Universidad de Valladolid

Facultad de Ciencias

TRABAJO FIN DE GRADO

Grado en Matemáticas

MÉTODOS DIRECTOS PARA SISTEMAS LINEALES CON MATRIZ DISPERSA

Autor: *Alonso Sánchez Rodríguez*

Tutora: *María Paz Calvo Cabrero*

Índice

Introducción

La resolución de sistemas de ecuaciones lineales es, sin duda, una de las tareas que con mayor frecuencia realizan los ordenadores, puesto que estos sistemas aparecen como paso intermedio en el estudio de diversos problemas en las distintas ramas de la Ciencia y la Técnica. En el Trabajo Fin de Grado que presentamos se estudian métodos directos para la resolución numérica eficiente de grandes sistemas lineales en los que la matriz es dispersa, es decir, el número de coeficientes no nulos en la matriz es pequeño si se compara con el cuadrado de su dimensión.

En la Sección 1 se revisan las factorizaciones LU y de Cholesky de una matriz, y se muestran ejemplos donde se pone de manifiesto la ventaja de sacar partido al carácter disperso de la matriz del sistema.

La Sección 2 se centra en el caso de sistemas lineales con matriz simétrica definida positiva y se estudian dos reordenaciones que aplicadas a la matriz antes de su factorización permiten reducir la aparición de elementos no nulos en el correspondiente factor triangular en posiciones en las que había ceros en la matriz de partida. Se incluyen también ejemplos, uno de ellos procedente de la discretización de la ecuación del calor, donde se refleja la reducción que se consigue en el tiempo de resolución de los sistemas lineales cuando se explota la estructura dispersa de las matrices.

La Sección 3 se dedica al estudio de una reordenación específica para el problema del cálculo del PageRank que aprovecha, por un lado la gran dispersión de la matriz asociada al grafo de conexiones de la web, y por otro, la existencia de páginas web que no tienen enlaces a otras páginas.

En el Apéndice se recogen las funciones que se han programado en Matlab implementando los algoritmos de reordenación estudiados en este trabajo y con los que se han realizado las gráficas y tablas incluidas en la memoria. También se ha incluido una breve decripción de la forma en que Matlab almacena las matrices dispersas.

Me gustaría agradecer a la Dra. Mari Paz Calvo Cabrero por su dedicación e implicación a lo largo de la realización de este trabajo.

1. Factorización LU y de Cholesky

En general, una matriz real $n \times n$ se almacena en el ordenador como un array lleno de n^2 números en coma flotante. La complejidad computacional de las operaciones básicas como la trasposición es proporcional a n^2. Sin embargo, cuando realizamos operaciones más complejas tales como la eliminación Gaussiana la complejidad aumenta a $O(n^3)$. Este hecho reduce el espectro de problemas que involucran matrices que podemos tratar numéricamente debido, en muchos casos, más a la dimensión del problema que al tiempo de cálculo. Por ejemplo, es compatible el hecho de que en un ordenador personal no se pueda resolver un sistema lineal con matriz de tamaño $n = 6800$ por problemas de memoria con poder resolver un sistema lineal para $n = 6500$ en menos de 20 segundos.

Aunque no hay una definición precisa de **matriz dispersa**, podríamos definir las matrices dispersas como aquellas que tienen un número de coeficientes nulos suficientemente grande como para poder obtener ventaja de ello. Por contraposición, a las matrices que no son dispersas se las llama **matrices densas**.

Las matrices dispersas las podemos encontrar en diversos campos, como análisis de circuitos y estructuras, optimización a gran escala, mecánica de fluidos, y la solución numérica de ecuaciones en derivadas parciales, entre otros.

Al evitar las operaciones aritméticas en las entradas nulas de la matriz, los algoritmos para matrices dispersas requieren menor tiempo computacional. Un parámetro descriptivo importante para una matriz dispersa A es el número de entradas no nulas, calculable en Matlab mediante `nnz(A)` (number of nonzero elements). El objetivo de un buen algoritmo para matrices dispersas es que el costo computacional requerido por el mismo sea proporcional al número de operaciones aritméticas correspondientes a elementos no nulos de la matriz.

En este trabajo estudiamos algunos algoritmos que permiten resolver de manera eficiente grandes sistemas lineales con matriz dispersa, utilizando para ello métodos directos. Comenzaremos revisando las factorizaciones LU y de Cholesky para matrices generales y simétricas definidas positivas, respectivamente.

1.1. Factorización LU

Cuando utilizamos el método de eliminación Gaussiana para factorizar una matriz A de dimensión $n \times n$ (con todos sus determinantes menores principales no nulos) como producto LU, con L una matriz triangular inferior con unos en la diagonal y U una matriz triangular superior invertible, obtenemos un método que nos proporciona las siguientes fórmulas para el cáculo de los elementos de L y U

$$u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}, \qquad j = 1, \ldots, n \quad i = 1, \ldots, j$$

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j})/u_{j,j} \qquad j = 1, \ldots, n \quad i = j+1, \ldots, n$$

Si L y U son matrices densas, el cálculo de L y U requiere $2n^3/3 + O(n^2)$ multiplicaciones. En el caso de que las matrices L y U fuesen dispersas, algunas de estas multiplicaciones no serían necesarias, puesto que alguno de los factores l_{ik} ó u_{kj} pueden ser nulos, y si

un multiplicador es cero, no hay que modificar la fila que lo contiene. Nuestro primer objetivo será desarrollar un algoritmo para hallar la factorización LU de una matriz A, de modo que el costo computacional sea proporcional al número de multiplicaciones no nulas en el proceso arriba descrito.

Para matrices o vectores X y Y, denotamos por flops(XY) el número de multiplicaciones de elementos no nulos que se han realizado al calcular el producto XY de forma usual. La eliminación Gaussiana necesita $O(\text{flops}(LU))$ para calcular la factorización LU de A, y nuestro objetivo será encontrar un algoritmo para la eliminación Gaussiana con pivotaje parcial que tenga un costo computacional de $O(\text{flops}(LU))$ al factorizar una matriz no singular A como $PA = LU$.

Descripción del algoritmo

El algoritmo que usamos para la factorización LU consiste en calcular en cada paso una columna de L y la misma columna de U resolviendo un sistema triangular con la parte de L conocida. De hecho, podemos conocer la estructura no nula de cada columna antes de calcularla, aunque, no podemos calcular a priori la cantidad de elementos no nulos de L y U.

Antes de entrar en el bucle general debemos calcular la primera columa de L y de U. Si el elemento a_{11} de A es nulo, ó bien no es el elemento de mayor módulo de la primera columna, se debe encontrar el elemento de mayor módulo de la columna y realizar un intercambio de filas; este intercambio se tiene en cuenta en el vector $\mathbf{p}$, asociado a la matriz de permutación P, tal que $PA = LU$. La primera columna de L será entonces la primera columna de A reordenada dividida entre la primera entrada de la misma (nótese que el primer elemento será un uno). La primera columna de U solo tendrá un elemento no nulo, que será su primera entrada y coincidirá con el primer elemento de A reordenada.

Para describir el algoritmo, denotamos por j al índice de la columa de L y de U que está siendo calculada. Se escribe $\mathbf{a}_j = (a_{1,j}, ..., a_{j-1,j})^T$, $\mathbf{a}'_j = (a_{j,j}, ..., a_{n,j})^T$ (separando la columna j-ésima de A en dos partes) y se denota:

$$L_j = \begin{pmatrix} l_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ l_{j-1,1} & \cdots & l_{j-1,j-1} \end{pmatrix}, \qquad L'_j = \begin{pmatrix} l_{j,1} & \cdots & l_{j,j-1} \\ \vdots & \ddots & \vdots \\ l_{n,1} & \cdots & l_{n,j-1} \end{pmatrix},$$

$\mathbf{l}_j = (l_{j,j}, ..., l_{n,j})$, y $\mathbf{u}_j = (u_{1,j}, ..., u_{j-1,j})$. También denotaremos por $\mathbf{b}_j = (b_{j,j}, ..., b_{n,j})$ a un vector intermedio que aparece en el bucle.

Algoritmo 1. Factorización LU con pivotaje parcial

for $j = 2, \ldots, n$ **do**
1.- Se resuelve el sistema $L_j \mathbf{u}_j = \mathbf{a}_j$.
2.- Se calcula $\mathbf{b}_j = \mathbf{a}'_j - L'_j \mathbf{u}_j$
3.- Se intercambia $b_{j,j}$ con el elemento de mayor módulo de $\mathbf{b}_j$ (éste será el pívot).
4.- $u_{j,j} = b_{j,j}$
5.- Se intercambia la fila pivotal con la fila j en L, en A, y $\mathbf{p}$.
6.- $\mathbf{l}_j = \mathbf{b}_j / u_{j,j}$
end for

El Algoritmo 1 describe el proceso de factorización cuando se trabaja por columnas. Hay que tener en cuenta que si encontramos un vector $\mathbf{b}_j$ idénticamente nulo no podremos

dividir por su elemento de mayor módulo y, por lo tanto, lo que hacemos es reescribir un 1 en la posición $l_{j,j}$, un 0 en $u_{j,j}$ y parar (A en este caso sería singular y no admitiría factorización $PA = LU$).

En relación con el algoritmo descrito se pueden formular algunos resultados relativos a su costo computacional, recogidos en [5]

Lema 1.1. *El sistema triangular del paso 1 del algoritmo se puede resolver con un costo computacional igual a O(flops ($L_j\mathbf{u}_j$)).*

Demostración: (ver [5])

Lema 1.2. *Calcular* $\mathbf{b}_j$ *en el paso 2 del algoritmo se puede hacer con un costo computacional igual a O(flops($L'_j\mathbf{u}_j$)).*

Demostración:
La operación que queremos realizar consiste en restar a $\mathbf{a}'_j$ el vector obtenido de multiplicar $L'_j\mathbf{u}_j$, por lo que las multiplicaciones elementales que estamos realizando son esencialmente aquellas que corresponden a elementos no nulos del producto $L'_j\mathbf{u}_j$. □

Teorema 1.1. *El algoritmo completo de la factorización LU con pivotaje parcial se puede implementar con un costo computacional igual a O(flops(LU)).*

Demostración:
Por los Lemas 1.1 y 1.2, los pasos 1 y 2 se realizan en $O(\text{flops}(LU))$, los pasos 3 y 6 trabajan sobre cada elemento no nulo de $\mathbf{b}_j$. El paso 4 modifica $O(n)$ elementos, y el 5 sólo realiza intercambios de fila. Por lo tanto, el número de multiplicaciones es del orden de $O(\text{flops}(LU))$. □

Veamos a continuación sobre un ejemplo concreto, cómo actúa el algoritmo que estamos estudiando.

Ejemplo 1.1. *Sea*

$$A = \begin{pmatrix} 7 & 0 & 1 & 3 \\ 0 & 5 & 0 & 9 \\ 6 & 0 & 1 & 0 \\ 10 & 2 & 0 & 1 \end{pmatrix}.$$

Inicialización:

$$A = \begin{pmatrix} 10 & 2 & 0 & 1 \\ 0 & 5 & 0 & 9 \\ 6 & 0 & 1 & 0 \\ 7 & 0 & 1 & 3 \end{pmatrix}, \qquad \mathbf{p} = (4, 2, 3, 1)^T,$$

$$L = \begin{pmatrix} 1 & \vdots \\ 0 & \vdots \\ 6/10 & \vdots \\ 7/10 & \vdots \end{pmatrix}, \qquad U = \begin{pmatrix} 10 & \vdots \\ 0 & \vdots \\ 0 & \vdots \\ 0 & \vdots \end{pmatrix},$$

Primer paso, $j = 2$:

$$L_2 = (1), \quad \mathbf{a}_2 = 2, \quad \mathbf{u}_2 = 2,$$

$$L_2' = \begin{pmatrix} 0 \\ 7/10 \\ 6/10 \end{pmatrix}, \quad \mathbf{a}_2' = \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{b}_2 = \mathbf{a}_2' - L_2'\mathbf{u}_2 = \begin{pmatrix} 5 \\ -14/10 \\ -12/10 \end{pmatrix}.$$

Intercambiamos los elementos para que el pívot sea el coeficiente de mayor módulo (aunque en este caso no hay intercambio),

$$\mathbf{b}_2 = \begin{pmatrix} 5 \\ -14/10 \\ -12/10 \end{pmatrix}, \quad \mathbf{p} = (4,2,3,1), \quad u_{2,2} = 5, \quad \mathbf{l}_2 = \begin{pmatrix} 1 \\ -14/50 \\ -12/50 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & \vdots \\ 0 & 1 & \vdots \\ 7/10 & -14/50 & \vdots \\ 6/10 & -12/50 & \vdots \end{pmatrix}, \quad U = \begin{pmatrix} 10 & 2 & \vdots \\ 0 & 5 & \vdots \\ 0 & 0 & \vdots \\ 0 & 0 & \vdots \end{pmatrix}, \quad A = \begin{pmatrix} 10 & 2 & 0 & 1 \\ 0 & 5 & 0 & 9 \\ 6 & 0 & 1 & 0 \\ 7 & 0 & 1 & 3 \end{pmatrix}.$$

Al final:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 7/10 & -14/50 & 1 & 0 \\ 6/10 & -12/50 & 1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 10 & 2 & 0 & 1 \\ 0 & 5 & 0 & 9 \\ 0 & 0 & 1 & 91/50 \\ 0 & 0 & 0 & 137/50 \end{pmatrix}, \quad \mathbf{p} = (4,2,3,1)^T.$$

Si tratásemos con una matriz densa cuadrada de tamaño $n \times n$, se tendría una implementación de éste algoritmo LU con pivotaje parcial con costo computacional $O(n^3)$. Sin embargo, cuando la matriz es dispersa, el costo computacional puede ser notablemente menor.

Veamos ahora otro ejemplo que nos permite ilustrar esta reducción en el costo computacional del algoritmo LU cuando se trabaja con matrices dispersas. Para ello consideremos la matriz tridiagonal con 1's en la diagonal superior e inferior y con -2's en la diagonal principal (esta matriz dispersa está relacionada con la discretización mediante diferencias finitas de la derivada segunda),

$$A = \begin{pmatrix} -2 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & \cdots & 0 & 1 & -2 \end{pmatrix}. \tag{1.1}$$

En este caso esperamos para el caso denso un costo computacional de $O(n^3)$, mientras que para el caso disperso debería reducirse a $O(3n-2)$, que es la cantidad de elementos no nulos en la matriz.

Veamos una tabla comparativa de los tiempos de CPU requeridos por el algoritmo implementado en Matlab para la versión densa y dispersa de la matriz (1.1) con diferentes

Orden n	Densa	Dispersa
125	0.0126	0.0108
250	0.0131	0.0111
500	0.0142	0.0109
1000	0.0370	0.0123
2000	0.2452	0.0130
4000	1.1318	0.0136
8000	6.7174	0.0134
16000	59.4724	0.0162

Orden n	Dispersa
32000	0.0233
64000	0.0312
128000	0.0580
256000	0.0976
512000	0.1467
1024000	0.3455
2040000	0.7266
4096000	1.2837
8192000	2.6960

Cuadro 1: Comparativa de tiempos de CPU cuando la matriz 1.1 se considera densa o dispersa

tamaños n (Cuadro 1) y una gráfica en la que se han representado en escala doblemente logarítmica el tiempo de CPU frente a la dimensión (Figura 1).

Fijándonos en el Cuadro 1 de tiempos de CPU vemos la diferencia drástica densa-dispersa según aumentamos n. Aunque los tiempos requeridos en el caso disperso son siempre menores que en el caso denso, las diferencias crecen notablemente con n. También es de remarcar la posibilidad de trabajar con dimensiones n mucho mayores cuando tratamos las matrices como dispersas. De hecho, las dimensiones que se han considerado en la tabla de la derecha, son sólo posibles en el caso disperso.

