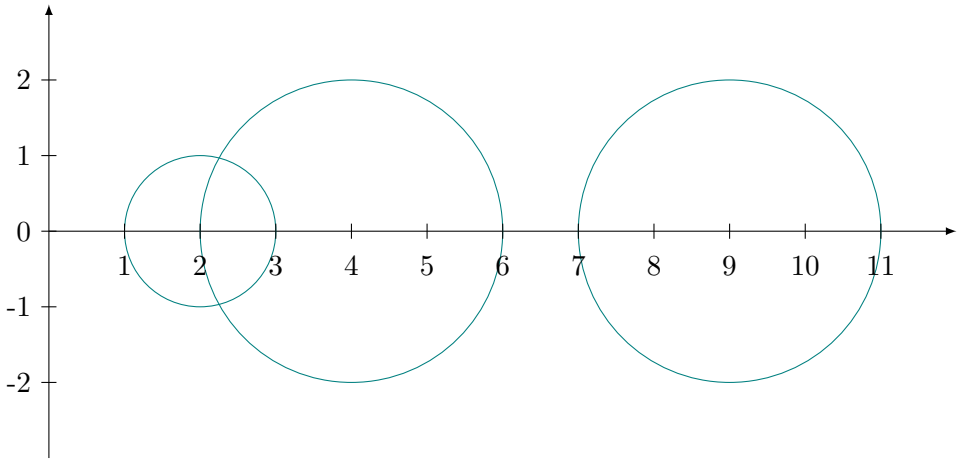



**Figura 1.1**: Círculos de gershgorin

*Así puede observarse que los valores propios efectivamente se encuentran en la unión de estos círculos.*

### 1.2.6. Sistemas de ecuaciones

Se basan en econtrar $x$ tal que $Ax = b$

Es un problema muy frecuente en modelos numéricos (sistemas físicos, economía, procesamiento de señales, optimización, ...)

**Teorema 1.2.6.1.** *Sea $A$ una matriz de elelemntos reales de tamaño $n \times n$, entonces son equivalentes las siguientes afirmaciones:*

- *$Ax = b$ tiene una única solución*
- *$Ax = 0$ implica $x = 0$*
- *$A^{-1}$ existe*
- *$det(A) \neq 0$*
- *$rango(A) = n$*

# Capítulo 2

# Errores y representaciones

El objetivo de los métodos numéricos es llegar a una solución "correcta", es decir, con un nivel de error aceptable, en forma "eficiente", es decir, en un tiempo razonable utilizando los recursos disponibles.

Para lograr llevar esto a cabo son necesarias 2 cosas:

- El estudio de la propagación de errores.
- El estudio del número de operaciones.

Para el estudio de la propagación de errores a estos se los separa en dos tipos:

- Errores de truncamiento: So aquellos debidos a las aproximaciones realizadas al resolver el modelo numérico.
- Errores de redondeo: Son aquellos debidos al número finito de dígitos con los que se trabaja en la computadora.

## 2.1. Representaciones

Se plantea entonces el problema de representar número reales en una computadora. Se tiene que $\mathbb{R}$ es un cuerpo infinito y denso mientras que la computadora es capaz de representar los números con una cantidad finita de dígitos.

Para reoslver esto se utilizan 2 tipos de representaciones:

- Rpresentación de punto fijo
- Representación de punto flotante

### 2.1.1. Representación de punto fijo

Representa los números reales como enteros divididos por cierto factor de escala.

| Tipo | Número máximo a representar sin signo |
|---|---|
| 1 byte | 255 |
| 2 bytes | 65.535 |
| 4 bytes | 4.294.967.295 |
| 8 bytes | 18.446.744.073.709.551.615 |

Ej.: Se toma al entero 12340 con escala 1000 entonces el número es 12.340.

Es útil ya que los procesadores simples o microprocesadores no poseen unidades de punto flotante. Pueden utilizarse de dos tipos: Decimal o binario.

Los tamañaos usuales para representar los enteros sin signo son:

### 2.1.2. Representación de punto flotante

Esta representación surge por la necesidad de representar números reales y enteros con un rango de representación mayor que el que ofrece el punto fijo y se basa en la notación científica utilizada en física, química y metemática:

$$n = (-1)^s \times f \times 10^{\exp} \tag{2.1}$$

La representación en punto flotante es la versión para computadoras de la notación científica utilizando base 2:

$$n = (-1)^s \times f \times 2^{\exp} \tag{2.2}$$

Se compone de tres partes:


- **s:** Signo, puede ser 0 (número positivo) 0 1 (número negativo).
- **e:** Exponente, representado con $q$ bits en exceso a $M$, es decir $M = 2^{q-1}$ y es $e = \exp + 2^{q-1}$.
- **f:** Mantisa, representada con $p$ bits en binario.

Si para la representación se pretenden utilizar $n$ bits entonces se tiene que cumplir $1 - p - q = n$.

#### Normalización

Un número en punto flotante normalizado cumple que el dígito más significativo de la mantisa es diferente de cero o, lo que es equivalente, la mantisa es máxima ya que su bit más significativo es 1.

Estos números representan la máxima precisión posible para los puntos de precisión flotante.

Para esto de define una representación que omite el bit más significativo y solo representa la porción después de la coma, dejando un 1 y una coma implícitos:

$$n = (-1)^s \times (1, f) \times 2^{\exp} \tag{2.3}$$

### Estándar IEEE 754

Este estándar define el formato y las operaciones a utilizar, permite que los números representados en punto flotante puedan ser intercambiados entre distintas arquitecturas.

El estándar IEEE 754 define cuatro formatos:

| Tipos de datos | s | e | f | Total bytes | Total bits |
|---|---|---|---|---|---|
| Media Precisión | 1 | 5 | 10 | 2 | 16 |
| Simple precisión (short real) | 1 | 8 | 23 | 4 | 32 |
| Doble precisión (long real) | 1 | 11 | 52 | 8 | 64 |
| Extended real | 1 | 15 | 64 | 10 | 80 |

Así se define para cada formato la cantidad de bits a utilizar en cada parte para luego definir cómo se representará cada una de esas partes:

- Signo: se asocia un bit que puede ser 0 (positivo) o 1 (negativo).
- Mantisa: Se representa como un binario puro.
- Se representa utilizando exceso a $M$, calculando $M$ como $2^{n-1} - 1$.
  - Precisión simple: $M = 2^{8-1} - 1 = 127$
  - Doble precisión: $M = 2^{11-1} - 1 = 1023$

Sin embargo hay casos particulares como:

- Cero: $e = 00 \cdots 00,\ f = 00 \cdots 00$
- Infinito: $e = 11 \cdots 11\ f = 00 \cdots 00$ el valor de $s$ define si es $+\infty$ o $-\infty$
- Not a Number (NaN): $e = 11 \cdots 11\ f \neq 00 \cdots 00$
- Desnormalizados: Cuando el resultado de un cálculo tiene una magnitud menor que el número normalizado más pequeño que se puede representar en ese sistema surge un problema, para solucionar esto se crearon los números desnormalizados:

$$n = (-1)^s \times (0, f) \times 2^{-126} \quad \text{(simple precisión)} \tag{2.4}$$

### Aritmética de punto flotante

Para sumar o restar dos números en punto flotante es necesario que los exponentes sean iguales.

Pasos para realizar la suma o resta:

1. Alinear las mantisas: Se desplaza hacia la derecha la mantisa que tiene el exponente más pequeño tantos lugares como la diferencia entre los exponentes.

2. Sumar o restar las mantisas.

3. Normalizar el resultado.

**Errores en la representación**

Con la representación en punto flotante se representan los números reales sin embargo existen algunas diferencias importantes.

Como ejemplo se considera una representación que:

- Utiliza tres dígitos y signo para la mantisa.
- Donde el valor absoluto de la mantisa esté comprendido entre $0.1 < |f| < 1$ o $f = 0$.
- Utilice un exponente de dos dígitos y signo.
- Se utilice base 10 para simplificar.

La recta real quedará dividida en 7 regiones, como muestra la Figura 2.1:

1. Números negativos menores que $-0.999 \times 10^{99}$
2. Números negativos entre $-0.99 \times 10^{99}$ y $-0.100 \times 10^{-99}$
3. Numéros negativos entre $-0.100 \times 10^{-99}$ y 0
4. 0
5. Números positivos entre 0 y $0.100 \times 10^{-99}$
6. Números positivos entre $0.100 \times 10^{-99}$ y $0.999 \times 10^{99}$
7. Números positivos mayores que $0.999 \times 10^{99}$

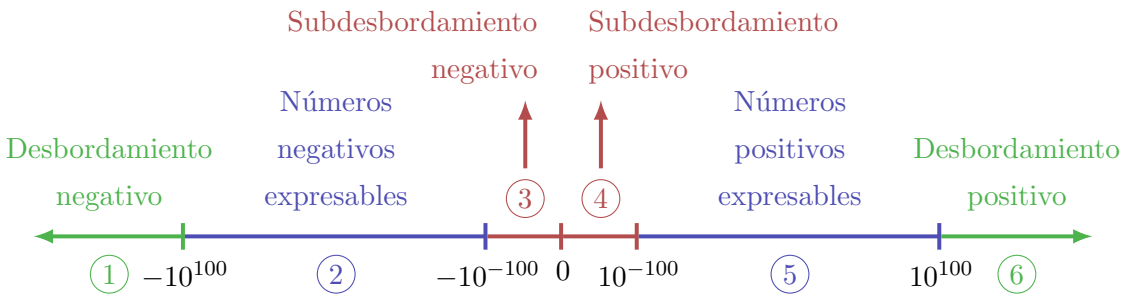



**Figura 2.1**

Así se notan las diferencias entre el conjunto de los números representables en punto flotante y el conjunto de los números reales:

- Los primeros no pueden representar ningún número en las regiones 1, 3, 4 y 5. Esto se da por la naturaleza finita de la representación.
  - Si una operación aritmética diera como resultado un número en las regiones 1 o 6 se produciría un error de desbordamiento u overflow y el resultado sería incorrecto.
  - De igual forma no se puede representar ningun resultado de las zonas 3 o 5 ya que se daría un error de subdesbordamiento o underflow.
- Existen diferencias en densidad.

  Con la representación elegida se pueden representar exactamente 179.000 números positivos, 179.000 números negativos y el 0, dando un total de 358.201 números representables. Es posible que el resultado de alguna operación no caiga dentro de estos números aunque sí pertenezca a la región 2 o 6

Si el resultado de una operación no se puede expresar en la representación elegida se debe aproximar este resultado por un número representable, siendo las alternativas para esto el redondeo o el truncamiento. Estas aproximaciones generaran errores.

**Definición 2.1.2.1** (Error absoluto)**.** *Se llamará* ***error absoluto*** *a la diferencia entre el número que se quiere representar y el número efectivamente representado.*

$$Ex = x - \overline{x} \tag{2.5}$$

*donde $x$ es el número que se quiere representar y $\overline{x}$ es el número efectivamente representado.*

**Definición 2.1.2.2** (Épsilon de máquina)**.** *El* ***épsilon de máquina****, o machine's epsilon, es una cota del error relativo que se comete al redondear un número en determinada representación de punto flotante.*

Tipos de errores:

- Not a Number (NaN)
- Overflow: $x > \text{máx}_{z \in FP} |z|$
- Underflow: $x < \text{mín}_{z \in FP} |z|$
- Cancelación catastrófica: Se produce al restar dos números grandes pero cercanos.
- Shift out: Se produce al sumar un número pequeño a uno grande.

### Número de Condición de un algoritmo

En la Figura 2.2 se tiene que $A(x)$ es el valor exacto calculado con el algoritmo $A$ mientras que $\overline{A}(x)$ es el valor calculado con una máquina.

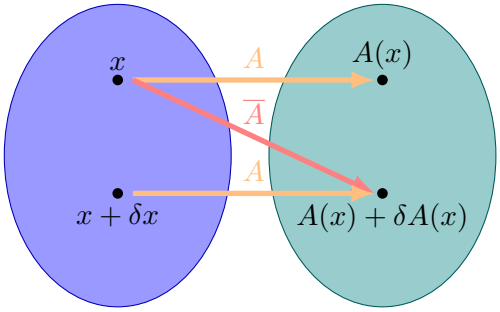



**Figura 2.2**

**Definición 2.1.2.3** (Número de condición de un algoritmo)**.** *El* ***número de condición de un algoritmo*** *se define entonces como:*

$$C_{A(x)} = \frac{\|\delta x\|}{\|x\|\epsilon_{mach}} = \frac{\left\|A^{-1}\left(\overline{A}(x)\right) - x\right\|}{\|x\|\epsilon_{mach}} \tag{2.6}$$

Con esto se ven reflejados los errores *"hacia atrás"*, es decir, hacia los datos, incidiendo la máquina a través del $\epsilon_{\text{mach}}$.

### Número de Condición de un problema

En la Figura 2.3 se tiene que $P(x)$ es la solución exacta de un problema con dato $x$, mientras que $P(x + \delta x)$ es la solución exacta del problema con dato $x + \delta x$.

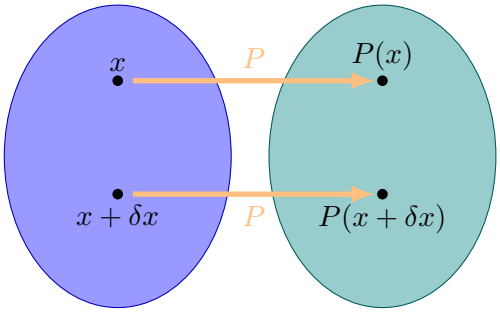



**Figura 2.3**

**Definición 2.1.2.4** (Número de condición de un problema)**.** *El* ***número de condición de un problema*** *se define entonces como:*

$$C_{P(x)} = \max_{\frac{|\delta x|}{|x|} \leq \epsilon} \frac{\frac{\|P(x+\delta x) - P(x)\|}{\|P(x)\|}}{\frac{\|\delta x\|}{\|x\|}} \tag{2.7}$$

Permite estimar la sensibilidad relativa de los resultados respecto a los datos. Es independiente tanto de los errores numéricos como del algoritmo utilizado.

# Capítulo 3

# Matrices y Máquinas

## 3.1. Órdenes

Una forma de estimar tiempos de ejecución es calculando la cantidad de operaciones (sumas-multiplicaciones) en función de N, siendo $N$ el tamaño de los datos, esto se denominará $T(N)$.

**Ejemplos:**

- ```
  for (int i = 0; i < N; i++)
      a = a + t[i]*v[i];
  ```

  Requiere $N$ operaciones (suma-multiplicación) por lo que $T(N) = N$

- ```
  for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
          a = a + t[i]*v[j];
  ```

  Requiere $T(N) = N^2$

- ```
  for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++) {
          a = a + t[i]*v[j];
          b = b + a*b;
      }
  ```

  Requiere $T(N) = 2N^2$

De forma informal se puede definir entonces el orden $\mathcal{O}()$ de un algoritmo como el orden de la función $T(N)$.

**Ejemplo:** Retomando el ejemplo anterior se tiene:

Medir el orden de un algoritmo da una vista rápida de la complejidad computacional del mismo sin embargo solo mide operaciones a realizar, sin tener en cuenta otros aspectos como pueden ser
  ```

| $T(N)$ | $\mathcal{O}(N)$ |
|---|---|
| $N$ | $N$ |
| $N^2$ | $N^2$ |
| $2N^2$ | $N^2$ |

**Tabla 3.1**: Órdenes

los patrones de acceso a memoria o la localidad espacial y temporal, que pueden ser determinantes a la hora de medir tiempos de ejecución y eficiencia.

## 3.2. Evaluación de Desempeño

Existen distintas métricas para medir la velocidad de un procesador:

- **CPU time:**

$$CPU = NI \cdot CPI \cdot CCT$$

  donde se tiene que $NI$ es el número de instrucciones, $CPI$ es el número de ciclos de reloj por instrucción y $CCT$.

- **Millions of Instructions Per Second (MIPS):**

$$MIPS = \frac{\text{frecuencia del reloj}}{CPI \cdot 10^6}$$

  donde $CPI$ es el número de ciclos de reloj por instrucción. Este depende del conjunto de instrucciones del equipo y a su vez en un equipo depende del problema a resolver.

- **Floating Point Operations Per Second (FLOPS):**

$$FLOPS = NFPU \cdot FPC \cdot CPS$$

  donde $NFPU$ es el número de unidades de punto flotante, $FPC$ son los flops por ciclo y $CPS$ es la frecuencia del reloj (ciclos por segundo, Hz).

Cuando se trata de medir el desempeño es fundamental entender cómo se divide el tiempo durante la ejecución de un proceso. Hay varios aspectos clave que contribuyen a esta comprensión:

- **Tiempo de usuario:** Este es el tiempo que se dedica a operaciones a nivel de usuario. Representa el tiempo en el que el programa está realizando tareas específicas requeridas por el usuario o por el proceso mismo.

- **Tiempo del sistema:** El tiempo del sistema está relacionado con las llamadas al sistema, como operaciones de entrada/salida (E/S). Estas operaciones implican la interacción del programa con el sistema operativo y otros recursos externos.

- : **Tiempo de CPU:** se refiere al tiempo durante el cual la unidad central de procesamiento (CPU) de un sistema está ocupada ejecutando instrucciones de un proceso específico.

- **Walltime:** Es el tiempo real que transcurre entre dos eventos o puntos de referencia. Mide el tiempo total desde el inicio hasta el final de un proceso o programa. Incluye el tiempo de usuario, el tiempo del sistema y cualquier tiempo de espera o inactividad.

Por ejemplo se tiene que le comando `time` en Linux proporciona 3 estadísticas de tiempo: `real`, `user` y `system`. por un lado`real` proporciona el tiempo "reloj" entre el inicio y el fin del programa, es una medida que incluye todos los factores, como la latencia de E/S, la espera del usuario y el tiempo de cálculo del programa. Por otro lado `user` mide el tiempo de CPU en el espacio de usuario, es decir, es la cantidad total de tiempo de CPU dedicado a la ejecución de instrucciones del programa en el espacio de usuario, el tiempo en el que la CPU está activamente ejecutando el código del programa. Por último `system` es el tiempo de CPU en el espacio del sistema, es decir, es la cantidad total de tiempo de CPU dedicado a las llamadas del sistema y operaciones del sistema realizadas por el programa, este incluye el tiempo que la CPU pasa en tareas relacionadas con el sistema operativo, como operaciones de E/S. Se cumple que `real` $\leq$ `user + system`.

Por otro lado para medir y evaluar el rendimiento de sistemas y programas, se utilizan benchmarks (pruebas de rendimiento), como pueden ser:

- **Micro benchmarks:** Están diseñados para medir el rendimiento de un código pequeño y específico o para evaluar una característica particular del hardware. Los micro benchmarks pueden centrarse en aspectos como la velocidad del procesador o el acceso a la memoria.

- **Kernel benchmarkks:** Evalúan el rendimiento mediante la ejecución de conjuntos de rutinas simples. Estas pruebas evalúan tanto el hardware como los compiladores y son utilizadas para comprender el desempeño en niveles más bajos del sistema. Ejemplos de estos son Livermore Loops y LINPACK.

En resumen, a la hora de evaluar es necesario hacerlo de variados puntos de vista y no enfocarse en uno solo como la velocidad del procesador, el conjunto de instrucciones, la cantidad de ciclos de reloj, etc. Pueden considerarse otros factores, como por ejemplo el consumo energético del programa.

## 3.3. Máquina de Von Neumann

La Máquina de Von Neumann, también conocida como arquitectura de Von Neumann, es un modelo fundamental en el diseño de computadoras y se basa en el concepto de un procesador que sigue un flujo de trabajo específico.

Los pasos que sigue una Máquina de Von Neumann para ejecutar instrucciones son los siguientes::

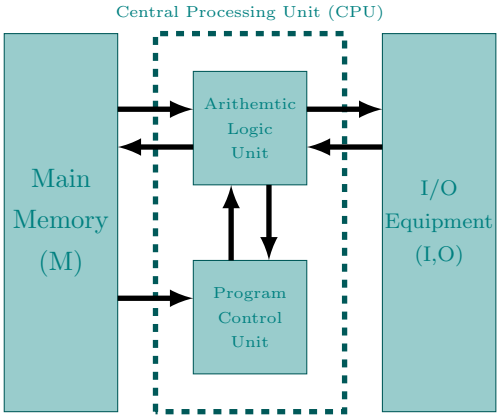



**Figura 3.1**: Máquina de Von Neumann

1. Obtiene la Siguiente Instrucción desde la Memoria: El proceso comienza obteniendo la siguiente instrucción que debe ejecutarse desde la memoria principal. Esta instrucción se recupera utilizando el contador de programa, que almacena la dirección de memoria de la siguiente instrucción a ejecutar.

2. Aumenta el Contador de Programa: Una vez que la instrucción se ha recuperado, el contador de programa se incrementa en uno para apuntar a la siguiente dirección de memoria. Esto asegura que la próxima instrucción se recupere en la siguiente iteración.

3. Decodifica la Instrucción Mediante la Unidad de Control: La instrucción recuperada se envía a la unidad de control. La unidad de control interpreta la instrucción y determina qué operación debe realizar la CPU. Esta etapa implica la decodificación de la instrucción y la preparación para su ejecución.

4. Se Ejecuta la Instrucción: Una vez que la unidad de control ha decodificado la instrucción y ha preparado las operaciones necesarias, se ejecuta la instrucción. Esto puede implicar cálculos matemáticos, operaciones lógicas, acceso a datos en memoria o cualquier otra acción requerida por la instrucción.

Este proceso se repite continuamente mientras el programa se ejecuta.

Un aspecto clave de esta arquitectura es que tanto los datos como las instrucciones se almacenan en la misma memoria principal, lo que se conoce como “almacenamiento de programas en memoria”. Esto permite una gran flexibilidad en la programación y ejecución de tareas, ya que los programas pueden ser modificados y adaptados fácilmente sin requerir cambios físicos en el hardware de la computadora.

## 3.4. Pipeline

Un "pipeline"se refiere a un flujo secuencial de datos o tareas interconectadas en la que la salida de una etapa se utiliza como entrada para la siguiente. Cada etapa del pipeline realiza una función o procesamiento específico, y la información se va pasando de una etapa a la siguiente en una forma ordenada y automatizada. En un pipeline una tarea se divide en sub-tareas y la unidad funcional se divide en sub-segmentos.

Un pipeline básico se basa en la ejecución de una instrucción dividida en dos etapas:

1. **Fetch:** Traer la siguiente instrucción desde la memoria.
2. **Execute:** Ejecuta la instrucción.

Siendo cada etapa realizada en una unidad funcional independiente.

Siguiendo la siguiente nomenclatura:

- FI: Fetch de una instrucción
- DI: Decodificar instrucción
- CO: Calcular los Operandos
- FO: Fetch de los datos
- EI: Ejecutar instrucciones
- WO: Escribir en la memoria

Se tiene que el pipeline se ejecuta según le siguiente esquema:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Instrucción 1 | FI ⟷ | DI ⟷ | CO ⟷ | FO ⟷ | EI ⟷ | WO ⟷ | | | | |
| Instrucción 2 | | FI ⟷ | DI ⟷ | CO ⟷ | FO ⟷ | EI ⟷ | WO ⟷ | | | |
| Instrucción 3 | | | FI ⟷ | DI ⟷ | CO ⟷ | FO ⟷ | EI ⟷ | WO ⟷ | | |
| Instrucción 4 | | | | FI ⟷ | DI ⟷ | CO ⟷ | FO ⟷ | EI ⟷ | WO ⟷ | |
| Instrucción 5 | | | | | FI ⟷ | DI ⟷ | CO ⟷ | FO ⟷ | EI ⟷ | WO ⟷ |

Algunos de los problemas comunes con los pipelines son:

- Llenar el Pipeline: Para mantener un pipeline eficiente, es importante asegurarse de que siempre haya instrucciones disponibles para ser procesadas en cada etapa. Si no hay suficientes instrucciones en el pipeline, se puede producir un "cuello de botella" que ralentiza el rendimiento.

- Hazards (Obstáculos): Los hazards son situaciones que pueden impedir que la próxima instrucción se ejecute en el ciclo correspondiente. Algunos ejemplos de hazards incluyen bifurcaciones en el código (como saltos condicionales), dependencias de datos (cuando una instrucción depende del resultado de otra) y conflictos de hardware (por ejemplo, cuando múltiples instrucciones intentan acceder a la misma ubicación de memoria).

  Cuando se encuentran hazards, a veces es necesario detener el pipeline y, en casos extremos, vaciarlo. Esto puede resultar en una pérdida de eficiencia y tiempo.

  Para abordar estos problemas y minimizar las penalizaciones en el pipeline, se han desarrollado estrategias y técnicas. Por ejemplo, se utilizan estrategias predictivas para predecir el flujo de instrucciones y tomar decisiones anticipadas sobre el camino a seguir, como en el caso de saltos condicionales. Esto ayuda a mantener el pipeline lleno y funcionando de manera más eficiente.

- Latencia: La latencia se refiere al tiempo que tarda en procesarse una tarea o una instrucción desde el momento en que se inicia hasta que se termina por completo. Puede convertirse en un problema cuando una instrucción necesita más tiempo que otras para completarse debido a su naturaleza complicada o requerimientos especiales, ralentizando el flujo general de instrucciones.

- Throughput (Rendimiento Medido): El throughput se refiere al rendimiento medido en términos de unidades producidas por unidad de tiempo. Un bajo throughput puede indicar ineficiencias en el pipeline que limitan la capacidad de la CPU para procesar instrucciones de manera rápida y eficiente.

El pipelining entonces no mejora la latencia de cada tarea, sino el throughput de toda la carga de trabajo. Sin embargo, el throughput está limitado por el estado más lento en el pipeline. Un pipeline con etapas de longitud desbalanceada puede reducir la aceleración general, y para aumentar la aceleración, a veces es necesario aumentar la cantidad de pasos en el pipeline. Esto es esencial para asegurar que el flujo de instrucciones se mantenga constante y que el rendimiento general de la CPU sea óptimo.

## 3.5. Máquinas Vectoriales

El procesamiento vectorial se refiere a una técnica en la que las instrucciones operan sobre arreglos de largo variable, conocidos como vectores. Para llevar a cabo este tipo de procesamiento, se utilizan registros vectoriales, que son áreas de memoria especializadas para trabajar con datos en forma vectorial.

En el procesamiento vectorial, los datos se cargan en forma vectorial en estos registros, y las operaciones se realizan también en forma vectorial, lo que significa que una sola instrucción puede

operar en múltiples elementos de un vector simultáneamente.

Este enfoque tuvo gran relevancia en la década de 1970 y fue empleado en sistemas informáticos notables como las supercomputadoras Cray (como la Cray 1 y Cray X-MP), así como por empresas como Fujitsu y NEC. Sin embargo, con el tiempo, esta tecnología cayó en desuso debido a su elevado costo de implementación, ya que requería hardware especializado.

La idea subyacente en el procesamiento vectorial se ha convertido en un componente fundamental de otro tipo de hardware: las unidades de procesamiento gráfico (GPUs), que son ampliamente utilizadas en tareas que requieren cálculos intensivos, como gráficos y aprendizaje profundo.

En la actualidad, los procesadores escalares modernos han incorporado extensiones SIMD (Single Instruction, Multiple Data) de largo fijo, como AVX (Advanced Vector Extensions) y AVX-512, que permiten realizar operaciones similares en conjuntos de datos, aunque con longitudes fijas. Además, los equipos suelen contar con compiladores que pueden transformar automáticamente el código para aprovechar estas extensiones.

Una práctica común al programar en este contexto es utilizar notación matricial, como en el lenguaje de programación FORTRAN, para facilitar la escritura de código que aproveche al máximo el procesamiento vectorial y las extensiones SIMD disponibles en el hardware.

## 3.6. Encadenamiento

El encadenamiento se refiere a la técnica de combinar múltiples pipelines de manera que el resultado de un proceso en uno de ellos se convierta directamente en la entrada de otro. Esto permite un flujo continuo y eficiente de datos a través de múltiples etapas de procesamiento.

Un ejemplo típico de encadenamiento se encuentra en la operación "Multiply-Accumulate" (MAC), que se representa comúnmente como:

```
a = a + bc
```

En esta operación, se realiza una multiplicación entre los valores `b` y `c`, y luego el resultado se suma al valor existente de `a`. Este cálculo se puede dividir en dos etapas secuenciales: la etapa de multiplicación y la etapa de acumulación.

Al aplicar el encadenamiento, la salida de la etapa de multiplicación se alimenta directamente como entrada a la etapa de acumulación sin necesidad de almacenarla en un registro intermedio. Esto mejora la eficiencia y reduce la latencia, ya que los datos fluyen de una etapa a la siguiente sin interrupciones.

## 3.7. Estrategias de Compilación

Cada compilador dispone de diversas opciones, la mayoría permite aplicar optimizaciones básicas o compilar para una arquitectura en específico.

Tomando como ejemplo Intel Fortran en el entorno de desarrollo Linux, se tiene que algunas opciones comunes disponibles son las siguientes:

- Opciones de Optimización:
  - `-O0, -O1, -O2, -O3, -Ofast`: Estas opciones controlan el nivel de optimización aplicado al código. A medida que se aumenta el nivel, se aplican más optimizaciones, lo que generalmente mejora el rendimiento, pero también puede aumentar el tiempo de compilación.
- Generación de Código:
  - `-x{code}`: Permite generar código específico que se ejecutará de manera eficiente en los procesadores compatibles con la arquitectura indicada por {`code`}, como SSE2, SSE3, SSSE3, SSE4.2, AVX, y otros conjuntos de instrucciones.
  - `-xHost`: Genera código utilizando el conjunto de instrucciones más avanzado disponible en el host de compilación, maximizando el rendimiento para la arquitectura de la máquina de destino.
  - `-mtune=cpu`: Esta opción permite optimizar el código generado para una CPU específica, como "skylake", "haswell", "broadwell", "skylake-avx512", y otras. Ver link.
- Optimización Interprocedural:
  - `-ip` o `-ipo`: La opción `-ip` busca oportunidades de optimización dentro de funciones del mismo archivo, mientras que `-ipo` amplía la búsqueda a todas las funciones en todos los archivos del sistema. Estas opciones habilitan optimizaciones interprocedurales, como la expansión de funciones inline, la propagación de constantes, la eliminación de código muerto, el pasaje de argumentos por registros y la extracción de códigos invariantes de los bucles.
- Punto Flotante:
  - `-fp-model {nombre}`: Controla el modelo de punto flotante utilizado, permitiendo opciones como "Fast", "Precise","Source" y "Strict".
  - `-pc{32|64|80(default)}`: Controla la precisión interna de la unidad de punto flotante (FPU).
  - `-rcd` y `-fp-port`: Estas opciones controlan la utilización de redondeos y truncamientos en operaciones de punto flotante.
- Otras Opciones:

- `-openmp`: Habilita la API (Application Program Interface) OpenMP, que permite programar "paralelismo" de memoria compartida para mejorar el rendimiento en sistemas multiprocesador.

Algunas prácticas que pueden permitir al optimizador realizar su tarea de una mejor forma:

- Minimizar la utilización de variables globales, ya que pueden dificultar la optimización y aumentar la complejidad del programa.
- Evitar el uso de controles de flujo complejos.
- Evitar el uso de instrucciones de conversión (cast) ya que estas pueden introducir operaciones costosas y potencialmente reducir la eficiencia del código.
- Evitar las refencias indirectas como la utilización de punteros.
- **Programación Pipelining con Programación por Busqueda (Fetch Scheduling):** Utilizar técnicas de programación por búsqueda para organizar las instrucciones de manera eficiente y reducir los cuellos de botella en el acceso a datos.
- **Desenrollado de Bucles (Loop Unrolling):** Aplicar el desenrollado de bucles para mejorar la paralelización y reducir las instrucciones de salto, lo que puede mejorar significativamente el rendimiento en bucles largos.
- **Expansión Escalar (Scalar Expansion):** Expandir expresiones escalares para reducir la dependencia de datos y permitir que el optimizador realice mejoras más efectivas.

## 3.8. Jerarquías de memoria

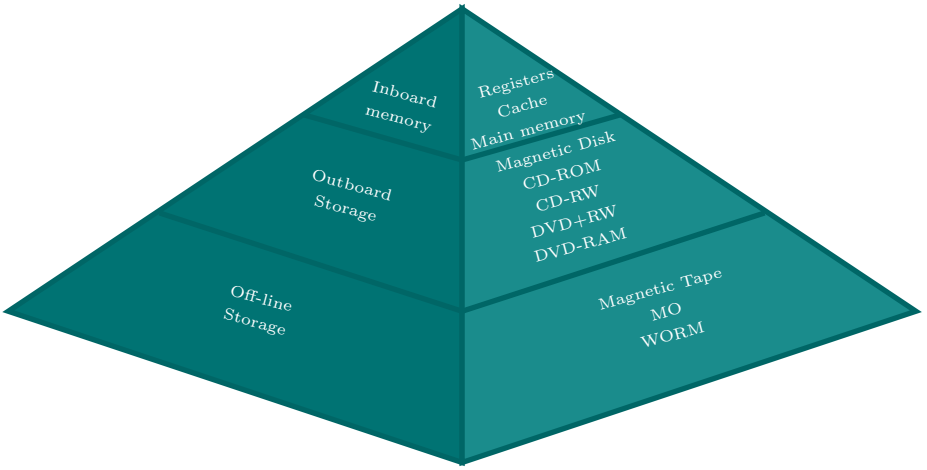
Uno de los desafíos fundamentales radica en la diferencia entre la velocidad de acceso a la memoria y la velocidad de cómputo de la CPU. A medida que las CPU se vuelven más rápidas y eficientes, la velocidad de acceso a la memoria tiende a aumentar a un ritmo más lento. Esto se debe a varias razones, incluidas limitaciones físicas.

Existen varios tipos de memorias, cada una con diferentes costos, tamaños y velocidades. Por lo general, se observa que cuanto más grande y más lejana está una memoria con respecto al procesador, más lenta es su velocidad de acceso. Esta discrepancia en la velocidad se debe en parte a consideraciones físicas.

Para comprender esto, podemos tomar como ejemplo la velocidad de la luz en el vacío, que es aproximadamente $c = 3.0x10^8 \frac{m}{s}$. Si consideramos un reloj de CPU funcionando a una frecuencia de $3GHz$, la relación entre la velocidad de la luz y la velocidad del reloj es de aproximadamente $10 \frac{cm}{ciclo}$. Esto significa que, en cada ciclo de reloj, la señal eléctrica o los datos solo pueden recorrer una distancia de unos 10 centímetros, lo que limita la velocidad de acceso a la memoria.

Además, otras propiedades físicas, como la capacitancia de los cables y la resistencia eléctrica, también influyen en la velocidad de acceso a la memoria. Cuanto más largos sean los cables, mayor será la capacitancia, lo que ralentizará la transmisión de datos.

Debido a estas limitaciones físicas, los sistemas informáticos han desarrollado una jerarquía de memorias para equilibrar el rendimiento y la capacidad de almacenamiento. Esto implica el uso de diferentes tipos de memorias, como la memoria caché, la memoria RAM y el almacenamiento en disco, cada una diseñada para satisfacer diferentes necesidades de velocidad y capacidad.




### 3.8.1. Principio de localidad

Los programas tienden a acceder a datos e instrucciones cercanas a las accedidas recientemente.

#### Localidad temporal

Si un ítem de datos o una instrucción se ha referenciado recientemente, es altamente probable que se vuelva a referenciar en un futuro cercano. En otras palabras, los datos o instrucciones que se utilizaron recientemente tienen una alta probabilidad de ser necesarios nuevamente en el corto plazo. Esto se debe a que los programas a menudo realizan bucles y repiten tareas, lo que resulta en el acceso frecuente a los mismos datos e instrucciones.

#### Localidad espacial

Si un ítem de datos o una instrucción se ha referenciado, es probable que los ítems con direcciones cercanas también sean referenciados en un futuro próximo. En otras palabras, los datos e instrucciones suelen estar agrupados en áreas cercanas de la memoria, y cuando se accede a uno de ellos, es probable que se acceda a otros cercanos en el espacio de direcciones. Esto se debe a la forma en que los programas organizan la información y la estructura de los datos.

### 3.8.2. Memoria Cache

Se trata de una memoria pequeña pero extremadamente rápida que se encuentra ubicada entre el procesador y la memoria principal. Su diseño se basa en la explotación del "Principio de Localidad" mencionado anteriormente.

Cuando la CPU necesita acceder al contenido de una posición de memoria, primero se verifica si ese contenido ya se encuentra en la caché. Si la información está presente en la caché, se produce lo que se conoce como un *"caché hit"*, y la CPU puede leer los datos de la caché de manera extremadamente rápida. Por otro lado, si la información no se encuentra en la caché, se produce un *"caché miss"*. En este caso, el bloque necesario se copia desde la memoria principal a la caché y luego se entrega desde la caché a la CPU. Este proceso lleva más tiempo en comparación con un caché hit, lo que se conoce como la *"penalización por mis"*.

Para gestionar de manera efectiva los datos en la caché y asegurarse de que los datos correctos estén en el lugar adecuado en el momento adecuado, la caché utiliza etiquetas o *"tags"* que identifican qué bloque de la memoria principal está almacenado en cada línea de la caché.

La memoria caché tiene como objetivo principal mejorar el rendimiento del sistema de computación en varios aspectos:

- Reducir el Miss Rate: El objetivo es minimizar la frecuencia de caché misses. Esto se logra al predecir de manera inteligente qué datos se necesitarán en el futuro y manteniéndolos en la caché antes de que se requieran.

- Reducir la Penalización por Miss: Aunque los caché misses son inevitables, se busca reducir el tiempo que lleva recuperar datos de la memoria principal cuando ocurre un miss. Esto implica técnicas como el almacenamiento de datos adyacentes al bloque faltante para aprovechar la localidad espacial.

- Reducir el Tiempo de Hit en la Caché: Se trabaja en optimizar el tiempo de acceso a la caché para maximizar la velocidad de recuperación de datos cuando ocurre un caché hit.

### 3.8.3. Memoria virtual

La memoria física de una computadora es un recurso valioso pero limitado. Para superar esta limitación y permitir que los programas funcionen eficazmente, se utiliza un concepto fundamental conocido como memoria virtual. La memoria virtual es una técnica que aprovecha el espacio de almacenamiento en disco para proporcionar un espacio de memoria aparentemente más amplio y abundante que la memoria física real de la computadora.

Cuando un programa necesita acceder a datos o instrucciones, normalmente busca en la memoria física. Sin embargo, debido a que la memoria física es limitada, no es posible cargar todos los datos y programas en ella de una vez. En lugar de eso, la memoria virtual divide la memoria en "páginas"

o “bloques” de datos y permite que solo se carguen en la memoria física las páginas que se necesitan en un momento dado.

Cuando un programa intenta acceder a una página que no está cargada en la memoria física en ese momento, se produce un evento llamado “Page Fault” o “Fallo de Página”. En este punto, el sistema operativo interviene y mueve la página requerida desde el disco a la memoria física, lo que permite que el programa continúe su ejecución. Este proceso de mover páginas entre el disco y la memoria física se realiza de manera transparente para el usuario y es gestionado por el sistema operativo.

La memoria virtual es una técnica esencial para gestionar eficazmente la memoria en sistemas informáticos modernos. Permite que los programas utilicen más memoria de la que realmente está disponible en la RAM física, lo que facilita la ejecución de aplicaciones grandes y múltiples al mismo tiempo. Además, proporciona una forma eficiente de administrar el uso de la memoria y garantizar que los recursos se asignen de manera óptima a las tareas en ejecución.

# Capítulo 4

# Matrices Dispersas

Existen distintas formas de definir a las **matrices dispersas**. Intuitivamente, son aquellas que poseen una fracción relativamente pequeña (digamos $\mathcal{O}(n)$ donde $n$ es la menor dimensión de la matriz) de sus coeficientes distintos a cero, lo que motiva **el uso de estructuras de almacenamiento que aprovechen esta particularidad**. Específicamente, se suelen almacenar únicamente los coeficientes distintos de cero, acompañados por índices que permitan deducir sus coordenadas en la matriz. Su importancia en el ámbito de la ciencia y la ingeniería radica en que son una herramienta fundamental para la resolución de problemas de gran escala que no pueden ser modelados por matrices densas, por ejemplo, algunos problemas de optimización (1) o la resolución de ecuaciones diferenciales parciales utilizando el Método de Elementos Finitos (o FEM por Finite Element Method) (2). La variedad de problemas a los que puede aplicarse ha crecido enormemente. Una completa revisión del uso de matrices dispersas en la computación científica, en las etapas tempranas de desarrollo, hasta el año 1977, se puede encontrar en el trabajo de I. Duff (3).

En los últimos años, las matrices dispersas han ido cobrando cada vez más relevancia en el campo de la computación científica, debido, por ejemplo, al impulso del crecimiento de las aplicaciones relacionadas con las redes sociales, big data e inteligencia artificial. En este contexto, las matrices representan, en general, un grafo de las relaciones entre los diferentes usuarios, tal como se presenta en (4; 5), alcanzando matrices de dimensiones extraordinariamente grandes y, al mismo tiempo, con muy pocos coeficientes no nulos.

**Definición 4.0.0.1** (Matriz dispersa)**.** *Una **matriz dispersa** es aquella que posee un número tal de ceros que amerita su aprovechamiento. (James H. Wilkinson)*

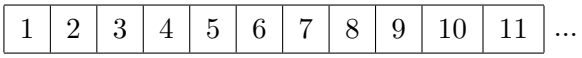
**Definición 4.0.0.2** (Matriz densa)**.** *Una **matriz densa** $A$ de tamaño $m \times n$ se caracteriza por la propiedad de que existe un número significativo de elementos no nulos en comparación con el número total de elementos en la matriz.*

Cuando trabajamos con matrices de gran tamaño, el almacenamiento de datos y el acceso a la memoria pueden convertirse en desafíos críticos que afectan significativamente los tiempos de

ejecución de los algoritmos. Para abordar este problema, se busca una representación eficiente de estas matrices que minimice el uso de memoria y, al mismo tiempo, acelere los cálculos. Esta optimización es especialmente importante cuando se trabaja con matrices dispersas, ya que el almacenamiento y la manipulación de los valores nulos pueden resultar en una utilización ineficiente de recursos. Por lo tanto, se buscan formas de representación para mitigar estos problemas y lograr un mejor rendimiento computacional en aplicaciones que involucran matrices de gran tamaño.

## 4.1. Representación en memoria de una matriz densa

La memoria es una secuencia de bytes identificados por una dirección. Podemos pensarla como un gran arreglo lineal:




Consideremos, por ejemplo, el caso de una matriz $A \in \mathcal{M}(\mathbb{R})_{n\times m}$ de reales de 4 bytes. Si los elementos son almacenados por fila, es decir, los elementos de una fila se encuentran contiguos en la memoria, la posición en bytes de la celda $(i, j)$ en la memoria es $inicio\_matriz + 4((i-1)m + j - 1)$.

Las matrices densas son naturalmente muy eficientes en el uso de memoria para almacenar datos y en el uso de procesador para accederlos.

## 4.2. Representación de matrices dispersas

Los formatos de almacenamiento de matrices dispersas suelen almacenar únicamente los coeficientes distintos de cero, acompañados por ındices que permitan deducir sus coordenadas en la matriz. Bajo ciertas condiciones, algunos formatos pueden almacenar algunos ceros si esto representa cierta ventaja a la hora de realizar algunos cálculos.

En general, se debe tratar de preservar los siguientes principios de economía.

- Mínimo consumo de memoria.
- Mínimo uso de procesador
  - "Tiempo" para acceder a los datos.
  - "Tiempo" de cálculos.
  - Localidad de los datos.

En muchos casos, factores como el "Tiempo" de las operaciones o la localidad en el acceso a los datos dependerá del problema que se intenta resolver, por lo que que existen numerosos formatos de almacenamiento disperso que presentan ventajas y desventajas en distintas situaciones.

Existen dos grandes familias de estrategias para almacenamiento de matrices dispersas:

- **Formatos estáticos:** Útiles cuando se conoce la estructura de la matriz, o cuando esta se genera en una única ocasión.
- **Formatos dinámicos:** Cuando la estructura de la matriz dispersa cambia en forma continua.

### 4.2.1. Formatos estáticos

**Formato simple o elemental (COO)**

La estrategia más simple para almacenar matrices dispersas es el formato de coordenadas (conocido como COO)(2). En este formato, se almacenan únicamente los valores no nulos de la matriz y las coordenadas son almacenadas en forma explícita. Típicamente, se utiliza un arreglo para almacenar el valor de cada elemento no nulo, otro para almacenar sus índices de fila y otro para el de columna.

A modo de ejemplo, consideremos la matriz

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & a_{15} & a_{16} & 0 \\ 0 & a_{22} & 0 & a_{24} & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 & 0 & 0 \\ 0 & a_{52} & 0 & 0 & a_{55} & a_{56} & 0 \\ 0 & 0 & a_{63} & 0 & a_{65} & a_{66} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{77} \end{pmatrix}$$

Entonces, se tiene

$$d = (a_{11}\ a_{12}\ a_{15}\ a_{16}\ a_{22}\ a_{24}\ a_{32}\ a_{33}\ a_{34}\ a_{41}\ a_{44}\ a_{52}\ a_{55}\ a_{56}\ a_{63}\ a_{65}\ a_{66}\ a_{77})$$
$$f = (1\ 1\ 1\ 1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5\ 5\ 6\ 6\ 6\ 7)$$
$$c = (1\ 2\ 5\ 6\ 2\ 4\ 2\ 3\ 4\ 1\ 4\ 2\ 5\ 6\ 3\ 5\ 6\ 7)$$

¿Cuanta memoria se utiliza para almacenar una matriz de dimensiones $n \times n$ y $nz$ elementos distintos de cero? Para guardar los $nz$ datos en punto flotante con doble precisión se utilizan $8nz$ bytes, para el vector columnas, que guarda enteros (32 bits), $4nz$ bytes y para el vector filas lo mismo. Por lo tanto, se utilizan $16nz$ bytes.

**Ventajas:**

- Intuitivo.
- Se utilizan estructuras del mismo tamaño.
- Es igual si queremos acceder por fila que por columna.

**Desventajas:**

- Utiliza más memoria que otras implementaciones.
- Acceder por fila o columna es más difícil que en otros formatos.

**Formato comprimido por fila (CSR - Compressed Sparse Row) o Old Yale Format**

Si los elementos de COO se ordenan por fila, entonces puede afirmarse que el arreglo de filas almacena información redundante, ya que las zonas del arreglo que corresponden a una misma fila almacenan el mismo valor. El formato comprimido por fila (CSR) atiende este problema almacenando el índice del comienzo de cada fila en los demás arreglos.

CSR es igual que el formato anterior excepto por el vector $f$. El mismo es ahora de largo $n+1$ siendo $n$ la cantidad de filas de la matriz $A$. En este vector se guarda el índice $k$ del $k$-ésimo número distinto de 0 que está primero en cada fila contando de izquierda a derecha y de arriba a abajo en la matriz.

Es decir, se necesitan 3 vectores:

- Un vector de punto flotante de tamaño $nz$ en el que se almacenan los valores de los coeficientes distintos de cero ordenados por fila.
- Un vector de enteros de tamaño $nz$ en el que se almacenan el número de columna de los elementos distintos de cero.
- Un vector de tamaño $n+1$ siendo n la cantidad de filas de la matriz, en el cual se almacena el índice en el cual comienza cada fila en los dos vectores anteriores.

En el ejemplo de la matriz anterior, se tiene

$$d = (a_{11}\ \ a_{12}\ \ a_{15}\ \ a_{16}\ \ a_{22}\ \ a_{24}\ \ a_{32}\ \ a_{33}\ \ a_{34}\ \ a_{41}\ \ a_{44}\ \ a_{52}\ \ a_{55}\ \ a_{56}\ \ a_{63}\ \ a_{65}\ \ a_{66}\ \ a_{77})$$
$$f = (1\ \ 5\ \ 7\ \ 10\ \ 12\ \ 15\ \ 18\ \ 19)$$
$$c = (1\ \ 2\ \ 5\ \ 6\ \ 2\ \ 4\ \ 2\ \ 3\ \ 4\ \ 1\ \ 4\ \ 2\ \ 5\ \ 6\ \ 3\ \ 5\ \ 6\ \ 7)$$

Análogamente al formato anterior, la memoria necesaria para almacenar la misma matriz en este caso es $8nz + 4nz + 4(n+1) = 12nz + 4(n+1)$ bytes. Lo cual representa un ahorro de $16nz - 12nz - 4(n+1) = 4nz - 4(n+1)$ respecto a COO.

**Ventajas:**

- Es óptimo desde el punto de vista de la información almacenada.
- Es fácil de acceder a una fila "completa".

**Desventajas:**

- No se utilizan estructuras del mismo tamaño.
- Es difícil acceder a una columna "completa".

### Formato comprimido por columna (CSC - Compressed Sparce Column) o CCS

Este es el formato que emplea MatLab.

Es igual al CSR pero almacena por columna. Siguiendo el mismo ejemplo, se tiene

$$d = (a_{11}\ a_{12}\ a_{15}\ a_{16}\ a_{22}\ a_{24}\ a_{32}\ a_{33}\ a_{34}\ a_{41}\ a_{44}\ a_{52}\ a_{55}\ a_{56}\ a_{63}\ a_{65}\ a_{66}\ a_{77})$$
$$f = (1\ 1\ 1\ 1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5\ 5\ 6\ 6\ 6\ 7)$$
$$c = (1\ 2\ 8\ 6\ 3\ 4\ 18\ 19)$$

Cada vez que se agrega un elemento se tiene que insertar ordenado en la matriz por lo que es necesario correr los demás elementos y rearmar la matriz.

Probemos en Octave: llenando una matriz CSC por columnas en distinto orden se obtiene

```
>> n = 512;
>> m = n*n;
>> A = spalloc( n, n, m );
>> for j = 1:n
       for i = 1:n
           A(i, j)=1+i+j;
       end
   end
```

0,88s