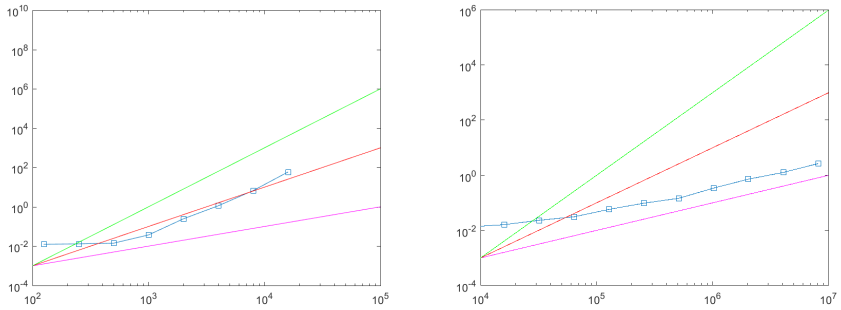


Figura 1: Tiempos de CPU (en sg.) frente a dimensión

En la Figura 1 la línea azul con cuadrados es representación gráfica en escala doblemente logarítmica de la relación 'Dimensión de la matriz-Tiempo de CPU', para matrices densas en la gráfica de la izquierda y para matrices dispersas en la de la derecha (utilizando los datos del Cuadro 1). Representado en escala doblemente logarítmica tenemos en morado la función $c_1 n$, en rojo $c_2 n^2$ y en verde $c_3 n^3$ para constantes c_1, c_2 y c_3 adecuadas. En la gráfica de las matrices dispersas hemos podido llegar a calcular tiempos de factorización de matrices cuadradas de dimensión 8192000 (2.6960 sg.).

Se puede observar, comparando las pendientes de las gráficas, que el tiempo de CPU

tratando la matriz como densa crece más deprisa que n^2 (línea roja), casi tendiendo a n^3 (línea verde), contrariando en cierto modo los resultados teóricos, pero su explicación reside en que al tratarla como densa, Matlab no tiene en cuenta la estructura tridiagonal de la matriz. Además, hay que tener en cuenta que en el tiempo de CPU no se contabiliza sólo el debido a las operaciones aritméticas. Para el caso disperso, tenemos una pendiente próxima a 1, como corresponde a $O(n)$ (línea morada).

Los resultados de esta comparación justifican sobradamente el estudio de métodos específicos para la resolución de grandes sistemas lineales con matriz dispersa.

1.2. Factorización de Cholesky

Consideramos en esta subsección el caso particular de matrices simétricas definidas positivas. Es conocido que una matriz simétrica A, ($A = A^T$) es definida positiva si $\mathbf{x}^T A\mathbf{x} > 0$, para cualquier vector no nulo $\mathbf{x}$. Típicamente, $\mathbf{x}^T A\mathbf{x} > 0$ representa la energía de algún sistema físico que es positivo para cualquier configuración $\mathbf{x}$.

En una matriz A simétrica definida positiva las entradas diagonales son siempre positivas debido a que

$$\mathbf{e}_i^T A\mathbf{e}_i = a_{i,i},$$

siendo $\mathbf{e}_i$ el i-ésimo vector de la base ordenada canónica.

Incluimos a continuación un resultado de factorización para matrices simétricas definidas positivas ya estudiado en el Grado, pero aportando una demostración constructiva alternativa que proporciona un algoritmo para el cálculo eficiente de dicha factorización en el caso disperso.

Teorema 1.2. *Si A es una matriz simétrica definida positiva $n \times n$, entonces existe una única factorización $A = LL^T$, donde L es una matriz triangular inferior con todas sus entradas diagonales positivas. Dicha factorización se llama factorización de Cholesky.*

Demostración:

Aunque el resultado se puede demostrar como consecuencia de la factorización LU de una matriz, indicamos aquí una demostración constructiva por inducción en el orden de la matriz A.

El resultado es trivialmente cierto para matrices 1×1 ya que la entrada $a_{1,1}$ es positiva y basta tomar $l_{1,1} = \sqrt{a_{1,1}}$.

Suponemos ahora el resultado cierto para matrices de orden $n-1$, y sea A una matriz simétrica definida positiva de orden n. Se puede dividir de la siguiente forma

$$A = \begin{pmatrix} d & \mathbf{v}^T \\ \mathbf{v} & \overline{H} \end{pmatrix},$$

donde d es un escalar positivo (el elemento (1,1) de A), $\mathbf{v} \in \mathbb{R}^{n-1}$ y $\overline{H}$ es la submatriz inferior derecha de tamaño $(n-1)\times(n-1)$ de A. La matriz así dividida se puede reescribir como el siguiente producto de matrices

$$A = \begin{pmatrix} \sqrt{d} & \mathbf{0}^T \\ \dfrac{\mathbf{v}}{\sqrt{d}} & I_{N-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & H \end{pmatrix} \begin{pmatrix} \sqrt{d} & \dfrac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & I_{N-1} \end{pmatrix},$$

donde $H = \overline{H} - \dfrac{\mathbf{v}\mathbf{v}^t}{d}$.

Es claro que H es simétrica, y es también definida positiva ya que para cualquier vector $\mathbf{x}$ no nulo de dimensión $n-1$ se tiene

$$\mathbf{x}^T H \mathbf{x} = \mathbf{x}^T \left(\overline{H} - \frac{\mathbf{v}\mathbf{v}^T}{d} \right) \mathbf{x} = \begin{pmatrix} -\dfrac{\mathbf{x}^T\mathbf{v}}{d} & \mathbf{x}^T \end{pmatrix} \begin{pmatrix} d & \mathbf{v}^T \\ \mathbf{v} & \overline{H} \end{pmatrix} \begin{pmatrix} -\dfrac{\mathbf{x}^T\mathbf{v}}{d} \\ \mathbf{x} \end{pmatrix} = \mathbf{y}^T A \mathbf{y} > 0,$$

donde $\mathbf{y} = [-\dfrac{\mathbf{x}^T\mathbf{v}}{d}, \quad \mathbf{x}^T]^T$.

Por la hipótesis de inducción, H tiene una factorización $L_H L_H^T$, con L_H triangular inferior con elementos diagonales positivos. Por tanto A se puede expresar como

$$A = \begin{pmatrix} \sqrt{d} & \mathbf{0}^T \\ \dfrac{\mathbf{v}}{\sqrt{d}} & I_{N-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & L_H \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & L_H^T \end{pmatrix} \begin{pmatrix} \sqrt{d} & \dfrac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & I_{N-1} \end{pmatrix} =$$

$$= \begin{pmatrix} \sqrt{d} & \mathbf{0}^T \\ \dfrac{\mathbf{v}}{\sqrt{d}} & L_H \end{pmatrix} \begin{pmatrix} \sqrt{d} & \dfrac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & L_H^T \end{pmatrix} = LL^T,$$

con lo que queda probada la existencia.

Veamos ahora la unicidad también por inducción sobre las columnas de la matriz.

Sean L y C triangulares inferiores con todos sus elementos diagonales positivos de modo que

$$A = LL^T = CC^T.$$

La igualdad del elemento $a_{1,1}$ conduce a $l_{1,1}^2 = c_{1,1}^2 \Rightarrow l_{1,1} = c_{1,1}$, por ser ambos positivos.

Para $j = 2, \ldots, n$ la igualdad del elemento $a_{1,j}$ de la matriz A da lugar a

$$l_{1,1} l_{j,1} = c_{1,1} c_{j,1} \Rightarrow l_{j,1} = c_{j,1}.$$

En definitiva, la primera fila de L^T coincide con la primera fila de C^T, es decir, la primera columna de L coincide con la primera columna de C.

Supongamos ahora que coinciden las primeras $k-1$ columnas de L con las primeras $k-1$ columnas de C. Veamos que entonces también coinciden las columnas k de L y C. En efecto,

$$a_{k,k} = \sum_{j=1}^{k} l_{k,j}^2 = \sum_{j=1}^{k} c_{k,j}^2,$$

y al tener L y C las primeras $k-1$ columnas iguales, resulta: $l_{k,k}^2 = c_{k,k}^2 \Rightarrow l_{k,k} = c_{k,k}$, por ser ambos positivos.

Veamos ahora que $l_{i,k} = c_{i,k}$, si $i > k$. Examinando la igualdad para el elemento $a_{i,k}$ de A se obtiene que

$$a_{i,k} = \sum_{j=1}^{n} l_{i,j} l_{k,j} = \sum_{j=1}^{n} c_{i,j} c_{k,j},$$

y como L y C son ambas triangulares inferiores y coinciden sus columnas hasta la $k-1$ se tiene

$$a_{i,k} = \sum_{j=1}^{k} l_{i,j} l_{k,j} = \sum_{j=1}^{k} c_{i,j} c_{k,j} \Rightarrow l_{i,k} l_{k,k} = c_{i,k} c_{k,k} \Rightarrow l_{i,k} = c_{i,k},$$

con lo que se tiene que también coincide la columna k de ambas matrices.
Por tanto se tiene la unicidad de la factorización de Cholesky. □

Descripción del algoritmo

La demostración constructiva del Teorema 1.2 nos proporciona un método para calcular el factor triangular inferior L. Es la llamada forma de producto externo *(outer product form)* del algoritmo de factorización de Cholesky. Veámoslo paso por paso, partiendo de $A_0 = A$.

$$\begin{aligned}
A_0 &= \begin{pmatrix} d_1 & \mathbf{v}_1^T \\ \mathbf{v}_1 & \overline{H}_1 \end{pmatrix} = \begin{pmatrix} \sqrt{d_1} & \mathbf{0^T} \\ \dfrac{\mathbf{v}_1}{\sqrt{d_1}} & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & \overline{H}_1 - \dfrac{\mathbf{v}_1\mathbf{v}_1^t}{d_1} \end{pmatrix} \begin{pmatrix} \sqrt{d_1} & \dfrac{\mathbf{v}_1^T}{\sqrt{d_1}} \\ \mathbf{0} & I_{n-1} \end{pmatrix} \\
&= L_1 \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & H_1 \end{pmatrix} L_1^T = L_1 A_1 L_1^T, \qquad (1.2) \\
A_1 &= \begin{pmatrix} 1 & 0 & \mathbf{0^T} \\ 0 & d_2 & \mathbf{v}_2^T \\ \mathbf{0} & \mathbf{v}_2 & \overline{H}_2 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & \mathbf{0^T} \\ 0 & \sqrt{d_2} & \mathbf{0^T} \\ \mathbf{0} & \dfrac{\mathbf{v}_2}{\sqrt{d_2}} & I_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 0 & \mathbf{0^T} \\ 0 & 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{0} & \overline{H}_2 - \dfrac{\mathbf{v}_2\mathbf{v}_2^T}{d_2} \end{pmatrix} \begin{pmatrix} 1 & 0 & \mathbf{0^T} \\ 0 & \sqrt{d_2} & \dfrac{\mathbf{v}_2^T}{\sqrt{d_2}} \\ \mathbf{0} & \mathbf{0} & I_{n-2} \end{pmatrix} \\
&= L_2 A_2 L_2^T \\
&\cdots \\
A_{n-1} &= L_n I_n L_n^T,
\end{aligned}$$

donde, para $1 \le i \le n$, $\quad d_i$ es un escalar positivo, $\mathbf{v}_i$ un vector de dimensión $n-i$, y $\overline{H}_i$ es una matriz simétrica definida positiva de dimensión $(n-i)\times(n-i)$.

Después de n pasos se tiene $A = L_1 L_2 \cdots L_n L_n^T \cdots L_2^T L_1^T = LL^T$, con $L = L_1 L_2 \cdots L_n$. Veamos a continuación que se puede dar una expresión alternativa más sencilla para L.

Lema 1.3. *Sean E y F dos matrices triangulares inferiores $n \times n$ que para algún k $(1 \le k \le n)$ satisfacen*

$$\begin{aligned}
f_{i,j} &= 1, \quad j \le k, i = j, \\
f_{i,j} &= 0, \quad j \le k, i > j, \\
e_{i,j} &= 1, \quad j > k, i = j, \\
e_{i,j} &= 0, \quad j > k, i > j.
\end{aligned}$$

Se verifica que $EF = E + F - I$.

Demostración:

Podríamos escribir las matrices E y F de la siguiente forma por bloques

$$E = \left(\begin{array}{c|c} E_k & O \\ \hline \overline{E}_k & I_{n-k} \end{array}\right), \qquad F = \left(\begin{array}{c|c} I_k & O \\ \hline O & F_k \end{array}\right),$$

con E_k la submatriz de E formada por sus primeras k filas y k columnas (es triangular inferior), $\overline{E}_k$ la submatriz de E formada por primeras k columnas y sus últimas $n-k$

filas, O es la parte de E correspondiente a los ceros que se encuentran en las k primeras filas y $n-k$ últimas columnas de E y, por último, I_{n-k} es la identidad de tamaño $n-k$ que corresponde según lo descrito en el enunciado del ejercicio a las últimas $n-k$ filas y columnas de E. Para la matriz F se tiene una identidad I_k de tamaño k que se corresponde con sus primeras k filas y columnas, dos submatrices de ceros en las partes correspondientes a las k primeras filas y $n-k$ últimas columnas, y a las últimas $n-k$ filas y primeras k columnas, y también se tiene una submatriz cuadrada de tamaño $n-k$ denotada por F_k, triangular inferior, formada por sus últimas $n-k$ filas y columnas.

Estamos ahora en condiciones de realizar el producto y la suma de E y F trabajando por bloques para obtener

$$E \cdot F = \left(\begin{array}{c|c} E_k & O \\ \hline \overline{E}_k & F_k \end{array}\right), \qquad E + F = \left(\begin{array}{c|c} E_k + I_k & O \\ \hline \overline{E}_k & I_{n-k} + F_k \end{array}\right) = I + E \cdot F, \tag{1.3}$$

lo que concluye la demostración. □

Teorema 1.3. *El factor $L = L_1 L_2 \cdots L_n$ construido mediante (1.2) satisface*

$$L \quad = \quad L_1 + L_2 + \cdots + L_n - (n-1)I_n. \tag{1.4}$$

Demostración:

Veamos como se llega al resultado (1.4) utilizando el Lema 1.3. Razonamos por inducción.

Si tomamos $n = 2$ el resultado se tiene por el Lema 1.3, tomando $E = L_1$ y $F = L_2$ con $k = 1$, por lo que se tiene el resultado (1.4) con $n = 2$

$$L_1 \cdot L_2 = L_1 + L_2 - (2-1)I.$$

Suponemos cierto el resultado para $n-1$, y lo probamos para n. Podemos separar $L = L_1 L_2 \cdots L_n = L_1 L_2 \cdots L_{n-1} \cdot L_n$, y es cierto que la matriz $L_1 L_2 \cdots L_{n-1}$ tiene la forma de E y L_n la de F con $k = n-1$. Por tanto, por (1.3) podemos separar $L = L_1 L_2 \cdots L_{n-1} \cdot L_n = L_1 L_2 \cdots L_{n-1} + L_n - I$, y por la hipótesis de inducción se llega definitivamente a

$$L = L_1 + L_2 + \cdots + L_n - (n-2)I - I = L = L_1 + L_2 + \cdots + L_n - (n-1)I_n,$$

quedando probada la igualdad (1.4). □

La primera observación que se puede hacer es que por (1.2) y (1.4), si $\eta(X)$ es el número de elementos no nulos de X, donde X puede ser un vector o una matriz, se tiene que el número de elementos no nulos de L está dado por

$$\eta(L) = n + \sum_{i=1}^{n-1} \eta(\mathbf{v}_i),$$

siendo los $\mathbf{v}_i$ los vectores de dimensión $n-i$ que se encuentran en la primera fila y columna de la matriz $H_i = \overline{H_i} - \dfrac{\mathbf{v}_i \mathbf{v}_i^T}{d_i}$ en cada paso del algoritmo constructivo.

Además se tiene el siguiente resultado

Teorema 1.4. *El número de operaciones requeridas para calcular el factor triangular L de la matriz A está dado por*

$$\frac{1}{2}\sum_{i=1}^{n-1}\eta(\mathbf{v}_i)[\eta(\mathbf{v}_i)+3] = \frac{1}{2}\sum_{i=1}^{n-1}[\eta(L_{*i})-1][\eta(L_{*i})+2],$$

donde L_{*i} denota la columna i-ésima de L.

Demostración:

En el paso i-ésimo, necesitamos $\eta(\mathbf{v}_i)$ operaciones para calcular $\dfrac{\mathbf{v}_i}{\sqrt{d_i}}$, y $\dfrac{1}{2}\eta(\mathbf{v}_i)[\eta(\mathbf{v}_i)+1]$ para calcular la matriz simétrica

$$\frac{\mathbf{v}_i\mathbf{v}_i^T}{d_i} = (\frac{\mathbf{v}_i}{\sqrt{d_i}})(\frac{\mathbf{v}_i}{\sqrt{d_i}})^T,$$

ya que cuando estamos calculando la fila j-ésima ya conocemos $j-1$ elementos de ella, por lo que, esencialmente esta multiplicación conlleva $1+2+\cdots+\eta(\mathbf{v}_i) = \dfrac{1}{2}\eta(\mathbf{v}_i)[\eta(\mathbf{v}_i)+1]$ operaciones.

Al resultado global se llega sumando el número de operaciones requeridas en estos dos procesos en los $n-1$ pasos. □

En el caso denso, la matriz triangular L tiene $\dfrac{1}{2}n(n+1)$ elementos no nulos y el costo operativo de la factorización, según el Teorema 1.4 es

$$\frac{1}{2}\sum_{i=1}^{n-1} i(i+3) = \frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n.$$

Consideremos ahora la matriz tridiagonal A de la Subsección 1.1 y como A es definida negativa, trabajamos con $-A$. Se puede demostrar que el factor L satisface

$$\eta(L_{*i}) = 2, \qquad i = 1, \ldots n-1,$$

(ver Capítulo 5 de [2]), y se tiene que $\eta(L) = 2n-1$.

El costo computacional de hallar el factor L de Cholesky en este caso es

$$\frac{1}{2}\sum_{i=1}^{n-1} 1\cdot 4 = 2n-2,$$

con lo que la complejidad se reduce de ser cúbica a ser lineal, lo cual es una mejora considerable.

2. Reordenaciones

A la vista de la importancia que tiene el número de elementos no nulos de la matriz L en el costo computacional de la factorización de Cholesky de una matriz A, estudiamos en esta sección algunas reordenaciones que permiten reducir en la medida de lo posible dicho número. Se han seguido los libros [1] y [2].

Consideramos el sistema de ecuaciones $n \times n$

$$A\mathbf{x} = \mathbf{b},$$

donde n es grande y A es una matriz simétrica definida positiva y dispersa. Esta hipótesis se mantendrá a lo largo de toda esta sección.

Al calcular el factor L de la factorización de Cholesky, muchas veces se produce **llenado**, definido coloquialmente como la generación de elementos no nulos durante el proceso de factorización en posiciones donde la matriz original presentaba entradas nulas. Lo mismo ocurre al calcular la factorización LU de una matriz dispersa no necesariamente simétrica.

Una matriz de permutación P es una matriz que se obtiene a partir de la identidad reordenando sus filas (o columnas). En otras palabras, es una matriz ortogonal que al multiplicar por la izquierda (respectivamente por la derecha) a otra matriz A, reordena las filas (respectivamente columnas) de A. Cuando se tiene una matriz de permutación P, la matriz PAP^T sigue siendo simétrica y definida positiva (en el caso de que A lo sea), por lo que, en vez de resolver el sistema lineal $A\mathbf{x} = \mathbf{b}$, se puede resolver el sistema reordenado

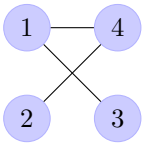
$$(PAP^T)(P\mathbf{x}) = P\mathbf{b}$$

La elección de P puede tener un efecto drástico en la cantidad de llenado que se produce durante la factorización de Cholesky.
Veamos algunos algoritmos de ordenación para la elección de P.

2.1. Algoritmo de Mínimo Grado

Antes de describir el algoritmo de mínimo grado vamos a revisar algunos conceptos y resultados que se van a utilizar más adelante.

Dada una matriz simétrica $A = (a_{i,j})_{1 \le i,j \le n}$, su **grafo no dirigido**, denotado por $G^A = (X^A, E^A)$, tiene n nodos numerados de 1 a n, y una arista uniendo los nodos i y j , $i \neq j$ (sin apuntar en ningún sentido), si $a_{i,j} = a_{j,i} \neq 0$. X^A es el conjunto de los nodos y E^A son los pares de índices (i, j) que definen las aristas. Por ejemplo,

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}, \qquad G^A = \quad \text{  }$$

La **estructura no nula** de una matriz simétrica A se define como

$$Nonz(A) = \{(i, j) : a_{i,j} \neq 0 \quad \text{e} \quad i \neq j\},$$

y determina las aristas del grafo no dirigido asociado a dicha matriz. De hecho $E^A = Nonz(A)$.

Supongamos ahora que $A = LL^T$ es la factorización de Cholesky de A. Se denota por $F(A)$ a la matriz suma $L + L^T$, que refleja el llenado que se ha producido en la factorización. La estructura no nula de $F = F(A)$ (para abreviar se suprime A) es

$$Nonz(F) = \{(i,j) : \quad l_{i,j} \neq 0 \quad \text{ó} \quad l_{j,i} \neq 0, \quad \text{e} \quad i \neq j\}.$$

Suponiendo que la cancelación numérica exacta no ocurre, dada una estructura no nula $Nonz(A)$, la correspondiente $Nonz(F)$ está totalmente determinada por $Nonz(A)$, es decir, $Nonz(F)$ es independiente de los valores numéricos de los coeficientes de A. Además,

$$Nonz(A) \subset Nonz(F),$$

y el llenado de la matriz A se puede definir como

$$Fill(A) = Nonz(F) - Nonz(A).$$

Sea L el factor triangular inferior obtenido con la factorización de Cholesky de la matriz A, y sea $G^A = (X^A, E^A)$ el grafo asociado a la matriz A. Si $F = L + L^T$, el grafo $G^F = (X^F, E^F)$ tiene los mismos nodos que G^A, es decir, $X^F = X^A$, y el conjunto de aristas E^F está formado por todas las aristas del grafo G^A junto con las aristas añadidas durante la factorización. Más precisamente, el conjunto de aristas E^F se relaciona con E^A a través del siguiente teorema

Teorema 2.1. *El par no ordenado $(i,j) \in E^F$ si y solo si $(i,j) \in E^A$ ó $(i,k) \in E^F$ y $(k,j) \in E^F$ para algún $k < min\{i,j\}$.*

Demostración:

$\boxed{\Rightarrow}$ Si $(i,j) \in E^F$, por definición se tiene que, ó bien $(i,j) \in E^A$, ó bien es una arista formada durante la factorización. Si $(i,j) \in E^A$, está clara la implicación. Si se trata de una arista formada debida a la factorización, podemos suponer, sin pérdida de generalidad, que $i > j$. Para que se haya podido generar un elemento no nulo en la posición (i,j) del factor triangular inferior L no existiendo ningún elemento en esa posición en la matriz original A, quiere decir que existe una fila con índice k anterior a i, de modo que el elemento (k,k) del factor L en una etapa intermedia ha actuado de pívot para eliminar el elemento (i,k) de la fila i y la entrada en la posición (k,j) con $j > k$era no nula, para que se pudiera formar un elemento no nulo en la posición (i,j). Además se ha supuesto la no cancelación numérica exacta, por lo que al final del proceso de factorización ese elemento se mantiene. Por lo tanto se tienen elementos no nulos en las posiciones (i,k) y (k,j), de la matriz $F = L + L^T$ como se quería.

$\boxed{\Leftarrow}$ Supongamos ahora que es cierta la parte derecha de la doble implicación. Tenemos también dos casos. Si $(i,j) \in E^A$, $(i,j) \in E^F$ por tener que $E^A \subset E^F$. Si estamos en el caso en el que $(i,k) \in E^F$ y $(k,j) \in E^F$ para algún $k < min\{i,j\}$, supongamos sin pérdida de generalidad que $i > j$. Se tiene que el elemento $(i,k) \in L$ es no nulo y se habrá utilizado en el proceso de eliminación como multiplicador para restar a la fila i un múltiplo de la fila k. En ese paso al elemento (i,j) (originalmente nulo en A) se le ha restado dicho multiplicador por el elemento (k,j) no nulo por hipótesis. En conclusión, se ha generado un elemento (i,j) no nulo en L y, por tanto, el par $(i,j) \in E^F$. □

El teorema anterior proporciona la forma en la que se produce el llenado, que viene dado por la diferencia $E^F - E^A$.

Además de los conceptos anteriores, para entender el funcionamiento del algoritmo de mínimo grado necesitamos introducir algunos términos adicionales de teoría de grafos (por simplicidad, se ha prescindido del superíndice A)

- Si v es un nodo de un grafo G, llamamos **nodos adyacentes a** v a aquellos que están unidos a v mediante una arista. Denotamos a este conjunto por $Adj_G(v)$.
- El **grado de un nodo** v en un grafo G es el número de nodos adyacentes a v (es decir, $|Adj_G(v)|$). Denotamos el grado del nodo v por $degree_G(v)$.
- Partiendo de un grafo G y de un nodo v de G, el **grafo de eliminación** G_v es el grafo obtenido después de eliminar del grafo G las aristas que parten de v, y añadiendo a continuación aristas de modo que los nodos adyacentes a v, al eliminar v, queden unidos entre sí (clique).

Descripción del algoritmo

En general, el algoritmo de mínimo grado trabaja solo con la estructura de A y simula de alguna forma los n pasos de la eliminación Gaussiana.

En cada paso se aplica un intercambio de una fila y su correspondiente columna a la parte de la matriz que aún no ha sido factorizada, de modo que el número de elementos en la fila/columna pivotal es mínimo. Después de n pasos la factorización entera ha sido simulada y el orden en el que elegimos las filas/columnas para el pivotaje parcial es la ordenación que define la matriz P.

Algoritmo 2. Algoritmo de Mínimo Grado

Se parte del grafo no dirigido G.
$i = 1$
while $G \neq \emptyset$ **do**
  Elegir el índice k de un nodo v de grado mínimo en G y guardarlo, $j_i \leftarrow k$.
  $G \leftarrow G_v$ (guardar en G el grafo de eliminación G_v).
  $i = i + 1$
end while

En este trabajo vamos a describir el ordenamiento de mínimo grado a través de grafos de eliminación.

Si partimos de una matriz simétrica, el primer paso es encontrar su grafo no dirigido etiquetado. Una vez entremos en el bucle, deberemos mantener un registro de los nodos que tomamos en cada paso (representando fila/columna), lo cual nos permitirá contruir la matriz de permutación.

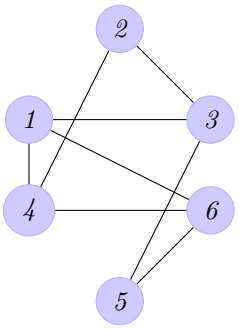
En relación al algoritmo, la teoría que subyace es la simulación del pivotaje parcial Gaussiano, pero en cada paso estamos tomando la fila/columna con el número mínimo de elementos no nulos distintos del pívot.

Al realizar el grafo de eliminación lo que estamos simulando es el 'llenado' que se produce en el resto de filas/columnas debido al intercambio de filas propuesto. El vector final $(j_1, \ldots, j_n)$ resultado del Algoritmo 2, proporciona el orden buscado.

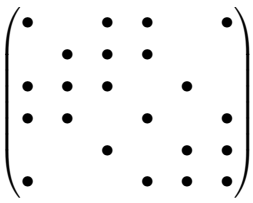
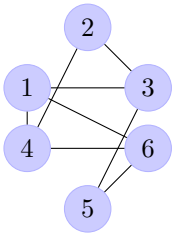
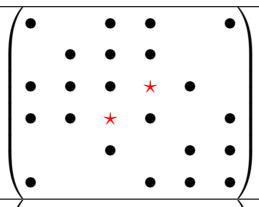
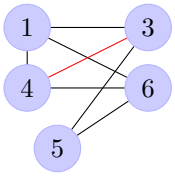
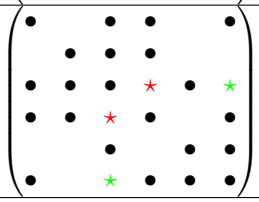
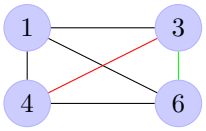
Veamos un ejemplo gráfico de esta simulación, de la elección de los nodos de mínimo grado, así como del llenado que predecimos.

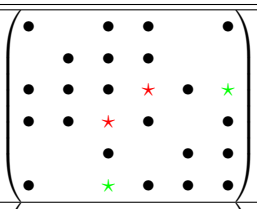
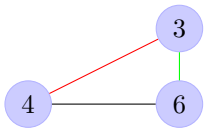
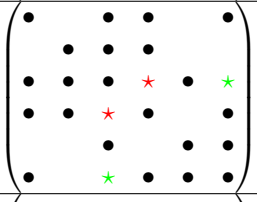
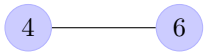
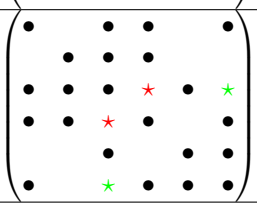
Ejemplo 2.1. *Partimos de la matriz* 6×6 *que se muestra a continuación, y de su grafo etiquetado* G^A*.*

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}, \qquad G^A = \quad$$



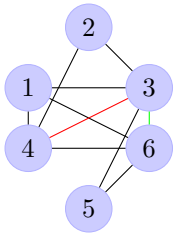
Para llevar a cabo el algoritmo no tendremos en cuenta el valor de los coeficientes de la matriz A. Representaremos con $\bullet$ los coeficientes originales de la matriz, mientras que el llenado que se produzca lo representaremos con $\star$.

Matriz	Nodo	Grado	Grafo asociado
	2	2	
	5	2	
	1	3	

Matriz	Nodo	Grado	Grafo asociado
	3	2	
	4	1	
	6	0	

Cuadro 2: Ilustración del funcionamiento del algoritmo de mínimo grado

En cada fila del Cuadro 2 se indica la estructura de los elementos no nulos de la matriz en cada etapa del proceso, el índice del nodo de mínimo grado y el grafo antes de eliminar dicho nodo (el grafo de eliminación del nodo seleccionado en la fila anterior). Las aristas de colores representan aristas que aparecen en el grafo de eliminación de un nodo que no estaban en el grafo antes de la eliminación. En este ejemplo, la secuencia de nodos reordenada sería $[2, 5, 1, 3, 4, 6]$, y la matriz de permutación asociada y el grafo G^F tras el llenado son

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad G^F = \quad \text{  }$$

Algunas mejoras del algoritmo

En la transformación del grafo de eliminación

$$G \leftarrow G_v$$

estamos realizando una actualización sobre los nodos adyacentes a v, siendo este el paso más costoso de todo el algoritmo. Se pueden implementar algunas mejoras a fin de re-

ducir el costo computacional del mismo. Revisamos aquí algunas de ellas, pero se puede encontrar más información en [3].