```
>> n = 512;
>> m = n*n;
>> A = spalloc( n, n, m );
>> for j = n:-1:1
       for i = n:-1:1
           A(i, j)=1+i+j;
       end
   end
```

14,18s !!!

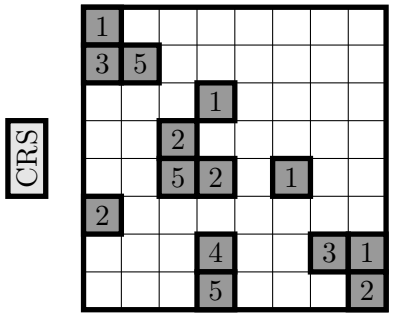
**Figura 4.1**: Prueba en Octave: llenar una matriz CSC por columnas en distinto orden.

### Formato comprimido por bloque de filas (BCRS: Block Compressed Row Storage)

En ocasiones es conveniente dividir la matriz original en bloques de igual tamaño y hacer que el arreglo de columnas represente posiciones de bloques en lugar de elementos distintos de cero. Un ejemplo de este esquema es el denominado Block Compressed Row Format (BCSR), cuyo arreglo comprimido de filas es análogo al arreglo correspondiente de CSR, pero considera filas de bloques en lugar de escalares. Este formato requiere que los elementos de un mismo bloque sean contiguos en el arreglo de valores y utiliza padding en dicho arreglo para asegurar que los bloques son del mismo tamaño.

Se considera una matriz de $n \times n$ con $nnzb$ bloques no nulos de dimensión $nb$. Para almacenarla, se utilizan tres vectores:

- Datos: punto flotante “val” de tamaño $nnzb \cdot nb \cdot nb$
- Columnas: enteros “col” de tamaño $nnzb$
- Bloques: enteros “bloque_fila” de tamaño $\frac{n}{nb} + 1$



CRS

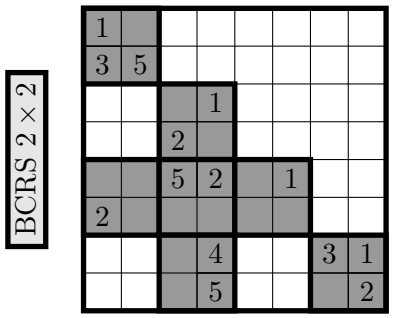
| Valores no nulos: | 1 | 3 | 5 | 1 | 2 | 5 | 2 | 1 | 2 | 4 | 3 | 1 | 5 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Índice de col: | 0 | 0 | 1 | 3 | 2 | 2 | 3 | 5 | 0 | 3 | 6 | 7 | 3 | 7 |

| Puntero a la fila | 0 | 1 | 3 | 4 | 5 | 8 | 9 | 12 | 14 |
|---|---|---|---|---|---|---|---|---|---|

Cantidad de elementos almacenados: 14 valores

Cantidad de accesos a memoria para un SpMV: 59 accesos

Porcentaje de llenado: 100 %



BCRS $2 \times 2$

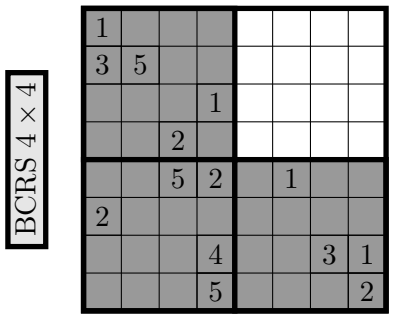
| Valores no nulos: | 1 | | | 1 | | | 5 | 2 | | 1 | | 4 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | 2 | | 2 | | | | | | | 5 | | 2 |
| Índice de col: | 0 | | 1 | | 0 | | 1 | | 2 | | 1 | | 3 | |

| Puntero a la fila | 0 | 1 | 2 | 5 | 7 |
|---|---|---|---|---|---|

Cantidad de elementos almacenados: 28 valores

Cantidad de accesos a memoria para un SpMV: 30 accesos

Porcentaje de llenado: 50 %



BCRS $4 \times 4$

| Valores no nulos: | 1 | | | | | | 5 | 2 | | 1 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | | | 2 | | | | | | | |
| | | | | 1 | | | | 4 | | | 3 | 1 |
| | | | 2 | | | | | 5 | | | | 2 |
| Índice de col: | 0 | | | | 0 | | | | 1 | | | |

| Puntero a la fila | 0 | 1 | 3 |
|---|---|---|---|

Cantidad de elementos almacenados: 48 valores

Cantidad de accesos a memoria para un SpMV: 23 accesos

Porcentaje de llenado: 29 %

**Figura 4.2**: Ejemplo de almacenamiento de una matriz por el formato BCRS.

**Formato comprimido por bloque de columnas (BCCS: Block Compressed Column Storage)**

Es análogo al anterior excepto que utiliza el formato CCS para acceder a los bloques

Estos dos últimos formatos presentan tanto ventajas como desventajas.

**Ventajas:**

- Disminuye el espacio destinado a almacenar índices.
- Las operaciones dispersas se descomponen en operaciones con pequeñas matrices densas.
- Se aprovecha la localidad espacial tanto en filas como en columnas.

**Desventajas:**

- Puede ser necesario almacenar demasiados 0's para respetar los tamaños de bloque.

- Sirve "solo" si la matriz posee bloques regulares.

**Formato comprimido por diagonales (CDS)**

Se almacena una matriz rectangular con las diagonales.

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & a_{15} & a_{16} \\ 0 & a_{22} & 0 & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 & 0 \\ 0 & a_{52} & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & a_{63} & 0 & a_{65} & a_{66} \end{pmatrix}$$

| | | | | | | |
|---|---|---|---|---|---|---|
| val$(:,-1)$ | 0 | 3 | 7 | 8 | 9 | 2 |
| val$(:,0)$ | 10 | 9 | 8 | 7 | 9 | -1 |
| val$(:,+1)$ | -3 | 6 | 7 | 5 | 13 | 0 |

**Jagged Diagonal Storage (JDS)**

1. Los elementos $n_z$ de la matriz se corren a la izquierda (se almacena un vector con los valores distintos de 0 y uno con el número de columna original de cada elemento).

2. Se ordenan las filas de acuerdo a la cantidad de $n_z$ (se almacena un vector de permutación).

3. Se almacena la matriz resultante por columna (se almacena en un vector con un puntero al comienzo de cada columna en los vectores anteriores).

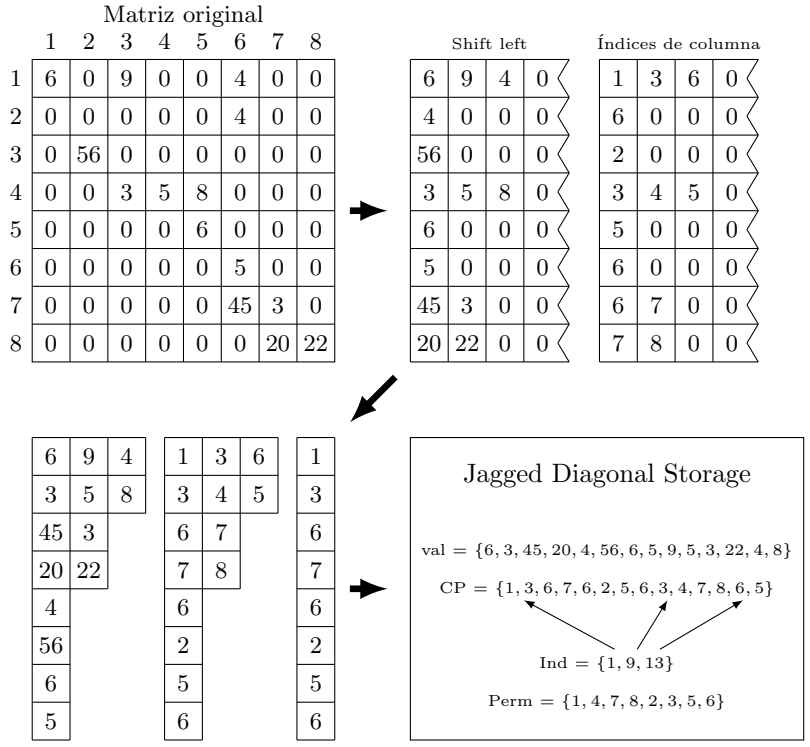



**Figura 4.3**: Formato Jagged Diagonal Storage

**Ventajas:**

- Como la cantidad de no ceros por fila suele ser mucho menor que la cantidad de no ceros por "jagged diagonal", este formato suele ocupar menos espacio que CSR.

**Desventajas:**

- Las filas se almacenan permutadas, por lo que hay que deshacer la permutación al operar con la matriz. La variante "transpuesta" de este formato (TJDS) apunta a solucionar esta desventaja.

### Bit Map

Es un formato pionero caído en desuso por mucho tiempo. Algunos trabajos recientes utilizan estos formatos para regularizar el acceso a memoria en dispositivos como GPUs.

Las matrices dispersas pueden representarse usando bloques de $8 \times 8$.

Se almacena un bitmap para cada bloque no nulo (entero de 64 bits)

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & a_{15} & a_{16} & 0 \\ 0 & a_{22} & 0 & a_{24} & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 & 0 & 0 \\ 0 & a_{52} & 0 & 0 & a_{55} & a_{56} & 0 \\ 0 & 0 & a_{63} & 0 & a_{65} & a_{66} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{77} \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$d = (a_{11}\ a_{12}\ a_{15}\ a_{16}\ a_{22}\ a_{24}\ a_{32}\ a_{33}\ a_{34}\ a_{41}\ a_{44}\ a_{52}\ a_{55}\ a_{56}\ a_{63}\ a_{65}\ a_{66}\ a_{77})$$

**ELLPACK (ELL)**

El formato *Ellpack-itpack*, utiliza una estructura densa de tamaño $dim \times k$, donde $dim$ es la cantidad de filas y $k = max_i((A_i))$, con $A_i$ la fila $i$-ésima de la matriz, es decir, la cantidad máxima de elementos no nulos por fila. La matriz dispersa es almacenada en dos matrices "densas" de tamaño $dim \times k$, una con las entradas no nulas de la matriz y otra con los índices de las columnas. Es necesario agregar explícitamente valores nulos para completar la primer matriz (*zero padding*). Este problema es menor cuando todas las filas de la matriz son de largos similares (el caso ideal son las matrices con cantidad constante de elementos por fila/columna). Dado que la estructura elegida es de tamaño $dim \times k$, es decir, tiene la misma cantidad de filas que la matriz original, no es necesario almacenar explícitamente los índices de fila ya que están implícitos en la estructura. Por otra parte, las matrices se almacenan por columnas para favorecer el acceso a memoria de la GPU durante la operación.

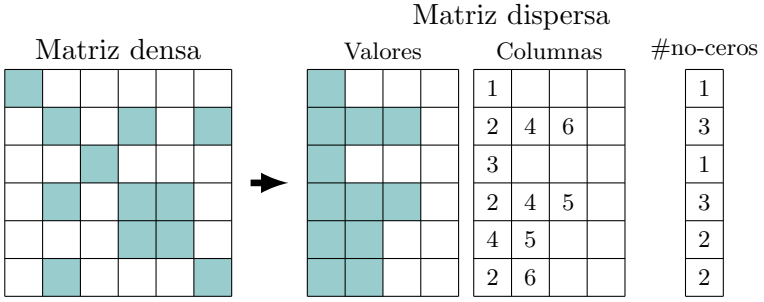



**Figura 4.4**: Formato ELLPACK

### 4.2.2. Formatos dinámicos

Resuelven el problema de los formatos estáticos al insertar elementos. Sin embargo, no son tan eficientes en almacenamiento/acceso como los formatos estáticos. Utilizan herramientas de almacenamiento dinámico (punteros) tales como listas enlazadas y vectores de listas.

Veamos algunos formatos dinámicos.

#### LLRCS (Linked List Row-Column Storage)

Se utiliza una multi-estructura bidimensional con dos vectores de tamaño filas (supongamos n) y columnas (supongamos n), en donde cada entrada es un puntero para recorrer la fila/columna correspondiente. A su vez, cada elemento guarda: valor, fila, columna, al siguiente en la fila y al siguiente en la columna.

La memoria necesaria para $n_z$ coeficientes no nulos es entonces

$n_z$(1 flotante (valor)+2 enteros (fila y columna)+2 punteros (siguiente fila y columna)+$2n$ punteros

#### LLRS (Linked List Row Storage)

Utiliza una estructura unidimensional de tamaño igual a la cantidad de filas, a partir de cada posición se puede obtener la lista de entradas de esa fila.

Memoria para $n_z$ coeficientes no nulos:

- $n_z$ puntos flotantes
- $n_z$ enteros
- $2n_z$ punteros (o $n_z$)
- el vector de entrada ($n$ punteros)

#### LLCS (Linked List Column Storage)

Utiliza una estructura unidimensional de tamaño igual a la cantidad de columnas, a partir de cada posición se puede obtener la lista de coeficientes no nulos de esa columna. Por ser análoga a LLRS, posee las mismas necesidades de memorias que este formato.

### 4.2.3. Otros formatos

#### Formatos compuestos

Se guarda la diagonal por un lado y el resto por separado.

#### Formatos multi-nivel

Utilizan estructuras tipo árbol: UB-tree, BUB-tree, R-tree, R*-tree, etc. Traslada ideas de índices, imágenes y otros.

# Capítulo 5

# High Performance Computing (HPC)

## 5.1. Arquitecturas paralelas

El modelo de computación estándar se basa en la Arquitectura de Von Neumann, en este modelo, se utiliza una única Unidad Central de Procesamiento (CPU) que ejecuta un solo programa y accede a una única memoria. La memoria, por su parte, consiste en operadores de lectura/escritura y dispositivos de almacenamiento.

Este modelo ha demostrado ser robusto y versátil, ya que separa al programador de la complejidad de la arquitectura subyacente. Además, ha permitido el desarrollo de las técnicas de programación estándar que utilizamos en la actualidad.

Sin embargo, con el avance de la computación, surgió la necesidad de abstraer aún más el hardware subyacente y lograr un mayor rendimiento. Para lograr esto, se extendió el modelo hacia la computación paralela, que ofrece varias alternativas:

- **Multicomputador:** Esta configuración consta de múltiples nodos, cada uno equipado con una CPU al estilo de Von Neumann. Estos nodos están interconectados mediante un mecanismo de comunicación. Esto permite que múltiples tareas se ejecuten simultáneamente en nodos independientes.

- **Computador Masivamente Paralelo:** En este enfoque, se utilizan muchos nodos, cada uno con una CPU simple de tipo Von Neumann. La interconexión entre nodos sigue una topología específica y diseñada para un rendimiento óptimo en tareas paralelas.

- **Multiprocesador de Memoria Compartida:** Esta configuración consta de nodos con CPU de Von Neumann, pero con acceso a una memoria compartida única. Esto permite que múltiples procesadores compartan datos en tiempo real y trabajen en tareas simultáneas.

- **Cluster:** Los cluster utilizan múltiples nodos, cada uno con su propia CPU tipo Von Neumann, y se comunican entre sí mediante una red de área local (LAN). Esto proporciona una infraestructura de alto rendimiento para aplicaciones distribuidas.

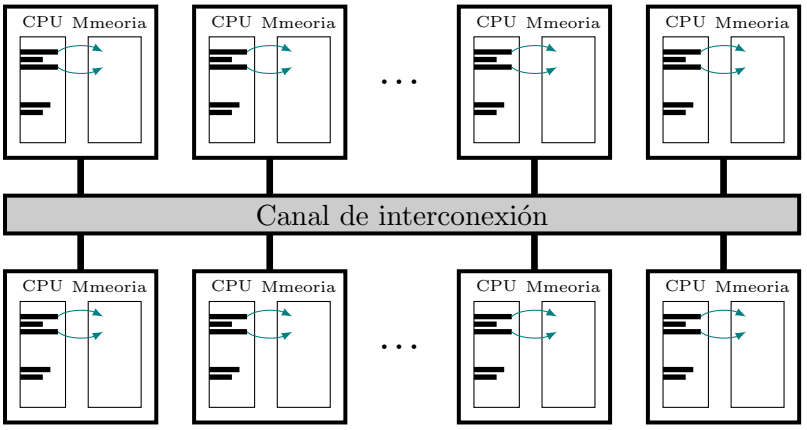



**Figura 5.1**: Mecanismo de interconexión entre nodos de un multicomputador

### 5.1.1. Categorización de Flynn

Una forma de categorizar las arquitecturas paralelas se basa en la cantidad de instrucciones y datos que pueden procesarse simultáneamente. Esta categorización se conoce como la categorización de Flynn. Se divide en cuatro categorías:

| | | Instrucciones | |
|---|---|---|---|
| | | Single Instrucción (SI) | Multi Instrucción (MI) |
| Datos | Single Datos (SD) | SISD | MISD |
| | Multi Datos (MD) | SIMD | MIMD |

donde:

- **SISD:** Es el modelo convencional de Von Neumann, donde una única instrucción opera sobre un solo conjunto de datos.
- **SIMD:** En este enfoque, se realiza el procesamiento paralelo de datos mediante una única instrucción que se aplica a múltiples conjuntos de datos simultáneamente. Esto se conoce comúnmente como computación vectorial.
- **MISD:** Involucra arrays sistólicos, donde múltiples instrucciones trabajan en un solo conjunto de datos. Aunque menos común, se utiliza en aplicaciones específicas que requieren este tipo de configuración.
- **MIMD:** Este es el modelo más generalizado, que admite múltiples instrucciones operando sobre múltiples conjuntos de datos en paralelo.

### Single Instruction Multiple Data (SISD)

El modelo SISD es un tipo de procesador capaz de realizar acciones secuencialmente. Estas acciones son controladas por un programa almacenado en una memoria que está conectada al procesador. En este enfoque, el hardware esta diseñado para dar soporte al procesamiento secuencial clásico, basado en el intercambio de datos entre memoria y registros del procesador, y la realización de operaciones aritméticas en ellos.

Sin embargo, con el tiempo, el modelo SISD no resultó ser suficiente para abordar los desafíos emergentes y las crecientes demandas de procesamiento. Surgieron nuevos problemas de dimensiones significativas, como la necesidad de manejar grandes volúmenes de datos y mejorar la precisión en las operaciones de cálculo.

Aunque se intentaron mejoras en las máquinas SISD, como el desarrollo de compiladores optimizadores de código y el aumento de la velocidad de reloj de los procesadores, estas mejoras tuvieron limitaciones. Además, se anticipa que el ritmo de mejora se desacelere debido a limitaciones físicas.

Es en este contexto que se desarrollaron las máquinas paralelas.

### Single Instruction Multiple Data (SIMD)

En este enfoque, un solo programa coordina y controla la ejecución de todos los procesadores. Esto es especialmente útil en aplicaciones con características uniformes. No obstante, se encuentra con limitaciones cuando las comunicaciones entre los procesadores están predeterminadas y no pueden adaptarse de manera dinámica. Este tipo de limitación es frecuente en tareas como el procesamiento de imágenes o cálculos que se basan en diferencias finitas, lo que restringe su aplicabilidad en estos contextos.

### Multiple Instruction Multiple Data (MIMD)

A diferencia de las arquitecturas SISD y MISD, los sistemas MIMD operan de manera asincrónica. Dentro de esta categoría, existen dos variantes: MIMD de memoria compartida, que se caracteriza por una fuerte interconexión entre sus componentes, y MIMD de memoria distribuida, que presenta una menor interdependencia entre ellos.

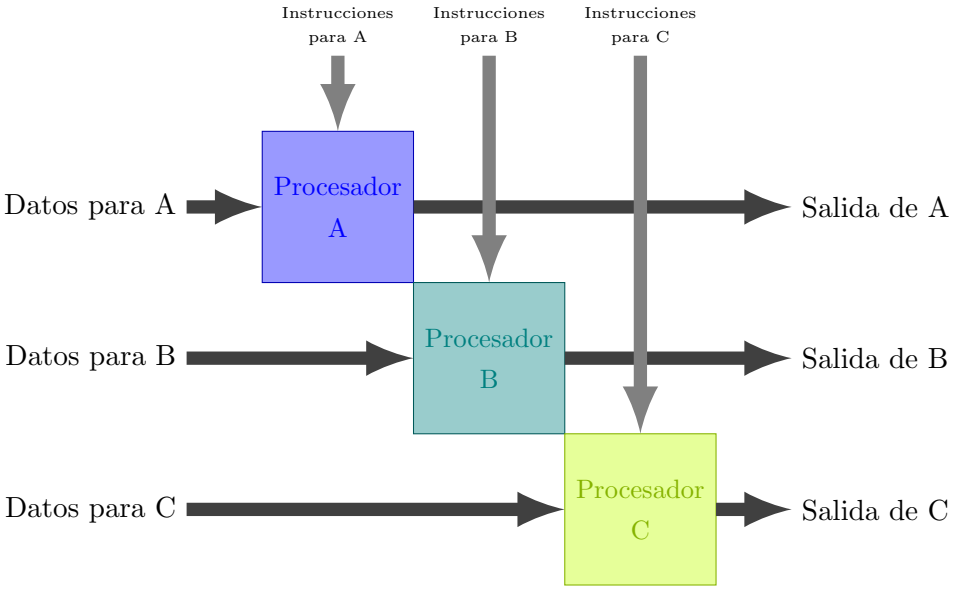



**Figura 5.2**: Arquitectura MIMD

- **Arquitectura MIMD con memoria compartida:** En esta configuración, la compartición de datos se lleva a cabo mediante dos enfoques principales:

  - **UMA (Uniform Memory Access):** Aquí, todos los procesadores pueden acceder a la memoria simultáneamente. Este modelo se encuentra comúnmente en sistemas multiprocesadores simétricos (SMP) con un número limitado de procesadores, generalmente en el rango de 32, 64 o 128, debido a las restricciones de ancho de banda del canal de acceso.

  - **NUMA (Non-Uniform Memory Access):** En este caso, se emplea una colección de memorias separadas que se combinan para formar un espacio de memoria direccionable. Sin embargo, algunos accesos a memoria son más rápidos que otros debido a la disposición física de las memorias, que están distribuidas físicamente de forma que cada memoria es local a un procesador/controlador de memoria y el acceso por parte de otros procesadores se realiza a través de este. Este enfoque se encuentra en sistemas multiprocesadores masivamente paralelos (MPP).

- **Arquitectura MIMD con Memoria Distribuida:** En contraste con la memoria compartida, en esta configuración no existe el concepto de memoria global. En su lugar, la comunicación y sincronización entre procesadores se realiza a través del intercambio de mensajes explícitos, lo que implica un mayor costo en comparación con la memoria compartida. Esta arquitectura es escalable y se adapta bien a aplicaciones diseñadas específicamente para esta topología, con la capacidad de admitir decenas de miles de procesadores.

### 5.1.2. Conectividad entre procesadores y factores determinantes de la eficiencia

A continuación se enumeran los factores críticos que influyen en la eficiencia de las arquitecturas mencionadas:

- **Ancho de Banda:** Este parámetro se refiere al número de bits que pueden transmitirse por unidad de tiempo.
- **Latencia de la Red:** Representa el tiempo necesario para que un mensaje viaje a través de la red de comunicación.
- **Latencia de las Comunicaciones:** Incluye no solo el tiempo de transmisión, sino también los tiempos de trabajo del software y los retrasos en la interfaz.
- **Latencia del Mensaje:** Es el tiempo que toma enviar un mensaje de longitud cero.
- **Valencia de un Nodo:** Hace referencia al número de canales que convergen en un nodo específico.
- **Diámetro de la Red:** Este valor corresponde al número mínimo de saltos requeridos para conectar los nodos más distantes entre sí. Ayuda a estimar el peor escenario en cuanto al retraso de un mensaje.
- **Ancho de Bisección:** Indica el número mínimo de enlaces necesarios para dividir la red en dos componentes independientes en caso de que estos enlaces no existieran.
- **Largo Máximo de un Tramo de Comunicación:** Es el máximo recorrido permitido para una comunicación efectiva.
- **Costo:** Representa la cantidad de enlaces de comunicación.

La configuración óptima de una red de este tipo cumple con los siguientes criterios:

- Ancho de Banda grande.
- Latencias (tanto de red como de comunicación y mensaje) bajas.
- Diámetro de la red reducido.
- Ancho de Bisección grande.
- Valencia constante e independiente del tamaño de la red.
- Largo máximo de un tramo de comunicación limitado y constante, independiente del tamaño de la red.
- Costo mínimo en términos financieros.

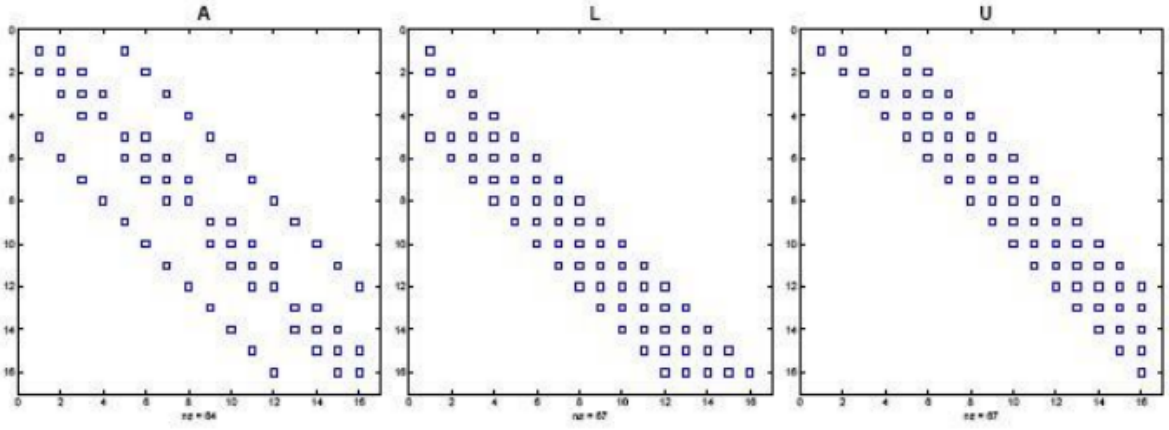



**Figura 5.3**: Modelos de conectividad entre procesadores (I)

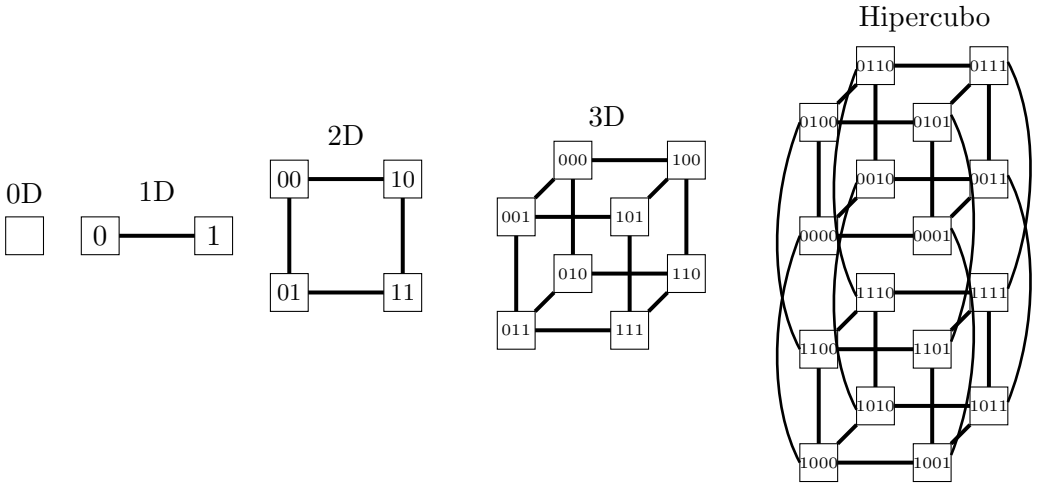



**Figura 5.4**: Modelos de conectividad entre procesadores (II)

## 5.2. Modelo de Programación Paralela

En el contexto de la programación paralela, es importante comprender que difiere significativamente de la programación tradicional en máquinas de Von Neumann. En el modelo de programación convencional, las operaciones se ejecutan secuencialmente, controladas por un programa almacenado en memoria. Esta abstracción se basa en una secuencia de operaciones que incluye cálculos aritméticos, operaciones de lectura/escritura en memoria y avance del contador de programa. Además, se aplican técnicas de programación modular para organizar el código.

Cuando se trata de programación en máquinas paralelas, se introducen complejidades adicionales. Esto implica la ejecución simultánea de múltiples procesos y requiere una comunicación y sincronización efectiva entre ellos. En este contexto, la modularidad se vuelve fundamental, ya que se deben manejar potencialmente una gran cantidad de procesos en ejecución simultánea.

### 5.2.1. Modelo conceptual de paralelismo

El paradigma de diseño y programación en sistemas paralelos es sustancialmente diferente al utilizado en aplicaciones secuenciales. La eficiencia de un algoritmo paralelo depende en gran medida de una estrategia efectiva para dividir el problema en tareas paralelas. Además, se debe considerar la disponibilidad de hardware específico.

En cuanto a los mecanismos de comunicación entre procesos, existen dos paradigmas principales en la computación paralela: la memoria compartida y el paso de mensajes. También existen modelos híbridos que combinan ambas técnicas según las necesidades de la aplicación.

La aplicación del paralelismo puede abordarse en distintos niveles:

- A nivel intrainstrucción, que involucra el hardware, como la utilización de pipelines.
- A nivel interinstrucción, que incluye el sistema operativo, la optimización del código y el trabajo del compilador.
- A nivel de procedimientos o tareas, donde se diseña y programan algoritmos específicos para tareas paralelas.
- A nivel de programas o trabajos, que se enfoca en aspectos más generales de diseño y funcionalidad.

En el diseño de aplicaciones paralelas, es importante mencionar la utilización de Grafos Dirigidos Acíclicos (ADGs). En estos grafos, los nodos representan tareas o procesos, es decir, partes del código que se ejecutan secuencialmente. Las aristas representan dependencias entre estas tareas, indicando cuáles deben ejecutarse antes que otras.

El objetivo de trabajar con ADGs es dividir el problema a resolver en tareas que puedan ejecutarse cooperativamente en múltiples procesadores. Para lograrlo, se deben definir las tareas

que pueden ejecutarse concurrentemente, lanzar y detener la ejecución de tareas, e implementar los mecanismos de comunicación y sincronización necesarios.

Es importante resaltar que no siempre es una buena decisión partir de un algoritmo secuencial para "paralelizar" una aplicación. En ocasiones, será necesario diseñar un nuevo algoritmo completamente diferente. En resumen, las etapas de diseño de aplicaciones paralelas involucran identificar el trabajo que puede realizarse en paralelo, dividir el trabajo y los datos entre los procesadores, y resolver los accesos a los datos, la comunicación entre procesos y las sincronizaciones para lograr un rendimiento eficiente en un entorno paralelo.

La planificación y diseño de aplicaciones paralelas se rige entonces por cuatro etapas esenciales:

1. **Descomposición:** En esta etapa, se identifica la concurrencia en el problema y se decide cómo explotarla. Se determina la cantidad de tareas, si serán estáticas o dinámicas, y se aplican criterios de división y asignación de recursos.

2. **Asignación:** Se asignan los datos a los procesos, considerando criterios estáticos o dinámicos, equilibrio de cargas y reducción de costos de comunicación y sincronización.

3. **Orquestación:** Aquí, se toman decisiones sobre la arquitectura, el modelo de programación, el lenguaje y las bibliotecas a utilizar. Se optimizan aspectos como la estructura de datos y la localidad de referencias para mejorar la comunicación y la sincronización.

4. **Mapeo (Scheduling):** Esta etapa involucra la asignación concreta de tareas a procesadores. Se utilizan criterios de rendimiento, utilización y reducción de costos de comunicación y sincronización.

Es esencial contar con mecanismos que permitan definir, controlar y sincronizar tareas. Estos mecanismos deben estar integrados en el lenguaje de programación utilizado o ser insertados por el compilador. Su función principal es facilitar la gestión eficiente del flujo de ejecución de tareas y el particionamiento de datos en un entorno paralelo.

En este sentido, se presentan algunos ejemplos de mecanismos utilizados en la programación paralela:

- Definición de Tareas y Control de Flujo: Algunos lenguajes, como C, ofrecen construcciones como "fork & join", que permiten definir tareas y controlar su ejecución concurrente. Pascal concurrente utiliza "parbegin-parend" para un propósito similar. Ada introduce el concepto de "task", mientras que PVM proporciona la funcionalidad de "spawn" para crear procesos paralelos.

- Coordinación y Sincronización: Para garantizar la coordinación entre tareas y evitar problemas de sincronización, se utilizan mecanismos como semáforos, monitores y barreras en entornos de memoria compartida. En sistemas de memoria distribuida, se recurre a mensajes

asincrónicos, como los proporcionados por C, PVM y MPI, o a mensajes sincrónicos, como el "rendezvouz" de Ada.

Es importante destacar que el particionamiento de datos es una tarea que generalmente recae en el diseñador del algoritmo paralelo, y la elección de los mecanismos adecuados depende en gran medida de las características específicas de la aplicación y del hardware disponible. La complejidad de este modelo aumenta debido a las particularidades de la computación paralela y distribuida, lo que resalta la importancia de una planificación cuidadosa y una implementación precisa para lograr un rendimiento óptimo en un entorno paralelo.

### 5.2.2. Problemas con la computación paralela-distribuida

La computación paralela y distribuida, si bien ofrece ventajas notables en términos de rendimiento y capacidad de procesamiento, conlleva diversos desafíos que deben abordarse cuidadosamente:

1. **Confiabilidad:**
   - Diversos componentes del sistema pueden experimentar fallas, desde nodos y tarjetas hasta caches y dispositivos de red. El uso de hardware no dedicado puede agravar estos problemas.
   - La compartición de recursos puede dar lugar a problemas de uso y congestión de tráfico, lo que hace que las condiciones de ejecución no sean repetibles.
2. **No Determinismo en la Ejecución:** Los programas en entornos paralelos tienen una gran cantidad de estados posibles, lo que dificulta la especificación de secuencias de control mediante enfoques tradicionales como los diagramas de flujo.
3. **Seguridad:** La necesidad de acceder a equipos remotos y gestionar datos distribuidos plantea preocupaciones en cuanto a la seguridad de los sistemas distribuidos.
4. **Dificultad en la Estimación de la Performance:** Debido al no determinismo inherente en la ejecución paralela, es complicado predecir con precisión la performance final de una aplicación. A menudo se recurre a criterios estadísticos.
5. **Dificultad en las Pruebas y Depuración:**
   - La multiplicidad de estados posibles y la asincronía dificultan la reproducción de escenarios y la depuración de programas.
   - La descentralización de datos y procesos requiere la implementación de herramientas de depuración en cada nodo del sistema.
6. **Incompatibilidades entre Productos y Plataformas (para Sistemas Heterogéneos):**

- La coexistencia de diversos sistemas operativos, como AIX, Solaris, Linux y otros, puede generar incompatibilidades.
- La diversidad de protocolos de comunicación, como TCP/IP, DEC-NET, IPX y otros, puede dificultar la interoperabilidad.
- La variedad de herramientas de software utilizadas, como PVM, MPI, C, HPF y más, requiere una cuidadosa consideración de la compatibilidad.

## 5.3. Técnicas de programación paralela

Se analizarán las técnicas de descomposición o particionamiento, que permiten dividir un problema en subproblemas a resolver en paralelo. El objetivo primario de la descomposición será dividir en forma equitativa tanto los cálculos asociados con el problema como los datos sobre los cuales opera el algoritmo.

**¿Cómo se puede alcanzar este objetivo de descomposición eficiente?**

- Es crucial definir al menos un orden de magnitud más de tareas que de procesadores disponibles, garantizando así una utilización óptima de los recursos disponibles.
- Evitar cálculos y almacenamientos redundantes, minimizando la duplicación de esfuerzos y datos.
- Generar tareas de tamaño comparable para mantener un equilibrio en la carga de trabajo entre los procesadores.
- Diseñar tareas escalables, de modo que su tamaño pueda adaptarse según las dimensiones del problema en cuestión.
- Considerar diversas alternativas de descomposición siempre que sea posible, evaluando enfoques tanto en la descomposición de datos como en la de tareas.

La elección entre una estrategia de descomposición centrada en los datos o en las tareas determinará la técnica específica de programación paralela a aplicar en un proyecto dado. Estas técnicas se adaptarán a la naturaleza del problema y los recursos disponibles, buscando maximizar la eficiencia y el rendimiento en la ejecución paralela.

### 5.3.1. Descomposición de dominio

La descomposición de dominio es una estrategia que se enfoca en el particionamiento de los datos de un problema, también conocida como programación de datos paralelos (data parallel). La idea fundamental es dividir los datos en fragmentos de tamaño aproximadamente igual. Posteriormente, se distribuyen los cálculos necesarios, asociando cada operación con los datos en los que debe operar.

Los datos que se pueden dividir utilizando esta técnica pueden ser diversos, incluyendo la entrada del programa, la salida calculada por el programa o datos intermedios generados durante la ejecución. Aunque no existe una regla general para determinar la forma de dividir los datos, algunas sugerencias surgen de la estructura o "geometría" del problema y la idea de comenzar con las estructuras de datos más grandes o las más frecuentemente accedidas.

La técnica de descomposición de dominio se asocia comúnmente con la estrategia *"Divide y Conquista"* y se implementa en modelos de programas paralelos como SIMD (Single Instruction Multiple Data) y SPMD (Single Program Multiple Data). Un ejemplo concreto de su aplicación es la resolución de ecuaciones, donde se discretiza la solución, se dividen los dominios de cálculo y se ejecuta el mismo programa en cada dominio, con comunicación necesaria para los cálculos en los bordes.

Esta técnica es aplicable tanto en modelos SIMD sobre arquitecturas de memoria compartida como en modelos SPMD sobre arquitecturas de memoria distribuida.

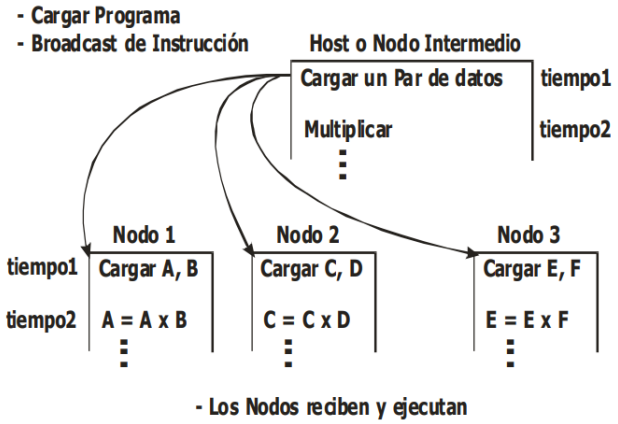



**Figura 5.5**: Modelo de programa SIMD

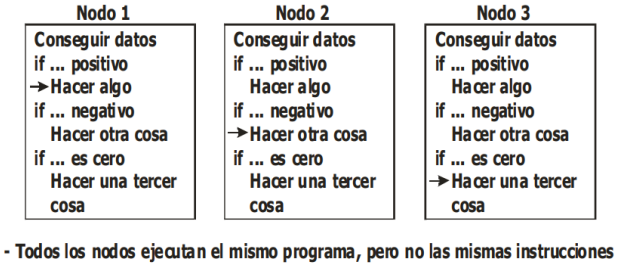



**Figura 5.6**: Modelo de programa SPMD

### 5.3.2. Descomposición funcional

La descomposición funcional es una estrategia de programación paralela que se enfoca en la partición de las operaciones de un problema, conocida como control paralelo (control parallel). El

objetivo principal es dividir el procesamiento en tareas que sean independientes entre sí.

En esta técnica, primero se dividen las operaciones en tareas disjuntas, lo que significa que no hay interdependencia entre ellas. Luego, se analizan los datos que serán utilizados por las tareas definidas. Si los datos son también disjuntos entre sí, se logra un particionamiento completo. Sin embargo, si los datos no son disjuntos y existen interdependencias, se produce un particionamiento incompleto. En este último caso, se requiere replicar los datos o establecer comunicación entre los procesos asociados a las diferentes tareas para garantizar que tengan acceso a la información necesaria.

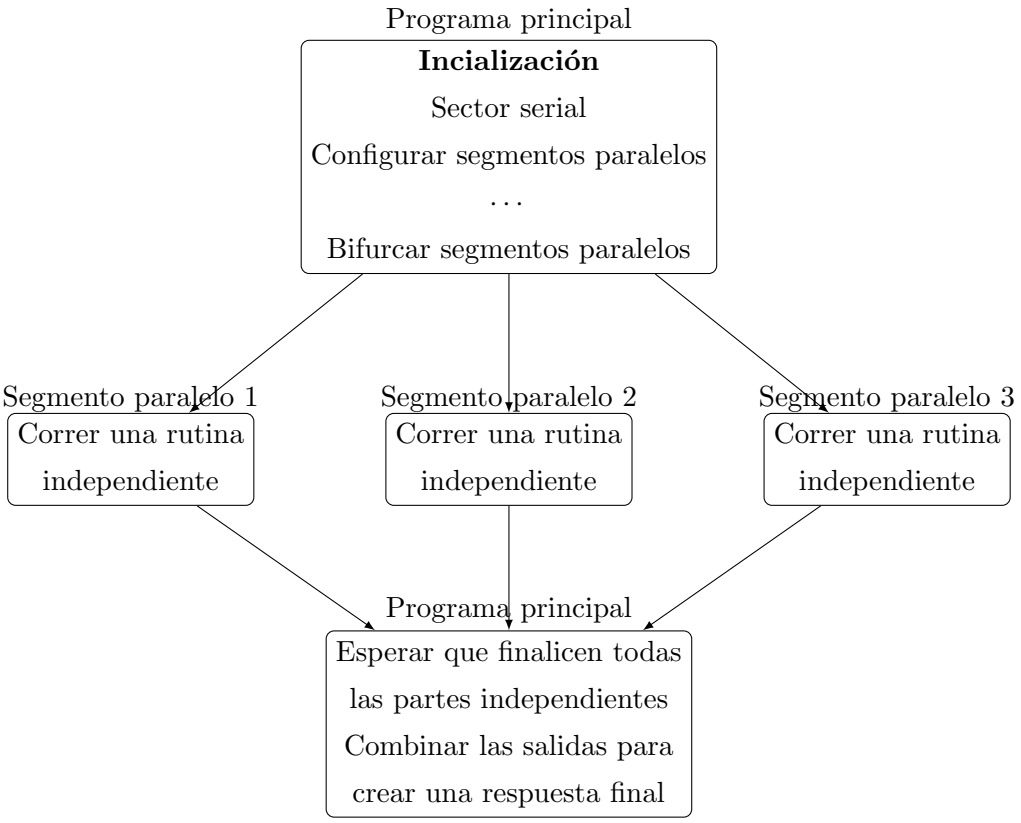



**Figura 5.7**: Caption

### 5.3.3. Modelos híbridos

Los modelos híbridos son aquellos que incorporan dos o más tipos de programación paralela dentro del mismo programa. Esta estrategia es comúnmente utilizada en programas paralelos distribuidos en entornos de Internet, donde casi siempre existe la posibilidad de aprovechar recursos ociosos adicionales.

### 5.3.4. Paralelización de aplicaciones existentes

En el diseño y la "paralelización" de aplicaciones, a menudo no se dispone de recursos, especialmente tiempo, para diseñar una aplicación paralela-distribuida desde cero siguiendo criterios y técnicas de programación específicos. En muchas ocasiones, la tarea consiste en obtener resultados aceptables en términos de eficiencia computacional adaptando programas secuenciales existentes a modelos de programación paralela. Esto se conoce comúnmente como "paralelizar" una aplicación existente.

Sin embargo, esta adaptación plantea desafíos. La utilización de un código existente que no fue diseñado para ejecutarse en múltiples recursos computacionales puede resultar en modelos de programación enmarañados y una especie de "contaminación" del código heredado.

Para abordar este proceso de paralelización, se deben analizar varios aspectos. Primero, se debe determinar si existe una partición funcional evidente en el código. Los programas modulares suelen ser más fáciles de paralelizar. Luego, se debe evaluar si es posible particionar los datos y cuál es la relación entre el procesamiento y los datos. Si existen muchas variables globales en el código, se debe abordar cuidadosamente el manejo de recursos compartidos, considerando la posibilidad de utilizar un servidor de variables globales.

En este contexto, también es esencial definir modelos de comunicación entre procesos paralelos. Estos modelos determinan cómo los procesos paralelos se comunican y sincronizan entre sí. Algunos ejemplos de modelos de comunicación incluyen el modelo maestro-esclavo (master-slave), el modelo cliente-servidor y el modelo peer to peer, que permiten establecer la interacción y coordinación necesarias entre los componentes paralelos de una aplicación.

#### Modelo Master-Slave

El modelo Master-Slave es uno de los paradigmas de comunicación más simples. Aquí se generan un conjunto de subproblemas y procesos destinados a resolverlos. La estructura de este modelo implica la presencia de un proceso distinguido conocido como *"master"* (maestro) y uno o varios procesos idénticos llamados *"slaves"* (esclavos).

En este esquema, el proceso master desempeña un papel central al controlar y coordinar la ejecución de los procesos slave. El proceso master se encarga de distribuir las tareas y los datos necesarios a los procesos slave para su procesamiento. Los procesos slave, por su parte, llevan a cabo las tareas asignadas y, en general, no mantienen comunicación directa entre sí, lo que implica que las dependencias entre las tareas realizadas por los esclavos suelen ser mínimas o nulas.

El flujo de trabajo en el modelo Master-Slave se inicia cuando el proceso master lanza a los procesos slave y les proporciona los datos necesarios para su ejecución. Una vez establecida esta jerarquía, el control del programa se ejerce desde el proceso master hacia los procesos slave. La comunicación entre los esclavos suele limitarse a la transmisión de los resultados de las tareas asignadas al proceso master.

Es importante destacar que existen variantes tanto sincrónicas como asincrónicas en el modelo Master-Slave. La elección entre estos enfoques depende de las necesidades específicas de la aplicación.

Aunque este paradigma es sencillo en su estructura, su implementación requiere abordar aspectos como la programación del mecanismo de lanzamiento de tareas, la distribución eficiente de datos, el control del proceso master sobre los procesos slave y, en algunos casos, la sincronización entre los componentes. La selección adecuada de recursos es fundamental para garantizar un equilibrio de cargas y, por lo tanto, optimizar el rendimiento de las aplicaciones que siguen este modelo.

#### Modelo Cliente-Servidor

El modelo Cliente-Servidor es un enfoque de comunicación que divide los procesos en dos categorías distintas: los procesos cliente y los procesos servidor. En este paradigma, los procesos de una clase, denominados *"clientes"*, solicitan servicios a los procesos de otra clase, conocidos como *"servidores"*, que responden a estos pedidos proporcionando los servicios solicitados.

Este modelo puede ser utilizado tanto para comunicar procesos que se ejecutan en un único equipo como para aplicaciones distribuidas en una red de computadoras. La principal ventaja del modelo Cliente-Servidor radica en su versatilidad, ya que permite establecer una comunicación eficiente entre aplicaciones distribuidas que funcionan simultáneamente como clientes y servidores para diferentes servicios.

En un entorno distribuido, el modelo Cliente-Servidor resulta especialmente poderoso, ya que posibilita la interacción entre múltiples nodos de una red de manera organizada y eficiente. Los procesos cliente envían solicitudes de servicios a través de la red a los procesos servidor, que se encargan de atender estas peticiones y proporcionar los servicios correspondientes.

## 5.4. Medidas de performance

Las medidas de rendimiento desempeñan un papel fundamental en la evaluación y optimización de algoritmos paralelos. Su objetivo principal es proporcionar una estimación precisa del desempeño de estos algoritmos y permitir su comparación con sus contrapartes secuenciales. Algunos de los factores intuitivos clave que se utilizan para evaluar el rendimiento incluyen el tiempo de ejecución y la utilización de los recursos disponibles.

El tiempo de ejecución, una medida tradicionalmente utilizada, se refiere al período de tiempo que lleva completar la ejecución de una aplicación o algoritmo. Sin embargo, en el contexto de la programación paralela, el tiempo de ejecución puede verse influenciado por varios factores adicionales, como el almacenamiento de datos en dispositivos y la transmisión de datos entre procesos. Estos elementos pueden tener un impacto significativo en el rendimiento global de una aplicación paralela.

La utilización de los recursos disponibles y la capacidad de aprovechar al máximo el poder de cómputo para resolver problemas más complejos o de mayor dimensión son características altamente deseables en las aplicaciones paralelas. Por lo tanto, evaluar la utilización eficiente de los recursos de cada plataforma de hardware se vuelve crucial al medir su rendimiento.

En este contexto, el tiempo total de ejecución de una aplicación paralela se convierte en una medida fundamental del rendimiento. Esta medida es relativamente simple de medir y proporciona información valiosa sobre el esfuerzo requerido para resolver un problema en particular utilizando un algoritmo paralelo. En resumen, las medidas de rendimiento son herramientas esenciales para comprender y mejorar la eficiencia de los algoritmos paralelos y garantizar su óptimo funcionamiento en aplicaciones del mundo real.

El tiempo en estado ocioso, es decir, el tiempo en el que los procesadores no están realizando tareas de cómputo, puede ser significativo y afectar el rendimiento general.

Las razones detrás del tiempo en estado ocioso pueden incluir la ausencia de recursos de cómputo disponibles o la falta de datos sobre los cuales operar en un momento dado. Para abordar este problema, es fundamental que el diseño de algoritmos paralelos tenga como objetivo minimizar el tiempo en estado ocioso.

Existen diversas soluciones para mitigar este problema, como la implementación de técnicas de balance de cargas que distribuyan equitativamente los requerimientos de cómputo entre los procesadores disponibles. Además, en algunos casos, puede ser necesario rediseñar el programa para garantizar una distribución adecuada de los datos, lo que contribuirá a reducir el tiempo en estado ocioso y mejorar el rendimiento general de la aplicación paralela.

### 5.4.1. Speed-Up

El *Speed-Up* es una medida esencial para evaluar la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores utilizados en comparación con el rendimiento obtenido al ejecutar la misma aplicación en un solo procesador. Distinguimos entre dos tipos de Speed-Up:

**Definición 5.4.1.1** (Speed-Up absoluto)**.** *El Speed-Up absoluto se define como:*

$$S_N = \frac{T_0}{T_N} \tag{5.1}$$

*donde:*

- $T_0$ *representa el tiempo de ejecución del mejor algoritmo serial disponible para resolver el problema (el más rápido).*
- $T_N$ *es el tiempo de ejecución del algoritmo paralelo cuando se utiliza un total de N procesadores.*

**Definición 5.4.1.2** (Speed-Up algorítmico)**.** *El Speed-Up algorítmico se define como:*

$$S_N = \frac{T_1}{T_N} \tag{5.2}$$

*donde $T_1$ es el tiempo serial y $T_N$ es el tiempo paralelo de ejecución.*

El Speed-Up algorítmico compara el mismo programa utilizando un hilo o proceso y utilizando varios hilos. Es utilizado para evaluar la mejora de rendimiento en términos de tiempos de ejecución en programas paralelos cuando el speedup absoluto es complicado de calcular debido a la falta de conocimiento sobre el mejor algoritmo serial posible para un problema determinado.

Una métrica que evalúa qué tan bien se está utilizando un conjunto dado de procesadores para acelerar una aplicación en paralelo, o qué tan lejos de lograr un speedup ideal se encuentra el programa es la *Eficiencia*.

**Definición 5.4.1.3** (Eficiencia)**.** *La eficiencia se define como:*

$$E_N = \frac{T_1}{N \cdot T_N} \tag{5.3}$$

*es decir, $E_N = \frac{S_N}{N}$ Corresponde al valor normalizado del Speed-Up algorítmico respecto de la cantidad de procesadores utilizados.*

Cuando la eficiencia se acerca a uno, representa una situación casi ideal en términos de mejora de rendimiento. En este escenario, los recursos computacionales se utilizan de manera extremadamente eficaz, lo que significa que la aplicación paralela está obteniendo un beneficio casi lineal del aumento en el número de procesadores utilizados. En resumen, una eficiencia cercana a uno indica una utilización muy efectiva de los recursos paralelos disponibles.

La situación ideal es lograr un Speed-Up lineal, es decir, al utilizar $N$ procesadores obtener una mejora de factor $N$. En algunas situaciones, los programas se componen de un gran número de tareas independientes que requieren casi nula sincronización y pueden ser distribuidas de forma pareja entre varios procesadores. Este tipo de problemas es comúnmente referido como *embarrassingly parallel* y el speedup al utilizar varios procesadores se acercará a un speedup lineal.

A pesar de los esfuerzos para lograr un speedup lineal, muchos factores afectan el desempeño de los programas paralelos existen con frecuencia tiempos ociosos en algunos procesadores. Para que el speedup sea lineal, todos los procesadores deben estar ocupados todo el tiempo (al mismo nivel que el procesador que resuelve el programa secuencial). De existir tiempos ociosos, algunos procesadores tomarán un tiempo mayor que $T_1/N$. Esto significa que utilizar $N$ procesadores no garantiza una mejora de factor $N$ en el rendimiento y, por lo general, el speedup será sublineal.

Existen varios factores que pueden impedir el crecimiento lineal del speedup, como las demoras introducidas por las comunicaciones, el overhead en el intercambio de datos, el overhead en el trabajo de sincronización, la existencia de tareas no paralelizables y los cuellos de botella en el acceso al hardware necesario. Estos factores pueden llevar a que el uso de más procesadores no sea necesariamente beneficioso para la performance de la aplicación.

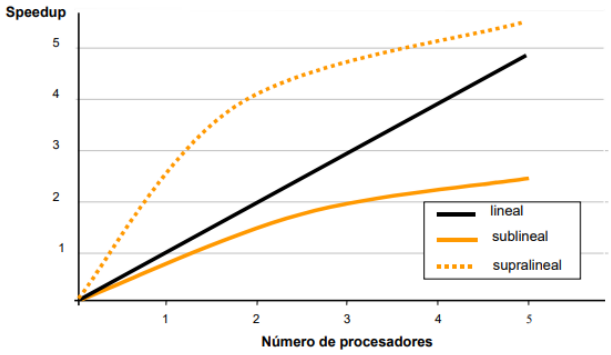



**Figura 5.8**: Tipos de Speed-Up

En otros casos es incluso posible lograr un speedup superlineal, es decir, $T_1/T_N > N$. Por ejemplo, un programa secuencial puede utilizar tanta memoria que requiera el intercambio constante de datos con el disco o memorias más lentas. Al repartir estos datos entre varios procesadores o memorias, es posible que todos puedan alojarse en memorias rápidas, por lo que la mejora será mayor que la derivada de paralelizar la tarea que antes era secuencial.

#### Ley de Amdahl - 1967

La ley de Amdahl indica que existe un límite respecto a cuánto se puede mejorar el rendimiento de un programa agregando más procesadores. Podría enunciarse de la siguiente manera:

*"La parte serial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo técnicas de paralelismo."*

Esta ley subraya que la parte secuencial del programa actúa como un cuello de botella que limita la mejora del rendimiento, es decir, la razón para utilizar un número mayor de procesadores debe ser resolver problemas más grandes o más complicados, y no para resolver más rápido un problema de tamaño fijo.

Esto se relaciona con los conceptos de escalabilidad fuerte y débil que se presentan a continuación.

### 5.4.2. Escalabilidad

La escalabilidad se refiere a la capacidad de un programa parlelo de mejorar su rendimiento al aumentar el número de procesadores. Es una característica fundamental y altamente deseable en los algoritmos paralelos y distribuidos.

Distinguimos entre dos tipos de escalabilidad:

- La *escalabilidad fuerte* equivale a la definición de speed-up algorítmico que se dio anteriormente. En otras palabras, indica hasta qué cantidad de procesadores se puede obtener un speedup cercano a lineal para un tamaño de entrada dado. Coloquialmente se puede decir

que “tal programa para un tamaño de entrada de tanto escala hasta 200 procesadores”.

- La *escalabilidad débil* se refiere al comportamiento del programa paralelo al aumentar el tamaño de la entrada y la cantidad de procesadores de forma de que cada procesador mantenga aproximadamente la misma carga de trabajo. En otras palabras, diremos que un programa tiene escalabilidad débil si se puede lograr que ante un aumento del tamaño de la entrada, el tiempo de ejecución se mantenga aproximadamente constante aumentando la cantidad de procesadores en la misma proporción.

### 5.4.3. Utilización de Recursos Disponibles

Una perspectiva adicional en la medición del rendimiento de aplicaciones paralelas se centra en la utilización de los recursos disponibles. Esta métrica mide el porcentaje de tiempo durante el cual un procesador está siendo utilizado durante la ejecución de una aplicación paralela.

$$\text{USO} = \frac{\text{Tiempo ocupado}}{\text{Tiempo ocioso} + \text{Tiempo ocupado}}$$

Un objetivo importante es mantener valores equitativos de utilización entre todos los procesadores en una máquina paralela. Esto asegura que los recursos se estén utilizando de manera eficiente y que no haya desequilibrios significativos en la carga de trabajo entre los procesadores.

#### Factores que Afectan el Rendimiento

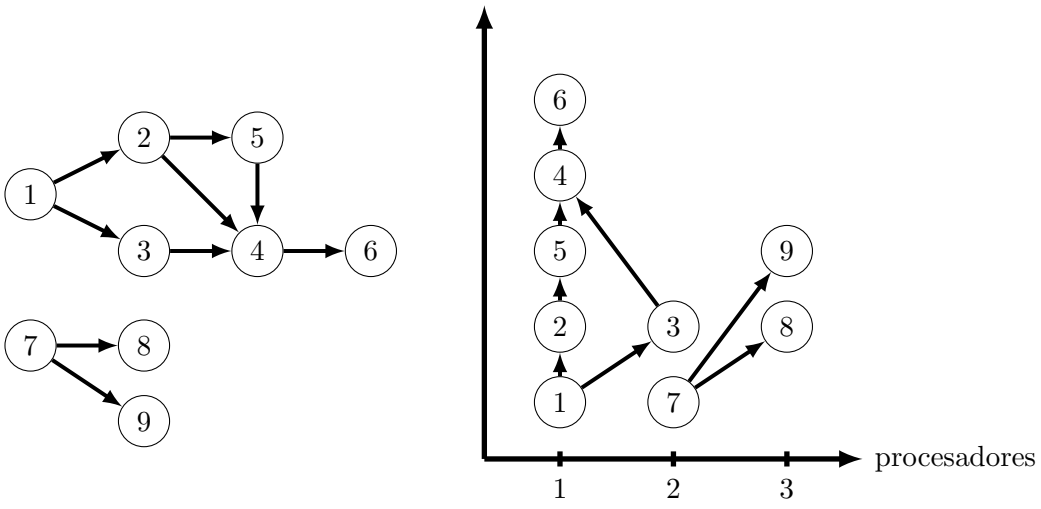
Uno de los factores cruciales que influyen en la eficiencia de las aplicaciones paralelas es la “granularidad”, que se refiere a la cantidad de trabajo realizada por cada nodo o procesador. Puede variar desde una granularidad extremadamente fina (operaciones muy pequeñas) hasta una granularidad gruesa (procesos completos). Aumentar la granularidad puede disminuir el sobrecosto de control y comunicación, pero también puede reducir el grado de paralelismo disponible.

El objetivo del diseño de aplicaciones paralelas es lograr un equilibrio entre el sobrecosto de sincronización y comunicación y el grado de paralelismo obtenido.

## 5.5. Scheduling o Mapeo

El “scheduling” o mapeo es un proceso que implica asignar recursos a los múltiples procesos que se ejecutarán en paralelo. Esta asignación determina dónde y cuándo se llevará a cabo cada tarea paralela. Las dependencias entre estas tareas juegan un papel importante en la toma de decisiones sobre la asignación de recursos.

Al realizar el proceso de scheduling se enlaza el algoritmo, que se puede representar con un grafo, con el hardware, es decir los procesadores y el tiempo relativo a cada tarea.




El objetivo principal del mapeo es optimizar ciertos criterios clave para mejorar el rendimiento de la aplicación paralela, como el tiempo total de ejecución, la utilización eficiente de los recursos disponibles y el balance de cargas entre los procesadores.

Para lograr esta optimización, se utilizan técnicas de programación que aplican algoritmos de planificación y asignación de recursos. Estos algoritmos toman en cuenta las dependencias de datos y comunicaciones entre las tareas paralelas, así como la topología de la red en sistemas distribuidos.

Algunas estrategias comunes en el mapeo incluyen:

- **Asignación en Procesadores Diferentes:** Las tareas que pueden ejecutarse simultáneamente se asignan a procesadores distintos para aprovechar al máximo el paralelismo.
- **Agrupación de Tareas Comunicativas:** Las tareas que se comunican con alta frecuencia se asignan al mismo procesador o a procesadores cercanos para minimizar la latencia de comunicación.

Es importante destacar que existen mecanismos teóricos para asignar recursos en función de los diferentes modelos de descomposición en tareas y arquitecturas paralelas estudiadas. Por ejemplo, en sistemas con topologías específicas, como mallas 2D o hipercubos, se aplican estrategias particulares de asignación de recursos para maximizar la eficiencia y el rendimiento de la aplicación paralela.

## 5.6. Balance de Cargas

El balance de cargas es un factor crítico en el rendimiento de aplicaciones distribuidas que se ejecutan en una red. El objetivo principal de esta técnica es evitar que el desempeño global del sistema se vea comprometido debido a demoras en tareas individuales. Este aspecto adquiere una importancia significativa en entornos no dedicados, donde múltiples tareas compiten por recursos compartidos.

### 5.6.1. Técnicas de balance de cargas

Las técnicas de balance de cargas, también conocidas como "técnicas de despacho", se utilizan para lograr un uso eficiente de los recursos en aplicaciones distribuidas, es decir, tratar de mantener los procesadores ocupados la mayor cantidad posible de tiempo y minimizar las comunicaciones entre los procesos. Estas técnicas se pueden clasificar en tres categorías principales:

- **Técnicas Estáticas (Planificación):** En estas técnicas, las decisiones de despacho se toman tempranamente y se mantienen sin cambios durante la ejecución de la aplicación. Son efectivas en entornos de redes poco cargadas, pero pueden fallar en ambientes compartidos con carga variable, ya que no tienen en cuenta las fluctuaciones de carga de la red.
- **Técnicas Dinámicas (Al Momento del Despacho):** Estas estrategias determinan qué procesador se asigna a una tarea durante la ejecución de la aplicación. Son comunes en el modelo maestro-esclavo (Master-Slave) y consideran la situación en el momento del despacho exclusivamente. Son efectivas en entornos compartidos con carga variable, ya que tratan de aprovechar las fluctuaciones de carga de la red.
- **Técnicas Adaptativas:** Las técnicas adaptativas realizan el despacho de tareas teniendo en cuenta el estado actual de la red. Pueden incluso incorporar herramientas de predicción del futuro y utilizan técnicas de migración de procesos para aprovechar al máximo las fluctuaciones de carga de la red.

### 5.6.2. Parámetros relevantes para la determinación de la carga

La determinación precisa de la carga en una aplicación distribuida implica considerar varios parámetros importantes:

- **Consumo de CPU:** Esto se refiere al porcentaje de uso de la CPU o la cantidad de operaciones por segundo que realiza.
- **Uso de Disco:** Se refiere a los bloques transferidos desde el controlador al dispositivo de disco, lo que indica la carga de trabajo en el almacenamiento.
- **Tráfico de Red:** Involucra el número de paquetes transmitidos y recibidos en la red, lo que proporciona información sobre la carga en las comunicaciones.

## 5.7. Herramientas

### 5.7.1. Multi-Threading

Los threads, o hilos de ejecución, son una característica fundamental en sistemas operativos modernos que permiten una mayor utilización de los recursos del procesador y una comunicación

más eficiente entre tareas. A continuación, se explorarán los conceptos clave relacionados con los threads y las ventajas y desventajas de su implementación en sistemas operativos.

En los sistemas operativos tradicionales de tipo UNIX, cada proceso tenía su propio espacio de direccionamiento y un único hilo de control. Sin embargo, a finales de los años 80, con la creciente necesidad de procesamiento, surgieron los sistemas multiprocesadores. Los diseñadores de sistemas operativos se centraron en desarrollar sistemas escalables para aprovechar al máximo estos multiprocesadores.

Los threads ofrecen una serie de beneficios significativos:

1. **Mejor Aprovechamiento de los Recursos del Procesador:** En entornos multiprocesadores, los threads permiten una distribución más efectiva de la carga de trabajo, lo que lleva a un uso más eficiente de los recursos del procesador. **Rapidez en la Comunicación:** Los threads pueden comunicarse entre sí de manera más rápida y eficiente que los procesos independientes.

2. **División de Aplicaciones en Módulos Funcionales:** Los threads facilitan la división de aplicaciones en módulos funcionales, lo que simplifica el diseño y la programación.

3. **Mejor Rendimiento en Tiempo de Ejecución:** En comparación con procesos independientes, los threads suelen lograr un mejor rendimiento en términos de tiempo de ejecución.

Sin embargo, también existen desventajas en la implementación de threads:

1. **Programación Más Difícil:** La programación con threads puede ser más complicada debido a la necesidad de gestionar la concurrencia y la sincronización.

2. **Dificultad en la Detección de Errores:** En entornos con múltiples threads, puede ser más difícil detectar y depurar errores.

Es importante comprender cómo se gestionan y comparten los recursos entre los threads en un proceso. Aquí se describen los aspectos clave de la compartición de recursos en threads:

- Compartición del Espacio de Direccionamiento: Todos los threads dentro de un proceso comparten el mismo espacio de direccionamiento. Esto significa que pueden acceder a las mismas variables globales y estructuras de datos sin restricciones. Sin embargo, esta compartición conlleva el riesgo de condiciones de carrera y la necesidad de sincronización para evitar problemas.

- Ausencia de Protección entre Threads: A diferencia de los procesos independientes, no existe ningún tipo de protección entre los threads dentro de un proceso. Esto significa que un thread puede modificar los recursos de memoria utilizados por otro thread sin restricciones. Es responsabilidad del programador implementar mecanismos de sincronización para garantizar un acceso seguro a los recursos compartidos.

- Información Privada de Cada Thread: Cada thread dentro de un proceso mantiene su propia información privada, que incluye un contador de programa, una pila (stack) para el almacenamiento de llamadas a funciones, un conjunto de registros y su propio estado. Esta información es específica de cada thread y se utiliza para mantener el contexto de ejecución de ese thread en particular.

- Recursos Compartidos: A pesar de la información privada de cada thread, los threads dentro de un proceso comparten ciertos recursos, como el espacio de direccionamiento de memoria, archivos abiertos y señales del sistema. Esto permite una comunicación más eficiente y una gestión más sencilla de estos recursos compartidos.

Los threads se pueden implementar a nivel de usuario o a nivel de núcleo del sistema operativo. En el caso de los threads a nivel de usuario, la gestión de los threads se realiza mediante una biblioteca de usuario sin la intervención del sistema operativo. En cambio, los threads a nivel de núcleo son gestionados directamente por el sistema operativo y pueden aprovechar mejor las capacidades del multiprocesador.

#### Threads a Nivel de Núcleo del Sistema

Los threads a nivel de núcleo son soportados directamente por el sistema operativo. El sistema operativo proporciona soporte para la creación, planificación y administración de threads en el entorno del núcleo. Esto permite que el sistema operativo reconozca a cada thread como una unidad planificable de manera independiente. Esto es beneficioso ya que el bloqueo de un thread de un proceso no afecta a los demás threads del mismo proceso y permite un mayor nivel de paralelismo en sistemas multiprocesador.

Sin embargo, las desventajas incluyen un cambio de contexto más costoso en términos de ciclos de CPU y la necesidad de mantener más estructuras en memoria por parte del sistema operativo.

#### Threads a Nivel de Usuario

Los threads a nivel de usuario son implementados en una biblioteca de usuario. Esta biblioteca proporciona soporte para la creación, planificación y administración de threads sin la intervención del sistema operativo. El sistema operativo no es consciente de la existencia de threads a nivel de usuario. Esto resulta en un rápido cambio de contexto entre los threads, ya que el núcleo del sistema no interviene.

Sin embargo, en entornos multiprocesador, solo se ejecuta un thread a la vez, lo que puede llevar a recursos de procesador ociosos. Además, si el thread en ejecución se bloquea, puede bloquear a todos los demás threads en el sistema.

## 5.8. OpenMP

OpenMP es una extensión para los lenguajes de programación C y Fortran que facilita la paralelización y el manejo de hilos. Por ejemplo, utiliza directivas del compilador para describir la ejecución paralela de las iteraciones de un *for*.

Una ventaja significativa de OpenMP sobre MPI es su facilidad de programación, ya que permite convertir un código secuencial en uno paralelo de manera incremental, mientras que MPI requiere una conversión completa de código secuencial a un programa con memoria distribuida.

Muchos compiladores, como `gcc` o el compilador de Intel, soportan las extensiones de OpenMP. En Fortran, las directivas se colocan en comentarios, y en C, se utilizan directivas `#pragma`. Esto hace que el código OpenMP sigua siendo válido para compiladores que no soportan OpenMP. Para poder utilizar OpemMP, además de compilar el programa utilizando la flag correspondiente (por ejemplo -fopenmp en gcc), y incluir el archivo de cabecera correspondiente (`#include <omp.h`

en C), los programas deben estar enlazados a la biblioteca que maneja el *runtime* de OpenMP (por ejemplo `libopenmp.so`).

Un ejemplo sencillo puede ser el siguiente:

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

En ese caso, todas las iteraciones son independientes y el *runtime* de OpenMP intentará distribuirlas en tantos threads como procesadores tenga el sistema o en los que se especifique mediante la variable de entorno `OMP_NUM_THREADS`.

OpenMP permite configurar la forma en que las iteraciones del for se asignan a los distintos hilos mediante la directiva `schedule`. Si se omite o si se selecciona `schedule(auto)`, el *runtime* de OpenMP asigna los hilos de la forma que encuentre más conveniente para cada for. Con `schedule(dynamic,n)` los hilos se asignan en *chunks* de tamaño $n$ (o si se omite el $n$ los chunks son de tamaño igual a la cantidad de iteraciones dividido por la cantidad de hilos), y cada chunk se asigna dinámicamente al primer hilo disponible. El comportamiento de `schedule(static,n)` es similar, excepto que los *chunks* se pre-asignan a hilos de forma estática.

Si escribimos el ejemplo anterior de la siguiente forma

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    x = a[i] + b[i];
    c[i] = x;
}
```