- **Eliminación en masa** (Mass elimination):
  Esta mejora se basa en el siguiente resultado

  Teorema 2.2. *Si y es el nodo de mínimo grado en el grafo G, entonces todos los nodos del subconjunto*

  $$Y = \{z \in Adj_G(y) : degree_{G_y}(z) = degree_G(y) - 1\}$$

  pueden ser seleccionados en los pasos sucesivos (en cualquier orden) en el algoritmo de mínimo grado.

  Demostración: La base de la demostración reside en darse cuenta de que si en el grafo de eliminación G_y un nodo z tiene un grado que es una unidad menos que el grado del nodo y que se está eliminando, dicho nodo z tenía el mismo grado que y, (y por tanto tenía mínimo grado) en el paso previo. □

  Este teorema permite evitar alguna transformación de grafos y actualización de grados, ya que en vez de hacer la transformación del grafo de eliminación para los nodos de $Y \cup \{y\}$, se pueden eliminar todos ellos simultaneamente (básicamente, tienen la misma secuencia de nodos adyacentes).

- **Eliminación múltiple**
  Esta mejora hace uso de la observación siguiente: en la eliminación del nodo y del grafo G, la estructura asociada a los nodos que no se encuentran en $Adj_G(y)$ permanece inalterada. La idea es suspender la actualización de grados para nodos en $Adj_G(y)$ y seleccionar un nodo con el mismo grado que y en el subgrafo $G - (Adj_G(y) \cup \{y\})$. Si no existe tal nodo, se procede a la actualización de grados de los nodos adyacentes a y.

- **Grado externo**
  Esta última mejora está basada en la definición de **nodos indistinguibles** de un grafo.

  Definición 2.1. *Se dice que dos nodos u y v son indistinguibles en G si*

  $$Adj_G(u) \cup \{u\} = Adj_G(v) \cup \{v\}.$$

  Después de introducir esta definición, hablaremos del **grado externo** de un nodo como el número de sus nodos adyacentes no distinguibles de él. Se puede llevar a cabo el algoritmo de mínimo grado, pero considerando el grado externo de los nodos en lugar del grado definido al principio de esta sección. Podríamos hablar de un algoritmo de mínimo "grado externo", que reduce el número de elementos no nulos en los factores de las decomposiciones que se están considerando en este trabajo.

Para finalizar el estudio del algoritmo de mínimo grado veamos un ejemplo en el que se puede apreciar cómo dicho algoritmo (implementado en el comando `symamd` en Matlab

y en la función `MDOmass` incluida en el Apartado 4.2.1 del Apéndice) reduce el llenado durante la factorización de Cholesky para una matriz 1138×1138 con 4054 elementos no nulos tomada de [8] ('1138_bus')

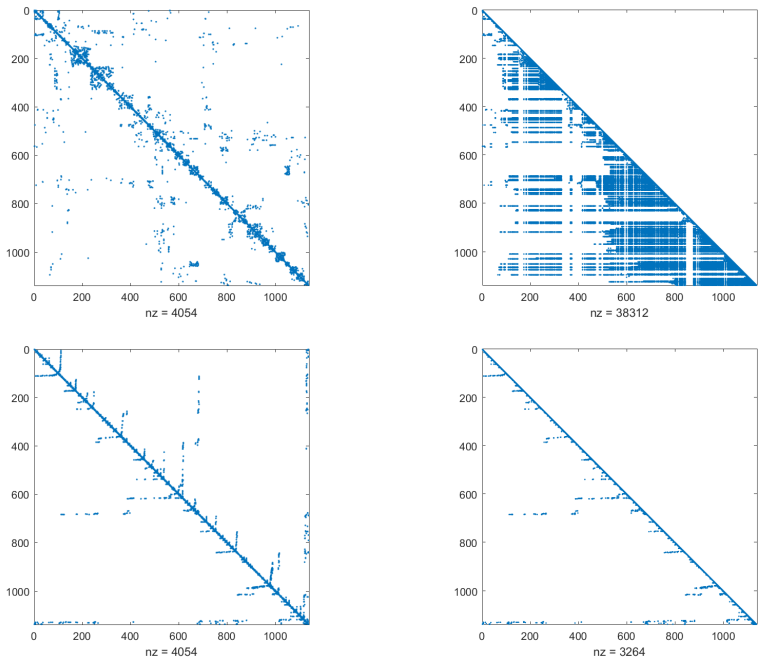


Figura 2: Factorización de Cholesky y algoritmo de mínimo grado

En la Figura 2 tenemos representados (usando el comando `spy` de Matlab) los elementos no nulos de la matriz original, los del factor de Cholesky de la matriz sin reordenar, los elementos no nulos de la matriz reordenada con el algoritmo de mínimo grado PAP^T, y los del factor de Cholesky de la matriz reordenada (de izquierda a derecha y de arriba abajo).

Como podemos observar se obtiene una división por más de 10 del número de los elementos no nulos en el factor triangular inferior de la factorización de Cholesky al reordenar la matriz con el algoritmo de mínimo grado, pasando 38312 a 3264.

2.2. Algoritmo de Cuthill-Mckee inverso

En esta subsección nuestro objetivo es ordenar las columnas y filas de la matriz A de modo que los elementos no nulos de PAP^T se acumulen en torno a la diagonal principal. Como veremos, esta propiedad se mantiene en el factor L de la factorización de Cholesky. Estos métodos se utilizan para reducir el llenado, aunque no son óptimos para este fin. La ventaja que presentan es que los programas involucrados son más simples en términos de almacenamiento e implementación. Como ya sucedió en la subsección anterior, necesitamos introducir algunos conceptos y resultados previos.

Sea A una matriz simétrica definida positiva de tamaño n, con elementos $a_{i,j}$. Para la i-ésima fila de A, $i = 1, \ldots, n$, se definen los indicadores

$$f_i(A) = min\{j : a_{ij} \neq 0\}, \quad \beta_i(A) = i - f_i(A).$$

El valor $f_i(A)$ es simplemente el índice de la columna en la que está la primera (más a la izquierda) componente no nula en la fila i de A y el valor $\beta_i(A)$ se llama el ancho de banda i-ésimo de A. Como los elementos diagonales $a_{i,i}$ son positivos, tenemos

$$f_i(A) \leq i, \quad 0 \leq \beta_i(A).$$

Se define, el **ancho de banda** de una matriz A como el entero

$$\beta(A) = max\{\beta_i(A) : 1 \leq i \leq n\} = max\{|i - j| : \quad a_{ij} \neq 0\},$$

y la **banda** de A como

$$Band(A) = \{(i, j) : 0 < i - j \leq \beta(A)\}$$

Como demostraremos más adelante, se tiene que $Band(A) = Band(L + L^T)$,

Teorema 2.3. *El número de operaciones necesarias para hallar la factorización de Cholesky de una matriz A con ancho de banda β, suponiendo que $Band(L + L^T)$ es densa, viene dado por*

$$\frac{1}{2}\beta(\beta + 3)n - \frac{\beta^3}{3} - \beta^2 - \frac{2}{3}\beta.$$

Demostración: La prueba se sigue del Teorema 1.4 y de la observación siguiente

$$\eta(L_{\star i}) = \begin{cases} \beta + 1, & \text{si} \quad 1 \leq i \leq n - \beta, \\ n - i + 1, & \text{si} \quad n - \beta < i \leq n. \end{cases}$$

□

Teorema 2.4. *Sea A una matriz simétrica con ancho de banda β. Entonces el número de operaciones necesarias para resolver el sistema lineal $A\mathbf{x} = \mathbf{b}$, conocido el factor de Cholesky L de A es*

$$2(\beta + 1)n - \beta(\beta + 1).$$

Demostración: El resultado se sigue del Teorema 1.4 y de los valores de $\eta(L_{\star i})$ dados en el teorema anterior. □

Si el ancho de banda $\beta_i(A)$ de las distintas filas de A varía mucho, se puede explotar mejor el carácter disperso de la matriz A si se introduce el concepto de envolvente.
La **envolvente** de A, denotada por $Env(A)$, se define como

$$Env(A) = \{(i, j) : 0 < i - j \leq \beta_i(A)\}.$$

En términos de los índices $f_i(A)$, se puede ver que

$$Env(A) = \{(i, j) : f_i(A) \leq j < i\}.$$

El número de elementos de la envolvente de A se denota por $|Env(A)|$, y viene dado por

$$|Env(A)| = \sum_{i=1}^{n} \beta_i(A).$$

La Figura 3 muestra la banda y la envolvente de una matriz que aparecerá más adelante al describir el algoritmo de Cuthill-Mckee inverso.

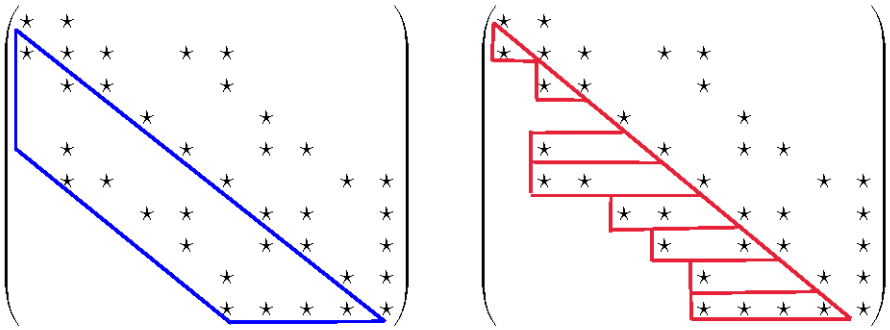


Figura 3: Ejemplo de Banda (azul) y Envolvente (rojo) de una matriz

Lema 2.1. *Se verifica que si $A = LL^T$ es la factorización de Cholesky de A, entonces*

$$Env(A) = Env(L + L^T).$$

Demostración: Ver [2], Capítulo 4, página 53.

Teorema 2.5.

$$Env(A) \subset Band(A).$$

Demostración: Se sigue de las definiciones de banda y envolvente. □

Se introduce ahora la noción de **anchura de frente** de modo que se puedan contar las operaciones de una forma más simple. Para una matriz A, se define la i-ésima anchura de frente de A como

$$\omega_i(A) = |\{k : k > i \quad \text{y} \quad a_{k,l} \neq 0 \quad \text{para algún} \quad l \leq i\}|$$

El valor $\omega_i(A)$ es el número de filas de la envolvente de A que intersecan a la columna i-ésima.
La anchura de frente de A es la cantidad

$$\omega(A) = max\{\omega_i(A) : \quad 1 \leq i \leq n\}.$$

Lema 2.2. *Se verifica que*

$$|Env(A)| = \sum_{i=1}^{n} \omega_i(A).$$

Teorema 2.6. *Si sólo se explotan los 0's que se encuentran fuera de la envolvente de A, el número de operaciones requeridas para factorizar A en LL^T viene dado por*

$$\frac{1}{2} \sum_{i=1}^{n} \omega_i(A)(\omega_i(A) + 3).$$

Demostración: Si se trata la envolvente de A como densa, el número de elementos no nulos en $L_{\star i}$ es simplemente $\omega_i(A) + 1$. El resultado se sigue entonces del Teorema 1.4. □

Descripción del algoritmo

El algoritmo de Cuthill-Mckee inverso proporciona una ordenación que intenta minimizar el ancho de banda donde se concentran los elementos no nulos de la matriz simétrica A y el número de elementos de la envolvente. Para ello, se tiene en cuenta la siguiente observación: si consideramos el grafo no dirigido como la representación de una matriz simétrica, si y es un nodo ya escogido y z un adyacente suyo, para minimizar la anchura de banda, es claro que queremos ordenar z lo antes posible después de y.

Antes de describir el algoritmo de Cuthill-Mckee inverso (RCM), hay que tener en cuenta que este algoritmo trabaja con grafos conexos, por lo que si el grafo asociado a una matriz simétrica A presenta varias componentes conexas, habrá que llevar a cabo el algoritmo RCM en cada una de ellas para llegar a la ordenación final de la matriz.

A continuación describimos el algoritmo RCM en un grafo conexo G de orden n.

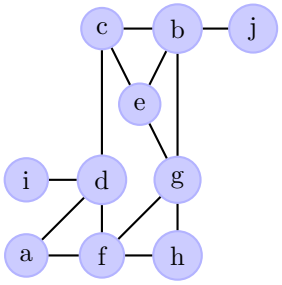
Algoritmo 3. Algoritmo de Cuthill-Mckee inverso

---

1. Determinar el nodo de comienzo r y guardarlo en x_1, $x_1 \leftarrow r$;
2. Bucle principal: for $i = 1, \ldots, n-1$, encontrar todos los nodos adyacentes a x_i no numerados y numerarles en orden creciente de grado;
3. Invertir el orden: La secuencia final es $y = (y_1, \ldots, y_n)$, donde $y_i = x_{n-i+1}$, $i = 1, \ldots, n$.

Veamos ahora sobre un ejemplo el funcionamiento del algoritmo. Consideramos el grafo G^A de la figura y supongamos que escogemos el nodo con la etiqueta g como nodo de comienzo, es decir, $x_1 \leftarrow g$. Veamos ahora cómo se determinan los nodos x_i en los sucesivos pasos.

$$A = \begin{pmatrix} \star & & & \star & & \star & & & \\ & \star & \star & & \star & & \star & & \star \\ & \star & \star & & \star & & & & \\ \star & & & \star & & \star & & \star & \\ & \star & \star & & \star & & \star & & \\ \star & & & \star & & \star & \star & \star & \\ & \star & & & \star & \star & \star & \star & \\ & & & & & \star & \star & \star & \\ & & & \star & & & & & \star \\ & \star & & & & & & & \star \end{pmatrix}, \qquad G^A =$$



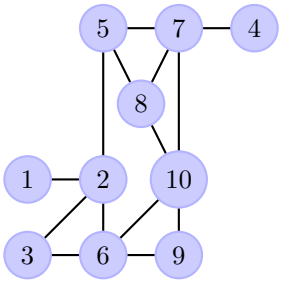
En el Cuadro 3 se ve el orden en el que se numeran los nodos en los sucesivos pasos. Se ha indicado entre paréntesis el grado de cada nodo en el grafo. La última parte del algoritmo es reodenar estos x_i en orden inverso: $y = (i, d, a, j, c, f, b, e, h, g)$.

i	Nodo j_i	Nodos adyacentes no numerados en orden creciente de grado
1	g(4)	h(2),e(3),b(3),f(4)
2	h(2)	-
3	e(3)	c(3)
4	b(3)	j(1)
5	f(4)	a(2),d(4)
6	c(3)	-
7	j(1)	-
8	a(2)	-
9	d(4)	i(1)
10	i(1)	-

Cuadro 3: Ordenación de los nodos de G^A con el algoritmo RCM

Si comparamos la estructura de elementos no nulos en la matriz original A y en la reodenada $B = PAP^T$, observamos que en la matriz reordenada el ancho de banda es $\beta(PAP^T) = 4$ y $|Env(PAP^T)| = 22$ (ver también Figura 3), mientras que la matriz original tenía $\beta(A) = 8$ y $|Env(A)| = 32$.

$$B = PAP^T = \begin{pmatrix} \star & \star & & & & & & & & \\ \star & \star & \star & & \star & \star & & & & \\ & \star & \star & & & \star & & & & \\ & & & \star & & & \star & & & \\ & \star & & & \star & & \star & \star & & \\ & \star & \star & & & \star & & & \star & \star \\ & & & \star & \star & & \star & \star & & \star \\ & & & & \star & & \star & \star & & \star \\ & & & & & \star & & & \star & \star \\ & & & & & \star & \star & \star & \star & \star \end{pmatrix}, \qquad G^B = \quad$$



El primer paso del algoritmo de Cuthill-Mckeeinverso es determinar el nodo de comienzo r. El problema de encontrar un nodo de comienzo con el que recorrer el grafo conexo realizando el algoritmo RCM no es trivial, y de su correcta resolución depende la efectividad del algoritmo, es decir, escogiendo un nodo de comienzo apropiado tendremos una reducción óptima de la banda de la matriz.

El problema consiste en encontrar el par de nodos dentro del grafo que se encuentran entre sí a una distancia máxima ó casi máxima. La experiencia nos indica que ambos nodos son buenos candidatos para comenzar el algoritmo RCM.

Vamos a introducir algunos términos necesarios para describir el algoritmo de búsqueda de ese primer nodo

- Un **camino** de longitud k de x_0 a x_k es un conjunto ordenado de vértices distintos $(x_0, x_1, \ldots, x_k)$ donde $x_i \in Adj_G(x_{i+1})$, para $0 \leq i \leq k-1$.
- La **distancia** $d(x, y)$ **entre dos nodos** x e y en el grafo conexo G es la longitud del camino mínimo que une x e y.

- La **excentricidad de un nodo** x se define como

$$l(x) = max\{d(x,y) : y \in G\}.$$

- El **diámetro de un grafo** G es

$$\delta(G) = max\{l(x) : x \in G\} = max\{d(x,y) : x,y \in G\}.$$

- Un **nodo** x es **periférico** si su excentricidad es igual al diámetro del grafo G, es decir $l(x) = \delta(G)$.

- La **estructura de niveles** de un grafo conexo G cuando se parte del nodo x es una partición del conjunto de nodos del grafo

$$L(x) = \{L_0(x), L_1(x), \ldots, L_{l(x)}(x)\},$$

donde $L_0(x) = \{x\}$, $L_1(x) = \{Adj_G(L_0(x))\}$, y

$$L_i(x) = Adj_G(L_{i-1}(x)) - L_{i-2}(x),$$

para i con $2 \leq i \leq l(x)$.

A $l(x)$ se le denomina longitud de la estructura de niveles $L(x)$, y la anchura de $L(x)$ se define como

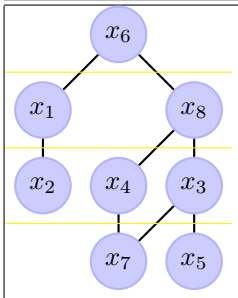
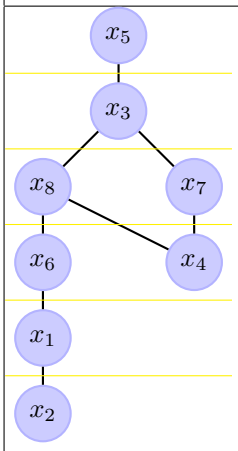
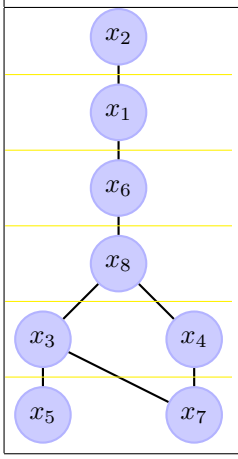
$$w(x) = max\{|L_i(x)|, \quad 0 \leq i \leq l(x)\}.$$

En el Cuadro 4 se muestra la estructura de niveles de un mismo grafo cuando se parte de tres nodos distintos x_6, x_5 y x_2. En el caso del grafo que está en la primera fila, podemos observar las distintas capas o niveles: $L_0(x) = \{x_6\}$, $L_1(x) = \{x_1, x_8\}$, $L_2(x) = \{x_2, x_4, x_3\}$, $L_3(x) = \{x_7, x_5\}$. Ademas, $l(x_6) = w(x_6) = 3$.

Nuestro objetivo ahora es encontrar nodos con alta excentricidad. El caso óptimo sería encontrar un nodo periférico (existen algoritmos que lo consiguen, pero son extremadamente costosos a nivel de implementación), pero con el algoritmo que presentamos a continuación no está asegurado el poder hallarlos. A lo que llegaremos será a un nodo con alta excentricidad. Los nodos producidos por el Algoritmo 4 reciben el nombre de **nodos pseudo-periféricos**.

Algoritmo 4. Búsqueda de un nodo pseudo-periférico

1. Escoger un nodo arbitrario r en G;
2. Construir la estructura de niveles $L(r)$ partiendo de r;
3. Escoger un nodo x en $L_{l(r)}(r)$ de mínimo grado
4. a.-Construir la estructura de niveles $L(x)$ partiendo de x
   b.- si $l(x) > l(r)$, $r \leftarrow x$ y volver al paso 3.;
5. Si $l(x) \geq l(r)$, x es un nodo pseudo-periférico.

Estructura de niveles	Nodo inicial r	$l(r)$	$w(r)$	$r \leftarrow x$
	x_6	3	3	x_5
	x_5	5	2	x_2
	x_2	5	2	

Cuadro 4: Estructura de niveles de un grafo y algoritmo de búsqueda de un nodo pseudo-periférico

El Cuadro 4, además de mostrar la estructura de niveles de un grafo partiendo de distintos nodos, recoge también el funcionamiento del Algoritmo 4. La primera línea muestra la estructura de niveles del paso 2 del Algoritmo 4 con $r = x_6$ (cada nivel separado por una línea amarilla), y las filas restantes ilustran los pasos 3 y 4 de dicho algoritmo. En los tres casos se incluye la longitud y la anchura de la estructura de niveles y la actualización de nodos. Como resultado se obtiene el nodo pseudo-periférico x_2, que es el que se debería utilizar como nodo de comienzo para el algoritmo RCM en caso de que se fuese a aplicar a una matriz con dicho grafo no dirigido.

Para finalizar esta subsección, veamos cómo trabaja el algoritmo de Cuthill-Mckee inverso de reducción de banda (implementado en el comando `symrcm` en Matlab y también en la función `RCM.m` del Apartado 4.2.2 del Apéndice) sobre la matriz real simétrica definida positiva considerada ya al estudiar el algoritmo de mínimo grado, y las implicaciones que conlleva al calcular la factorización de Cholesky de dicha matriz. Recordamos que se trata de una matriz 1138×1138 con 4054 elementos no nulos. En la Figura 4 están representados los elementos no nulos de la matriz original (izquierda), los de la matriz reordenada PAP^T (centro) con el algoritmo RCM y los del factor de Cholesky de la matriz reordenada (derecha). En este ejemplo concreto observamos que el número de elementos no nulos en el factor de Cholesky tras utilizar esta reordenación es ligeramente mayor (5222) que el obtenido con el algoritmo de mínimo grado (3264).

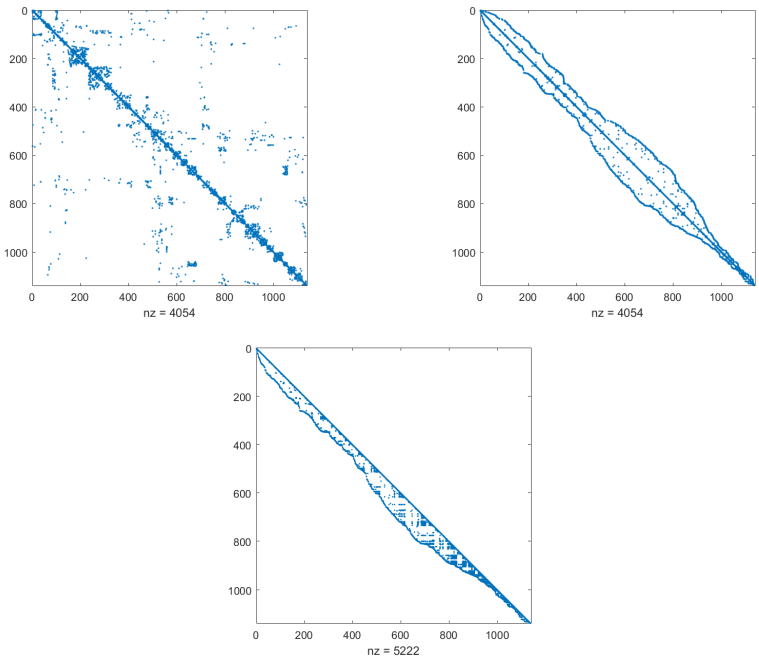


Figura 4: Factorización de Cholesky y algoritmo RCM

2.3. Ejemplos de aplicación

2.3.1. Discretización de la ecuación del calor

En esta subsección consideramos una primera aplicación de los métodos que hemos estudiado a lo largo de esta memoria. Nuestro objetivo es analizar la evolución de la temperatura de una placa cuadrada de lado unidad, con temperatura conocida en los cuatro lados, y sometida a unas fuentes de calor intermitentes. Para ello consideramos la ecuación del calor

$$\begin{cases} u_t(x,y,t) = K(u_{xx}(x,y,t) + u_{yy}(x,y,t)) + f(x,y,t), \\ u(x,y,0) = u_0(x,y), \\ u(0,y,t) = g_1(y,t), \\ u(1,y,t) = g_2(y,t), \\ u(x,0,t) = g_3(x,t), \\ u(x,1,t) = g_4(x,t), \end{cases}$$

donde K es la difusividad térmica del material del que está hecha la placa, $(x,y) \in [0,1] \times [0,1]$, $t \geq 0$, y u_0, f, g_1, g_2, g_3 y g_4 son funciones conocidas.

Para discretizar esta ecuación recurrimos al método de diferencias finitas en dos dimensiones sobre una malla equiespaciada con nodos (x_i, y_j), $0 \leq i,j \leq N$ con $x_i = i\Delta x$, $y_j = j\Delta y$ con $\Delta x = \Delta y = 1/N$. La discretización de las derivadas espaciales se realiza de la siguiente forma

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)}{(\Delta x)^2}$$

y de forma análoga para la variable y. Denotando por $u_{i,j}(t)$ la aproximación a $u(x_i, y_j, t)$, $1 \leq i,j \leq N-1$, tras la discretización espacial se llega a un sistema diferencial de la forma

$$\frac{d}{dt}u_{i,j}(t) = K\left[\frac{u_{i+1,j}(t) + u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t) - 4u_{i,j}(t)}{(\Delta x)^2}\right] + f(x_i, y_j, t).$$

Si dicho sistema diferencial se integra numéricamente con el método de Euler implícito, conociendo la temperatura en los nodos de la malla en el tiempo $t_n = n\Delta t$, la temperatura en los nodos de la malla en el tiempo $t_{n+1} = t_n + \Delta t$ viene dada por la solución del sistema lineal

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} + \Delta t K\left[\frac{u_{i+1,j}^{(n+1)} + u_{i-1,j}^{(n+1)} + u_{i,j+1}^{(n+1)} + u_{i,j-1}^{(n+1)} - 4u_{i,j}^{(n+1)}}{(\Delta x)^2}\right] + \Delta t f_{i,j}^{(n+1)},$$

para $1 \leq i,j \leq N-1$, donde $u_{i,j}^{(n)} \approx u(x_i, y_j, t_n)$, es la aproximación a la temperatura del nodo (x_i, y_j) en el instante t_n y $f_{i,j}^{(n+1)} = f(x_i, y_j, t_{n+1})$. Los valores $u_{i,j}^{(n+1)}$ para $i = 0, N$ y $j = 0, N$ se toman de la condición frontera y son conocidos para todo n.

En la Figura 5 se ha representado la malla correspondiente a $N = 4$, es decir, $\Delta x = \Delta y = \frac{1}{4}$.

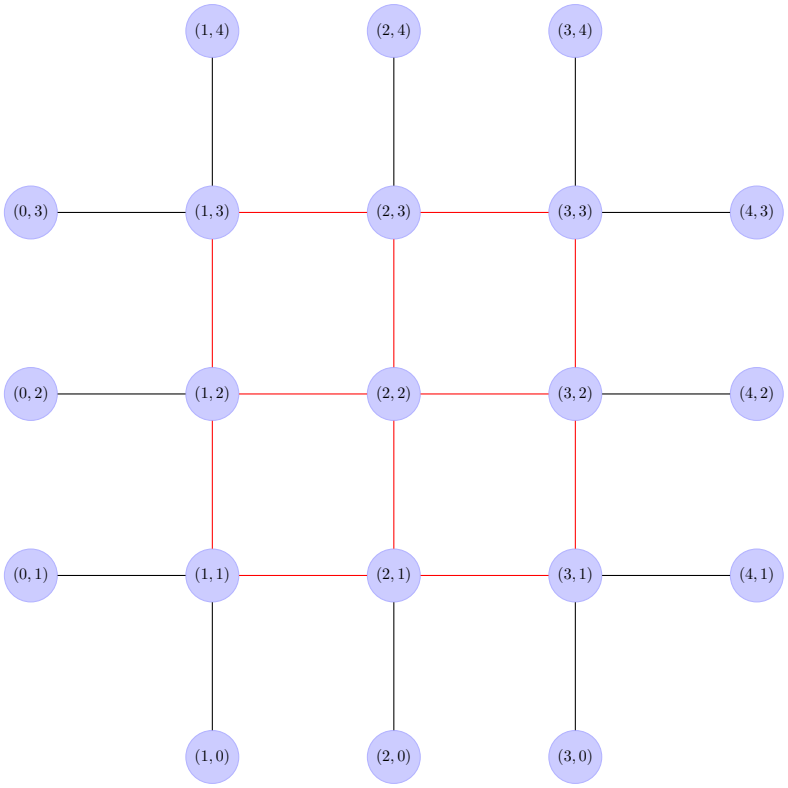


Figura 5: Malla equispaciada de nodos (x_i, y_j) con $x_i = i\Delta x$, $y_j = j\Delta y$, $\Delta x = \Delta y = 1/4$

Los nodos en los que hay que aproximar la temperatura son los nueve nodos interiores (unidos en la gráfica por líneas de color rojo), ya que la temperatura en los nodos de la frontera es conocida por la condición Dirichlet. Para $N = 4$, hay que resolver, por tanto, un sistema de nueve ecuaciones con nueve incógnitas en cada nivel de tiempos..