nos encontramos con el problema de que distintos hilos podrían modificar la variable `x` y no es posible saber qué valor terminará almacenado en `c[i]` en la mayoría de los casos. Par evitar este problema existe la clausula `private(a,b,...)`, que define las variables listadas como privadas de cada hilo. El valor de estas variables se copia al lanzar los hilos en una variable privada. El ejemplo anterior quedaría como sigue:

```
#pragma omp parallel for private(x)
for (int i=0; i<n; i++) {
    x = a[i] + b[i];
    c[i] = x;
}
```

Otra cláusula muy útil es `reduce`, que permite unificar el valor una variable local de cada hilo, mediante alguna operación, una vez que finaliza el for paralelo. Un ejemplo puede ser realizar una suma paralela de los valores en un vector `a`:

```
int suma=0;
#pragma omp parallel for reduce(+:suma)
for (int i=0; i<n; i++) {
    suma += a[i];
}
```

Además de permitir paralelizar bucles for de forma sencilla, la directiva `#pragma omp parallel` permite paralelizar cualquier sección del código lanzando múltiples hilos. También es posible establecer zonas críticas (donde los hilos deben sincronizarse y acceder de forma secuencial), barreras donde deben esperar los hilos, operaciones atómicas, y muchas cosas más, brindando un entorno completo y flexible para el manejo de hilos.

## 5.9. Message Passing Interface (MPI)

MPI, que significa “Message Passing Interface” (Interfaz de Paso de Mensajes), es una biblioteca estándar utilizada en programación paralela para facilitar la comunicación entre procesos mediante el intercambio de mensajes. Fue desarrollada por la industria, desarrolladores de software y científicos, incluyendo a empresas como IBM, Intel y nCUBE, con el objetivo principal de proporcionar un estándar portátil y eficiente para la programación paralela, especialmente en sistemas con memoria distribuida.

Las características clave de MPI incluyen su enfoque en plataformas de memoria distribuida, la necesidad de que los programadores definan y controlen explícitamente el paralelismo en sus aplicaciones, y su modelo de programas SPMD (Single Program, Multiple Data). MPI no permite la creación de nuevos procesos durante la ejecución y se compone de alrededor de 125 funciones que facilitan la comunicación y sincronización entre procesos en aplicaciones paralelas.

MPI (Message Passing Interface)

MPI, que significa "Message Passing Interface"(Interfaz de Paso de Mensajes), es una biblioteca estándar utilizada en programación paralela para facilitar la comunicación entre procesos mediante el intercambio de mensajes. A continuación, se presentan los aspectos clave de MPI:

Origen y Objetivo de MPI:

MPI es una biblioteca estándar desarrollada por la industria, desarrolladores de software, aplicaciones y científicos, con contribuciones de empresas como IBM, Intel y nCUBE. Su objetivo principal es proporcionar un estándar portable y eficiente para la programación paralela, especialmente en sistemas con memoria distribuida. Características Principales de MPI:

Plataforma Objetivo: MPI está diseñado principalmente para entornos de memoria distribuida, donde múltiples procesadores se ejecutan en máquinas independientes que se comunican a través de la red. Paralelismo Explícito: MPI requiere que el programador defina y controle explícitamente el paralelismo en la aplicación. Modelo de Programas: MPI sigue el modelo SPMD (Single Program, Multiple Data), donde cada proceso ejecuta el mismo programa, pero con diferentes datos. Número de Tareas: El número de tareas (procesos) se fija antes de la ejecución, y MPI no incluye primitivas para crear nuevos procesos dinámicamente. Funciones MPI: MPI proporciona alrededor de 125 funciones que permiten a los procesos comunicarse y sincronizarse en aplicaciones paralelas.

**Objetivos Específicos de MPI:**

- **Portabilidad:** MPI busca definir un entorno de programación único que sea independiente de la plataforma y que pueda ejecutarse en diferentes sistemas de cómputo.
- **Eficiencia:** Busca aprovechar al máximo las capacidades de hardware especializado para lograr un alto rendimiento.
- **Funcionalidad:** Proporciona herramientas avanzadas para estructurar aplicaciones, manejar grupos de procesos y optimizar la comunicación entre procesos.

El estándar MPI incluye una variedad de características y funciones que permiten a los programadores desarrollar aplicaciones paralelas eficientes:

- Comunicaciones punto a punto.
- Operaciones colectivas para comunicación entre grupos de procesos.
- Agrupación de procesos para una gestión eficiente.
- Contextos de comunicación para controlar los flujos de mensajes.
- Definición de topologías de procesos.
- Soporte tanto para lenguaje Fortran como para C.

- Manejo del entorno de programación para controlar aspectos como la inicialización y finalización.
- Interfaz personalizada para adaptarse a las necesidades específicas de las aplicaciones.

El estándar MPI ofrece una amplia gama de capacidades y características que facilitan la programación paralela en sistemas con memoria distribuida. Entre las principales ventajas y posibilidades que brinda se encuentran la capacidad de utilizar un conjunto completo de rutinas de comunicación tanto punto a punto como entre grupos de procesos. Además, MPI permite definir contextos de comunicación separados entre grupos de procesos y especificar diferentes topologías de comunicación, lo que brinda flexibilidad y control en la organización de la comunicación en una aplicación.

Una característica importante de MPI es su capacidad para crear tipos de datos derivados, lo que facilita el envío de mensajes que contienen datos no contiguos en memoria, lo que es especialmente útil en aplicaciones con estructuras de datos complejas. MPI también admite la comunicación asincrónica, lo que significa que los procesos pueden continuar ejecutándose sin tener que esperar la finalización de una operación de comunicación, lo que mejora la eficiencia de la aplicación.

Otro punto fuerte de MPI es su capacidad para administrar eficientemente el pasaje de mensajes, minimizando el overhead y garantizando un rendimiento óptimo. Esto permite a los desarrolladores crear aplicaciones paralelas altamente eficientes que se ejecutan de manera eficiente en sistemas de memoria compartida (MPP) y en clusters.

Además de su capacidad técnica, MPI se destaca por ser altamente portátil, lo que significa que las aplicaciones desarrolladas en MPI pueden ejecutarse en una variedad de plataformas sin modificaciones significativas. MPI también cuenta con una especificación formal que proporciona una base sólida para su implementación y uso.

En cuanto a la implementación, existen varias opciones de alta calidad de MPI disponibles, como LAM, MPICH y CHIMP.

Aunque MPI es una biblioteca poderosa, no aborda todos los aspectos relacionados con la programación paralela. Algunas de las cosas que no incluye son:

- Comunicaciones de memoria compartida.
- Soporte para recepciones por interrupción del sistema operativo.
- Ejecución remota de procesos.
- Facilidades de depuración.
- Soporte para threads (hilos).
- Soporte para control de tareas.
- Manejo limitado de E/S (entrada/salida).

**Rutinas básicas de MPI**

- `ierr = MPI_Iinit(&argc,&argv)`

  Inicializa el entorno MPI

- `ierr = MPI_Comm_size(MPI_COMM_WORLD,&npes);`

  Obtiene el número total de procesos en el comunicador.

- `ierr = MPI_Comm_rank(MPI_COMM_WORLD,&iam);`

  Obtiene el identificador (rango) del proceso actual en el comunicador.

- `ierr = MPI_Send(buffer,count,datatype,destino,tag,comm)`

  Envía un mensaje desde un proceso a otro.

  - `buffer:` Dirección inicial del buffer de envío.
  - `count:` Número de elementos en el buffer de envío (entero no negativo).
  - `datatype`: Tipo de dato de cada elemento del buffer de envío (identificador).
  - `destino:` Rango de destino (entero).
  - `tag:` Etiqueta del mensaje (entero).
  - `comm:` Comunicador (identificador).

- `ierr = MPI_Recv(buffer,count,datatype,source,tag,comm,estado)`

  Recibe un mensaje en un proceso desde otro.

  - `buffer:` Dirección de inicio del buffer de envío o de recepción.
  - `count:` Número de elementos a enviar o a recibir.
  - `datatype:` Tipo de dato de cada elemento. `source:` Identificador del remitente.
  - `tag:` Etiqueta del mensaje.
  - `comm:` Representa el dominio de comunicación.
  - `status:` Permite obtener fuente, tag y contador del mensaje recibido

- `ierr = MPI_Get_Processor_name(char *name, int *resultlen)`

  Retorna el nombre del procesador ( o equipo ) donde ejecuta el proceso.

- `ierr = MPI_Finalize()`

  Finaliza el entorno MPI.

### Comunicaciones colectivas

Las comunicaciones colectivas en MPI permiten la transferencia de datos entre todos los procesos que forman parte de un grupo específico. En lugar de utilizar etiquetas para los mensajes, se emplean identificadores de grupos, conocidos como comunicadores. Las operaciones colectivas abarcan a todos los procesos dentro del ámbito del comunicador al que pertenecen.

Por defecto, todos los procesos se encuentran incluidos en el comunicador genérico `MPI_COMM_WORLD`.

Las operaciones colectivas en MPI se dividen en tres categorías principales:

1. **Sincronización (Operaciones de barrera):** Estas operaciones implican que los procesos esperan a que otros miembros del grupo alcancen un punto de sincronización antes de continuar.

2. **Movimiento (transferencia) de datos:** Estas operaciones permiten la difusión, recolección y dispersión de datos entre los procesos, facilitando el intercambio de información entre ellos.

3. **Cálculos colectivos:** Estas operaciones se utilizan para realizar reducciones globales, como sumas, máximos, mínimos u otras funciones definidas por el usuario, en los datos distribuidos entre los procesos.

Es importante destacar que las operaciones colectivas son bloqueantes, lo que significa que pueden bloquear el progreso de un proceso hasta que se complete la operación. Cuando se realizan operaciones colectivas que involucran a un subconjunto específico de procesos, es necesario definir previamente un particionamiento de esos subconjuntos y relacionarlos con nuevos grupos a través de la creación de nuevos comunicadores.

Operaciones colectivas de MPI:

- **Sincronización:**

  `MPI_Barrier(comm)`

  Esta función crea una barrera de sincronización en un grupo de procesos. Cuando se alcanza una invocación a `MPI_Barrier`, cada tarea se bloquea hasta que todas las tareas en el grupo lleguen a la misma invocación de `MPI_Barrier`. Esto asegura que todas las tareas en el grupo estén sincronizadas antes de continuar.

- **Difusión:**

  `MPI_Bcast(buffer, count, datatype, root, comm)`

  Se utiliza para enviar un mensaje desde el proceso con el rango especificado como `root` a todos los demás procesos en el grupo. El mensaje se transmite desde el proceso `root` a todos los demás procesos en el comunicador `comm`. La ejecución se bloquea hasta que todos los procesos ejecuten la rutina. Automáticamente actúa como punto de sincronización.

- `MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`: Esta función se emplea para distribuir diferentes mensajes desde una tarea a todas las tareas en el grupo. El proceso raíz envía porciones de datos desde `sendbuf` a cada uno de los procesos en el grupo y los recibe en `recvbuf`.

- **Recolección:**

  `MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`

  Se utiliza para recopilar diferentes mensajes desde todas las tareas del grupo y reunirlos en una única tarea, que se especifica como `root`. Cada tarea envía su propio mensaje desde `sendbuf` al proceso raíz, donde se reúnen en `recvbuf`.

- **Esparcir:**

  `MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`

  Esta operación concatena los datos de cada tarea en el grupo y los envía a todas las demás tareas. Cada tarea realiza un broadcast uno a uno con todas las demás tareas en el grupo, lo que permite que todos tengan acceso a los datos de los demás.

- **Cálculos colectivos**

  `MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)`

  Se utiliza para aplicar una operación de reducción a todos los datos de las tareas en el grupo y almacenar el resultado en la tarea especificada como `root`. La operación de reducción puede ser una suma, un máximo, un mínimo u otra función definida por el usuario. El usuario puede combinar cálculos parciales de todos los procesos. Los resultados están disponibles en un proceso particular o en todos los procesos.

  Todos almacenan el resultado con `MPI_Allreduce`.

# Capítulo 6

# Sistemas Lineales

Se estudiará el problema de la forma:

$$Ax = b \tag{6.1}$$

donde $A$ es una matriz invertible de tamaño $n \times n$, $b$ es un vector de tamaño $n \times 1$ y $x$ el vector incógnita de tamaño $n \times 1$.

Este tipo de problemas tiene una amplia gama de aplicaciones, que incluyen la resolución de ecuaciones diferenciales, problemas de optimización y la solución de problemas no lineales, como el método de Newton-Raphson.

Existen varias estrategias para abordar la resolución de estos problemas. Por un lado, están los métodos directos, que llegan a la solución en una cantidad finita de pasos, y por otro, los métodos iterativos, que intentan aproximar la solución mediante una sucesión generada iterativamente.

## 6.1. Métodos directos

Los métodos directos son capaces de llegar a una solución en un número predefinido de pasos, dependiendo del tamaño del sistema a resolver. Además, estos métodos garantizan, excepto por errores numéricos derivados del uso de precisión finita para representar los números, la obtención de la solución exacta del sistema lineal. También incorporan estrategias diseñadas para mitigar los problemas derivados del mal condicionamiento de las matrices, como el uso de técnicas de pivoteo.

En el caso de la resolución de matrices dispersas mediante métodos directos, es crucial controlar la aparición de nuevos coeficientes no nulos, lo que se conoce como "fill in".

### 6.1.1. Resolución de sistemas particulares

#### Sistemas de una diagonal

El sistema de la ecuación $Ax = b$ cumple que la matriz $A$ es diagonal:

$$\begin{pmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \implies x_i = \frac{b_i}{a_i} \tag{6.2}$$

**Sistemas triangulares**

El sistema de la ecuación $Ax = b$ cumple que la matriz $A$ es triangular (superior o inferior).

Cuando el sistema es triangular superior se aplica lo que se conoce como sustitución hacia atrás:

$$\begin{pmatrix} u_{11} & \cdot & \cdot & \cdot & u_{1n} \\ 0 & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & 0 & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{pmatrix} \implies \begin{cases} x_n = \frac{b_n}{u_{nn}} \\ x_i = \frac{b_i - \sum_{k=i+1}^{n} x_k u_{ik}}{u_{ii}} \end{cases} \tag{6.3}$$

En este caso la cantidad de operaciones que se deben realizar para resolver el sistemas es:

$$op \approx \sum_{k=1}^{n} (k-1) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

Lo que implica que este algoritmo se de orden $\mathcal{O}(n^2)$.

Si el sistema es triangular inferior se realiza lo que se conoce como sustitución hacia adelante, la cual es análoga a la sustitución hacia atrás ya vista.

### 6.1.2. Eliminación Gaussiana

La Eliminación Gaussiana, también conocida como escalonamiento y eliminación de Gauss en honor a Karl Gauss (1777-1855), es uno de los métodos más ampliamente utilizados para resolver sistemas de ecuaciones lineales. Este método asume que la matriz $A$ es invertible, lo que garantiza que el sistema tiene una única solución.

La Eliminación Gaussiana opera transformando la matriz $A$ y el vector $b$ mediante una serie de operaciones elementales, como intercambiar dos filas, multiplicar una fila por un escalar distinto de cero o sumar dos filas, con el objetivo de convertir el sistema en uno triangular.

El proceso se lleva a cabo de la siguiente manera: para cada columna $j$, desde la primera hasta la $n$-ésima (en orden), se buscan y eliminan los coeficientes distintos de cero a partir de la fila $j$. Para eliminar el coeficiente $a_{ij}$ de la columna $j$ se resta la fila $i$ menos la fila $j$ multiplicada por $m_{ij} = \frac{a_{ij}}{a_{jj}}$. Paralelamente, se aplican los mismos cambios al vector $b$ para mantener la equivalencia del sistema.

La segunda etapa del método implica resolver el sistema lineal triangular resultante. Esta resolución se realiza utilizando estrategias previamente establecidas.

**Ejemplo:**

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 5 & 4 \\ 1 & 3 & 9 & 6 \\ 2 & 3 & 7 & 9 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 2 \\ 0 & 2 & 8 & 5 \\ 0 & 1 & 5 & 7 \end{pmatrix} x = \begin{pmatrix} 1 \\ -1 \\ 0 \\ -1 \end{pmatrix} \rightarrow \cdots \rightarrow \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 2 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ -1 \\ 2 \\ -2 \end{pmatrix} \tag{6.4}$$

En el caso de la eliminación Gaussiana para cada columna $k$, donde $k \in \{1, \cdots, n-1\}$ (todas las columnas excepto la última), se realizan los siguientes pasos:

1. Se calculan los $n-k$ multiplicadores, lo que implica realizar $n-k$ divisiones.

2. Se actualizan los $(n-k)^2$ elementos, lo que involucra una multiplicación y una suma por cada uno de estos elementos.

Por lo tanto, la cantidad de operaciones realizadas será:

$$\sum_{k=1}^{n-1}(n-k) + 2\sum_{k=1}^{n-1}(n-k)^2 = \frac{2n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2} = \frac{4n^3 - 3n^2 - n}{6} \approx \frac{2}{3}n^3 \tag{6.5}$$

Finalmente, se puede concluir que el orden del algoritmo de Eliminación Gaussiana es $\mathcal{O}(n^3)$, lo que significa que el número de operaciones crece cúbicamente en función del tamaño del sistema, $n$.

### 6.1.3. Factorización LU

En el primer paso de la escalerización Gaussiana, la eliminación del coeficiente $i$ de la primera columna de $A$ se logra restando a la fila $i$ el resultado de multiplicar la primera por $m_{i1} = \frac{a_{i1}}{a_{11}}$. Para una matriz de dimensión 4, esto equivale al siguiente producto matricial:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -m_{21} & 1 & 0 & 0 \\ -m_{31} & 0 & 1 & 0 \\ -m_{41} & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \tilde{a}_{34} \\ 0 & \tilde{a}_{42} & \tilde{a}_{43} & \tilde{a}_{44} \end{pmatrix} \tag{6.6}$$

De la misma forma, los coeficientes de la segunda columna se volverán cero al multiplicar la matriz resultante del paso anterior por

$$M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -m_{32} & 1 & 0 \\ 0 & -m_{42} & 0 & 1 \end{pmatrix}$$

Podemos, entonces, reescribir el proceso de escalerización Gaussiana como el producto de las matrices $M_n \dots M_1 A = U$, donde la matriz $U$ será triangular superior. En particular, $A = M_1^{-1} \dots M_n^{-1} U$. Es fácil ver que

$$M_i^{-1} = \begin{pmatrix} I & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & m_{i+1,i} & 0 & 0 \\ \vdots & \vdots & I & 0 \\ 0 & m_{n,i} & 0 & I \end{pmatrix} \tag{6.7}$$

y que el producto $L = M_1^{-1} \dots M_n^{-1}$ será una matriz triangular inferior donde la columna $i$ es igual a la columna $i$ de $M_i^{-1}$. De esta forma se obtiene la *factorización LU* de la matriz $A$.

Esto permite resolver el sistema de ecuaciones en dos etapas: primero, resolviendo $Ly = b$, y luego, $Ux = y$.

Una característica importante de la factorización LU es que, si la matriz $L$ tiene unos en su diagonal principal, la descomposición es única. Esto puede ser útil cuando se necesita resolver varios sistemas de ecuaciones con la misma matriz $A$ pero con diferentes vectores de términos independientes ($b$).

Ejemplos de algoritmos:

**Algorithm 1** Algoritmo

```
U = A;
for k = 1 : n − 1 do
    for j = k + 1 : n do
        L(j, k) = U(j, k)/U(k, k);
        U(j, k : n) = U(j, k : n) − L(j, k)U(k, k : n);
    end for
end for
```

**Algorithm 2** Algoritmo “in place”

```
for i = 1 : n − 1 do
    for j = i + 1 : n do
        A(j, i) = A(j, i)/A(i, i);
    end for
    for k = i + 1 : n do
        U(j, k : n) = U(j, k : n) − L(j, k)U(k, k : n);
    end for
end for
```

En la implementación de estos algoritmos, el acceso a la matriz $A$ puede realizarse por bloques, lo que mejora la localidad de datos en el acceso, especialmente en lectura y escritura. Además, los patrones de acceso por acceso a la matriz por bloques pueden adaptarse según qué datos calcula. Esto es particularmente beneficioso porque realiza operaciones matriz-matriz que son más eficientes que las operaciones vector-vector.

Como se dijo, en la factorización LU por bloques sedescompone a la matriz A en sub-bloques:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \tag{6.8}$$

**Algorithm 3** Algoritmo por bloques

Se resuelve $L_{11}$ y $U_{11}$
Se resuelve $L_{21}$
Se resuelve $U_{12}$
Luego se actualiza $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$
Por último se factoriza $\tilde{A}_{22}$

Existen diferentes estrategias de acceso:

- Left-looking
- Right-looking
- Crout-LU

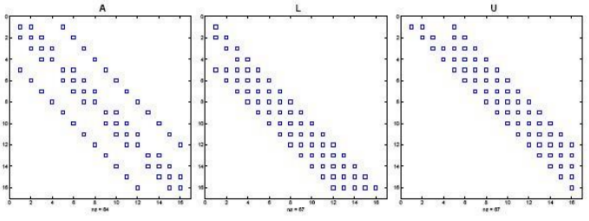



**Figura 6.1**

Como conclusión las estrategias de acceso a los datos presentan varias ventajas. En primer lugar, mejoran la localidad de datos, lo que significa que los datos necesarios para una operación están más cerca entre sí en la memoria, reduciendo así el tiempo de acceso y mejorando el rendimiento general. Además, estas estrategias son flexibles y permiten adaptarse según las necesidades del algoritmo, lo que facilita el acceso tanto por columna como por fila, dependiendo de lo que sea más eficiente.

Otro punto importante es que estas estrategias mejoran el trabajo en computadoras vectoriales, ya que estas máquinas se basan en la ejecución eficiente de operaciones en paralelo, y tener datos

bien organizados facilita esta tarea. Además, al utilizar estrategias de acceso optimizadas, se puede aprovechar al máximo el rendimiento de bibliotecas de alto rendimiento como las bibliotecas BLAS (Basic Linear Algebra Subprograms). Estas bibliotecas son fundamentales para operaciones matriciales y vectoriales de alto rendimiento, y el acceso eficiente a los datos es esencial para aprovechar al máximo su potencial.

**Estrategias de pivoteo**

En el proceso de escalerización durante la Eliminación Gaussiana, los elementos en las filas y columnas se denominan pivotes. Sin embargo, puede ocurrir que uno de estos pivotes sea (o esté cerca de ser) igual a cero, lo que puede causar problemas en el proceso de resolución. Para abordar esta situación, se utiliza una técnica llamada pivoteo, que implica intercambiar filas o columnas para asegurarse de que los pivotes sean distintos de cero.

Existen dos estrategias principales de pivoteo:

1. **Pivoteo parcial:** En esta estrategia, se intercambian las filas de manera que el pivote en la columna actual sea distinto de cero. Se elige la fila con el pivote más grande en valor absoluto entre las filas disponibles.

2. **Pivoteo completo:** En esta estrategia, además de intercambiar filas, también se intercambian columnas. Se buscan tanto filas como columnas de manera que el pivote sea distinto de cero y que las filas y columnas intercambiadas maximicen el valor absoluto del pivote.