Ordenando las incógnitas utilizando el orden natural (de izquierda a derecha y de abajo hacia arriba) se introduce el vector de incógnitas

$$\mathbf{U}^{(n)} = [u_{11}^{(n)}, u_{21}^{(n)}, u_{31}^{(n)}, \ldots, u_{33}^{(n)}]^T$$

y se resuelve para cada n, el sistema lineal

$$\left(I_M - \frac{k\Delta t}{(\Delta x)^2} A\right) \mathbf{U}^{(n+1)} = \mathbf{U}^{(n)} + \frac{k\Delta t}{(\Delta x)^2} \mathbf{U}_{\Omega}^{(n+1)} + \Delta t \mathbf{F}^{(n+1)} \tag{2.1}$$

donde $M = 9$,

$$\begin{aligned} \mathbf{U}_{\partial\Omega}^{(n+1)} \quad = \quad & [u_{1,0}^{(n+1)} + u_{0,1}^{(n+1)}, u_{2,0}^{(n+1)}, u_{3,0}^{(n+1)} + u_{4,1}^{(n+1)}, \\ & u_{0,2}^{(n+1)}, 0, u_{4,2}^{(n+1)}, \\ & u_{0,3}^{(n+1)} + u_{1,4}^{(n+1)}, u_{2,4}^{(n+1)}, u_{3,4}^{(n+1)} + u_{4,3}^{(n+1)}]^T \end{aligned}$$

$$\mathbf{F}^{(n+1)} = [f_{1,1}^{(n+1)}, f_{2,1}^{(n+1)}, f_{3,1}^{(n+1)}, f_{1,2}^{(n+1)}, f_{2,2}^{(n+1)}, f_{3,2}^{(n+1)}, f_{1,3}^{(n+1)}, f_{2,3}^{(n+1)}, f_{3,3}^{(n+1)}]^T$$

y

$$A = \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix}$$

En general, si $\Delta x = \frac{1}{N}$, introduciendo los vectores de $\mathbb{R}^{N-1}$

$$\begin{aligned} \mathbf{v}_1 &= [u_{0,1}^{(n+1)} + u_{1,0}^{(n+1)}, u_{2,0}^{(n+1)}, u_{3,0}^{(n+1)}, \ldots, u_{N-2,0}^{(n+1)}, u_{N-1,0}^{(n+1)} + u_{N,1}^{(n+1)}] \\ \mathbf{v}_j &= [u_{0,j}^{(n+1)}, 0, \ldots, 0, u_{N,j}^{(n+1)}] \qquad 2 \le j \le N-2, \\ \mathbf{v}_{N-1} &= [u_{0,N-1}^{(n+1)} + u_{1,N}^{(n+1)}, u_{2,N}^{(n+1)}, u_{3,N}^{(n+1)}, \ldots, u_{N-2,N}^{(n+1)}, u_{N-1,N}^{(n+1)} + u_{N,N-1}^{(n+1)}] \end{aligned}$$

se construye el vector con los datos de la frontera

$$\mathbf{U}_{\partial\Omega}^{(n+1)} = [\mathbf{v}_1^T, \mathbf{v}_2^T, \cdots, \mathbf{v}_{N-2}^T, \mathbf{v}_{N-1}^T]^T,$$

y denotando

$$\begin{aligned} \mathbf{U}^{(n)} &= [u_{1,1}^{(n)}, \ldots, u_{N-1,1}^{(n)}, u_{1,2}^{(n)}, \ldots, u_{N-1,2}^{(n)}, \ldots, u_{1,N-1}^{(n)}, \ldots, u_{N-1,N-1}^{(n)}]^T \\ \mathbf{U}^{(n+1)} &= [u_{1,1}^{(n+1)}, \ldots, u_{N-1,1}^{(n+1)}, u_{1,2}^{(n+1)}, \ldots, u_{N-1,2}^{(n+1)}, \ldots, u_{1,N-1}^{(n+1)}, \ldots, u_{N-1,N-1}^{(n+1)}]^T \\ \mathbf{F}^{(n+1)} &= [f_{1,1}^{(n+1)}, \ldots, f_{N-1,1}^{(n+1)}, f_{1,2}^{(n+1)}, \ldots, f_{N-1,2}^{(n+1)}, \ldots, f_{1,N-1}^{(n+1)}, \ldots, f_{N-1,N-1}^{(n+1)}]^T \end{aligned}$$

el sistema que hay que resolver para hallar $\mathbf{U}^{(n+1)}$ a partir de $\mathbf{U}^{(n)}$ es (2.1), con $M = (N-1)^2$, y

$$A = \begin{pmatrix} B & I_{N-1} & 0 & \cdots & \cdots & \cdots & 0 \\ I_{N-1} & B & I_{N-1} & 0 & \cdots & \cdots & 0 \\ 0 & I_{N-1} & B & I_{N-1} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I_{N-1} & B & I_{N-1} & 0 \\ 0 & \cdots & 0 & 0 & I_{N-1} & B & I_{N-1} \\ 0 & \cdots & 0 & 0 & 0 & I_{N-1} & B \end{pmatrix}$$

donde

$$B = \begin{pmatrix} -4 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & -4 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 \end{pmatrix}$$

La matriz del sistema lineal es claramente dispersa puesto que en cada ecuación intervienen a lo sumo 5 incógnitas. Aunque se puedan utilizar métodos específicos para "matrices banda", hay que notar que la anchura de la banda aumenta con N, por lo que nos planteamos el ver si con alguna de las reordenaciones que se han considerado en esta memoria, los resultados pueden mejorar.

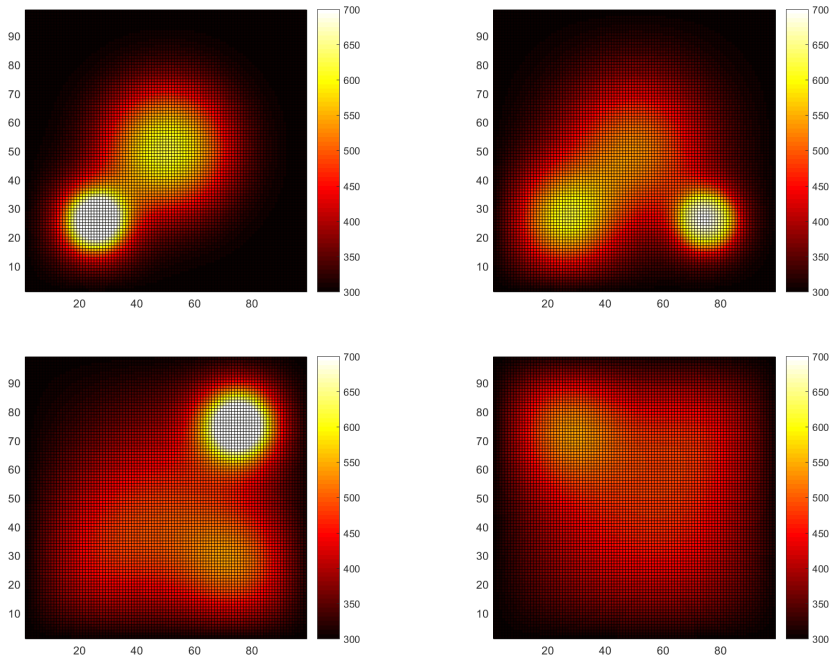


Figura 6: Evolución de la temperatura de una placa cuadrada

Para ello consideramos un ejemplo concreto, en el que $K = 4 \times 10^{-3}$. Como condición inicial se toma

$$u(x,y,0) = \begin{cases} 300, & \text{si } (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 \geq \frac{1}{5^2} \\ 700, & \text{si } (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 < \frac{1}{5^2}, \end{cases}$$

y como fuente de calor se elige

$$f(x,y,t) = \begin{cases} 500, & \text{si } (x-\frac{1}{4})^2 + (y-\frac{1}{4})^2 < 0.01; \quad t < 2 \\ 500, & \text{si } (x-\frac{1}{4})^2 + (y-\frac{3}{4})^2 < 0.01; \quad 2 \leq t < 4 \\ 500, & \text{si } (x-\frac{3}{4})^2 + (y-\frac{3}{4})^2 < 0.01; \quad 4 \leq t < 6 \\ 500, & \text{si } (x-\frac{3}{4})^2 + (y-\frac{1}{4})^2 < 0.01; \quad 6 \leq t < 8 \\ 0, & \text{en otro caso.} \end{cases}$$

Se fijan las condiciones frontera Dirichlet $g_1(y,t) = g_2(y,t) = g_3(x,t) = g_4(x,t) = 300$.

Para la discretización se ha tomado $N = 100$ (99^2 nodos internos), y $\Delta t = 0.1$. Se ha representado en la Figura 6 la temperatura de la placa en $t = 1, 3, 6, 10$.

Veamos ahora una comparativa de tiempos usando distintas técnicas para resolver el sistema lineal que nos aparece en (2.1):

En el ordenador que se ha utilizado, se podría trabajar en mallas con $N = 170$, como máximo, cuando tratamos la matriz como densa, cosa que no ocurre cuando estamos trabajando con matrices dispersas, ya que el espacio en memoria se reduce de forma drástica y podríamos hallar resultados en mallas de tamaños mucho mayores.

Consideremos simplemente mallas con, como máximo, $N = 80$, y calculemos el tiempo que se tarda en hallar la temperatura sobre la malla en el tiempo $t = 1$ tratando la matriz como densa y como dispersa (con intervalos temporales $\Delta t = 0.1, 0.01$). Los resultados se han recogido en el Cuadro 5.

N	Δt	Denso	Disperso
10	0.1	0.0026	0.0022
20	0.1	0.0174	0.0064
40	0.1	0.5617	0.0186
50	0.1	1.4957	0.0311
80	0.1	21.962	0.0311
10	0.01	0.018	0.0139
20	0.01	0.1388	0.0442
40	0.01	4.1612	0.2332
50	0.01	13.6226	0.318
80	0.01	$>$ 1min	1.041

Cuadro 5: Comparativa del tiempo de CPU requerido para resolver el problema de la temperatura de la placa

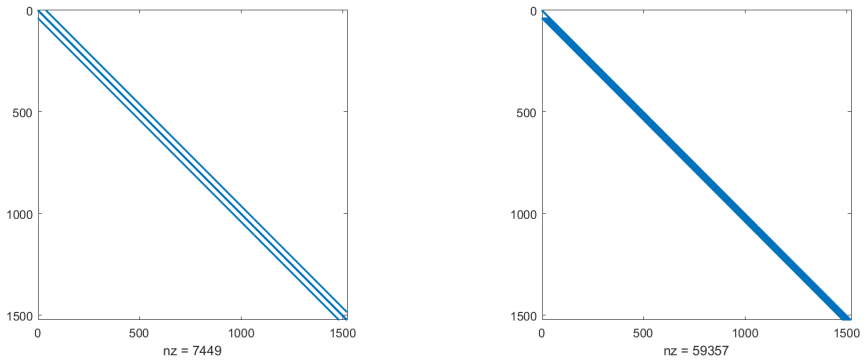


Figura 7: Elementos no nulos en A (izqda.) y en el factor de Cholesky (dcha.)

Tomamos ahora $N = 40$ y como ejemplo de matriz dispersa la opuesta de la matriz A, $(-A)$, de dimensión $(N-1)^2$, explicada en esta sección, con los bloques B y las identidades de dimensión $N - 1$. En la Figura 7 se tiene la estructura de elementos no nulos de la matriz sin reordenar (izquierda) y su correspondiente factor de Cholesky (derecha), en la Figura 8 se tiene la matriz reordenada usando el algoritmo de mínimo grado y en la Figura

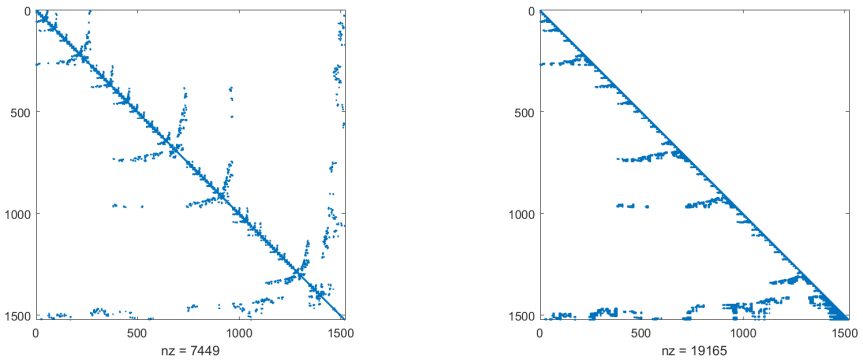


Figura 8: Elementos no nulos en A (izqda.) y en el factor de Cholesky (dcha.) tras utilizar el algoritmo de mínimo grado

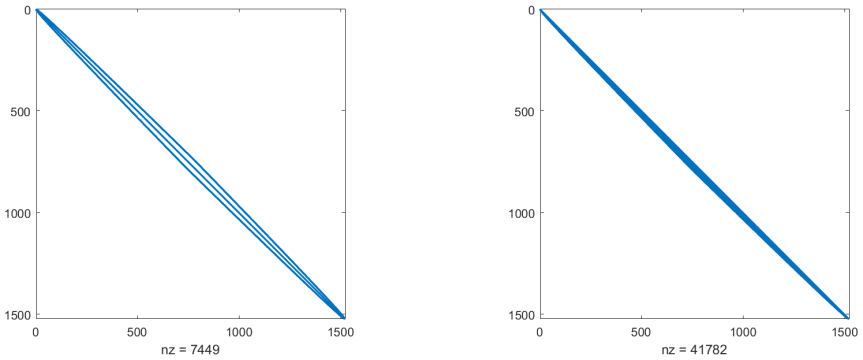


Figura 9: Elementos no nulos en A (izqda.) y en el factor de Cholesky (dcha.) tras utilizar el algoritmo de Cuthill-Mckee inverso

9 se tiene la matriz reordenada usando el algoritmo de Cuthill-Mckee inverso. Se puede obsevar la reducción del número de los elementos no nulos en el factor de Cholesky cuando utilizamos las reordenaciones frente a la matriz sin reordenar. Esta reducción es mayor cuando utilizamos el algoritmo de mínimo grado frente al algoritmo de Cuthill-Mckee (unas 3 veces mayor), ya que el algoritmo de mínimo grado proporciona la reordenación óptima para que se produzca el menor "llenado"posible durante la factorización. También se hace visible la reducción de la banda de la matriz cuando utilizamos la reordenación de Cuthill-Mckee (tanto en la matriz reordenada como en el factor de Cholesky, ya que es una propiedad que se mantiene tras la factorización).

2.3.2. Otros ejemplos de matrices dispersas

En este apartado llevaremos a cabo la resolución de sistemas lineales $A\mathbf{x} = \mathbf{b}$, con matriz dispersa de grandes dimensiones, de modo directo, con la factorización LU y con la de Cholesky. Implementaremos los dos métodos de ordenación vistos hasta el momento en

este trabajo. Las matrices test se han tomado de la Suite Sparse Matrix Collection [8], y pretenden ilustrar el comportamiento de los métodos estudiados para matrices dispersas de distintas dimensiones que van desde $n = 479$ hasta $n = 13965$. Más precisamente, se ha trabajado con las matrices denominadas 'west0479', '1138_bus', 'add20', 'Kuu' y 'crystm02', que surgen en distintas aplicaciones que van desde problemas de simulación de procesos químicos hasta problemas relacionados con la teoría de materiales, pasando por la simulación de circuitos eléctricos. La primera y la tercera no son simétricas y para poder aplicar los métodos de reordenación que se han estudiado y su efecto cuando se va a calcular la factorización, se ha trabajado con la matriz que se obtiene multiplicando la matriz original por su matriz traspuesta.

n	Res. densa	LU	Resol.	Min. Gr. + LU	Resol.	RCM + LU	Resol.
479	0.02821	0.03280	0.02163	0.03798	0.03942	0.03565	0.04399
1138	0.05779	0.03409	0.02163	0.02673	0.03698	0.02739	0.03797
2395	0.51869	4.64191	0.05526	0.03222	0.03458	0.03737	0.03955
7102	3.15873	3.50311	0.03706	0.13278	0.03321	0.48583	0.03055
13965	20.32426	0.86528	0.02855	1.23712	0.03817	0.96325	0.03704

Cuadro 6: Tiempo de CPU cuando se utiliza la factorización de LU

n	Res. densa	Cholesky	Resol.	Min. Gr. + Chol.	Resol.	RCM + Chol.	Resol.
479	0.02821	0.02638	0.02038	0.02848	0.03788	0.02696	0.03854
1138	0.05779	0.03152	0.02215	0.02877	0.03404	0.02358	0.03867
2395	0.51869	0.04408	0.07068	0.02777	0.03786	0.02729	0.03971
7102	3.15873	0.02551	0.08668	0.04540	0.03867	0.08979	0.04655
13965	20.32426	1.58065	0.06889	0.16664	0.06958	0.13825	0.06815

Cuadro 7: Tiempo de CPU cuando se utiliza la factorización de Cholesky

En el Cuadro 6 se muestran los resultados cuando se utiliza la factorización LU mientras que el Cuadro 7 corresponde al uso de la factorización de Cholesky. En ambas tablas se han separado los tiempos de CPU necesarios para la factorización matricial de los requeridos para la resolución de los correspondientes sistemas triangulares. Se ha incluido también el tiempo de CPU necesario para resolver el correspondiente sistema lineal cuando la matriz es considerada como densa (columna 2).

Se puede observar que, como se esperaba, el tiempo de factorización para matrices dispersas se ve reducido cuando reordenamos la matriz original, ya que disminuye el 'llenado' que se produce en la factorización y por tanto el número de operaciones realizadas. Se visualiza que cuando tratamos las matrices como dispersas, la variación de tiempos al indicar a Matlab que realice las reordenaciones antes de resolver y al no indicarselo

explícitamente es mínima, debido al hecho que cuando definimos la matriz como dispersa, Matlab ya resuelve los sistemas de forma eficiente (realizando las reordenaciones oportunas).

Se observa con bastante claridad la diferencia en los tiempos de resolución cuando tratamos la matriz como densa o como dispersa, diferencia que se acentúa según aumenta el tamaño de la matriz.

n	$\eta(A)$	Reorden.	$\eta(L)$ de LU	$\eta(L)$ de Chol.
479	7553	Ninguna	32042	30183
479	-	MDO	19647	8285
479	-	RCM	22154	23823
1138	4054	Ninguna	38312	38312
1138	-	MDO	3256	3256
1138	-	RCM	4693	4693
2395	13151	Ninguna	2006049	2006049
2395	-	MDO	9850	9850
2395	-	RCM	19619	19619
7102	340200	Ninguna	2993061	2993061
7102	-	MDO	385826	385826
7102	-	RCM	1024873	102487
13965	322905	Ninguna	2058861	2058900
13965	-	MDO	2025252	2025303
13965	-	RCM	2203371	2203452

Cuadro 8: Cantidad de elementos no nulos en el factor triangular cuando se utilizan las distintas reordenaciones

En el Cuadro 8 se ha incluido la información relativa al número de elementos no nulos de cada matriz y del factor triangular L que se obtiene tanto con factorización LU como con la factorización de Cholesky si se aplican a la matriz o a las matrices reordenadas con los algoritmos de mínimo grado (MDO) y de Cuthill-Mckee inverso (RCM). En todos los casos se observa que la matriz L contiene una cantidad de elementos no nulos notablemente menor cuando se realiza la reordenación antes de la factorización, alcanzando el valor mínimo cuando se utiliza la reordenación de mínimo grado.

3. Una reordenación para el problema del PageRank

En esta última sección se presenta un algoritmo de reordenación específico para las matrices dispersas que se originan al calcular el PageRank, que se introduce en [7] y [6], junto con un par de resultados teóricos.

3.1. Cálculo eficiente del PageRank

Como es sabido, podemos visualizar la web como un grafo dirigido con un gran número de nodos, en el que los nodos representan las páginas web, y los arcos dirigidos del grafo representan los hiperenlaces que hay de unas páginas a otras. Con esta notación de grafos se puede desarrollar el estudio de la importancia de las distintas páginas web y su correspondiente ordenación al realizar una búsqueda en Google (problema del cálculo del PageRank). Coloquialmente hablando, una página con más recomendaciones que otra (entendemos recomendaciones como enlaces que apuntan hacia a ella) debería tener una importancia mayor y aparecer en las primeras posiciones cuando se da respuesta ordenada a una búsqueda. Así es como funciona el marcador de popularidad PageRank de Google. En resumen, una página es importante si está siendo apuntada por otras páginas importantes.

Exponemos a continuación una formulación matemática para el problema de calcular el PageRank.

Si n es el número de nodos en el grafo que representa la estructura de la web (el número de páginas) se introduce la matriz H de tamaño $n \times n$, cuyo elemento (i, j) viene dado por $H_{i,j} = \dfrac{1}{|O_i|}$ si existe un enlace del nodo i al nodo j, y 0 en caso contrario. El escalar $|O_i|$ es el número de enlaces salientes desde la página i. Se tiene entonces que las filas no nulas de la matriz H suman 1. Por otro lado la matriz H contiene filas de ceros para cada nodo que corresponde a una página de la que no salen enlaces (nodos 'dangling'). Estos nodos 'dangling' pueden corresponder a imágenes o pdf's, a una página solo con tablas de datos, etc.

La matriz H es extremadamente dispersa, pues de cada página parten un número de enlaces que puede considerarse pequeño frente al número total de páginas web.

Junto con la matriz H, se introduce un vector fila de tamaño $1 \times n$, que denotamos por π^T y que es conocido como el vector del PageRank. Dicho vector mide la importancia de cada página web, y es un vector estacionario para la cadena de Markov asociada a la matriz H, es decir, es un autovector de la matriz H asociado al autovalor $\lambda = 1$ (ver [7] para más detalles).

Utilizando esta formulación surje un primer problema, y es el debido a que la matriz H no es, en general, estocástica puesto que la suma de sus filas no siempre es 1, por la existencia de filas idénticamente nulas. Para ello Brin y Page, los fundadores de Google, sugieren reemplazar cada fila de ceros de H por un vector de probabilidad denso (un vector cuyas componentes están entre 0 y 1 y cuya suma sea igual a 1) y así crear una matriz H estocástica.

La solución original fue considerar el vector uniforme $\mathbf{e}^T/n$, con $\mathbf{e}^T$ el vector de unos de tamaño n. Más tarde, este vector uniforme fue reemplazado por un vector de probabilidad $\mathbf{v}^T$ conocido como vector de personalización. Para ahorrar almacenamiento, se crea el vector $\mathbf{a}$, con coeficientes $a_i = 1$ si la fila i de la matriz H corresponde a un nodo 'dangling', y 0 en caso contrario. Con esta notación, la nueva matriz S que se origina a

partir de H se puede escribir como

$$S = H + \mathbf{a}\mathbf{v}^T.$$

Se podría ya pasar a la búsqueda del vector estacionario para esta matriz estocástica e irreducible S, pero se debe realizar un último ajuste con el fin de evitar problemas de unicidad del vector π^T. Este último ajuste consiste en añadir la matriz de 'perturbación' $\mathbf{e}\mathbf{v}^T$ que crea conexiones directas entre cada página. El nuevo término añadido se interpreta como la probabilidad de que el usuario, cuando está navegando por la web, pueda decidir no hacer caso a los enlaces salientes que hay en la página web en la que se encuentra y opta por visitar cualquier otra página de la red.

Tras estas consideraciones, se llega a la matriz estocástica de Google

$$G = \alpha S + (1-\alpha)\mathbf{e}\mathbf{v}^T = \alpha H + (\alpha\mathbf{a} + (1-\alpha)\mathbf{e})\mathbf{v}^T, \qquad 0 < \alpha < 1, \tag{3.1}$$

donde α controla la prioridad que se da a la estructura de enlaces original de la web (H) frente a la idea de que desde cualquier página se puede acceder a cualquier otra, teniendo en cuenta las preferencias del usuario $\mathbf{v}^T$.

Una primera forma de hallar el vector del Page Rank π^T se basa en interpretarlo como el autovector asociado al autovalor dominante 1 de la matriz G, y en aproximarlo mediante el método de la potencia. Se llega entonces a la formulación iterativa

$$\begin{aligned}
\mathbf{x}^{(k)T} &= \mathbf{x}^{(k-1)T}G \\
&= \alpha\mathbf{x}^{(k-1)T}S + (1-\alpha)\mathbf{x}^{(k-1)T}\mathbf{e}\mathbf{v}^T \\
&= \alpha x^{(k-1)T}S + (1-\alpha)\mathbf{v}^T \\
&= \alpha\mathbf{x}^{(k-1)T}H + \alpha(\mathbf{x}^{(k-1)T}\mathbf{a})\mathbf{v}^T + (1-\alpha)\mathbf{v}^T \\
&= \alpha\mathbf{x}^{(k-1)T}H + (\alpha\mathbf{x}^{(k-1)T}\mathbf{a} + (1-\alpha))\mathbf{v}^T.
\end{aligned}$$

Se sabe (ver [6]) que la razón de convergencia asintótica del método de la potencia aplicado a la matriz de Google es la misma con l que $\alpha^k \to 0$ cuando $k \to \infty$. Como Google usa $\alpha = 0.85$, se pueden esperar como máximo unas 114 iteraciones hasta que se consigue una diferencia entre dos aproximaciones consecutivas inferior a 10^{-8}. Esto requeriría un número de operaciones proporcional a $114(\eta(H))$, donde $\eta(H)$ es el número de elementos no nulos en la matriz original H. El costo computacional cuando tenemos millones de páginas es de días.

Una segunda alternativa, que como veremos más adelante permite explotar la existencia de nodos 'dangling', pasa por notar que el vector π^T cumple $\pi^T = \pi^T G$, junto con $\pi^T\mathbf{e} = 1$. Se trata, por tanto, de hallar la solución de un gran sistema lineal con matriz muy dispersa. Esta segunda interpretación del problema permite relacionarlo con el objetivo de este trabajo.

Teorema 3.1. *Resolver el sistema lineal*

$$\mathbf{x}^T(I - \alpha H) = \mathbf{v}^T, \tag{3.2}$$

y tomar $\pi^T = \mathbf{x}^T/\mathbf{x}^T\mathbf{e}$*, produce el vector del PageRank.*

Demostración: Sabemos que el vector del PageRank π^T debe satisfacer $\pi^T G = \pi^T$ y $\pi^T\mathbf{e} = 1$. La segunda propiedad se deduce trivialmente si se normaliza el vector $\mathbf{x}^T$ como

indica el enunciado del teorema. Para demostrar que si se cumple (3.2) se tiene la primera condición, hay que tener en cuenta que demostrar que $\pi^T G = \pi^T$, es equivalente a probar que $\pi^T(I-G) = \mathbf{0}^T$, y esto a su vez es equivalente a probar que $\mathbf{x}^T(I-G) = \mathbf{0}^T$. Ahora bien, usando (3.1) y teniendo en cuenta que $S = H + \mathbf{a}\mathbf{v}^T$ se sigue

$$\begin{aligned}\mathbf{x}^T(I-G) &= \mathbf{x}^T(I - \alpha H - \alpha\mathbf{a}\mathbf{v}^T - (1-\alpha)\mathbf{e}\mathbf{v}^T)\\ &= \mathbf{x}^T(I-\alpha H) - \mathbf{x}^T(\alpha\mathbf{a} + (1-\alpha)\mathbf{e})\mathbf{v}^T\\ &= \mathbf{v}^T - \mathbf{v}^T.\end{aligned}$$

La última igualdad $\mathbf{x}^T(\alpha\mathbf{a} + (1-\alpha)\mathbf{e}) = 1$, es cierta debido a que por (3.2), por la forma en que se ha definido $\mathbf{a}$, y por ser $\mathbf{v}$ un vector de probabilidad se tiene

$$\begin{aligned}1 = \mathbf{v}^T\mathbf{e} &= \mathbf{x}^T(I-\alpha H)\mathbf{e} = \mathbf{x}^T\mathbf{e} - \alpha\mathbf{x}^T H\mathbf{e}\\ &= \mathbf{x}^T\mathbf{e} - \alpha\mathbf{x}^T(\mathbf{e}-\mathbf{a}) = (1-\alpha)\mathbf{x}^T\mathbf{e} + \alpha\mathbf{x}^T\mathbf{a}.\end{aligned}$$

La matriz de coeficientes $(I-\alpha H)$ del sistema lineal (3.2) satisface varias propiedades interesantes ([6]):

- $(I-\alpha H)$ es no singular.
- La suma de los elementos no nulos de la fila i-ésima de $(I-\alpha H)^{-1}$ es igual a uno si i corresponde a un nodo 'dangling', y menor o igual que $\dfrac{1}{1-\alpha}$ en otro caso.
- La fila de $(I-\alpha H)^{-1}$ correspondiente al nodo 'dangling' i es $\mathbf{e}_i^T$, donde $\mathbf{e}_i$ es la $i-$ésima columna de la matriz identidad.

La última propiedad permite que el cálculo del vector del PageRank se pueda realizar de un modo especialmente eficiente. Supongamos que las filas y columnas de H se permutan, de modo que las filas correspondientes a nodos 'dangling' se encuentren en la parte inferior de la matriz

$$PHP^T = \begin{array}{c} \\ n_1 \\ n_2 \end{array}\begin{array}{c} \begin{array}{cc} n_1 & n_2 \end{array} \\ \begin{pmatrix} H_{11} & H_{12} \\ 0 & 0 \end{pmatrix} \end{array} \tag{3.3}$$

donde n_1 es el cardinal del conjunto de nodos no 'dangling' y n_2 número de nodos 'dangling'.

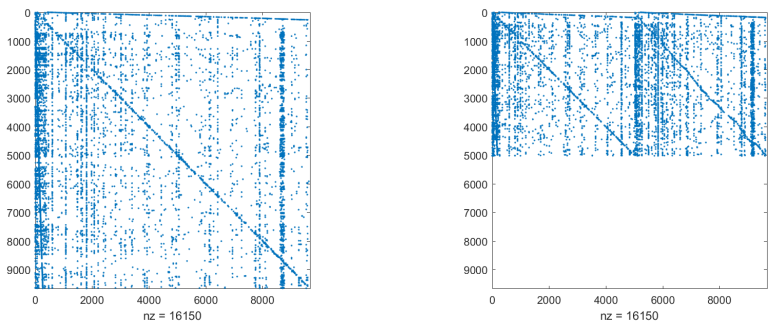


Figura 10: Ilustración de (3.3) para una matriz

Veamos un ejemplo de esta reordenación de las filas y columnas en una matriz 9664×9664 que representa la búsqueda de la palabra 'california' en la web ([6],[9]). La correspondiente matriz H tiene 16150 elementos no nulos, de los casi cien millones de elementos que tiene. Se puede interpretar como un grafo dirigido con 9664 nodos y 16150 aristas dirigidas. El número de nodos 'dangling' es 5027.
Cuando estudiamos el sistema lineal asociado a la matriz reordenada se puede ver que

$$(I - \alpha PHP^T) = \begin{pmatrix} I - \alpha H_{11} & I - \alpha H_{12} \\ 0 & I \end{pmatrix},$$

y para la inversa

$$(I - \alpha PHP^T)^{-1} = \begin{pmatrix} (I - \alpha H_{11})^{-1} & \alpha(I - \alpha H_{11})^{-1} H_{12} \\ 0 & I \end{pmatrix}.$$

Por lo tanto, el vector no normalizado $\mathbf{x}^T = \mathbf{v}^T(I - \alpha PHP^T)^{-1}$ puede escribirse como

$$\mathbf{x}^T = (\mathbf{v}_1^T(I - \alpha H_{11})^{-1} \quad | \quad \alpha \mathbf{v}_1^T(I - \alpha H_{11})^{-1} H_{12} + \mathbf{v}_2^T),$$

donde $\mathbf{v}_1^T$ contiene las primeras n_1 componentes de $\mathbf{v}^T$ (la parte correspondiente a los nodos que no son 'dangling'), y $\mathbf{v}_2^T$ corresponde a las últimas n_2 componentes de $\mathbf{v}^T$ (asociadas a los nodos 'dangling'). Naturalmente, no se calcularán explícitamente las matrices inversas que aparecen en la expresión anterior, sino que se resolverán los correspondientes sistemas lineales. En el ejemplo de la Figura 10 se han representado los elementos no nulos de la matriz original (izquierda) y los de la matriz reordenada cuando los nodos 'dangling' se pasan a las últimas posiciones (derecha). Observamos que en este ejemplo $n_1 < 5000$, por lo que se ha reducido a menos de la mitad el tamaño del sistema lineal que hay que resolver, pasando de $n = 9664$ a $n_1 = 4637$.