Las estrategias de pivoteo son esenciales para garantizar la estabilidad y la convergencia del proceso de eliminación Gaussiana, especialmente cuando se trabaja con matrices que pueden tener elementos cercanos a cero en sus pivotes.

Además, es importante destacar que con este método se pueden resolver varios sistemas de ecuaciones lineales independientes simultáneamente. Primero, se escalerizan todas las matrices juntas y luego se resuelve cada uno de los sistemas triangulares resultantes de manera individual.

El pivoteo, que implica intercambiar filas o columnas para evitar divisiones por cero o problemas de precisión numérica, también se aplica en la factorización LU. En particular, el pivoteo parcial se utiliza de la siguiente forma:

$$Ly = Pb \quad Ux = y \tag{6.9}$$

Sin embargo, el pivoteo en bloques completo puede ser más complejo, por lo que solo se debe realizar dentro del bloque . Otras soluciones pueden ser el pivoteo $2 \times 2$ y el enfoque de matrices aleatorias.

Finalmente, se menciona la descomposición Cholesky para matrices simétricas y definidas positivas, que es un caso especial de la factorización LU. Esta descomposición es útil en situaciones donde se trabaja con matrices simétricas y se busca una factorización eficiente y única, la misma

es:

$$A = LL^T \tag{6.10}$$

## 6.2. Errores

Cuando se perturba la matriz $A$ esto produce cambios en la solución, se esperaría que si se modifica "poco" la variación en la solución también sea pequeña, sin embargo esto no siempre sucede como muestra el siguiente ejemplo:

$$A = \begin{pmatrix} 1 & 1 \\ 10.05 & 10 \end{pmatrix} \quad b = \begin{pmatrix} 2 \\ 12 \end{pmatrix} \implies x = \begin{pmatrix} 20 \\ -18 \end{pmatrix}$$
$$A = \begin{pmatrix} 1 & 1 \\ 10.1 & 10 \end{pmatrix} \quad b = \begin{pmatrix} 2 \\ 12 \end{pmatrix} \implies x = \begin{pmatrix} 10 \\ -8 \end{pmatrix}$$

Ahora se quiere analizar cómo se pueden acotar las variaciones en la solución $x$ frente a las perturbaciones en $b$ y $A$:

- Perturbaciones en $b$:

$$A(x+\delta x) = b + \delta b \implies A\delta x = \delta b \implies \delta x = A^{-1}\delta b \implies \|\delta x\| \leq \|A^{-1}\|\|\delta b\|$$
$$\frac{\|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|}$$

- Perturbaciones en $A$:

$$(A+\delta A)(x+\delta x) = b \implies A\delta x = -\delta A(\delta x + x) \implies \delta x = A^{-1}(-\delta A)(\delta x + x)$$
$$\frac{\|\delta x\|}{\|x+\delta x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta A\|}{\|A\|}$$

  Si además se cumple que $\|A^{-1}\|\|\delta A\| < 1$ entonces:

$$\|\delta x\| \leq \|A^{-1}\|\|\delta A\|\|x+\delta x\| \leq \|A^{-1}\|\|\delta A\|\|x\| + \|A^{-1}\|\|A\|\|\delta x\|$$
$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\frac{\|A\|\|A^{-1}\|\|\delta A\|}{\|A\|}}{1 - \frac{\|A\|\|A^{-1}\|\|\delta A\|}{\|A\|}}$$

**Definición 6.2.0.1** (Número de condición del sistema)**.** *El número de condición es el factor que controla la amplificación ded los errores relativos de los datos:*

$$cond(A) = \|A\|\|A^{-1}\| \tag{6.11}$$

Por lo tanto, las acotaciones anteriores quedan:

- Perturbaciones en $b$:

$$\frac{\|\delta x\|}{\|x\|} \leq cond(A)\frac{\|\delta b\|}{\|b\|} \tag{6.12}$$

- Perturbaciones en $A$:

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{cond(A)\frac{\|\delta A\|}{\|A\|}}{1 - cond(A)\frac{\|\delta A\|}{\|A\|}} \tag{6.13}$$

- Perturbaciones en $A$ y $b$:

$$(A + \delta A)(x + \delta x) = (b + \delta b) \implies \frac{\|\delta x\|}{\|x\|} \leq \frac{cond(A)\|A\|}{1 - cond(A)\frac{\|\delta A\|}{\|A\|}} \left( \frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right) \tag{6.14}$$

**Teorema 6.2.0.1.** *Cualquier norma matricial es una función continua de los elementos de la matriz.*

**Teorema 6.2.0.2.** *Para cada par de normas matriciales existen constantes positivas $m$ y $M$ tales que:*

$$m\|A\|_i \leq \|A\|_j \leq M\|A\|_i \tag{6.15}$$

**Teorema 6.2.0.3.** *Para cualquier norma matural y cualquier matriz cuadrada se verifica:*

$$\rho(A) \leq \|A\| \tag{6.16}$$

**Teorema 6.2.0.4.** *Para cualquier matriz cuadrada y cualquier valor existe alguna norma natural tal que:*

$$\rho(A) \leq \|A\| \leq \rho(A) + \epsilon \tag{6.17}$$

**Corolario 6.2.0.1.** *Para cualquier matriz cuadrada se cumple:*

$$\rho(A) \leq \text{ínf}\, \|A\| \tag{6.18}$$

Además se cumple que para la norma euclidiana y una amtriz normal (aquella que cumple $AA^* = A^*A$):

$$cond(A) = \frac{\lambda_1}{\lambda_n} \tag{6.19}$$

donde $\lambda_1$ es el mayor valor propio de $A$ y $\lambda_n$ el menor.

Otra forma de obtener en forma exacta el número de condición es mediante la fórmula:

$$cond(A) = \frac{\sigma_1}{\sigma_n} \tag{6.20}$$

donde $\sigma_1$ es el mayor valor singular de $A$ y $\sigma_n$ el menor.

El condicionamiento numérico indica cómo se amplifican los errores, entonces si se introducen errores de redondeo en las cuentas los resultados se verán afectados.

Se pueden realizar dos enfoques diferentes para evaluar y comprender la propagación de errores:

1. **Análisis Directo:** En este enfoque, se estudia cómo se propagan los errores a lo largo de un cálculo numérico. Este análisis tiende a ser conservador y, a menudo, proporciona estimaciones muy pesimistas de los errores.

2. **Análisis Inverso:** En contraste, el análisis inverso se enfoca en estimar los errores al considerar perturbaciones en los datos iniciales o de entrada. En lugar de estudiar cómo se propagan los errores a lo largo del cálculo, se parte de la premisa de que los datos de entrada pueden contener errores y se busca cuantificar cómo estos errores afectan los resultados. Este enfoque puede proporcionar estimaciones más realistas de los errores en situaciones prácticas.

Si se realiza una factorización (sin pivoteo) de una matriz $A$ no singular se obtiene $LU = A+\delta A$ por causa de los errores de redondeo. Se cumple el siguiente reultado:

$$\|\delta A\| \leq n\epsilon\|L\|\|U\| \tag{6.21}$$

**Definición 6.2.0.2** (Factor de crecimiento)**.**

$$\rho = \frac{\|L\|\|U\|}{\|A\|} \tag{6.22}$$

Este parámetro refleja el crecimiento en magnitud de las entradas LU respecto de las de A.

Con la definición anterior se tiene que la Ecuación 6.21 queda:

$$\frac{\|\delta A\|}{\|A\|} \leq n\epsilon\rho \tag{6.23}$$

Si se utiliza pivoteo los valores de la matriz $L$ se obtienen mediante divisiones en las cuales el divisor siempre es mayor al dividendo. Se puede asumir que la norma de la matriz $L$ no crecerá sustancialmente y que $\rho$ está determinado por el crecimiento de $U$.

El factor de crecimiento que utiliza normas matriciales se sustituye por:

$$\rho = \frac{\text{máx}_{i,j}\,|u_{ij}|}{\text{máx}_{i,j}\,|a_{ij}|} \implies \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\rho n\epsilon_{mach}) \tag{6.24}$$

# Capítulo 7

# Bibliotecas

## 7.1. Historia

La historia de las bibliotecas y estándares de álgebra lineal numérica está llena de importantes hitos y contribuciones. Algunos de los pioneros en este campo incluyen a James Hardy Wilkinson, un matemático británico que desempeñó un papel fundamental en la construcción de la primera computadora británica con programas almacenados en la década de 1950, conocida como la ACE. Wilkinson realizó cálculos con matrices en la ACE y se convirtió en el principal experto en álgebra lineal numérica de su época.

La NAG (Numerical Algorithms Group) es otra figura destacada en esta historia, ya que desarrolló una biblioteca de 98 rutinas numéricas escritas en Algol 60 y Fortran. Esta biblioteca sentó las bases para el desarrollo posterior de estándares en álgebra lineal numérica.

El trabajo de investigadores, como Jack Dongarra y Cleve Moler del Argonne National Laboratory, condujo a la traducción de las rutinas numéricas del libro de Wilkinson a Fortran, lo que resultó en la creación de la biblioteca EISPACK. Esta biblioteca fue precursora de estándares más amplios, como LINPACK y LAPACK.

La revista TOMS (Transactions on Mathematical Software) de la ACM desempeñó un papel fundamental en la publicación de avances en software numérico y ayudó a difundir nuevas técnicas y herramientas en este campo.

La estandarización de las operaciones de álgebra lineal numérica comenzó con las BLAS (Basic Linear Algebra Subprograms), que se dividieron en BLAS-1, BLAS-2 y BLAS-3, según las operaciones y complejidad computacional. NETLIB surgió como una colección de software, artículos y bases de datos matemáticos que promovió la disponibilidad de herramientas numéricas.

Otras bibliotecas y herramientas, como ARPACK, UMFPACK, SCALAPACK, PETSc, SuperLU y GotoBLAS, siguieron contribuyendo al campo de álgebra lineal numérica, brindando soluciones para sistemas lineales, ecuaciones en derivadas parciales y cálculos numéricos en una variedad de plataformas y dispositivos de cómputo.

Hoy en día, se siguen utilizando implementaciones de BLAS y LAPACK optimizadas para diversos dispositivos de cómputo, como procesadores multicore, plataformas distribuidas, GPU, FPGAs, entre otros. Estas implementaciones a menudo son desarrolladas por los propios fabricantes de hardware y son esenciales para una amplia gama de aplicaciones numéricas en la computación actual.

## 7.2. Basic Linear Algebra Subprograms (BLAS)

BLAS, o Basic Linear Algebra Subprograms, es una especificación lo que significa que es un estándar que define una serie de operaciones matemáticas y una interfaz de programación para realizar operaciones de álgebra lineal numérica. Especifica cómo deben funcionar estas operaciones y cómo se comunican con otros componentes de software. Se divide en tres niveles, y se ha observado que las operaciones de niveles más altos tienden a obtener mejores resultados de optimización. Los tres niveles de BLAS son:

- BLAS-1
- BLAS-2
- BLAS-3

Se aplica a matrices generales y estructuradas, lo que incluye matrices triangulares y matrices en formato banda. Lo que lo distingue es su nomenclatura estandarizada, que consta de 5 letras agrupadas en 3 bloques: TMMOO, que representan el tipo de datos, el tipo de matriz y el código de operación.

En el nivel BLAS-1 se implementan operaciones tipo vector-vector y escalares. En este caso se traba con datos de orden $n$ teniendo también un orden de operaciones del tipo $\mathcal{O}(n)$. Las operaciones de movimiento de datos incluyen copia e intercambio, y se realizan operaciones vectoriales como escalado vectorial. Además, se llevan a cabo operaciones de reducción, como el producto escalar, la norma vectorial, la sumatoria y la búsqueda de máximos. Las operaciones BLAS-1 siguen una nomenclatura específica, donde la letra $x$ representa el tipo de datos ($S$ para un real de simple precisión, $C$ para un complejo de simple precisión, $D$ para relaes de doble precisión y $Z$ para complejos de soble precisión), seguido de una letra que describe la operación (por ejemplo, $COPY$, es decir, sería $xCOPY$).

BLAS-2, el siguiente nivel, se centra en implementar operaciones básicas tipo matriz-vector. Estas operaciones involucran una cantidad de datos proporcional a $n^2$ y un número equivalente de cálculos, es decir, una cantidad de operacones de orden $\mathcal{O}(n^2)$. Esto incluye el producto matriz-vector y actualizaciones de rango 1 y 2.

En el nivel más alto, BLAS-3, se implementan operaciones tipo matriz-matriz. Aquí, la cantidad de datos se incrementa a $n^2$ y los cálculos se escalan a un orden $\mathcal{O}(n^3)$. Esto incluye el producto matriz-matriz y actualizaciones de rango $k$ y $2k$.

BLAS ha sido implementado en diversas plataformas, y cada implementación puede adaptarse a las características del hardware. Algunas de las implementaciones más notables incluyen:

- ACML de AMD
- ESSL de IBM
- MKL de Intel, utilizada por MATLAB
- Sun Performance Library de SUN
- GotoBLAS (anteriormente)
- OpenBLAS (basada en GotoBLAS2)
- ATLAS (una implementación portable)
- cuBLAS para GPUs NVIDIA
- RocBLAS para GPUs AMD.

Algunas de estas implementaciones proporcionan paralelismo a nivel de hilos en arquitecturas con memoria compartida, como Intel MKL, OpenBLAS y Atlas.

Por último, vale la pena mencionar que existen variantes de BLAS para trabajar con formatos de matrices dispersas, como NIST S-BLAS, PSBLAS y SparseLib++. Además, para GPUs NVIDIA, cuSparse ofrece operaciones compatibles con BLAS en formatos dispersos. Estas implementaciones no siguen exactamente el estándar BLAS, pero proporcionan operaciones eficientes en matrices dispersas.

## 7.3. Linear Algebra PACKage (LAPACK)

LAPACK (Linear Algebra PACKage) es una especificación que abarca una amplia variedad de rutinas diseñadas para resolver problemas de álgebra lineal estándar. Estas rutinas incluyen la resolución de sistemas de ecuaciones lineales, problemas de mínimos cuadrados, así como la determinación de valores propios y singulares. Además, LAPACK proporciona algoritmos para descomposiciones matriciales, como LU, QR, SVD, Cholesky, entre otros. Está diseñado para trabajar con matrices densas y de banda.

En su implementación de referencia, LAPACK utiliza estrategias de bloques y se basa en llamadas a rutinas de BLAS (Basic Linear Algebra Subprograms). El código fuente de LAPACK está disponible en FORTRAN y C, lo que permite su uso en una variedad de entornos y plataformas. Puedes encontrar más información y acceder al código fuente en el Proyecto CLapack en el sitio web de Netlib.

LAPACK se compone de diferentes tipos de rutinas, como drivers que resuelven problemas completos, rutinas computacionales que realizan tareas específicas, y rutinas auxiliares que realizan subprocesos o operaciones de bajo nivel. Cada rutina driver utiliza un conjunto de rutinas computacionales para lograr su objetivo.

Para abordar implementaciones distribuidas de alto rendimiento en arquitecturas HPC, se encuentra ScaLAPACK, que extiende las capacidades de LAPACK. Este módulo utiliza componentes como PBLAS (Parallel BLAS) y BLACS (Basic Linear Algebra Communication Subroutines) para mejorar el rendimiento de las operaciones lineales distribuidas.

En el entorno de las unidades de procesamiento gráfico (GPU), existen variantes de LAPACK adaptadas a estas arquitecturas. Algunas de estas implementaciones incluyen cuSOLVER y MAGMA, que ofrecen soluciones eficientes en GPU para problemas de álgebra lineal. Además, GINKGO es un proyecto en desarrollo que busca proporcionar soporte multiplataforma para matrices densas y dispersas en entornos multicore y GPU. Estas implementaciones son esenciales para aprovechar al máximo el rendimiento de las GPU en cálculos de álgebra lineal.

# Capítulo 8

# Sistemas Lineales Dispersos

## 8.1. Factorización de matrices dispersas

Como se vio en el capítulo anterior la factorización de matrices dispersas es una técnica fundamental en el ámbito de la computación numérica y la resolución de sistemas de ecuaciones lineales.

Sin embargo, es importante tener en cuenta que al factorizar una matriz dispersa $A$ en sus componentes $L$ y $U$ mediante una descomposición $LU$, no se puede garantizar que las matrices $L$ y $U$ resultantes también sean dispersas. Este escenario presenta dos problemas:

1. Utilizar estructuras dispersas para matrices que son completas, lo que resulta en ineficiencia en el uso de la memoria.

2. Engañarnos en cuanto al número de operaciones realizadas, ya que la factorización $LU$ convencional requiere más cálculos que una factorización optimizada para matrices dispersas.

Este último problema se conoce como el problema de *"llenado"* o *"fill-in"*, donde el llenado se refiere a la creación de elementos no nulos en las matrices $L$ y $U$.

La clave al trabajar con matrices dispersas radica en el ahorro de dos aspectos críticos: espacio de memoria y cantidad de cálculos. En el caso de matrices dispersas, almacenar y operar con todos los ceros es ineficiente y consume recursos innecesarios.

La resolución de sistemas de ecuaciones completos (es decir, de matrices densas) consta de dos etapas: la factorización (la escalerizazción) y las sustituciones. Sin embargo, en el caso de matrices dispersas, estas etapas no son tan sencillas como en las matrices densas. Aquí, además de resolver los sistemas, se deben abordar dos desafíos adicionales:

- Minimizar o, al menos, controlar el llenado en las matrices $L$ y $U$, para evitar el consumo innecesario de memoria y recursos de cálculo.

- Prever la estructura de las matrices $L$ y $U$ de antemano, ya que esta estructura puede variar significativamente según la técnica de factorización utilizada.

La mayoría de los trabajos de investigación y aplicaciones prácticas inicialmente se centraron en la resolución de sistemas de ecuaciones con matrices simétricas y definidas positivas, ya que estas matrices tienen propiedades que permiten un tratamiento más eficiente. Por ejemplo, se utilizó con frecuencia la factorización de Cholesky para descomponer matrices simétricas en una matriz triangular inferior L y su traspuesta.

## 8.2. Sistemas dispersos y grafos

**Definición 8.2.0.1** (Envolvente de una matriz)**.** *Se define la envolvente de una matriz A como:*

$$Env(A) = \{(i,j) : \quad f_i \leq j \leq l_i, \quad 1 \leq i \leq n\} \tag{8.1}$$

*donde:*

$$f_i = \text{mín}\{j : \quad a_{ij} \neq 0\}$$
$$l_i = \text{máx}\{j : \quad a_{ij} \neq 0\}$$

$$\boldsymbol{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} \times & \times & \times & \times & & & \\ \times & \times & 0 & \times & & & \\ & & \times & 0 & \times & \times & \\ & \times & 0 & \times & & & \\ & & & \times & \times & \times & \times \\ & \times & \times & 0 & 0 & \times & \times \\ & & & & & & \times \end{bmatrix} \end{array}$$

**Figura 8.1**: Ilustración gráfica de $Env(A)$

**Definición 8.2.0.2** (Grafo)**.** *Un **grafo** es una dupla $G = (V, E)$ donde $V = \{v_1, v_2, \cdots, v_n\}$ es un conjunto finito y no vacío de elementos llamados **vértices** o **nodos** y $E \subset v \times V$ es el conjunto cuyos elementos se componen de subconjuntos de $V$ de cardinalidad dos denominados **aristas**.*

**Definición 8.2.0.3** (Grafo dirigido)**.** *Se dice que $G = (V, E)$ es un **grafo dirigido** si las aristas son pares ordenados.*

**Definición 8.2.0.4** (Grafo completo)**.** *Un grafo se dice **completo** si el conjunto de aristas cumple que existe una arista $(u, v) \in E \forall u, v \in V$. Es decir, para todo par de vértices existe al menos una arista que los une.*

**Definición 8.2.0.5** (Camino)**.** *Un **camino** (de longitud $r$) en un grafo $G = (V, E)$ del nodo $v_i$ al nodo $v_k$ es una secuencia de aristas $(v_i, v_j), (v_j, v_l), \cdots, (v_p, v_k)$ donde $r = k - l$.*

**Definición 8.2.0.6** (Ciclo)**.** *Un **ciclo** es un camino cerrado, $v_i = v_k$, que no repite vértices (salvo el primero y el último).*

**Definición 8.2.0.7** (Grafo conexo)**.** *Se dice que un grafo es* ***conexo*** *si existe por lo menos un camino entre todo par de nodos.*

**Definición 8.2.0.8** (Cuerda)**.** *Una* ***cuerda*** *en un camino es una arista que una dos vértices no consecutivos.*

**Definición 8.2.0.9** (Grafo cordal)**.** *Un* ***grafo cordal*** *es aquel que no posee ciclos ni cuerdas.*

**Definición 8.2.0.10** (Ciclo sin cuerdas)**.** *Un* ***ciclo sin cuerdas*** *es un ciclo de largo mayor o igual a 4 sin cuerdas.*

**Definición 8.2.0.11** (Árbol)**.** *Un* ***árbol*** *es un grafo conexo y acíclico.*

**Definición 8.2.0.12** (Subgrafo)**.** $H = (U, F)$ *es un* ***subgrafo*** *de* $G = (V, E)$ *si* $U \subset V$*,* $F \subset E$ *y* $F \subset U \times U$*.*

**Definición 8.2.0.13** (Clique)**.** *Se define un* ***clique*** *como un subgrafo completo, es decir, un subgrafo donde para todo par de vértices existe al menos una arista que los une.*

**Definición 8.2.0.14** (Conjunto de vértices adyacentes)**.** *Se define el* ***conjunto de vértices adyacentes*** *al vértice* $v$*,* $AdjG(v)$*, a aquellos vértices que poseen aristas incidentes desde el vértice* $v$*.*

**Definición 8.2.0.15** (Grado de un vértice)**.** *El* ***grado de un vértice*** $v$ *del grafo* $G$ *se define como la cardinalidad del conjunto de vértices adyacentes:* $grado_G(v) = \#Adj_G(v)$

Una matriz dispersa $A$ cuadrada de dimensión $n$ se puede representar mediante un grafo etiquetado dirigido $G = (V, E)$ donde el conjunto de vértices es $V = \{1, \cdots, n\}$ y el conjunto de aristas se compone de $(i, j)$ si $a_{ij} \neq 0$.

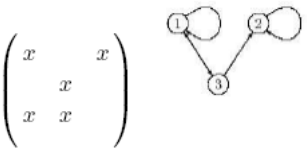



**Figura 8.2**

Para representar una matriz con estructura simétrica se puede utilizar un grafo etiquetado no dirigido $G = (V, E)$ donde $V = \{1, \cdots, n\}$ y $(i, j)$ pertenece a $E$ si $a_{ij} \neq 0 \quad (a_{ji} \neq 0)$.

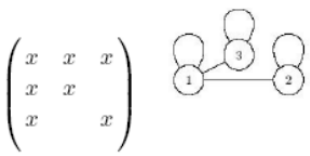



**Figura 8.3**

Los vértices adyacentes a cierto vértice $x$ son los índices de columna/fila de los elementos no nulos en la fila/columna $x$.

### 8.2.1. Árbol de eliminación

El concepto del *"Árbol de Eliminación"* es una herramienta valiosa en el contexto de la factorización de Cholesky y se utiliza para describir las dependencias entre las filas y columnas en este proceso. Los nodos en el árbol de eliminación se representan como enteros que van desde 1 hasta $n$, y estos números representan las columnas (o filas) de la matriz. La relación entre los nodos se basa en las propiedades de la matriz y las relaciones entre los elementos de la factorización de Cholesky.

En el árbol de eliminación, el padre de la fila $i$ es el valor más pequeño $j$ tal que $j > i$ y el elemento $L(i, j)$ es diferente de cero. Esto refleja cómo las filas y columnas de la matriz están relacionadas entre sí durante el proceso de factorización.

El patrón de $L$ a partir de $A$ se establece de la siguiente manera:

- Los elementos no nulos de $A$ se convierten en elementos no nulos de $L$.
- Si $i < j < k$, y $L(j, i)$ y $L(k, i)$ son diferentes de cero, entonces $L(k, j)$ también será diferente de cero. Esto muestra cómo las relaciones en el árbol de eliminación se traducen en la factorización de Cholesky.

El uso del árbol de eliminación en la factorización de matrices se atribuye a autores como Schreiber, Duff y Reid. Este concepto es especialmente útil para identificar dónde se generarán aristas de llenado (fill-in) en la matriz. Si $(i, j)$ es una arista en el grafo resultante y $i < j$, entonces $j$ es el ancestro de $i$ en el árbol de eliminación.

Esto tiene aplicaciones en la factorización simbólica y puede ser valioso para sincronizar el trabajo en factorizaciones paralelas, ya que refleja las dependencias entre las filas y columnas.

En el contexto de matrices no simétricas, Gilbert y Liu ampliaron el concepto de árbol de eliminación introduciendo los **DAGs de eliminación** (Directed Acyclic Graphs). Estos DAGs se derivan de reducciones transitivas de los grafos dirigidos asociados a las matrices L y U, lo que permite aplicar el concepto del árbol de eliminación a un rango más amplio de matrices y sistemas lineales.

## 8.3. Etapas de resolución de matrices dispersas

- Ordenamiento
- Factorización simbólica
- Factorización Numérica
- Sustitución

### 8.3.1. Ordenamientos

Cuando se trabaja con matrices dispersas, es crucial evitar que se pierda su dispersión en el proceso de factorización, lo que se conoce como el problema de "fill-in". La pérdida de dispersión hace que los métodos de resolución sean inviables, ya que se pierden los beneficios de trabajar con matrices dispersas.

Para abordar este problema, se emplean estrategias de reordenamiento que reorganizan las filas y columnas de la matriz de manera inteligente. Estas estrategias buscan preservar la dispersión de la matriz durante la factorización, lo que a su vez mejora la eficiencia y el rendimiento de los métodos de resolución en sistemas lineales con matrices dispersas.

El número de posibles reordenamientos es igual a $n!$, donde $n$ es el tamaño de la matriz.

Existen tres familias de estrategias de reordenamiento:

1. **Estrategias Globales:** Estas estrategias abordan el problema en su totalidad, dividiéndolo en nuevos subproblemas de ordenamiento. El objetivo es llegar a instancias del problema con soluciones triviales.

2. **Estrategias Locales:** En contraste con las estrategias globales, las estrategias locales toman un enfoque más incremental. Se selecciona un pivote a la vez basado en la información local disponible y se ajusta el orden de la submatriz en consecuencia. Luego, se procede a calcular el orden de los siguientes pivotes.

3. **Estrategias Híbridas:** Las estrategias híbridas combinan elementos de las estrategias globales y locales.

#### Estrategias Globales

Las estrategias de reordenamiento globales desempeñan un papel esencial en la optimización de matrices dispersas durante la factorización. A continuación, se describen algunas de estas estrategias:

- **Cuthill y McKee (CM):** El objetivo principal de esta estrategia es minimizar el ancho de la banda de las matrices dispersas. La metodología implica un proceso jerárquico que procede por niveles. Comienza seleccionando el vértice de menor grado, denominado $v$, y forma el conjunto $L_1 = Adj_G(v)$, es decir que $L_1$ consiste en los vértices adyacentes a $v$. Luego, se construye el conjunto $L_2$, compuesto por los vértices adyacentes a los vértices de $L_1$ que no pertenecen a $L_1$, y así sucesivamente hasta que se incluyen todos los vértices en niveles sucesivos. En cada nivel, los vértices se ordenan según su grado, y se les asignan números correlativos en función de este orden.

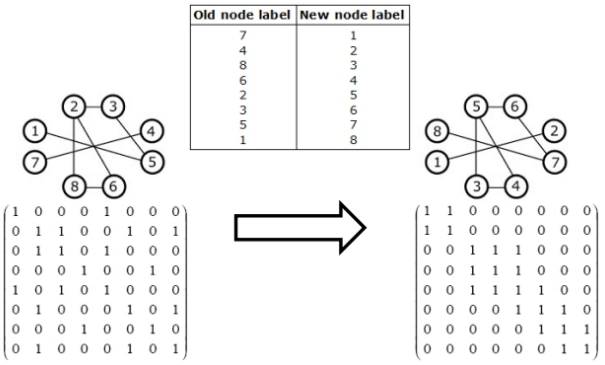



**Figura 8.4**

- **Reverse Cuthill y McKee (RCM):** George propuso una variante de Cuthill y McKee que implica utilizar el ordenamiento inverso del algoritmo CM. Además, se emplea una forma eficiente de identificar los nodos periféricos, lo que puede mejorar aún más el rendimiento de la estrategia.

- **Disección Recursiva (Nested Dissection):** La esencia de este método radica en su enfoque recursivo. Consiste en dividir una grilla o estructura en partes balanceadas a través de un conjunto divisor, que se numerará posteriormente. Luego, se aplica la heurística recursivamente a cada una de las partes, lo que contribuye a optimizar el ordenamiento global de la matriz.

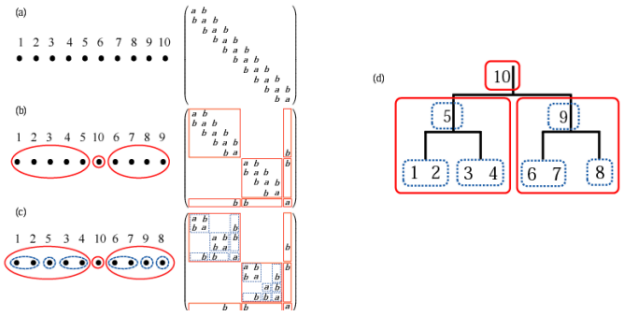



**Figura 8.5**

- **Variantes de Disección Recursiva (ND):** A lo largo de los años, se han presentado diversas variantes de la estrategia de Disección Recursiva. Por ejemplo, en 1978, George y Liu introdujeron la *"Disección Recursiva Automática"* (Automatic Nested Dissection - AND). Luego, en 1979, Lipton et al., y en 1980, Gilbert, extendieron la metodología al proponer el *"Método Recursivo Generalizado"* (Generalized Nested Dissection - GND), que definió un nuevo separador para dividir la matriz de manera más eficiente.

**Estrategias Locales**

En el contexto de estrategias locales de reordenamiento, generalmente asociadas a enfoques voraces, se han desarrollado varias técnicas para evitar el llenado de matrices dispersas. A continuación, se describen algunas de estas estrategias importantes:

- **Markowitz:** Es denominado Fill-in local mínimo (Minimum Local Fill - MF) o deficiencia mínima. La idea detrás de este algoritmo es minimizar la deficiencia, que se refiere a la cantidad de aristas necesarias para convertir el conjunto de vértices adyacentes a un nodo $v$ en un clique, que representa el llenado resultante al eliminar el nodo v. Este enfoque se ha demostrado efectivo para obtener reordenamientos con fill-in mínimo en grafos cordales. Sin embargo, ltiene un gran costo computacional que condicionó su desarrollo, lo que ha llevado a propuestas de variantes aproximadas para reducir el tiempo de cálculo, como el Fill-in Local Mínimo Aproximado (AMF), el Fill-in Local Mínimo Promedio Aproximado (AMMF) o la Mínima Deficiencia Modificada (MMDF).

- **S2**: Este método, contraparte para matrices simétricas del algoritmo de Markowitz, se basa en estimar el llenado utilizando las entradas no nulas por fila y columna de la submatriz pendiente de factorizar. La fórmula para estimar el llenado en el pivote $a_{ii}$ es $(c_i - 1) * (r_i - 1)$, donde $c_i$ representa la cantidad de entradas no nulas en la columna $i$ y $r_i$ la cantidad de entradas en la fila $i$. El producto de estos factores se conoce comúnmente como *"producto de Markowitz"*.

- **Grado Mínimo (Minimum Degree - MD):** Presentado por Rose, este algoritmo utiliza herramientas de grafos para reordenar la matriz. La selección realizada por el algoritmo S2 se puede entender como la elección del nodo con el menor grado de aristas incidentes (menor grado) en el grafo asociado. Por esta razón, el algoritmo S2 también se conoce como el algoritmo de grado mínimo (MD).

  ```
  while G != {} do:

      seleccionar el nodo v con menor grado del grafo G y numerarlo

     G = Gv

  end while
  ```

  El grafo Gv se obtiene borrando el nodo v y todas sus aristas incidentes del grafo G y agregando las aristas necesarias para crear un clique con los nodos adyacentes a v en G. En cada paso del método puede haber varios nodos con mínimo grado y la elección afecta los resultados. La disyuntiva es conocida como *problema de tie-breaking.*

- **Grafos Cocientes (Quotient Graph Model):** Esta técnica busca resolver dos desafíos importantes asociados con la actualización del grafo:

1. Costo Computacional Significativo: La actualización regular del grafo de adyacencias puede ser costosa computacionalmente, especialmente en el contexto de matrices grandes y complejas. Cada actualización implica revisar y modificar el grafo, lo que puede requerir recursos considerables.
2. Predicción de Memoria: Además, la actualización del grafo puede dificultar la predicción precisa de la cantidad de memoria que se necesitará en cada paso. A medida que se actualizan las aristas y se agregan conexiones, la cantidad de memoria requerida puede variar significativamente y ser difícil de estimar con precisión.

Para abordar estos desafíos, se propone trabajar con cliques utilizando la técnica de Grafos Cocientes. La idea principal detrás de los Grafos Cocientes es la siguiente:

- Dado un grafo $G(X, E)$ y una partición de sus vértices en conjuntos $P = \{X_1, X_2, \cdots, X_t\}$, donde cada $X_i$ es un subconjunto disjunto de vértices y la unión de todos los $X_i$ cubre todos los vértices de $G$ ($\cup_{k=1}^{t} X_k = X, \quad X_i \cap X_j = \emptyset$ para $i \neq j$), se crea un nuevo grafo denominado $G/P$.
- En el grafo $G/P$, una arista $\{X_i, X_j\}$ estará presente si y solo si existe una arista $\{u, v\}$ en el grafo original $G$, donde $u \in X_i$ y $v \in X_j$.

Este enfoque permite reducir el problema de actualización de grafo a la manipulación de cliques en el grafo cociente $G/P$. Al trabajar con cliques en lugar de aristas individuales, se simplifica el proceso de actualización y se facilita la estimación de la memoria necesaria.

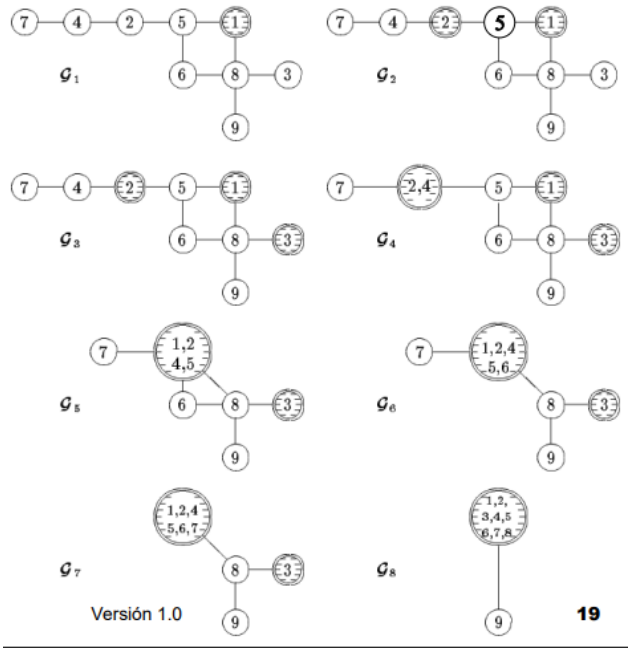



**Figura 8.6**

La Figura 8.6 muestra una secuencia (cualquiera) de grafos cociente. Los nodos rayados son los eliminados hasta el momento. Cuando el nodo a eliminar es adyacente a nodos ya eliminados se genera un "supernodo". La cantidad de nodos/aristas de los grafos cociente en la secuencia nunca es mayor que la del grafo original.

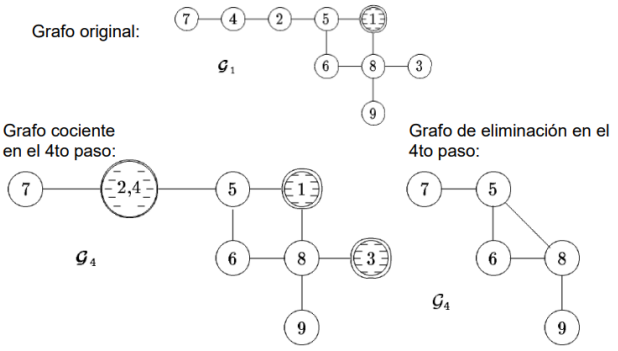



**Figura 8.7**

Además, el proceso de eliminación por Grafos Cocientes se puede ver desde otra perspectiva: cada arista en el grafo original $G$ se representa como un clique de tamaño dos en el grafo cociente $G/P$. Para eliminar un pivote, solo es necesario eliminar los cliques que contienen al nodo eliminado y crear un nuevo clique que incluya todos los vértices adyacentes. Esta tarea implica principalmente la unión de listas de vértices.

Otras mejoras pueden ser:

- **Eliminación de Masa (Mass Elimination):** George y McIntyre introdujeron la idea de eliminación de masa como una mejora en los métodos de ordenamiento. En algunas situaciones, el nodo $v$ de grado mínimo pertenece a un clique, lo que significa que el conjunto de nodos en ese clique se puede eliminar inmediatamente después de $v$, en cualquier orden.

- **Nodos Indistinguibles:** George y Liu propusieron el concepto de nodos indistinguibles. Demostraron que si dos nodos son indistinguibles en el grafo original $G$, entonces seguirán siendo indistinguibles en el grafo $Gv$ después de eliminar un nodo. Los nodos indistinguibles pueden tratarse como un solo nodo en la estructura, lo que se conoce como super-nodos (también llamados supervariables o nodos prototipos por otros autores). Esta técnica reduce la cantidad de cliques necesarios para representar los árboles de eliminación, lo que a su vez reduce la carga computacional durante el ordenamiento.

- **Actualización Incompleta:** Eisenstat, Gursky, Schultz y Sherman presentaron el concepto de actualización incompleta de grado. Se basa en la idea de que no es necesario actualizar el grado de todos los nodos en cada paso de la eliminación. En su lugar, se utiliza el concepto de *"dominado"* (outmatched), donde un nodo $v$ es dominado por $u$ si $u$ si $Adj_G(u) \cup \{u\} \subset$

$Adj_G(v) \cup \{v\}$, es decir que $u$ tiene un grado menor que $v$. Si un nodo $v$ se convierte en dominado por $u$ en algún punto de la eliminación, no es necesario actualizar el grado de $v$ hasta que $u$ sea eliminado.

- **Absorción de Elementos:** Duff y Reid introdujeron la idea de absorción de elementos. Esta técnica se basa en la eliminación de información redundante del grafo de eliminación. Significa que si existe un clique $K_s$ que está completamente incluido en otro clique $K_t$, el clique $K_s$ se puede eliminar del grafo $G$ sin pérdida de información. Esto simplifica el grafo de eliminación y reduce el llenado innecesario.

- **Eliminación Múltiple:** Liu propuso el concepto de eliminación múltiple en el algoritmo de grado mínimo (Multiple Minimum Degree - MMD). Si dos nodos tienen el mismo grado y no son adyacentes, se pueden eliminar en paralelo, o al menos suspender la actualización del grado de los nodos en $Adj_G(v)$ y seleccionar un nodo con el mismo grado que $v$ en el subgrafo $G - (vUAdjG(v))$. Este proceso se repite hasta que no queden nodos con el mismo grado.

- **Grado Externo:** Liu contribuyó con la técnica del grado externo, que trabaja con el grado externo de un nodo en lugar de su grado real. El grado externo se define como la cantidad de aristas que se conectan desde el nodo a otros nodos fuera del clique al que pertenece. Esta estrategia se enfoca en mejorar la calidad de los resultados en términos de llenado (fill-in).

- **Acotar el Grado:** Gilbert, Moler y Schreiber propusieron el uso de una cota superior para el grado, conocida como el grado externo. Además, Amestoy, Davis y Duff presentaron una aproximación más sofisticada llamada algoritmo de grado mínimo aproximado (Approximated MD - AMD), que logra resultados de calidad similar a los obtenidos con el grado mínimo y tiene un mejor rendimiento computacional.

#### Estrategias Híbridas

Las estrategias híbridas consisten en dividir en una primera etapa los grafos con estrategias globales una vez que cada partición posee pocos vértices (algunos cientos), se ordenan mediante estrategias locales.

Otras estrategias utilizadas:

- Simulated annealing

- Algoritmos genéticos

### 8.3.2. Factorización Simbólica

La motivación principal es prever la estructura de las matrices resultantes. Esta etapa de anticipación permite realizar una adecuada reserva de memoria para las diversas estructuras de datos

que se utilizarán en el proceso. Además, se busca facilitar el acceso a estas estructuras en una forma más directa y, lo que es especialmente relevante, no es necesario realizar operaciones en punto flotante durante este proceso.

En el enfoque simbólico, se trabaja principalmente con la estructura de la matriz, es decir, con las posiciones que contienen valores no nulos, en lugar de los propios coeficientes. Se asume que cualquier operación que involucre elementos no nulos resultará en un elemento no nulo, lo que proporciona una estimación de la estructura de la matriz, específicamente una cota superior.

El pionero Schreiber estableció las bases para muchos de los trabajos relacionados con la factorización simbólica al introducir el concepto del árbol de eliminación.

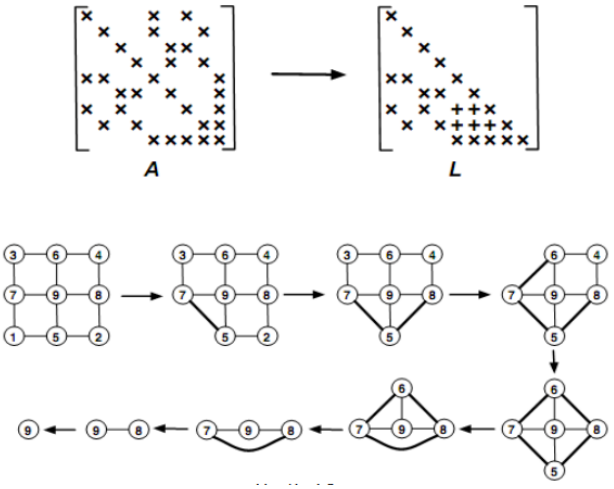



**Figura 8.8**

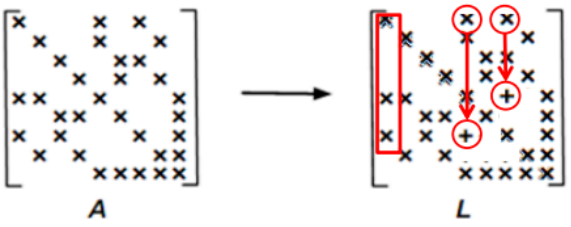



**Figura 8.9**

Durante el proceso de eliminación, en la etapa $k$-ésima, se resta la fila $k$ multiplicada por un coeficiente a todas las filas que tienen elementos no nulos en la columna $k$ (debajo del pivote). Los elementos no nulos de la fila $k$ "bajan" a estas filas y se convierten en elementos no nulos de las mismas. En términos del grafo de adyacencia, esto equivale a que los vértices adyacentes al nodo $k$ se convierten en adyacentes a los vértices correspondientes a las filas mencionadas. Por esta razón, al eliminar un vértice, se forma un clique entre sus adyacentes.

Otra manera de expresar esto es mediante conjuntos. Sea $S_i = \{x_1, \cdots, x_i\}$, $Alc(x_i, S_{i-1})$ representa el conjunto de vértices a los que se puede llegar desde $x_i$ a través de vértices con numeración

menor (que serán eliminados antes que $i$). Si $x_j \in Alc(x_i, S_{i-1})$, significa que existe un camino desde $x_i$ a $x_j$ a través de $S_{i-1}$, lo que indica que cuando se eliminen todos los vértices de $S_{i-1}$ se generará una arista de fill-in entre $x_i$ y $x_j$.

Por lo tanto, el grafo de adyacencia que incluye las aristas de fill-in queda caracterizado por:

$$E^F = \{(x_i, x_j) : \quad x_j \in Alc(x_i, S_{i-1})\} \tag{8.2}$$

El uso de conjuntos alcanzables permite expresar el fill-in únicamente en función de las aristas del grafo original. En cada paso $i$ de la eliminación gaussiana, los nodos adyacentes a cualquier nodo $y$ en el grafo de eliminación $G^i(X_i, E_i)$ se definen como $Alc(y, S_i)$.

## 8.4. Factorización numérica

Dentro de esta técnica, existen diversas sub-técnicas que varían en función de cómo se gestiona el acceso a la memoria y se organizan los datos. Algunas de las sub-técnicas más comunes incluyen:

- Super Nodo
- Frontal
- Multifrontal

### 8.4.1. Método frontal

El método frontal es una variante de los métodos de factorización que fue desarrollada en los años 70 por Irons para resolver problemas que implicaban matrices simétricas y definidas positivas de gran tamaño, las cuales se generaban al aplicar el método de elementos finitos en problemas de análisis estructural.

Este enfoque se inspira en la metodología de trabajo de los Métodos de Elementos Finitos (MEF) y establece una relación entre la factorización utilizada en el método frontal y la construcción de la matriz de rigidez global en los métodos de elementos finitos a través del ensamblaje de mas matrices de rigidez de cada elemento según:

$$M = \sum_e M_e \tag{8.3}$$

donde $B_e$ son las matrices de rigidez de los distintos elementos y $A$ es la matriz de rigidez global.

Cuando se discretiza un problema utilizando los métodos de elementos finitos, se obtiene una estructura específica para la matriz de rigidez asociada al problema, lo que significa que no todos los elementos de esta matriz son diferentes de cero. En otras palabras, solo algunas ubicaciones de la matriz contienen información significativa.

El método frontal se ilustra mediante un ejemplo en el que se utilizan 4 elementos triangulares, cada uno con 3 nodos, y una sola variable libre, lo que resulta en una matriz de rigidez global con

6 ecuaciones. Las matrices de rigidez asociadas a cada uno de los 4 elementos se presentan en el ejemplo:

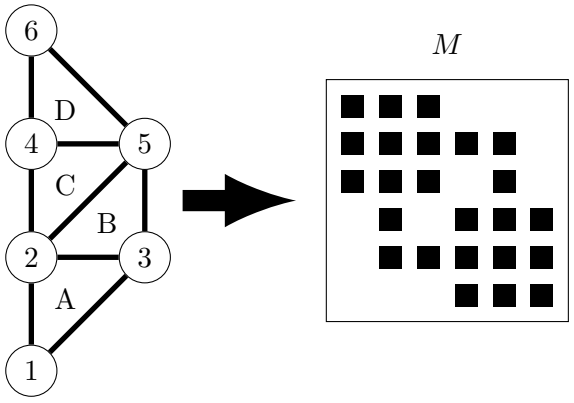
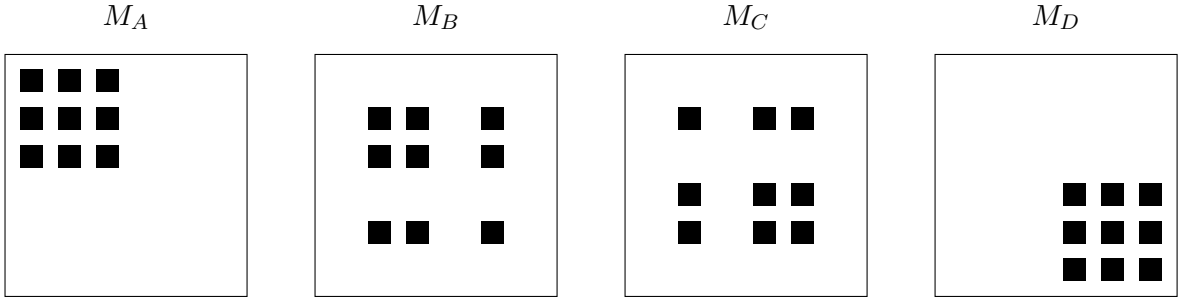



**Figura 8.10**: Cada la numeración de filas y columnas de la matriz $M$ identifica a los nodos del grafo, el elemento $a_{ij}$ de la matriz se llena si los nodos $i, j$ están conectados por una arista




(a) Al elemento triangular $A$ lo forman los nodos 1, 2 y 3, estando todos interconectados, por lo que la matriz $M_A$ tiene llenos todos los elementos tales que $i \in \{1, 2, 3\}$ y $j \in \{1, 2, 3\}$

(b) Al elemento triangular $B$ lo forman los nodos 2, 3 y 5, estando todos interconectados, por lo que la matriz $M_B$ tiene llenos todos los elementos tales que $i \in \{2, 3, 5\}$ y $j \in \{2, 3, 5\}$

(c) Al elemento triangular $C$ lo forman los nodos 2, 4 y 5, estando todos interconectados, por lo que la matriz $M_C$ tiene llenos todos los elementos tales que $i \in \{2, 4, 5\}$ y $j \in \{2, 4, 5\}$

(d) Al elemento triangular $D$ lo forman los nodos 4, 5 y 6, estando todos interconectados, por lo que la matriz $M_D$ tiene llenos todos los elementos tales que $i \in \{4, 5, 6\}$ y $j \in \{4, 5, 6\}$

**Figura 8.11**

La contribución principal de Irons, el desarrollador del método frontal, radica en la observación de que no es necesario calcular la matriz de rigidez global completa para luego factorizarla. En cambio, se pueden realizar secuencias de factorizaciones a medida que las submatrices (las matrices de rigidez de cada elemento, las $M_i$) se completen. Esto significa que se puede factorizar una fila y columna una vez que ésta haya recibido todos los aportes.

**Algorithm 4** Método frontal

1: Calcular los coeficientes que aportan los elementos mientras se puedan agregar a la matriz frontal.
2: Buscar un pivot en la matriz frontal, dentro de las filas/columnas que se encuentran completamente sumadas (aquellas que no recibirían más aportes).
3: Se elimina le pivot.
4: Mientras se puedan eliminar pıvot se repiten los pasos 2 y 3.
5: Luego se agregan los coeficientes de la matriz del siguiente elemento a la matriz frontal y se repiten los pasos 2 y 3.
6: Cuando se eliminan todos los pıvot se resuelve mediante sustitución.

**Ejemplo: Solución frontal de problema tridiagonal**

El objetivo es encontrar una distribución de temperaturas $x \mapsto u(x)$, $x, u(x) \in \mathbb{R}$ tal que:

$$\begin{cases} \frac{d^2u}{dx^2} = 0 & x \in [0,1] \\ u(0) = 0 \\ u(1) = 20 \end{cases} \tag{8.4}$$

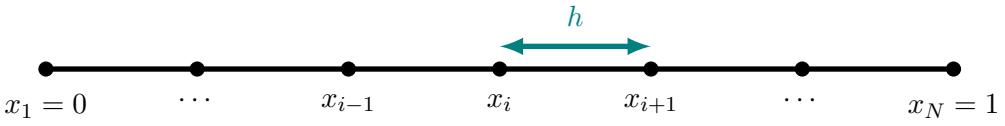



**Figura 8.12**

Donde la discretización por diferencias finitas queda:

$$\begin{cases} \frac{u_{i+1}-2u_i+u_{i-1}}{h^2} & i \in \{1, \cdots, N\} \\ u_0 = 0 \\ u_N = 20 \end{cases} \tag{8.5}$$

donde $h = \frac{1}{N}$, $u_i = u(x_i) = u\left(\frac{i}{N}\right)$ y $[0,1] = \bigcup_{i=1}^{N-1}[x_i, x_{i+1}]$ por lo que el sistema con matriz tridiagonal de este problema es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_i \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 20 \end{pmatrix} \tag{8.6}$$

En este caso se toma como matriz frontal de los indíces $1,1$ al $2,3$, aquella que tiene la información necesaria para eliminar la primera fila:

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \end{pmatrix} \implies F_2 = F_2 - \frac{1}{h^2}F_1 \implies \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{2}{h^2} & \frac{1}{h^2} \end{pmatrix} \tag{8.7}$$

quedando entonces el sistema:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_i \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 20 \end{pmatrix} \tag{8.8}$$

Aquí ya se eliminó la primera fila, la siguiente matriz frontal a tomar es aquella que va de los índices $2,2$ al $3,4$, es decir, aquella que tiene la información para eliminar la segunda fila

$$\begin{pmatrix} -\frac{2}{h^2} & \frac{1}{h^2} & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \end{pmatrix} \implies F_3 = F_3 - \frac{\frac{1}{h^2}}{-\frac{2}{h^2}}F_2 \implies \begin{pmatrix} -\frac{2}{h^2} & \frac{1}{h^2} & 0 \\ 0 & -\frac{3}{2h^2} & \frac{1}{h^2} \end{pmatrix} \tag{8.9}$$

Luego se inserta esto en el sistema como en el paso anteior. Al final del proceso se obtiene una matriz triangular superior.

### 8.4.2. Método multifrontal

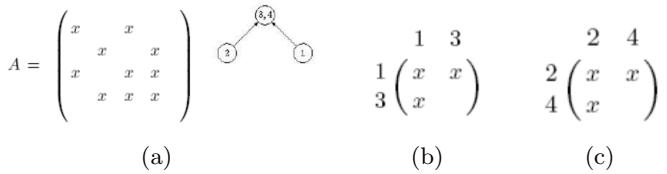
El método multifrontal es una extensión de los métodos frontales propuesta por Duff y Reid. La principal contribución de los métodos multifrontales radica en su capacidad para trabajar con varios frentes de manera independiente, calculando la factorización de cada frente y almacenando sus contribuciones para un uso futuro.

Para llevar a cabo esta tarea, primero se realiza un análisis de las dependencias de datos entre los diversos frentes. Una vez que se ha establecido un orden de eliminación, representado comúnmente mediante un árbol de eliminación, se procede a resolver cada frente de manera secuencial. Previo a la factorización de un frente, es esencial ensamblar las matrices de rigidez asociadas a los elementos de dicho frente y actualizarlas con las contribuciones de los frentes previamente resueltos.

Un ejemplo ilustrativo muestra una matriz A junto con su árbol de eliminación.

La técnica multifrontal comienza por factorizar el primer frente, que consiste en las entradas 1 y 3 de la matriz $A$. Las contribuciones de este frente se almacenan en una estructura auxiliar, ya que el segundo frente no depende de datos generados por el primero.

Una variante interesante propuesta por David y Duff combina estrategias frontales con el método multifrontal. Posteriormente, los avances en los métodos multifrontales se han centrado en las



(a) (b) (c)

**Figura 8.13**: Ejemplo de matriz $A$ y árbol de eliminación

etapas de ordenamiento y factorización simbólica, así como en la explotación de las características de diversas arquitecturas de computadoras y la optimización de la gestión de la memoria. El paralelismo también se ha vuelto fundamental en muchas aplicaciones.

En el trabajo de Davis y Duff, se presenta una versión del método multifrontal adaptada para matrices no simétricas. Su enfoque se basa en el uso de Grafos Acíclicos Dirigidos (DAGs) en lugar de árboles de eliminación para representar el proceso de factorización. Entre las características destacadas de esta propuesta se incluye la capacidad de manejar frentes rectangulares en contraposición al método multifrontal original, que se basa en frentes cuadrados, y la combinación de las etapas de análisis y factorización numérica para lograr un enfoque más eficiente en la resolución de sistemas de ecuaciones lineales complejos.

**Ejemplo: Solución multifrontal del problema anterior**

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & 0 & \cdots & 0 \\ 0 & \cdots & \frac{1}{h^2} & -\frac{2}{h^2}\frac{1}{h^2} & \cdots & & 0 \\ 0 & \cdots & 0 & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & 0 \\ 0 & 0 & \cdots & 0 & \frac{1}{h^2} & -\frac{2}{h^2}\frac{1}{h^2} & \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix}$$

**Figura 8.14**

Se construyen múltiples matrices frontales de forma que su suma sea igual a la matriz original. Las variables también se dividen en partes:

$$u_i = u_i^I + u_i^{II} \tag{8.10}$$

$$\begin{pmatrix} 1 & 0 \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_2^{II} \\ u_3^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_3^{II} \\ u_4^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{8.11}$$

$$\begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_4^{II} \\ u_5^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_5^{II} \\ u_6^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_6^{II} \\ u_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 20 \end{pmatrix} \tag{8.12}$$

La primer y segunda matriz (azul y celeste) se suman en una matriz de $3 \times 3$, así quedan la primer y segunda filas completamente definidas:

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \Rightarrow F_2 = F_2 - \left(-\frac{1}{h^2}\right) F_1 \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{8.13}$$

Esto se puede resumir como el frente:

$$\begin{pmatrix} -\frac{2}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_2 \\ u_3^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{8.14}$$

Por otro lado si se suman los frentes 3 y 4 (verde y amarillo) en una matriz de $3 \times 3$ se tiene que solo la segunda fila está completamente definida:

$$\begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & \frac{1}{h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_3^{II} \\ u_4 \\ u_5^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \Rightarrow \text{Reordeno} \begin{cases} F_1 \leftrightarrow F_2 \\ C_1 \leftrightarrow C_2 \end{cases} \Rightarrow \begin{pmatrix} -\frac{2}{h^2} & \frac{1}{h^2} & \frac{1}{h^2} \\ \frac{1}{h^2} & -\frac{1}{h^2} & 0 \\ \frac{1}{h^2} & 0 & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_4 \\ u_3^{II} \\ u_5^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{8.15}$$

La idea es eliminar los coeficientes debajo de la diagonal:

$$\begin{cases} F_2 = F_2 - \frac{\frac{1}{h^2}}{-\frac{2}{h^2}} F_1 \\ F_3 = F_3 - \frac{\frac{1}{h^2}}{-\frac{2}{h^2}} F_1 \end{cases} \Rightarrow \begin{pmatrix} -\frac{2}{h^2} & \frac{1}{h^2} & \frac{1}{h^2} \\ 0 & -\frac{1}{h^2} & \frac{1}{2h^2} \\ 0 & \frac{1}{2h^2} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} u_4 \\ u_3^{II} \\ u_5^I \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{8.16}$$

Se repite el proceso en los frentes 5 y 6 (naranja y rojo), ahora las ecuaciones 2 y 3 están completamente definidas:

$$\begin{pmatrix} -\frac{1}{h^2} & \frac{1}{h^2} & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_5^{II} \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 20 \end{pmatrix} \tag{8.17}$$

Así se generan todos los frentes en paralelo, para poder ejecutar las sumas y las eliminaciones de distintos pares de matrices frontales en simultáneo.

### 8.4.3. Bibliotecas

Existen diversas bibliotecas que implementan métodos frontales y multifrontales para la resolución eficiente de sistemas de ecuaciones lineales. Algunas de las bibliotecas más destacadas incluyen:

**Métodos Frontales**

- **HSL (Harwell Subroutine Library):** Una biblioteca que proporciona una amplia gama de solvers y rutinas para problemas matriciales, incluyendo métodos frontales.
  - **MA42:** Un solver frontal out-of-core no simétrico propuesto por Duff y Scott en 1996.
  - **MA62:** Un solver frontal out-of-core simétrico propuesto por Duff y Scott en 1997.

**Métodos Multifrontales:**

- **MUMPS:** Un método multifrontal no simétrico distribuido, que permite el procesamiento eficiente de sistemas de ecuaciones lineales en paralelo.
- **UMFPACK:** Un método multifrontal no simétrico propuesto por Davis, actualmente parte de SuiteSparse y distribuido con Matlab. Este método multifrontal se destaca por su capacidad de resolver sistemas de ecuaciones lineales de gran escala.
- **HSL (Harwell Subroutine Library):** Varias rutinas relacionadas con métodos multifrontales, incluyendo MA38, MA41, MA57, HSL_MA77 y HSL_MA78. Estas rutinas amplían la funcionalidad de HSL para abordar problemas matriciales complejos.

# Capítulo 9

# Métodos Iterativos Básicos

Los métodos directos para resolver sistemas de ecuaciones lineales presentan ventajas notables. Estos métodos son capaces de llegar a la solución en un número específico de pasos, lo que resulta conveniente en función del tamaño del sistema a resolver. Además, proporcionan la solución exacta del sistema lineal, a menos de errores numéricos. No obstante, su principal desventaja radica en su alto costo computacional, lo que los vuelve prohibitivos cuando se trata de matrices grandes.

Por otro lado, los métodos iterativos buscan aproximarse a la solución del sistema de ecuaciones de manera diferente. En lugar de buscar la respuesta exacta de manera directa, generan una sucesión de valores iterativamente, es decir, buscamos una sucesión $\{x_k\}_{k=0}^{\infty}$ tal que si $x^* = A^{-1}b$ cumpla:

$$\lim_{k\to\infty} x_k = x^* \tag{9.1}$$

Si estos métodos convergen, se obtiene una solución aproximada del problema, la cual satisface algún criterio de precisión predefinido. Es fundamental garantizar la convergencia de estos métodos e incluso estimar la tasa de convergencia, es decir, cómo varía el error al aumentar el número de iteraciones.

Sin embargo, surge la interrogante de cuándo detener la iteración en estos métodos iterativos. La respuesta depende del error deseado, pero determinar el error exacto no es posible ya que la solución del problema no lo es. Una forma detener las iteraciones, si se tiene $r_k = b - Ax_k$, es utilizando la siguiente condición de parada:

$$\frac{\|r_k\|}{\|r_0\|} < \text{tol} \tag{9.2}$$

En el contexto de trabajar con una máquina, la precisión alcanzable está relacionada con la precisión de la máquina ($\epsilon_{mach}$) y el número de condición de la matriz $A$ ($\kappa(A)$). Este número de condición influye en la estabilidad numérica del método, y la precisión de la máquina establece un límite en la exactitud que puede lograrse, siendo esta del orden $\epsilon_{mach} \times \kappa(A) \times \|b\|$. Por lo tanto, es esencial considerar estas precisiones al determinar cuándo detener la iteración en métodos iterativos.

El problema se puede reescribir como una iteración de punto fijo:

$$x_{k+1} = Bx_k + c \tag{9.3}$$

donde a la matriz $B$ se la denomina matriz de amplificación.

Para la resolución de este problema existen distintas familias de métodos:

- **Métodos Estacionarios:** Se caracterizan por el hecho de que tanto $B$ como $c$ no dependen de la iteración $k$. Este enfoque es generalmente más sencillo de implementar, pero suele tener una menor capacidad de convergencia.
- **Métodos No estacionarios:** La matriz de amplificación $B$ y el vector $c$ pueden variar en cada iteración, lo que puede aumentar la flexibilidad pero también complicar la implementación.

# Capítulo 10

# Métodos Estacionarios

En el contexto de los métodos estacionarios, se deduce una relación relevante. Partiendo de la ecuación original $Ax = b$, esta se puede reformular como $Mx = (M - A)x + b$, introduciendo una matriz invertible $M$ para simplificar el proceso. Por lo que se puede considerar la siguiente recursión:

$$Mx^{(k+1)} = (M - A)x^{(k)} + b \tag{10.1}$$

Si se parte de una solución inicial $x_0$, si la solución converge se tendrá:

$$M^{(\infty)} = (M - A)x^{(\infty)} + b \iff Ax^{(\infty)} = b \tag{10.2}$$

Restando las ecuaciones 10.1 y 10.2 se obtiene:

$$M(x^{(\infty)} - x^{(k+1)}) = (M - A)(x^{(\infty)} - x^{(k)}) \tag{10.3}$$

definiendo: $\begin{cases} e^{(k+1)} = x^{(\infty)} - x^{(k+1)} \\ e^{(k)} = x^{(\infty)} - x^{(k)} \end{cases}$ se tiene:

$$e^{(k+1)} = M^{-1}(M - A)e^{(k)} \tag{10.4}$$

Definiendo a su vez $M^{-1}(M - A) = \mathbb{I} - M^{-1}A = Q$ se tiene:

$$e^{(k+1)} = Qe^{(k)} \implies e^{(k+1)} = Q^k e^{(0)} \tag{10.5}$$

Por lo tanto:

$$e^{(k)} \xrightarrow{k} 0 \iff \|Q^k\| \xrightarrow{k} 0 \tag{10.6}$$

**Proposición 10.0.0.1.** *Considerando la iteración $e^{(k+1)} = Q^k e^{(0)}$ se tiene que es condición suficiente para asegurar la convergencia que $\|Q\| < 1$ para alguna norma matricial.*

**Demostración 10.0.0.1.** *En este caso caso basta basta utilizar la propiedad de normas matriciales:*

$$\|AB\| \leq \|A\| \cdot \|B\| \tag{10.7}$$

*Por lo tanto se cumple:*

$$\|Q^k\| \implies \|Q\|^k \tag{10.8}$$

**Proposición 10.0.0.2.** *Considerando la iteración $e^{(k+1)} = Q^k e^{(0)}$ se tiene que es condición necesaria y suficiente para asegurar la convergencia que el radio espectral de la matriz $Q$ sea menor a 1:*

$$\rho(Q) < 1 \tag{10.9}$$

**Demostración 10.0.0.2.** *(Suficiencia) Sea $\lambda$ un valor propio de $Q$ y $v$ el vector propio asociado tal que $\|v\| = 1$.*

*Se tiene que $\|Q^k v\| = \|\lambda^k v\| = |\lambda^k|$. Si $\rho(Q) < 1$ se tiene que $|\lambda^k| \to 0$ entonces $\|Q^k\| \to 0$.*

*(Necesidad) Si $\|Q^k\| \to 0$ y $v$ es un vector propio de norma unitaria entonces:*

$$\|Q^k v\| \leq \|Q^k\| \implies \|Q^k v\| \to 0 \tag{10.10}$$

*Por otro lado, utilizando la definición de valor propio se tiene que: $\|Q^k v\| = |\lambda^k|$, siendo $\lambda$ el valor propio asociado al vector propio de norma unitaria. Como $|\lambda^k| \to 0$ entonces $\lambda < 1$.*

**Proposición 10.0.0.3** (Velocidad de convergencia)**.** *Sea $q^{(k)} = \frac{\|e^{(k+1)}\|}{\|e^{(k)}\|}$. Para matrices normales se cumple que:*

$$q^{(k)} \approx \rho(Q) \tag{10.11}$$

## 10.1. Método de Jacobi

Este método consiste en tomar la matriz $M$ como la diagonal de $A$ (si $a_{ii} \neq 0$), en cuyo caso la inversa de $M$ es trivial.

Las iteraciones para cada entrada serían de la forma:

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq 1} a_{ij} x_j^{(k-1)}}{a_{ii}} \tag{10.12}$$

$$x^{(k)} = D^{-1}(-L - U)x^{k-1} + D^{-1}b \tag{10.13}$$

donde $D$ es la diagonal de la matriz $A$, $L$ la parte triangular inferior y $U$ la triangular superior, es decir $A = L + D + U$.

**Ejemplo**

Dado el sistema:

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} x = b \tag{10.14}$$

Por lo tanto el método de Jacobi queda:

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} x^{(k+1)} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} x^{(k)} + b \implies x^{(k+1)} = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} x^{(k)} + \frac{b}{2} \quad \lambda_i = \pm\frac{1}{2} \tag{10.15}$$

## 10.2. Método de Gauss-Seidel

Las iteraciones para cada entrada serían de la forma:

$$x_i^{(k)} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)}}{a_{ii}} \tag{10.16}$$

Escribiéndolo en forma matricial sería:

$$x^{(k)} = (D + L)^{-1}(-Ux^{k-1} + b) \tag{10.17}$$

donde $D$ es la diagonal de la matriz $A$, $L$ la parte triangular inferior y $U$ la triangular superior, es decir $A = L + D + U$.

**Ejemplo**

Dado el sistema:

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} x = b \tag{10.18}$$

Por lo tanto el método de Gauss-Seidel queda:

$$\begin{pmatrix} 2 & 0 \\ -1 & 2 \end{pmatrix} x^{(k+1)} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} x^{(k)} + b \implies x^{(k+1)} = \begin{pmatrix} 0 & \frac{1}{2} \\ 0 & \frac{1}{4} \end{pmatrix} x^{(k)} + \begin{pmatrix} \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{2} \end{pmatrix} b \quad \lambda_i = 0, \frac{1}{4} \tag{10.19}$$

## 10.3. Comparación de Jacobi - Gauss-Seidel

1. El método de Jacobi ofrece una ventaja significativa: no es necesario conocer las componentes del vector previamente calculadas.

2. Generalmente se observa que la velocidad de convergencia del método Gauss-Seidel es superior. En el método de Gauss-Seidel, se actualizan las componentes del vector a medida que se calculan, lo que conduce a una convergencia más rápida en muchos casos. La información actualizada en una iteración se utiliza inmediatamente en la siguiente, lo que puede mejorar la eficiencia del método.

## 10.4. Método SOR (Succesive Over-Relaxation)

El método SOR (Successive Over-Relaxation) se basa en la aplicación de una extrapolación al método de Gauss-Seidel. Donde la iteración en este caso será:

$$x_i^{(k)} = \omega \tilde{x}_i^{(k)} + (1 - \omega) x_i^{(k-1)} \tag{10.20}$$

En esta metodología, $\tilde{x}_i^{(k)}$ es la iteración de Gauss-Seidel y la variable $\omega$ es un parámetro que se define previamente. La idea subyacente es que este parámetro mejora la velocidad de convergencia del método, sin embargo, determinar el valor óptimo de $\omega$ puede ser una tarea complicada.

Kahan demostró que el método SOR solo converge cuando $\omega \in (0, 2)$.

# Capítulo 11

# Métodos No Estacionarios

Los métodos estacionarios hacen uso de información evaluada en cada iteración, lo que permite obtener la solución de forma dinámica. Esto suele resultar en una mayor eficiencia en comparación con los métodos estacionarios previamente abordados.

## 11.1. Métodos Basados en Subespacios de Krylov

**Definición 11.1.0.1** (Subespacio de Krylov)**.** *Se define el* ***subespacio de Krylov*** *de dimensión $m$ asociado a la matriz $A$ y al vector $v$ como el subespacio generado por la base:*

$$\{v, Av, A^2v, \cdots, A^{m-1}v\}$$

*Se lo denota por $K_m(A, v)$*

### 11.1.1. Propiedades de los subespacios de Krylov

- $AK_m \subset K_{m+1}$
- $K_1 \subset K_2 \subset \cdots \subset K_m \subset \cdots$
- Ortogonalidad: Si $v \perp K_m \implies Av \perp K_{m-1}$

  Cualquier vector ortogonal a un espacio de Krylov se transforma bajo A en otro también ortogonal, pero al espacio de Krylov un orden menor.

Volviendo a la Ecuación 10.1 se puede definir el residuo del $k$-ésimo paso como:

$$r^{(k)} = b - Ax^{(k)} \tag{11.1}$$

De esta forma se puede saber qué tan lejos está $Ax^{(k)}$ de $b$, pero no cuán lejos está $x^{(k)}$ de la solución $x$.

Si se introduce el residuo en Ecuación 10.1 se tiene que:

$$Mx^{(k+1)} = Mx^{(k)} - Ax^{(k)} + b \iff Mx^{(k+1)} = Mx^{(k)} + r^{(k)} \iff x^{(k+1)} = x^{(k)} + M^{-1}r^{(k)} \quad (11.2)$$

De esta forma la iteración $x^{(k)}$ se corrige en cada paso utilizando la información brindada por el residuo del sistema.

Si se toma como matriz $M$ a la matriz identidad y se considera el paso inicial como $x^{(0)} = b$ se tiene que las primeras iteraciones serán de la forma:

$$x^{(1)} = (\mathbb{I} - A)b = 2b - Ab \quad (11.3)$$

$$x^{(2)} = (\mathbb{I} - A)(2b - Ab) + b = 3b - 3Ab + A^2b \quad (11.4)$$

Como puede observarse $x^{(k)}$ será una combinación lineal del subespacio de Krylov de dimensión $k$ $K_k(A, b) = \{b, Ab, A^2b, \cdots, A^kb\}$.

Los métodos iterativos basados en subespacios de Krylov buscan la solución en $K(A, b)$ según distintos criterios.

Como se había considerado $M = \mathbb{I}$ se tiene que:

$$x^{(k+1)} = x^{(k)} + r^{(k)} \quad (11.5)$$

Para intentar mejorar la convergencia se puede incorporar un parámetro $\alpha^{(k)}$:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)}r^{(k)} \quad (11.6)$$

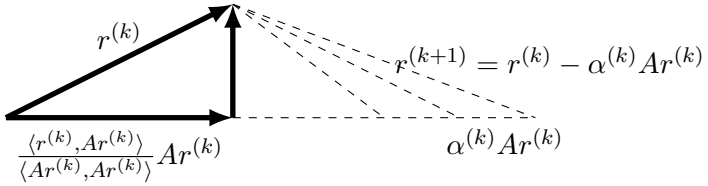
La pregunta es cómo elegir ese $\alpha^{(k)}$ de forma apropiada. Algunos criterios pueden ser:

- Minimizar la norma-2 del residuo de la siguiente iteración:

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)}Ar^{(k)} \quad (11.7)$$

Se elige $\alpha^{(k)}$ de tal forma que $\alpha^{(k)}Ar^{(k)}$ sea la proyección de $r^{(k)}$ sobre $Ar(k)$:

$$\alpha^{(k)} = \frac{\langle r^{(k)}, Ar^{(k)} \rangle}{\langle Ar^{(k)}, Ar^{(k)} \rangle} \quad (11.8)$$




Así se puede definir el error en el $k$-ésimo paso como:

$$e^{(k)} = x - x^{(k)} = A^{-1}b - x^{(k)} \quad (11.9)$$

que se relaciona con el residuo de la siguiente manera:

$$e^{(k)} = A^{-1}b - x^{(k)} = A^{-1}b - A^{-1}Ax^{(k)} = A^{-1}r^{(k)} \tag{11.10}$$

- Minimizar la norma-2 del error de la siguiente iteración:

$$e^{(k+1)} = e^{(k)} - \alpha^{(k)}r^{(k)} \tag{11.11}$$

Aquí el problema es que para realizar el cálculo de $\alpha^{(k)} = \frac{\langle e^{(k)}, r^{(k)}\rangle}{\langle r^{(k)}, r^{(k)}\rangle}$ se necesita conocer $A^{-1}$.

Si $A$ es simétrica y definida positiva una alternativa es minimizar el error de la siguiente iteración usando la norma-$A$:

$$\alpha^{(k)} = \frac{\langle e^{(k)}, r^{(k)}\rangle_A}{\langle r^{(k)}, r^{(k)}\rangle_A} = \frac{\langle e^{(k)}, Ar^{(k)}\rangle}{\langle r^{(k)}, Ar^{(k)}\rangle} = \frac{\langle A^{-1}r^{(k)}, Ar^{(k)}\rangle}{\langle r^{(k)}, Ar^{(k)}\rangle} = \frac{\langle r^{(k)}, r^{(k)}\rangle_A}{\langle r^{(k)}, Ar^{(k)}\rangle} \tag{11.12}$$

De esta manera se obtiene el **método de máximo descenso.**

### 11.1.2. Métodos de descenso

Para tomar $Ax = b$ como un problema de minimización se tiene que considerar, si $A \in \mathbb{R}^{n\times n}$ es simétrica y definida positiva y $b \in \mathbb{R}^n$, la función cuadrática:

$$\Phi : \mathbb{R}^n \to \mathbb{R} \quad / \quad \Phi(x) = \frac{1}{2}\langle Ax, x\rangle - \langle b, x\rangle \tag{11.13}$$

[1] la cual alcanza un único mínimo en el vector $x \in \mathbb{R}^n$ solución del sistema $Ax = b$.

Como $A$ es simétrica se cumple que el gradiente de $\Phi$ es de la siguiente forma:

$$\nabla\Phi(x) = \left(\frac{\partial\Phi}{\partial x}(x)\right) = \frac{1}{2}\left(Ax + ATx\right) - b = Ax - b \tag{11.14}$$

Entonces:

$$\nabla\Phi(x) = 0 \iff Ax = b \tag{11.15}$$

Además, la matriz hessiana asociada a $\Phi$ es:

$$H_\Phi(x) = \left(\frac{\partial^2\Phi}{\partial x_i \partial x_j}(x)\right) = (a_{ij}) = A \tag{11.16}$$

por lo que si $A$ es definida positiva, el punto crítico de $\Phi$ es un mínimo.

De forma general en los métodos de descenso se parte de un vector $u^{(0)} \in \mathbb{R}^n$, en cada paso $k = 0, , 1, \cdots$ se determina un nuevo punto:

$$u^{(k+1)} \in \mathbb{R}^n / \Phi\left(u^{(k+1)}\right) < \Phi\left(u^{(k)}\right) \tag{11.17}$$

de la siguiente manera:

[1] $\Phi(x) = \frac{1}{2}\langle Ax, x\rangle - \langle b, x\rangle = \frac{1}{2}x^TAx - x^Tb$

1. Se calcula la dirección de búsqueda $p^{(k)}$.

2. Se considera la recta $L_k$ que pasa por el punto $u^{(k)}$ con dirección $p^{(k)}$.

3. Se elige el punto $u^{(k+1)} \in L_k$ donde $\Phi$ alcanza su mínimo (sobre $L_k$).

Como $L_k = \{u^{(k)} + \alpha p^{(k)} : \alpha \in \mathbb{R}\}$ entonces:

$$\Phi\left(u^{(k)} + \alpha p^{(k)}\right) = \left[\frac{1}{2}\langle Au^{(k)}, u^{(k)}\rangle - \langle b, u^{(k)\rangle}\right] + \langle Au^k - b, p^{(k)}\rangle\alpha + \frac{1}{2}\langle Ap^{(k)}, p^{(k)}\rangle\alpha^2 \tag{11.18}$$

Por lo tanto:

$$\frac{\partial\Phi}{\partial\alpha}\left(u^{(k)+p^{(k)}}\right) = 0 \iff \alpha = \alpha^{(k)} = \frac{\langle r^{(k)}, p^{(k)}\rangle}{\langle Ap^{(k)}, p^{(k)}\rangle} \tag{11.19}$$

Donde $r^{(k)} = b - Au^{(k)}$ es el residuo de $u^{(k)}$. Luego se tiene que:

$$u^{(k+1)} = u^{(k)} + \alpha^{(k)}p^{(k)} \tag{11.20}$$

### 11.1.3. Método del máximo descenso o de descenso por gradiente

Los métodos de descenso se distinguen por la forma en la que se elige la dirección $p^{(k)}$, siendo la elección más simple la de máximo descenso de $\Phi$:

$$p^{(k)} = -\nabla\Phi\left(u^{(k)}\right) = b - Au^{(k)} = r^{(k)} \tag{11.21}$$

Los pasos a seguir son los siguientes:

1. Se calcula la dirección de máximo descenso en base a la solución inicial: $r = b - Ax$

2. Se halla la intersección entre la forma cuadrática y el plano determinado por la dirección de máximo descenso y la vertical.

3. La intersección es una parábola, se elige como siguiente punto el mínimo.

Aquí se tiene que la dirección del gradiente en el punto elegido es ortogonal a la dirección de máximo descenso inicial.

**Algorithm 5** Máximo descenso I

**Entradas:** Vector inicial $u^{(0)}$

1: $r^{(k)} = b - Au^{(k)}$
2: $\alpha^{(k)} = \frac{\langle r^{(k)}, r^{(k)}\rangle}{\langle Ar^{(k)}, r^{(k)}\rangle}$
3: $u^{(k+1)} = u^{(k)} + \alpha^{(k)}r^{(k)}$

Este algoritmo necesita de dos productos matriz-vector por iteración.

**Algorithm 6** Máximo descenso II

**Entradas:** Vector inicial $u^{(0)}$, $r^{(0)} = b - Au^{(0)}$

1: **while** A **do**lgún criterio de detención
2: $\alpha^{(k)} = \frac{\langle r^{(k)}, r^{(k)} \rangle}{\langle Ar^{(k)}, r^{(k)} \rangle}$
3: $u^{(k+1)} = u^{(k)} + \alpha^{(k)} r^{(k)}$
4: $r^{(k+1)} = b - Au^{(k+1)} = r^{(k)} - \alpha^{(k)} Ar^{(k)}$
5: **end while**

Este algoritmo necesita de un único producto matriz-vector por iteración.

**Propiedad de convergencia finita**

Tras $n$ iteraciones (como máximo) se alcanza la solución exacta, suponiendo que no haya efecto de los errores de redondeo (aritmética exacta).

Sin embargo en aritmética finita este resultado ya no es válido.

Debido a errores de redondeo no se puede alcanzar cualquier nivel de precisión en el residuo, por este motivo es que se establece una cota de la precisión alcanzable en relación con factores como la precisión de la máquina, las normas de los vectores que se utilizan y el número de condición del problema.

$$\|r^{(i)}\| \leq \text{cota}(cond(A), \|b\|, \epsilon_{mach}, \cdots) \tag{11.22}$$

### 11.1.4. Método del gradiente conjugado

El método de máximo descenso frecuentemente toma varios pasos en la misma dirección, realizando "zigzagueos", para corregir los movimientos en cada paso se toman direcciones conjugadas.

La mejora se basa en ortogonalizar el residuo $r^{(k)}$ respecto de todas las direcciones de descenso anteriores $p^{(j)} \quad j = 1; \cdots, k-1$ en el producto interno:

$$\langle x, y \rangle_A = \langle Ax, y \rangle \quad x, y \in \mathbb{R}^n \tag{11.23}$$

Las direcciones obtenidas satisfacen:

$$\langle Ap^{(k)}, p^{(j)} \rangle = 0 \quad \forall j \neq k \tag{11.24}$$

Estas son llamadas **direcciones conjugadas** o **direcciones A-ortogonales**.

La metodología se basa en la construcción de una base de vectores ortogonales, lo que permite realizar búsquedas de soluciones de manera eficiente. La ventaja fundamental de este método es que solamente se necesita asegurar la ortogonalidad del nuevo elemento respecto del último construido.

En cada iteración del método, se calcula el residuo correspondiente a las diversas soluciones, así como una dirección de búsqueda:

$$p^{(k)} = r^{(k)} + \beta^{(k-1)} p^{(k-1)} \tag{11.25}$$

La elección de la variable $\beta^{(k-1)}$ asegura que $r^{(k)}$ y $r^{(k-1)}$ sean ortogonales a todos los previos.

**Algorithm 7** Gradiente Conjugado I

**Entradas:** Vector inicial $u^{(0)}$, $r^{(0)} = b - Au^{(0)}$ $p^{(0)} = r^{(0)}$

1: **while** A **do**lgún criterio de detención
2: $r^{(k)} = b - Au^{(k)}$
3: $\beta^{(k-1)} = -\frac{\langle Ap^{(k-1)}, r^{(k)} \rangle}{\langle Ap^{(k-1)}, p^{(k-1)} \rangle}$
4: $p^{(k)} = r^{(k)} + \beta^{(k-1)} p^{(k-1)}$
5: $\alpha^{(k)} = \frac{\langle r^{(k)}, p^{(k)} \rangle}{\langle Ap^{(k)}, p^{(k)} \rangle}$
6: $u^{(k+1)} = u^{(k)} + \alpha^{(k)} p^{(k)}$
7: **end while**

**Algorithm 8** Gradiente Conjugado I

**Entradas:** Vector inicial $u^{(0)}$, $r^{(0)} = b - Au^{(0)}$ $p^{(0)} = r^{(0)}$

1: **while** A **do**lgún criterio de detención
2: $\beta^{(k-1)} = -\frac{\langle r^{(k)}, r^{(k)} \rangle}{\langle r^{(k-1)}, r^{(k-1)} \rangle}$
3: $p^{(k)} = r^{(k)} + \beta^{(k-1)} p^{(k-1)}$
4: $\alpha^{(k+1)} = \frac{\langle r^{(k)}, p^{(k)} \rangle}{\langle Ap^{(k)}, p^{(k)} \rangle}$
5: $u^{(k+1)} = u^{(k)} + \alpha^{(k)} p^{(k)}$
6: $r^{(k+1)} = r^{(k)} - \alpha^{(k)} Ap^{(k)}$
7: **end while**

El método se ejecuta con muy poca memoria adicional. La base del algoritmo se encuentra en la multiplicación matriz-vector.

En cada iteración del algoritmo básico, que prescinde de precondicionadores y condiciones de convergencia adicionales, es necesario realizar:

- Un producto matriz-vector.
- Pocas actualizaciones vectoriales.
- Pocos productos escalares entre vectores.

En casos dispersos, la técnica SPMV (Sparse Matrix-Vector Multiplication) es esencial.

La velocidad de convergencia en métodos iterativos como este es difícil de predecir con exactitud. Aunque existen cotas aproximadas, la convergencia depende en gran medida de las propiedades espectrales de la matriz involucrada. Un mal condicionamiento de la matriz puede ralentizar significativamente el proceso de convergencia.

Se verifica que:

$$\|u^{(i)} - x\| \leq 2 \left( \frac{\sqrt{\kappa_2} - 1}{\sqrt{\kappa_2} + 1} \right)^i \|u^{(0)} - x\| \tag{11.26}$$

donde:

- $u^{(i)}$ es la solución obtenida para la iteración $i$-ésima.
- $x$ es la solución exacta del sistema.
- $\kappa_2$ es el número de condición.

$$\kappa_2 = \frac{\lambda_{\text{máx}}(A)}{\lambda_{\text{mín}}(A)} \tag{11.27}$$

**Precondicionadores**

Los precondicionadores son matrices invertibles que se introducen en un sistema lineal para obtener una solución equivalente a la del sistema original, pero de manera más eficiente. Su aplicación proporciona una ventaja sustancial al presentar propiedades espectrales favorables en la nueva matriz de coeficientes, lo que conduce a una convergencia más rápida, como se evidencia en el método del Gradiente Conjugado Precondicionado (PCG). La matriz $M$, que será el precondicionador, se introduce en el problema $Ax = b$ de la siguiente manera:

$$M^{-1}Ax = M^{-1}b \tag{11.28}$$

La inclusión de un precondicionador implica resolver un sistema lineal $Mx = y$ en cada iteración del método.

La elección adecuada de $M$ es crucial, ya que se busca que $M^{-1}$, se asemeje de alguna manera a la inversa de la matriz original $A$ ($A^{-1}$), pero que el sistema $Mx = y$ sea más fácil de calcular directamente. Aunque el uso de precondicionadores suele reducir la cantidad de iteraciones necesarias para la convergencia, esto puede ir acompañado de un aumento en el costo computacional de cada iteración.

Existen diversos tipos de precondicionadores, cuyo rendimiento depende del tipo de problema abordado. Entre los precondicionadores más comunes se encuentran:

- **Jacobi, Gauss-Seidel, SOR (Relajación Sobrerrelajada), y SSOR (Relajación Sobrerrelajada Simétrica).** Las matrices de estos métodos pueden ser utilizadas como precondicionadores de otros métodos. Aunque son simples de calcular, pueden tener un rendimiento variable dependiendo del problema específico.
- **Factorizaciones incompletas.** Otro enfoque de precondicionamiento incluye las Factorizaciones Incompletas, como Incomplete LU (ILU). Estas consisten en factorizar la matriz $A$, permitiendo solo cierto grado de llenado. La variante ILU0, por ejemplo, produce una factorización con el mismo patrón de dispersión que $A$.
- **Inversa aproximada.** El precondicionador es una aproximación de la inversa de $A$, por lo que no es necesario resolver un sistema en cada iteración

### 11.1.5. Generalized Minimal Residual (GMRES)

El método de Residuos Mínimos Generalizados (GMRES, por sus siglas en inglés) es ampliamente empleado en la resolución de sistemas lineales. Su enfoque se basa en la construcción explícita de una base ortogonal para el subespacio de Krylov generado por la matriz $A$ y el residuo inicial $r_0$ ($K(A, r_0)$), utilizando el proceso de ortogonalización de Grahm-Schmidt, también conocido como el método de Arnoldi. En cada paso del método GMRES, el iterando $x^{(i)}$ se obtiene como una combinación lineal de esta base, buscando minimizar la norma del residuo $\|b - Ax^{(i)}\|$.

Aunque GMRES es eficaz para resolver sistemas lineales, presenta desafíos en términos de memoria, ya que requiere almacenar la base ortogonal, y el costo computacional aumenta con cada iteración. Para mitigar este problema, se suele emplear una variante conocida como Restarted GMRES o GMRES($m$), donde $m$ representa el número de iteraciones antes de reiniciar el método. Esta estrategia implica reiniciar el proceso después de cierto número predefinido de pasos, utilizando el iterando calculado hasta el momento como solución inicial para la siguiente fase del método. Este enfoque ayuda a controlar los requisitos de memoria y a gestionar eficientemente el crecimiento del costo computacional a lo largo de las iteraciones.

### 11.1.6. Método del Gradiente Bi-Conjugado (BiCG)

El método del Gradiente Bi-conjugado (BiCG) se caracteriza por su capacidad de operar eficientemente sin requerir que la matriz del sistema lineal sea simétrica. Su funcionamiento se basa en la sustitución de las secuencias ortogonales de residuos por dos secuencias que son mutuamente ortogonales.

En el proceso de actualización de las relaciones para los residuos en el método BiCG, se introducen relaciones aumentadas por términos basados en la matriz transpuesta $A^T$, en contraste con la matriz original $A$.

Una propiedad destacada del Gradiente Bi-conjugado es su capacidad para proporcionar resultados idénticos a los del método del Gradiente Conjugado (GC) cuando se trata de matrices simétricas y definidas positivas.

**Algorithm 9** Gradiente Bi-Conjugado

**Entradas:** Vector inicial $x^{(0)}$, $r^{(0)} = b - Ax^{(0)}$ $r'^{(0)} = r^{(0)}$ $p^{(0)} = p'^{(0)}$ $\rho^{(0)} = 0$

1: **while** A **do**lgún criterio de detención
2: $\beta^{(k+1)} = \frac{\rho^{(k+1)}}{\rho^{(k)}}$
3: $p^{(k+1)} = r^{(k)} + \beta^{(k+1)} p^{(k)}$
4: $p'^{(k+1)} = r'^{(k)} + \beta^{(k+1)} p'^{(k)}$
5: $r^{(k+1)} = r^{(k)} + \alpha^{(k+1)} A p^{(k+1)}$
6: $r'^{(k+1)} = r'^{(k)} + \alpha^{(k+1)} A^T p'^{(k+1)}$
7: $\rho^{(k+1)} = \langle r'^{(k)}, r^{(k)} \rangle$
8: $\alpha^{(k+1)} = \frac{\rho^{(k+1)}}{\langle p'^{(k+1)}, A p^{(k+1)} \rangle}$
9: $x^{(k+1)} = x^{(k)} + \alpha^{(k+1)} p^{(k+1)}$
10: **end while**

### 11.1.7. Método del Gradiente Conjugado Cuadrático

El Gradiente Conjugado Cuadrático (CGS) se presenta como una variante del método del Gradiente Bi-conjugado (BiCG) que comparte similitudes en la cantidad de operaciones, aunque se diferencia en que no requiere el uso de la matriz transpuesta $A^T$. Esta característica lo distingue al simplificar el proceso computacional en comparación con su contraparte, el BiCG.

Aunque el CGS mantiene una eficiencia computacional similar al BiCG en términos de la cantidad de operaciones necesarias, en la práctica, la convergencia del CGS puede mostrar una irregularidad significativa.

**Algorithm 10** Gradiente Conjugado Cuadrático

**Entradas:** Vector inicial $x^{(0)}$, $r^{(0)} = b - Ax^{(0)}$ $r'^{(0)} = r^{(0)}$ $p^{(0)} = q^{(0)}$ $\rho^{(0)} = 0$

1: **while** A **do**lgún criterio de detención
2: $\rho^{(k+1)} = \langle r'^{(0)}, r^{(k)} \rangle$
3: $\beta^{(k+1)} = \frac{\rho^{(k+1)}}{\rho^{(k)}}$
4: $u^{(k)} = r^{(k)} + \beta^{(k+1)} q^{(k)}$
5: $p^{(k+1)} = u^{(k)} + \beta^{(k+1)} \left( q^{(k)} + \beta^{(k+1)} p^{(k)} \right)$
6: $\alpha^{(k+1)} = \frac{\rho^{(k+1)}}{\langle p^{(0)}, A p^{(k+1)} \rangle}$
7: $q^{(k+1)} = u^{(k)} + \alpha^{(k+1)} p^{(k+1)}$
8: $x^{(k+1)} = x^{(k)} + \alpha^{(k+1)} \left( u^{(k)} + q^{(k+1)} \right)$
9: $r^{(k+1)} = x^{(k)} + \alpha^{(k+1)} \left( u^{(k)} + q^{(k+1)} \right)$
10: **end while**

## 11.2. Método del Gradiente Bi-Conjugado Estabilizado (BiCGSTAB)

El método del Gradiente Bi-conjugado Estabilizado (BiCGSTAB) resuelve sistemas lineales no simétricos. Este método aborda la posible irregularidad en la convergencia, una característica presente en el Gradiente Conjugado Cuadrático (CGS), mediante la modificación del polinomio utilizado para el cálculo del residuo.

A pesar de estas mejoras, el BiCGSTAB requiere dos productos internos adicionales en comparación con el Gradiente Bi-conjugado (BiCG) y el Gradiente Conjugado Cuadrático (CGS). Esta consideración debe tenerse en cuenta al evaluar el rendimiento y la eficiencia computacional del BiCGSTAB en comparación con otros métodos de resolución de sistemas lineales.

**Algorithm 11** Gradiente Bi-Conjugado Estabilizado

**Entradas:** Vector inicial $x^{(0)}$, $r^{(0)} = b - Ax^{(0)}$ $r'^{(0)} = r^{(0)}$ $p^{(0)} = q^{(0)} = 0$ $\rho^{(0)} = \alpha^{(1)} = \omega^{(0)} = 1$

1: **while** A **do**lgún criterio de detención
2: $\rho^{(k+1)} = \langle r'^{(0)}, r^{(k)} \rangle$
3: $\beta^{(k+1)} = \frac{\rho^{(k+1)}}{\rho^{(k)}} \frac{\alpha^{(k+1)}}{\omega^{(k)}}$
4: $p^{(k+)} = r^{(k)} + \beta^{(k+)} \left( p^{(k)} - \omega^{(k)} q^{(k)} \right)$
5: $q^{(k+1)} = Ap^{(k+1)}$
6: $\alpha^{(k+1)} = \frac{\rho^{(k+1)}}{\langle r'^{(0)}, q^{(k+1)} \rangle}$
7: $s^{(k+1)} = r^{(k)} - \alpha^{(k+1)} q^{(k+1)}$
8: $t^{(k+1)} = As^{(k+1)}$
9: $\omega^{(k+1)} = \frac{\langle t^{(k+1)}, s^{(k+1)} \rangle}{\langle t^{(k+1)}, t^{(k+1)} \rangle}$
10: $x^{(k+1)} = x^{(k)} + \alpha^{(k+1)} p^{(k+1)} + \omega^{(k+1)} s^{(k+1)}$
11: $r^{(k+1)} = s^{(k+1)} + \omega^{(k+1)} t^{(k+1)}$
12: **end while**

# Capítulo 12

# Precondicionadores

La velocidad de convergencia de los métodos iterativos está fuertemente vinculada al número de condición del sistema. En este contexto, el precondicionamiento surge como una estrategia clave para mejorar la eficiencia de estos métodos al reducir el número de iteraciones necesarias para lograr la convergencia.

La esencia del precondicionamiento radica en transformar el sistema lineal original $Ax = b$ en un sistema equivalente $A'x = b$, de manera que resolver $A'x = b$ no incremente significativamente la carga computacional necesaria en comparación con resolver $Ax = b$. Es esencial que ambos sistemas tengan la misma solución, es decir, $A^{-1}x = A'^{-1}x$

Para lograr esta transformación, se introduce una matriz fácilmente invertible llamada precondicionador, denotada como $M$, que se pre-multiplica o post-multiplica al sistema original.

La utilización del precondicionador implica la modificación de $Ax = b$ en un sistema equivalente con condiciones espectrales más favorables, lo que, a su vez, reduce el número de iteraciones requeridas para alcanzar la convergencia.

Se pueden distinguir dos grandes grupos de precondicionadores:

- **Implícitos:** Se construye la matriz $M$ y se resuelve $Mx = b$
- **Explícitos:** Se construye la matriz $M^{-1}$.

## 12.1. Precondicionadores implícitos

### 12.1.1. Método de Richardson Precondicionado

El método de Richardson precondicionado es:

$$x^{(k)} = x^{(k-1)} - M^{-1}\left(Ax^{(k-1)-b}\right) \tag{12.1}$$

El mismo surge de considerar el sistema precondicionado por la izquierda, $M^{-1}Ax = M^{-1}b$ y agregar la suma de $Nx$:

$$Nx^{(k)} = Nx^{(k-1)} - M^{-1}\left(Ax^{(k-1)} - b\right) \tag{12.2}$$

Tomando $N = \mathbb{I}$ se obtiene el método de Richardson precondicionado.

### 12.1.2. Precondicionador de Jacobi

Como se tenía en la Ecuación 10.13 el método de Jacobi es:

$$x^{(k)} = D^{-1}\left(-L - U\right)x^{(k-1)} + D^{-1}b \tag{12.3}$$

Por esto a la matriz $M = D$ se la llama precondicionador diagonal o de Jacobi. Para que pueda estar definido los valores de la diagonal deben ser distintos de cero.

La ventaja de este método es que no requiere de espacio extra de almacenamiento, además de que es fácil de implementar y paralelizar. A pesar de esto hay métodos que logran mejores resultados.

### 12.1.3. Método de Gauss-Seidel y SOR

Si se expresa la matriz $A$ en la forma $A = D + L + U$, donde $D$ es la matriz diagonal de $A$, $L$ es la parte estrictamente triangular inferior de $A$, y $U$ es la parte estrictamente triangular superior de $A$, se pueden introducir dos métodos iterativos precondicionados adicionales:

- **Método de Gauss-Seidel (GS):** Se define el precondicionador como $M = D + L$. En este caso, la iteración se conoce como el método de Gauss-Seidel.
- **Método de SOR (Successive Overrelaxation):** Se deifne el precondicionador como $M = \omega^{(} - 1)D + L$, donde $\omega$ es un parámetro en el intervalo $(0, 2)$.

### 12.1.4. Método del Gradiente jugado Precondicionado

El Método del Gradiente Conjugado Precondicionado (PCG) fue desarrollado por Saad, quien dedujo el algoritmo al emplear un sistema precondicionado por la izquierda y algunas ideas clave.

Primero, el concepto de $M$-producto interno, definido como:

$$(x, y)_M = (Mx, y)_2 = (x, My)_2 \tag{12.4}$$

donde $(\cdot, \cdot)_2$ representa el producto interno euclidiano convencional.

Además, se señaló que la matriz $M^{(} - 1)A$ es autoadjunta con respecto al $M$-producto interno:

$$\left(M^{-1}Ax, y\right)_M = (Ax, y)_2 = (x, Ay)_2 = \left(x, M^{-1}Ay\right)_M . \tag{12.5}$$

Como se vio en secciones anteriores, la función

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b \tag{12.6}$$

alcanza un único mínimo en el vector solución del sistema $Ax = b$. Esta expresión se puede reescribir de manera alternativa como:

$$\phi(x) = \frac{1}{2}\left(r, A^{-1}r\right)_2 \tag{12.7}$$

Ahora, al sustituir $r = b - Ax$ por $z = M^{-1}r$, $A$ por $M^{-1}A$, y el producto interno euclidiano por el $M$-producto interno, se llega a la conclusión de que el método PCG minimiza la función

$$\phi(x) = \frac{1}{2}\left(z, (M{-}1A)^{-1}\, z\right)_M \tag{12.8}$$

en el subespacio de Krylov $\left\{r_0, M^{-1}Ar_0, ..., \left(M^{-1}A\right)^{k-1} r_0\right\}$. Es crucial destacar que el precondicionador $M$ debe ser semi definido positivo (SDP) para garantizar la existencia del $M$-producto interior.

**Algorithm 12** Método del GC precondicionado

**Entradas:** Aproximación inicial $x_0$, $r_o = b - Ax_0$

1: Resolver $Mz_0 = r_o \quad p_0 = z_0$
2: **while do** $\frac{\|r_j\|}{\|r_0\|} \geq e\,(j = 0, 1, 2, \cdots)$
3: $\quad \alpha_j = \frac{\langle r_j, z\rangle}{\langle Ap_j, p_j\rangle}$
4: $\quad x_{j+1} = x_j + \alpha_j p_j$
5: $\quad$ Resolver $Mz_{j+1} = r_{j+1}$
6: $\quad \beta_j = \frac{\langle r_{j+1}, z_{j+1}\rangle}{\langle r_j, z_j\rangle}$
7: $\quad p_{j+1} = z_{j+1} + \beta_j p_j$
8: **end while**

### 12.1.5. Factorizaciones Incompletas

Una clase importante de precondicionadores se obtiene a partir de factorizaciones $LU$, aplicable a matrices arbitrarias. Sin embargo, se han creado variantes que explotan la estructura especial que pueda presentar la matriz, ya sea simétrica, definida positiva, dispersa, entre otras características.

En el contexto particular de **matrices dispersas**, se emplea un método conocido como factorización incompleta. La premisa esencial consiste en obtener matrices $M = LU$ de manera que la matriz original $A$ pueda ser descompuesta en la suma $A = LU + R$, siendo $R$ una matriz de residuo. Esto facilita la resolución del sistema $Mx = y$ en comparación con el sistema original.

Es importante destacar que si $A$ es una matriz dispersa, los factores $L$ y $U$ usualmente no mantienen el mismo patrón de dispersión que la matriz $A$ debido al fenómeno de fill-in ya visto en secciones anteriores. Para asegurar que el sistema $LUx = y$ sea manejable, se busca controlar el llenado de los factores $L$ y $U$. Esta estrategia implica descartar los elementos de llenado (fill-elements) que surgen durante el proceso de factorización, siguiendo ciertos criterios predefinidos.

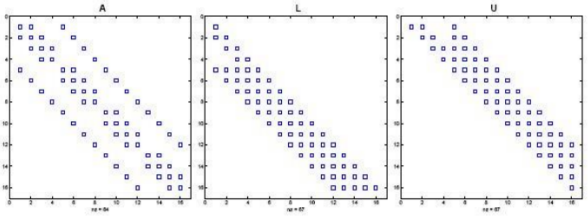



**Figura 12.1**: Se puede ver que en L y U existen "diagonales extra" que no están en la parte triangular superior o inferior de la matriz A. A tales elementos de estas "diagonales extra" se les denomina elementos de llenado.

Existen tres formas fundamentales de factorizaciones incompletas:

1. Factorizaciones sin llenado, denotadas como $ILU(0)$.

2. Factorizaciones con llenado, donde se introduce el llenado basándose en la posición dentro de la matriz, conocidas como $ILU(k)$.

3. Factorizaciones con llenado, que emplean umbrales numéricos como criterio para introducir el llenado, denominadas $ILU(t)$.

**Factorizaciones incompletas** $ILU(0)$

Es la más sencilla y no introduce llenado alguno, es decir, la factorización incompleta tiene la misma cantidad de elementos no nulos y en las mismas posiciones que en la matriz $A$. Esto significa que en $L$ y $U$ sólo se mantendrán aquellos valores tales que en la misma posición $A$ tiene un valor no nulo.

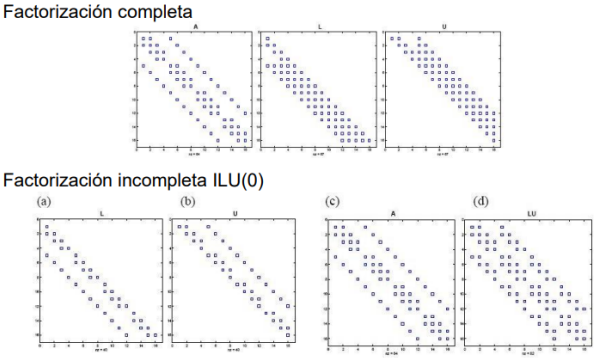



**Figura 12.2**: En este caso se puede ver que en $L$ y en $U$ se respetan las posiciones de las entradas no nulas de $A$

**Factorizaciones incompletas** $ILU(k)$

En estos casos, la variable $k$ representa el nivel de llenado permitido. Por ejemplo, el nivel-0 es equivalente a la factorización incompleta sin llenado, $ILU(0)$, lo que significa que conserva el patrón de dispersión de la matriz original. El nivel-1 de llenado incluye el nivel-0 y los elementos no nulos introducidos por la eliminación de elementos pertenecientes al nivel-0. El nivel-2 de llenado incluye el nivel-1 y cualquier elemento no nulo producido por la eliminación de elementos de llenado del nivel-1.

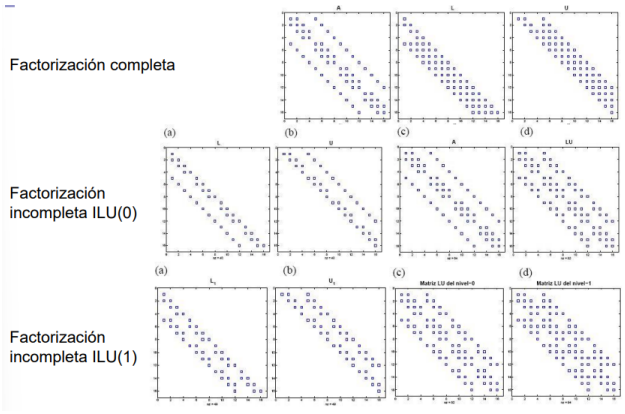



**Figura 12.3**: Se puede ver que por ejemplo las matrices $L$ y $U$ de la factorización $ILU(1)$ respetan la estructura de no nulos de la matriz $LU$ de la factorización $ILU(0)$.

En general, las factorizaciones de tipo $ILU(k)$ adolecen del defecto de considerar que la importancia numérica de un llenado $l_{ij}$, depende únicamente de la proximidad topológica entre los nodos $i$ y $j$, sin tener en cuenta el fenómeno físico que la matriz $A$ representa.

**Factorizaciones** $ILU(t)$

Las factorizaciones de este tipo deciden introducir o no un llenado $l_{ij}$ en función de si es superior o inferior a un umbral determinado. Dicho umbral se calcula relativo al valor de los elementos de la fila $i$-ésima de $A$ usando el parámetro $t$, por ejemplo, $t$ multiplicado por la norma original de la fila $i$-ésima. Se descartan los elementos de llenado de magnitud inferior al umbral.

Estas factorizaciones son de aplicación general, pero adolecen de una falta de control fino sobre la cantidad total de llenado que se permite. Esta falta de control genera problemas de dimensionamiento del espacio de memoria reservado al precondicionador, y sobre todo, impide un buen compromiso entre complejidad de cálculo del precondicionador y la aceleración que el precondicionador introduce en el método iterativo.

**Factorizaciones** $ILU(p,t)$

Es una variante muy utilizada para mitigar los problemas de la $ILU(t)$. Utiliza un criterio de descarte dual: primero aplica el criterio del umbral y luego se queda con los $p$ elementos de mayor magnitud en la fila que cumplen con el criterio.

## 12.2. Precondicionadores Explícitos

### 12.2.1. Precondicionadores Polinomiales

Los precondicionadores polinomiales se basan en aproximar la matriz $A^{-1}$ mediante un polinomio de grado $m$ de la matriz $A$.

**Ejemplo 12.2.1.1.** *Si se escribe $A = \mathbb{I} - B$ y el radio espectral de $B$ es menor a 1, utilizando las series de Neumann se puede escribir:*

$$A^{-1} = \sum_{k=0}^{\infty} B^k \tag{12.9}$$

*Por lo que al truncar esta serie se puede obtener una aproximación de la inversa de $A$.*

De forma más abstracta se define un polinomio de grado $n$ de la matriz $A$:

$$M^{-1} = P_n(A) \tag{12.10}$$

La selección del mejor precondicionador polinómico surge de seleccionar el poinomio que minimiza:

$$\|\mathbb{I} - M^{-1}A\| \tag{12.11}$$

**Ejemplo 12.2.1.2.** *Para la norma infinito se utilizan los polinomios de Chebyshev, requieren estimar límites superiores e inferiores del espectro de $A$, lo cual siempre es fácil.*

Los precondicionadores polinomiales, aunque requieren únicamente productos matriz-vector con $A$ y ofrecen un excelente potencial de paralelización, no son tan efectivos como los métodos de factorización incompleta para reducir el número de iteraciones.

Con estos precondicionadores, la disminución en el número de iteraciones suele compensarse con el costo de los productos matriz-vector que se realizan en cada iteración. De manera más precisa, Axelsson demostró que el costo por iteración aumenta linealmente con el número $k+1$ de términos en el polinomio, mientras que la disminución en el número de iteraciones es más lenta que $\mathcal{O}(\frac{1}{k}+1)$

Por lo tanto, los precondicionadores polinomiales pueden no ser tan efectivos, especialmente en sistemas en serie o con memoria compartida. Sin embargo, para sistemas en paralelo con memoria distribuida, estos precondicionadores son un poco más atractivos debido al menor número de productos escalares requeridos.

## 12.3. Inversas aproximadas

La idea básica de estos precondicionadores radica en encontrar una matriz $M$ que haga que $AM$ se asemeje lo más posible a la identidad. Una aproximación común a este problema es la siguiente: se construye una matriz inversa aproximada utilizando el producto escalar de Frobenius.

Sea $S \subset M_n$ el subespacio de las matrices $M$ en el que se busca una inversa aproximada explícita. La formulación del problema es encontrar $M_0 \in S$ tal que:

$$M_0 = \min_{M \in S} \|AM - \mathbb{I}\|_F \tag{12.12}$$

donde $\mathbb{I}$ es la matriz identidad, $A$ es una amtriz no simétrica y $\|\cdot\|_F$ denota la norma de Frobenius, que se ddefine como:

$$\|A\|_F^2 = \langle A, A \rangle \tag{12.13}$$

con $\langle A, B \rangle_F = tr\left(AB^T\right)$

Esta norma tiene la propiedad de que:

$$\|AM - \mathbb{I}\|_F^2 = \sum_{k=1}^{N} \|Am_k - e_k\|_2^2 \tag{12.14}$$

con $m_k$ y $e_k$ las $k$-ésimas columnas de $M$ e $\mathbb{I}$ respectivamente.

Así, se reduce el problema a resolver de manera independiente $N$ problemas de mínimos cuadrados para encontrar la solución de 12.12. De esta manera, las columnas de la inversa aproximada $M$ pueden calcularse y aplicarse en paralelo.

Para abordar eficientemente 12.14, es crucial observar que la inversa de $A$ generalmente es mucho más densa que $A$, pero los coeficientes significativos de $A^{-1}$ son solo unos pocos. Esto implica que se espera que $M$ sea una matriz dispersa. Por lo tanto, el problema 12.14 es de hecho de una dimensión muy pequeña y puede resolverse de manera eficiente.

# Capítulo 13

# Valores y Vectores propios

Los vectores propios de una matriz $A$, como se vió en la Subsección 1.2.5, son vectores $v \neq 0$ tales que $Av = \lambda v$ donde $\lambda$ es un escalar llamado valor propio de $A$. En otras palabras, los vectores propios de $A$ son vectores que no cambian de dirección al ser transformados por $A$, solo cambian su magnitud y su sentido. El valor propios asociado a un vector propio controla dicho cambio.

Como se vio en la Subsección 1.2.5 una primera forma de cálculo de los valores propios es la resolución del polinomio:

$$det(A - \lambda_j \mathbb{I}) = 0 \tag{13.1}$$

Las raíces de un polinomio de alto grado son muy sensibles a las perturbaciones en sus coeficientes, por lo que se vuelve un problema mal condicionado.

## 13.1. Resolución de la ecuación característica

### 13.1.1. Método de las potencias

Este método es aplicable cuando se requiere de unos pocos valores, o vectores, propios, encontrando el de mayor magnitud.

Aquí se asume que los valores propios cumplen que existe uno estrictamente dominante, es decir:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n| \tag{13.2}$$

Sea $\{x_1, x_2, \cdots, x_n\}$ una base de vectores propios, entonces se definen:

$$z_0 = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n = \sum_{j=1}^{n} \alpha_j x_j \tag{13.3}$$

$$z_1 = Az_0 = A\alpha_1 x_1 + \cdots + A\alpha_n x_n = \alpha_1 \lambda_1 x_1 + \cdots + \alpha_n \lambda_n x_n = \sum_{j=1}^{n} \alpha_j \lambda_j x_j \tag{13.4}$$

$$z_2 = Az_1 = A^2 z_0 = \alpha_1 \lambda_1^2 x_1 + \cdots + \alpha_n \lambda_n^2 x_n = \sum_{j=1}^{n} \alpha_j \lambda_j^2 x_j \tag{13.5}$$

$$\vdots \tag{13.6}$$

$$z_k = Az_k - 1 = A^k z_0 = \alpha_1 \lambda_1^k x_1 + \cdots + \alpha_n \lambda_n^k x_n = \sum_{j=1}^{n} \alpha_j \lambda_j^k x_j \tag{13.7}$$

Nótese que:

$$z_k = \lambda_1^k \sum_{j=1}^{n} \alpha_j \frac{\lambda_j^k}{\lambda_1^k} x_j = \lambda_1^k \left( \alpha_1 x_1 + \sum_{j=2}^{n} \alpha_j \frac{\lambda_j^k}{\lambda_1^k} x_j \right) \approx \lambda_1^k \alpha_1 x_1 \tag{13.8}$$

Es decir que el vector $z_k$ tiende a la dirección del vector propio $x_1$, mientras que el cociente de Rayleigh:

$$\sigma_k = \frac{z_k^T A z_k}{z_k^T z_k} \to \lambda_1 \tag{13.9}$$

**Algorithm 13** Método de las potencias

1: Se selecciona un vector inicial $z_0$
2: $y_0 = \frac{z_0}{\|z_0\|}$
3: **while** ! **do**(Finalizar)
4: $\quad z_{i+1} = Ay_i$
5: $\quad \sigma_k = \frac{y_i^t A y_i}{y_i^t y_i} = y_i^t z_{i+1}$
6: $\quad y_{i+1} = \frac{z_{i+1}}{\|z_{i+1}\|}$
7: **end while**

Es así entonces que el método de las potencias es una técnica fundamental para calcular los valores y vectores propios dominantes de una matriz. Algunas observaciones importantes sobre este método son las siguientes:

- El criterio de finalización del algoritmo puede estar basado en el valor propio o del vector propio.
- No es necesario almacenar explícitamente la potencia de la matriz original durante el proceso iterativo.

- La velocidad de convergencia del método está directamente relacionada con el cociente entre los valores propios dominantes sucesivos: $\frac{|\lambda_2|}{|\lambda_1|}$. Cuando este cociente se aproxima a uno, el progreso del algoritmo es más lento.

- Teóricamente, el método de las potencias puede fallar si $\alpha_1 = 0$. Sin embargo, en la práctica, los errores de redondeo pueden tener un efecto favorable en estos casos.

- El número de operaciones requeridas en cada iteración del algoritmo depende del producto matriz-vector $Ay_i$. En el caso de matrices llenas, esto implica un costo computacional de orden cuadrático, es decir, $\mathcal{O}(n^2)$, donde $n$ es el tamaño de la matriz.

### 13.1.2. Iteración inversa

Este método es útil cuando se busca el valor propio con módulo más pequeño puede utilizarse la misma idea anterior pero con la matriz inversa.

Es importante recordar que los valores propios de la inversa de una matriz son los inversos de los valores propios de dicha matriz.

**Algorithm 14** Iteración inversa

1: Se selecciona un vector inicial $z_0$
2: $y_0 = \frac{z_0}{\|z_0\|}$
3: **while** ! **do**(Finalizar)
4: $\quad Az_{i+1} = y_i$
5: $\quad \sigma_k = y_i^T z_{i+1}$
6: $\quad y_{i+1} = \frac{z_{i+1}}{\|z_{i+1}\|}$
7: **end while**

Algunas observaciones relevantes sobre este método son las siguientes:

- En cada iteración, se requiere resolver un sistema lineal. Este sistema lineal puede ser resuelto utilizando técnicas como la descomposición $LU$, que pueden ser eficientes en términos computacionales.

- La velocidad de convergencia del método está determinada por el coeficiente $\frac{\lambda_n}{\lambda_{n-1}}$.

- Si se desea calcular un valor propio cercano a un valor específico, $\lambda^*$, se pueden ajustar los parámetros del algoritmo para dirigir la convergencia hacia ese valor particular. Esto puede ser útil en situaciones donde se busca una solución específica o se necesita una precisión particular en el cálculo de los valores propios.

**Algorithm 15** Iteración inversa con desplazamiento

1: Se selecciona un vector inicial $z_0$
2: $y_0 = \frac{z_0}{\|z_0\|}$
3: **while** ! **do**(Finalizar)
4: $(A - \lambda^* \mathbb{I})z_{i+1} = y_i$
5: $\sigma_k = y_i^T z_{i+1}$
6: $y_{i+1} = \frac{z_{i+1}}{\|z_{i+1}\|}$
7: **end while**

### 13.1.3. Deflación

Una vez encontrado el mayor valor propio, para encontrar los demás se pueden utilizar técnicas denominadas deflación. Estas consisten en hallar otra matriz $B$ que tenga los mismos valores propios de $A$, excepto el ya conocido.

Sea $\lambda_1$ el valor propio con el vector $v_1$ asociado y un vector $x$ tal que $x^T v_1 = 1$ y $B = A - \lambda_1 v_1 x^T$. EL primer paso es comprobar que 0 es valor propio de $B$ con vector propio $v_1$:

$$Bv_1 = Av_1 - \lambda_1 v_1 x^T v_1 = (\lambda_1 - \lambda_1)v_1 = 0 \tag{13.10}$$

Luego se comprueba que los demás valores propios de $A$ son valores propios de $B$. Para esto se utilizarán los siguientes resultados sin demostrarlos:

- Los valores propios de la matriz $A^T$ son iguales a los de $A$.
- Los vectores propios de $A$ y $A^T$ que corresponden a distintos valores propios son ortogonales entre sí.

Sea $w_i$ el vector propio asociado al valor propio $\lambda_i$ de $A^T$. Si se traspone y se multiplica la ecuación original por $w_i$ se obtiene:

$$B^T w_i = A^T w_i - \lambda_1 x v_1^T w_i \tag{13.11}$$

$$v_1^T w_i = 0 \implies B^T w_i = \lambda_i w_i \tag{13.12}$$

Entonces los valores propios $\lambda_i$, $i \neq 1$, de $A/A^T$ son también valores propios de $B/B^T$.

## 13.2. Transformaciones de semejanza

Como ya se vio, las transformaciones de semejanza, realizadas a través de una matriz $P$ no singular, conservan los valores propios de la matriz original. Esta propiedad es fundamental en álgebra lineal y análisis numérico, ya que permite trabajar con matrices equivalentes que comparten las mismas características fundamentales. La búsqueda de una matriz semejante surge de la

necesidad de simplificar la tarea de encontrar los valores propios, lo que puede ser especialmente útil en situaciones donde la matriz original presenta una estructura complicada.

Para facilitar el proceso y evitar posibles problemas numéricos, es común utilizar matrices $P$ que sean ortonomales.

En muchos casos, se emplean transformaciones de semejanza para llevar la matriz original a una forma más conveniente, como la forma de Hessenberg. La matriz de Hessenberg es una forma particularmente útil en el estudio de problemas numéricos y análisis de matrices, ya que presenta una estructura casi triangular con ceros por debajo de la primera subdiagonal.

Uno de los métodos más utilizados es el Método de Givens, que se basa en rotaciones para manipular las matrices. Estas rotaciones son especialmente útiles para introducir ceros en posiciones específicas de la matriz, lo que puede simplificar significativamente los cálculos subsiguientes.

Otro enfoque importante es el uso de reflexiones, implementadas a través de matrices de Householder. Estas reflexiones también son eficaces para generar ceros en la matriz, contribuyendo así a su simplificación. Es importante observar que, para un vector dado, siempre es posible encontrar una transformación de Householder $H$ que lo convierta en un múltiplo de un vector específico, en particular, el primer vector canónico $e = (1, 0, \cdots, 0)$. Matemáticamente, esta transformación se expresa como $Hy = ke$, donde $k$ es un escalar y $e$ es el vector canónico mencionado.

## 13.3. Iteración QR

Toda matriz real puede descomponerse en forma única en un producto $A = QR$ donde $Q$ es ortogonal (o unitaria) y $R$ es triangular superior.

El costo computacional dependerá de con qué tipo de matriz se esté trabajando,, por ejemplo:

- Si $A$ es una matriz densa general $n \times n$ la factorización $QR$ tiene un costo de $\mathcal{O}\left(n^3\right)$ operaciones.
- Si la matriz $A$ es Hessenberg superior la matriz $R$ puede computarse usando rotaciones de Givens en $\mathcal{O}(n)$.

La iteración $QR$ es una técnica fundamental para abordar el problema de encontrar los valores propios de una matriz. Su fundamento radica en la posibilidad de transformar cualquier matriz cuadrada $A$ de tamaño $n \times n$ en una matriz triangular superior $T$, cuyos elementos en la diagonal corresponden a los valores propios de $A$, mediante el uso de transformaciones unitarias.

Esta transformación, expresada como $U^*AU = T$, se conoce como la forma de Schur de $A$, donde $U$ es una matriz unitaria. La iteración $QR$ aprovecha la factorización $QR$ para llevar a cabo este proceso de transformación, lo que implica descomponer la matriz original $A$ en un producto de una matriz ortogonal (o unitaria) $Q$ y una matriz triangular superior $R$. Este proceso se repite iterativamente hasta que la matriz $A$ se aproxima lo suficiente a una forma triangular superior, es decir, su forma de Schur.

Versiones optimizadas de este método son ampliamente utilizadas en la práctica debido a su eficiencia y precisión en la determinación de los valores propios de una matriz.

**Algorithm 16** Iteración $QR$

1: Se define $A_1 = A$
2: **while** ( **do**Algún criterio de convergencia)
3: $[Q_k, R_k] = qr(A_k)$
4: $A_{k+1} = R_k Q_k$
5: **end while**

Se puede observar que:

$$R = Q'AyA = RQ \implies A = Q'AQ \tag{13.13}$$

Se tiene que el límite de $A_k$ es una matriz triangular superior a bloques, y si es simétrica, es diagonal a bloques.

El método de iteración de $QR$ implica calcular una factorización $QR$ en cada paso del proceso. Sin embargo, cuando se trabaja con matrices densas generales, este método tiende a volverse poco práctico debido al costo computacional asociado con la factorización $QR$ repetida. Para abordar este problema, antes de comenzar la iteración, es común reducir la matriz original $A$ a una forma llamada matriz Hessenberg. Esta transformación se realiza mediante una serie de transformaciones de similitud que conservan los valores propios de $A$. De esta manera, al reducir A a una matriz Hessenberg, se puede realizar el método de iteración de $QR$ de manera más eficiente.

### 13.3.1. Iteración QR con desplazamiento

La velocidad de convergencia puede aumentarse si se incorporan desplazamientos con una aproximación de un valor propio:

$$Q_k R_k = A_{k-1} - \sigma_k \mathbb{I} \tag{13.14}$$

$$A_k = R_k Q_k + \sigma_k \mathbb{I} \tag{13.15}$$

donde $\sigma_k$ es una aproximación de un valor propio.

### 13.3.2. Extensión del método de las potencias

La iteración $QR$ puede interpretarse como una variación del método de las potencias donde, en lugar de utilizar un solo vector, se utiliza una base ortonormal completa $X_i$:

$$X_{i+1} = AX_i \tag{13.16}$$

Se utiliza la factorización $QR$ para re-ortornormalizar la matriz que se multiplica por $A$:

$$X_i = Q_i R_i \tag{13.17}$$

$$X_{i+1} = AQ_i R_i = Q_{i+1} R_{i+1} \tag{13.18}$$

Para una matriz $A$ simétrica, el algoritmo converge a $AQ = QD$, donde $D$ es una matriz diagonal que contiene los valores propios de $A$. Por lo tanto, $Q$ también contiene una base ortonormal de vectores propios.

## 13.4. Métodos basados en subespacios de Krylov

Otra opción, además del método $QR$ que trabaja con un conjunto de columnas ortonormal, es utilizar una base del espacio de Krylov $\mathcal{K}_n(A, x_0)$ generado por una matriz $A$ y un vector inicial $x_0$, definido como:

$$\mathcal{K}_n(A, x_0) = \text{span}\{x_0, Ax_0, A^2x_0, \ldots, A^{n-1}x_0\} \tag{13.19}$$

Operando con la matriz $A$, se puede obtener una matriz de Hessenberg $H_n$ y una matriz $Q_n$ cuyas columnas forman una base ortonormal para el subespacio de Krylov $\mathcal{K}_n(A, x_0)$, tal que:

$$AQ_n = Q_n H_n + q_{n+1} h_{n+1,n} e_n^T \tag{13.20}$$

$$= Q_n \begin{bmatrix} H_n & h_{n+1,n} e_n^T \\ 0 & 0 \end{bmatrix} \equiv Q_{n+1} \overline{H}_n \tag{13.21}$$

Donde $H_n$ es una matriz de Hessenberg de dimensión $n \times n$, $q_{n+1}$ es el vector de dimensión $n+1$ y $e_n$ es el $n$-ésimo vector canónico.

La matriz $\overline{H}_n$ es una matriz de Hessenberg extendida de dimensión $(n+1) \times (n+1)$, que contiene los valores propios de $A$ dentro de sus valores propios.

### 13.4.1. Iteración de Arnoldi

Sea $Q_n$ una matriz cuyas columnas $(q_i)$ forman una base ortonormal para $\mathcal{K}_n(A, x_0)$. Entonces, igualando $AQ_n = Q_n H$, se obtiene:

$$Aq_k = h_{1k} q_1 + \cdots + h_{kk} q_k + h_{k+1k} q_k \implies h_j k = q_j^H A q_k \tag{13.22}$$

La iteración de Arnoldi calcula $Q_n$ y $H_n$ de manera iterativa, utilizando un proceso de ortogonalización (similar a Gram-Schmidt), lo cual la hace muy costosa e inestable. Luego, se calculan los valores propios de $H_n$.

**Algorithm 17** Iteración de Arnoldi

$x_1 = \frac{x_0}{\|x_0\|}$ para el vector inicial $x_0$
**for** $j = 1, 2, \cdots, m$ **do**
    $w = Ax_j$
    **for** $i = 1, 2, \cdots, j$ **do**
        $h_{ij} = w^* x_i$
        $w = w - h_{ij} x_i$
    **end for**
    $h_{j+1,j} = \|w\|_2$
    **if** $h_{j+1,j} = 0$ **then**
        **stop**
    **end if**
    $v_{j+1} = \frac{w}{h_{j+1,j}}$
**end for**

### 13.4.2. Iteración de Lanczos

Para matrices simétricas (o hermitianas), la iteración de Lanczos es una variante de Arnoldi. En este caso, $H_n$ es una matriz tridiagonal, lo que reduce significativamente los cálculos requeridos.

**Algorithm 18** Iteración de Lanczos

$r = x_0$ vector inicial
$\beta_0 = \|r\|_2$
**for** $j = 1, 2, \cdots$ **do**
    $v_j = \frac{r}{\beta_{j-1}}$
    $r = Av_j$
    $r = r - v_{j-1}\beta_{j-1}$
    $\alpha_j = v_j^* r$
    $r = r - v_j \alpha_j$
    Reortogonalizar si es necesario.
    $\beta_j = \|r\|_2$
    Calcular los valores propios aproximados de $T_j = S\Theta^{(j} S^*$
    Calcular límites para evaluar la convergencia
**end for**Calcular los vectores propios aproximados de $X = V_j S$

### 13.4.3. Reinicio Implícito

Tanto Arnoldi como Lanczos mantienen una base de vectores. En el paso $k$, hay $k$ vectores almacenados, por lo que es necesario ortogonalizar el nuevo vector contra todos los anteriores,

ocupando mucha memoria y realizando demasiadas operaciones.

Una solución es reiniciar el método periódicamente. Se establece un tamaño fijo $k$ para la base. Cada $k$ pasos, el método se reinicia con un vector inicial "mejorado".

ARPACK implementa los métodos IRAM (Implicitly Restarted Arnoldi Method) e IRLM (Implicitly Restarted Lanczos Method) que utilizan esta técnica de reinicio implícito.

## 13.5. Método Strum

Para matrices tridiagonales simétricas, el determinante se puede calcular desarrollando por la última fila:

$$\det(A - \lambda I) = D_n(\lambda) \tag{13.23}$$

$$= (a_n - \lambda)D_{n-1}(\lambda) - b_{n-1}M \tag{13.24}$$

$$= (a_n - \lambda)D_{n-1}(\lambda) - b_{n-1}^2 D_{n-2}(\lambda) \tag{13.25}$$

Donde $D_n(\lambda)$ es un polinomio en $\lambda$ de grado $n$, $a_n$ es el elemento de la diagonal principal y $b_n$ es el elemento de la subdiagonal.

Para cualquier número $\mu$ dado, se demuestra que el número de cambios de signo entre elementos sucesivos de la secuencia $D_i(\mu)$ es igual al número de valores propios de $A$ mayores que $\mu$.

Si se encuentra un intervalo $[\lambda_1, \lambda_2]$ en el que varía el número de valores propios mayores, se puede localizar el valor propio mediante, por ejemplo, el método de bisección en dicho intervalo.

# Capítulo 14

# Descomposición SVD

**Definición 14.0.0.1** (Descomposición en valores singulares)**.** *Sea $m, n$ enteros positivos y $A \in \mathbb{C}^{m\times n}$. Entonces una descomposición en valores singulares de $A$ es una factorización:*

$$A = U\Sigma V^H \tag{14.1}$$

*Tal que $U \in \mathbb{C}^{m\times m}$ y $V \in \mathbb{C}^{n\times n}$ son unitarias mientras que $\Sigma$ es diagonal de tamaño $m \times n$*

**Definición 14.0.0.2** (Valores y vectores singulares)**.** *Se definen los **valores singulares** de la matriz $A$ como los elementos no nulos de la matriz $\Sigma$ y se los denota como $\sigma_i$.*

*Mientras que a los vectores $u_1, \cdots, u_m$ y $v_1, \cdots, v_n$ que forman las columnas de $U$ y $V$ respectivamente se los denomina **vectores singulares** de $A$ por la izquierda y por la derecha.*

**Observación 14.0.0.1.** *Si $A$ tiene entradas reales en vez de ser matrices unitarias son matrices ortogonales.*

Gráficamente, las transformaciones mediante matrices unitarias (u ortogonales en el caso real) no afectan la norma de los vectores, representando rotaciones en el espacio.

En la Figura 14.1 se puede ver el efecto de aplicar una matriz M al círculo unidad. Luego se muestra cómo llegar a esa misma matriz $A$ partiendo del círculo unidad, pero a través de la descomposición SVD. La primera transformación, dada por la matriz $V^*$, rota el círculo unidad, luego al aplicar $\Sigma$ se escalan los vectores según los valores singulares contenidos en la matriz diagonal generando una elipse. La tercera transformación, dada por la matriz $U$, rota la elipse resultante a la figura final representada por $A$.

Por lo tanto, la SVD descompone cualquier transformación lineal en un producto de tres operaciones geométricas: una rotación inicial $V^*$, un escalamiento por los valores singulares en $\Sigma$, y una rotación final $U$. Esta interpretación geométrica ilustra cómo la SVD captura de manera óptima la estructura intrínseca de deformación y rotación presente en una transformación lineal.

**Teorema 14.0.0.1.** *Toda matriz $A \in \mathbb{R}^{m\times m}$ admite una descomposición en valores singulares y estos valores singulares están determinados en forma única.*

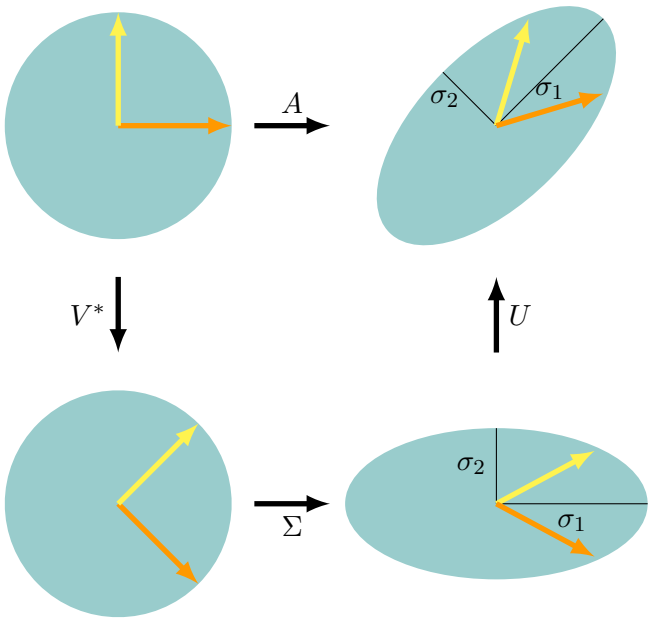



**Figura 14.1**: Descripción gráfica de la descomposición en valores singulares $A = U\Sigma V^*$

## 14.1. Propiedades de los valores singulares

- Los valores singulares distintos de cero de $A \in \mathbb{C}^{m\times n}$ son las raíces cuadradas de los valores propios distintos de cero de $A^H A$ y también los de $AA^H$ (si $A \in \mathbb{R}^{m\times n}$ entonces $A^H$ es $A^T$). Dem:

$$A = U\Sigma V^H \implies \begin{cases} A^H A = V\Sigma^H U^H U\Sigma V^H = V\Sigma^H\Sigma V^H \\ AA^H = U\Sigma V^H V\Sigma^H U^H = U\Sigma\Sigma^H U^H \end{cases} \tag{14.2}$$

- El rango de $A$ es la cantidad de valores singulares de $A$ distintos de cero.
- El valor absoluto de $\det(A) = \sigma_1 \cdot \cdots \cdot \sigma_n$ siendo $A$ cuadrada de tamaño $n$.
- Si $\sigma_n \neq 0$ entonces $\frac{\sigma_1}{\sigma_n}$ es el número de condición de la matriz.
- Si $A \in \mathbb{C}^{n\times n}$ es invertible y $\sigma_1 \geq \cdots \geq \sigma_n$ son sus valores singulares. Entonces los valores singulares de $A^{-1}$ son $\frac{1}{\sigma_n} \geq \cdots \geq \frac{1}{\sigma_1}$
- Todo elemento de la diagonal $\sigma_i$ cumple:

$$Au_i = \sigma_i v_i \quad A^T v_i = \sigma_i u_i \tag{14.3}$$

  siendo $u_i$ la columna de $U$ correspondiente a $\sigma_i$ y $v_i$ la columna de $V$ corres´pondiente a $sigma_i$.

- Si $A = U\Sigma V^H$ es una descomposición de $A$ en valores singulares y $\text{rango}(A) = r$ entonces.

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^H \tag{14.4}$$

donde $u_i, v_i$ son los vectores de las columnas $i$-ésimas de $U$ y $V$ respectivamente.

- La descomposición se puede representar como la suma de $r$ matrices de rango 1.
- El término $\sigma_i u_i v_i^H$ es conocido como tripleta singular.
- El rango de la matriz $A$ da el máximo número de tripletas necesarias.

## 14.2. Aproximación de $A$ por $A_k$

**Teorema 14.2.0.1.** *Dada la SVD de una matriz $A$ con rango $r$, siendo $p = \min(m, n)$ y $r < p$, sea $A_k$ definida como:*

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{14.5}$$

*con $k < r$. Entonces se cumple que:*

$$\|A - A_k\|_F^2 = \min_{r(B)=k} \|A - B\|_F^2 = \sigma_{k+1}^2 + \cdots + \sigma_p^2 \tag{14.6}$$

Por lo tanto, $A_k$ es la mejor aproximación de rango $k$ de la matriz $A$, para cualquier norma unitariamente invariante, como la norma de Frobenius. Mientras mayor sea $k$, mejor será la calidad de la aproximación, pero menor será la compresión lograda, y viceversa. Se desea encontrar un valor de $k$ que permita comprimir la matriz sin perder demasiada información relevante.

## 14.3. Métodos para calcular la descomposición SVD en matrices densas

### 14.3.1. Método de Golub-Kahan-Reinsch

Los métodos para calcular SVD en matrices densas incluyen el método de Golub-Kahan-Reinsch, reconocido por su eficacia basada en la iteración QR. Este método se divide en dos fases.

En la primera fase, se busca reducir la matriz A a una forma bidiagonal mediante una secuencia de transformaciones ortogonales de Householder:

$$Q_B^T A \Pi_B = B = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} \quad B_1 = \begin{pmatrix} d_1 & f_2 & 0 & \cdots & 0 \\ 0 & d_2 & f_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & f_n \\ 0 & 0 & 0 & \cdots & d_n \end{pmatrix} \tag{14.7}$$

donde $Q_B = Q_1 \cdots Q_n \in \mathbb{R}^{m \times m}$ y $\Pi_B = \Pi_1 \cdots \Pi_{n-2} \in \mathbb{R}^{n \times n}$.

En la segunda fase, una vez que la matriz A ha sido bidiagonalizada, se procede a anular los elementos que no se encuentran en la diagonal principal utilizando un algoritmo que obtenga:

$$\Sigma = Q_S^T B_1 \Pi_S = \text{diag}(\sigma_1, \cdots, \sigma_n) \tag{14.8}$$

donde $Q_S$ y $\Pi_S$ son matrices ortogonales de tamaño $n \times n$.

La descomposición en valores singulares de $A$ es:

$$A = U \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} V^T \tag{14.9}$$

donde

$$U = Q_B \text{diag}(Q_S, \mathbb{I}_{m-n}) \tag{14.10}$$

$$V = \Pi_B \Pi_S \tag{14.11}$$

Para obtener los valores singulares de la matriz bidiagonal $B_1$, se procede iterativamente buscando transformaciones ortogonales $U_k$ y $V_k$ que reduzcan los valores sobre la diagonal de $B$:

$$B_{k+1} = U_k^T B_k V_k \tag{14.12}$$

### 14.3.2. Otros métodos

Otros métodos pueden ser considerados en el cálculo de la descomposición en valores singulares (SVD). A partir de la primera propiedad presentada donde los valores singulares son iguales a las raíces cuadradas de los valores propios no nulos de $A^T A$, se pueden emplear métodos iterativos alternativos. En lugar de calcular los valores singulares directamente, se podría optar por calcular los valores propios de una nueva matriz $B = A^T A$. Sin embargo, este enfoque conlleva un costo adicional debido al cálculo de $A^T A$.

Estos métodos, aunque aprovechables, no resultan óptimos para matrices grandes o dispersas. La aplicación de transformaciones ortogonales como Householder o Givens sobre matrices dispersas conlleva un aumento en el llenado de la matriz. Además, el almacenamiento requerido por estas transformaciones es considerable. Es importante destacar que estos métodos calculan todas las tripletas, incluso cuando solo se necesitan unas pocas de las de mayor magnitud, ya sean valores singulares o valores propios. El uso de bibliotecas optimizadas como BLAS o LAPACK facilita la ejecución de estas operaciones.

## 14.4. Métodos para calcular la descomposición SVD en matrices dispersas

Algunos métodos para el cálculo de la descomposición en valores singulares (SVD) en matrices dispersas incluyen:

- SISVD: Subspace Iteration.
- TRSVD: Trace minimization.
- LASVD: Single-vector Lanczos.
- BLSVD: Block Lanczos.

Estos métodos ofrecen enfoques específicos para manejar la particularidad de matrices dispersas en el cálculo de la SVD, cada uno con sus propias ventajas y aplicaciones.

## 14.5. Aplicaciones de la descomposición SVD

### 14.5.1. Técnicas de regularización

Una aplicación común es resolver sistemas de ecuaciones mal condicionadas, conocidas como técnicas de regularización.

### 14.5.2. Cálculo de inversa

El cálculo de la inversa de una matriz $A$ (incluso si es mal condicionada) es una aplicación importante de la descomposición en valores singulares (SVD). Se puede calcular de forma rápida utilizando la descomposición SVD, expresada como:

$$A^{-1} = VS^{-1}U^T \tag{14.13}$$

Es importante recordar que las matrices $U$ y $V$ son ortogonales, y $S$ es diagonal.

### 14.5.3. Mínimos cuadrados

Una aplicación importante de la descomposición en valores singulares (SVD) es en el problema de mínimos cuadrados, donde se busca obtener $x$ que minimice $\|Ax - b\|$. Este proceso implica los siguientes pasos:

- Se calcula la SVD de $A$, expresada como $A = UDVT$.
- La solución $x$ se calcula como $x = VD^{-1}U^Tb$. Es importante notar que si $A$ es invertible, $VD^{-1}U^T = A^{-1}$.

La descomposición SVD es particularmente útil cuando la matriz $A$ es singular, proporcionando una solución robusta al problema de mínimos cuadrados.

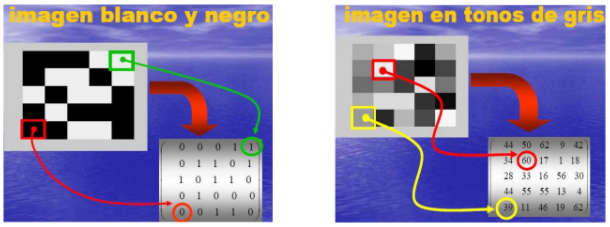



**Figura 14.2**: Imagen en escala de grises

### 14.5.4. Compresión de imágenes

La compresión de imágenes es una aplicación importante de la descomposición en valores singulares (SVD) que permite reducir el tamaño de archivos de imagen al eliminar la información redundante sin comprometer significativamente la calidad visual. Las imágenes se pueden pensar de dos formas distintas:

- Una imagen a color se representa como una matriz tridimensional $(n, m, 3)$, donde cada píxel está asociado a un vector en $\mathbb{R}^3$ que representa la composición RGB del color. Por ejemplo el vector $(1, 0, 0)$ representa el color rojo: ■
- Para una imagen en escala de grises, se reduce a una matriz $n \times m$, donde cada píxel tiene un valor entre 0 y $k$, que son los diferentes niveles de gris posibles.

La descomposición SVD permite aplicar técnicas de compresión de imágenes al descomponer la matriz de imagen y luego comprimir la información utilizando solo algunos valores singulares, dependiendo de la calidad deseada.

La calidad de la imagen comprimida se controla mediante el número de valores singulares $(k)$ utilizados en la reconstrucción. Un valor más alto de $k$ proporciona una mejor calidad de imagen pero una menor compresión, y viceversa. Para recuperar la matriz original después de aplicarle la SVD podemos utilizar la definición de suma de tripletas.

La relación de compresión $(r)$ se calcula como:

$$r = \frac{(n + m + 1)k}{nm} \tag{14.14}$$

Por ejemplo, para una imagen de $480 \times 640$ píxeles con $k = 50$, la imagen comprimida requeriría solo el 18 % de la información original.

En resumen, la aplicación de SVD en la compresión de imágenes ofrece una manera eficaz de reducir el tamaño de los archivos de imagen mientras se mantiene una calidad visual aceptable.

# Bibliografía


[1] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, "Sparse matrix methods in optimization," *SIAM Journal on Scientific and Statistical Computing*, vol. 5, pp. 562–589, Sept. 1984.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second ed., 2003.

[3] I. S. Duff, "A survey of sparse matrix research," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 500–535, 1977.

[4] N. P. Hummon and P. Doreian, "Computational methods for social network analysis," *Social Networks*, vol. 12, pp. 273–288, Dec. 1990.

[5] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011.