Esta ordenación proporciona un nuevo algoritmo para calcular el vector del PageRank mucho más eficiente

---

Algoritmo 1 para el cálculo del PageRank

---

1. Reordenar la matriz H utilizando los nodos 'dangling';
2. Resolver $\mathbf{x}_1^T(I - \alpha H_{11}) = \mathbf{v}_1^T$;
3. Calcular $\mathbf{x}_2^T = \alpha \mathbf{x}_1^T H_{12} + \mathbf{v}_2^T$;
4. Normalizar $\pi^T = [\mathbf{x}_1^T \quad \mathbf{x}_2^T]/\|[\mathbf{x}_1^T \quad \mathbf{x}_2^T]\|_1$;

Si nos fijamos, el único sistema que hay que resolver directamente es el del paso 2. que es de dimensiones menores que las del problema inicial. Para hallar $\mathbf{x}_2^T$ solo hay que llevar a cabo una serie de multiplicaciones por la matriz dispersa H_{12} y sumas.

Este proceso de ir ordenando las filas de 0's se puede repetir recursivamente en submatrices de H cada vez más pequeñas. Es decir, en el segundo paso se buscan las filas de ceros en la submatriz H_{11} y se pasan a las últimas posiciones dentro del bloque, en el tercero se buscan las filas de ceros en la submatriz que corresponde a filas no nulas de la submatriz diagonal del paso anterior y así hasta que no haya filas de ceros en la submatriz en la que se buscan. El resultado de este proceso es una descomposición de la

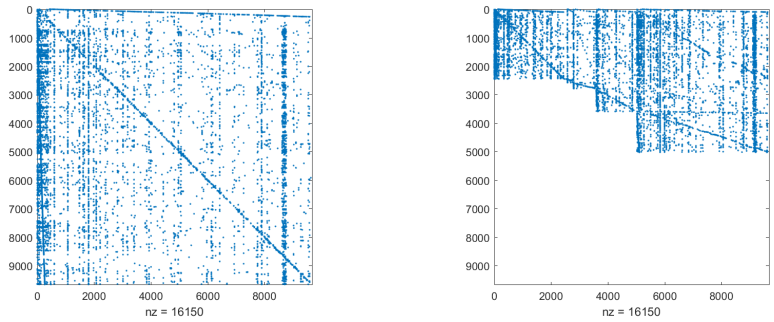


Figura 11: Ilustración de (3.4) para la matriz de la Figura 10

matriz H del modo

$$PHP^T = \begin{pmatrix} H_{11} & H_{12} & H_{13} & \cdots & H_{1k} \\ & 0 & H_{23} & \cdots & H_{2k} \\ & & 0 & \cdots & H_{3k} \\ & & & \ddots & \\ & & & & 0 \end{pmatrix} \tag{3.4}$$

donde los ceros representan matrices cuadradas idénticamente nulas y $k \geq 2$ es el número de bloques cuadrados diagonales en la matriz reordenada.
Veamos en la Figura 11 el ejemplo gráfico de esta ordenación para la matriz considerada en la Figura 3.4.

La matriz del sistema lineal que hay que resolver es, por tanto,

$$I - \alpha PHP^T = \begin{pmatrix} I - \alpha H_{11} & -\alpha H_{12} & -\alpha H_{13} & \cdots & -\alpha H_{1k} \\ & I & -\alpha H_{23} & \cdots & -\alpha H_{2k} \\ & & I & \cdots & -\alpha H_{3k} \\ & & & \ddots & \\ & & & & I \end{pmatrix},$$

y el algoritmo de resolución para hallar el vector del PageRank es el que se presenta a continuación

Algoritmo 2 para el cálculo del PageRank

1. Reordenar la matriz H utilizando los nodos 'dangling' recursivamente;
2. Resolver $\mathbf{x}_1^T(I - \alpha H_{11}) = \mathbf{v}_1^T$;
3. Para $i = 2 \cdots k$ calcular $\mathbf{x}_i^T = \alpha \sum_{j=1}^{i-1} \mathbf{x}_j^T H_{ji} + \mathbf{v}_i^T$;
4. Normalizar $\pi^T = [\mathbf{x}_1^T \quad \mathbf{x}_2^T \ldots \mathbf{x}_k^T]/|||[\mathbf{x}_1^T \quad \mathbf{x}_2^T \ldots \mathbf{x}_k^T]|||_1$;

Como resultado, después de esta reordenación, el único sistema que hay que resolver es: $\mathbf{x}_1^T(I - \alpha H_{11}) = \mathbf{v}_1^T$, el resto del vector $\mathbf{x}^T$ se halla mediante productos por matrices dispersas. Los vectores $\mathbf{x}^T$ y $\mathbf{v}^T$ se han particionado de acuerdo con el número y tamaño de los bloques diagonales de la matriz que se obtiene tras la reordenación de los nodos.

Comparamos ahora los tiempos de CPU invertidos para resolver el problema del PageRank para nuestra estructura de enlaces asociada a la búsqueda de la palabra 'california' usando el método de la potencia (con tolerancia 10^{-10} en el criterio de parada), las dos ordenaciones explicadas en esta sección, el algoritmo de mínimo grado y el RCM. En el Cuadro 9 se pueden ver los resultados obtenidos para distintos valores del parámetro α.

α	Potencia	Alg 1	Alg 2	Min. Grado	RCM
0.95	0.7266	0.0326	0.03875	0.06513	0.06421
0.9	0.2778	0.0332	0.0376	0.06784	0.06863
0.85	0.1834	0.03612	0.03322	0.0881	0.0982
0.7	0.1393	0.03437	0.03681	0.06443	0.06512

Cuadro 9: Tiempos de CPU para hallar el PageRank de la matriz 'california'.

Como podemos observar, la convergencia del método de la potencia es más rápida a medida que se reduce el valor de α. La primera ordenación explicada en esta sección (columna Alg.1) no es peor que la segunda (columna Alg2) en este caso, como se podría esperar, ya que el tiempo empleado en ordenar es menor aunque después haya que resolver un sistema lineal de mayor tamaño. Aún así, ambos algoritmos son más eficientes que el método de la potencia. Para el caso $\alpha = 0.85$ (Google), el factor de reducción es aproximadamente 5. Respecto al uso de las ordenaciones de mínimo grado y RCM, se observa que no son competitivos frente a las ordenaciones que aprovechan la existencia de nodos 'dangling'.

Tomemos ahora otras dos matrices de Google de dimensiones 2583 y 5109, y veamos en el Cuadro 10 los tiempos de calcular el correspondiente vector del PageRank con $\alpha = 0.85$ y con los diferentes métodos utilizados en el Cuadro 9.

Potencia	Dimensión	Alg1	Alg2	Min. Grado	RCM
0.04042	2583	0.01416	0.02384	0.05248	0.04617
0.09647	5109	0.02744	0.033	0.06617	0.06312

Cuadro 10: Tiempos de CPU para hallar el PageRank de otras dos búsquedas.

Las conclusiones que se obtienen para estas dos matrices son totalmente análogas a las que se tenía en el Cuadro 9 para la matriz 'california'. Los Algoritmos 1 y 2 se han implementado en Matlab en las funciones que se incluyen en el Apartado 4.2.3 del Apéndice.

3.2. Análisis de la dependencia de π^T respecto de α

Para concluir esta sección, vamos a demostrar dos resultados que permiten analizar el efecto que tiene el parámetro α en el vector de PageRank calculado. Para ello vamos a calcular la derivada de π^T respecto de α, denotada como $\frac{d\pi^T(\alpha)}{d\alpha}$, que muestra como varían las componentes del vector del PageRank cuando se producen pequeñas variaciones en α. Si el elemento j de $\frac{d\pi^T(\alpha)}{d\alpha}$, denotado por $\frac{d\pi_j^T(\alpha)}{d\alpha}$, es grande en valor absoluto, podemos

concluir que ante cambios pequeños de α, la componente j-ésima del vector del PageRank varía de forma notable. El signo de las derivadas también es importante. Si $\frac{d\pi_j^T(\alpha)}{d\alpha} > 0$, incrementos en α implican incremento de la componente j del vector de PageRank, y si $\frac{d\pi_j^T(\alpha)}{d\alpha} < 0$ se produce decremento al aumentar α.

Teorema 3.2. *El vector del PageRank, viene dado por*

$$\pi^T(\alpha) = \frac{1}{\sum_{i=1}^n D_i(\alpha)}(D_1(\alpha), D_2(\alpha), \cdots, D_n(\alpha)),$$

donde $D_i(\alpha)$ es el menor principal i-ésimo de orden $n-1$ en $I-G(\alpha)$. Debido a que cada menor principal $D_i(\alpha) > 0$ es solo una suma de productos de coeficientes de $I - G(\alpha)$, se sigue que cada componente en $\pi^T(\alpha)$ es una función diferenciable de α en el intervalo $(0, 1)$.

Demostración: Por conveniencia, denotamos $G = G(\alpha)$, $\pi^T = \pi^T(\alpha)$, $D_i(\alpha) = D_i$ y ponemos $A = I - G$. Si A^{adj} denota la traspuesta de la matriz de cofactores de A (también denominada matriz adjunta de A), entonces

$$AA^{adj} = 0 = A^{adj}A.$$

Esto es debido a que 1 es autovalor de G (por ser una matriz estocástica), y entonces 0 es autovalor de $I - G$, lo que implica que $det(A) = 0$.

Del teorema de Perron-Fröbenius, se sigue que $rango(A) = n - 1$ y, en consecuencia, rango $(A^{adj}) = 1$. Además, la teoría de Perron-Fröbenius asegura que cada columna de A^{adj} es un múltiplo de $\mathbf{e}$,de donde se sigue que $A^{adj} = \mathbf{e}\mathbf{w}^T$, para algún vector $\mathbf{w}$. Como $A_{ii}^{adj} = D_i$, entonces $\mathbf{w}^T = (D_1, \ldots, D_n)^T$. Análogamente, $A^{adj}A = 0$ asegura que cada fila de A^{adj} es un múltiplo de π^T y, consecuentemente $\mathbf{w}^T = \beta\pi^T$ para algún β. Este escalar β no puede ser 0, pues eso implicaría que $A^{adj} = 0$, lo que es imposible. Por tanto, $\mathbf{w}^T\mathbf{e} = \beta \neq 0$, y $\mathbf{w}^T/(\mathbf{w}^T\mathbf{e}) = \mathbf{w}^T/\beta = \pi^T$. □

Teorema 3.3. *Si $\pi^T(\alpha) = (\pi_1(\alpha), \pi_2(\alpha), \ldots, \pi_n(\alpha))^T$ es el vector del PageRank, entonces*

$$\left|\frac{d\pi_j(\alpha)}{d\alpha}\right| \leq \frac{1}{1-\alpha} \qquad j = 1, 2, \ldots, n, \tag{3.5}$$

y

$$\left|\left|\frac{d\pi(\alpha)}{d\alpha}\right|\right|_1 \leq \frac{2}{1-\alpha}. \tag{3.6}$$

Demostración: Utilizando que $\pi^T\mathbf{e} = 1$ y derivando en dicha igualdad se obtiene $\frac{d\pi^T(\alpha)}{d\alpha}\mathbf{e} = 0$.
Derivando ahora en ambos lados de

$$\pi^T(\alpha) = \pi^T(\alpha)(\alpha S + (1-\alpha)\mathbf{e}\mathbf{v}^T),$$

y utilizando la igualdad anterior se llega a

$$\frac{d\pi^T(\alpha)}{d\alpha}(I - \alpha S) = \pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T).$$

La matriz $(I - \alpha S)$ es no singular ya que $\alpha < 1$ garantiza $\rho(\alpha S) < 1$ y, por tanto

$$\frac{d\pi^T(\alpha)}{d\alpha} = \pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T)(I - \alpha S)^{-1}. \tag{3.7}$$

Notamos ahora que para todo vector real $\mathbf{x} \in \mathbf{e}^{\perp}$ (subespacio ortogonal a $\mathbf{e}$), y para todo vector real $\mathbf{y} \in \mathbb{R}^n$ se tiene

$$|\mathbf{x}^T\mathbf{y}| \leq ||\mathbf{x}||_1(\frac{y_{max} - y_{min}}{2}), \tag{3.8}$$

que es una consecuencia de la desigualdad de Hölder ya que para todo β real,

$$|\mathbf{x}^T\mathbf{y}| = |\mathbf{x}^T(\mathbf{y} - \beta\mathbf{e})| \leq ||\mathbf{x}||_1|(\mathbf{y} - \beta\mathbf{e})||_\infty$$

y $min_\beta||\mathbf{y} - \beta\mathbf{e}||_\infty = \dfrac{y_{max} - y_{min}}{2}$, donde el mínimo se obtiene cuando $\beta = \dfrac{y_{max} + y_{min}}{2}$. Se sigue de (3.7) que

$$\frac{d\pi_j(\alpha)}{d\alpha} = \pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T)(I - \alpha S)^{-1}\mathbf{e}_j,$$

donde $\mathbf{e}_j$ es el j-ésimo vector de la base canónica. Notamos que

$$\pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T)\mathbf{e} = \pi^T(\alpha)S\mathbf{e} - \pi^T(\alpha)\mathbf{e}(\mathbf{v}^T\mathbf{e}) = \pi^T(\alpha)\mathbf{e} - \pi^T(\alpha)\mathbf{e} = 0,$$

por tanto, la desigualdad (3.8) puede ser aplicada con

$$\mathbf{x} = \pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T), \mathbf{y} = (I - \alpha S)^{-1}\mathbf{e}_j$$

para obtener

$$|\frac{d\pi_j(\alpha)}{d\alpha}| \leq ||\pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T)||_1(\frac{y_{max} - y_{min}}{2}).$$

Ahora bien, $||\pi^T(\alpha)(S - \mathbf{e}\mathbf{v}^T)||_1 \leq \pi^T(\alpha)S\mathbf{e} + \pi^T(\alpha)\mathbf{e} \leq 2$, por lo que:

$$|\frac{d\pi_j(\alpha)}{d\alpha}| \leq y_{max} - y_{min}.$$

Ahora hacemos uso de que $0 \leq (I - \alpha S)^{-1}$ (una propiedad derivada de la teoría de Perron-Fröbenius), junto con la observación siguiente

$$(I - \alpha S)\mathbf{e} = (1 - \alpha)\mathbf{e} \Rightarrow (I - \alpha S)^{-1}\mathbf{e} = (1 - \alpha)^{-1}\mathbf{e},$$

para concluir que $0 \leq y_{min}$ y que

$$y_{max} \leq max_{ij}[(I - \alpha S)^{-1}]_{ij} \leq ||(I - \alpha S)^{-1}||_\infty = ||(I - \alpha S)^{-1}\mathbf{e}||_\infty = \frac{1}{1 - \alpha}$$

Se sigue entonces que

$$|\frac{d\pi_j(\alpha)}{d\alpha}| \leq \frac{1}{1 - \alpha}.$$

La desigualdad (3.6) es una consecuencia directa de (3.7) junto con la obsevación de arriba:

$$||(I - \alpha S)^{-1}||_\infty = ||(I - \alpha S)^{-1}\mathbf{e}||_\infty = \frac{1}{1 - \alpha}$$

□

La utilidad del Teorema 3.3 se reduce a valores pequeños de α; asegura que, para valores pequeños de α, el vector PageRank no es demasiado sensible como función del parámetro α. Veamos este efecto en la práctica; tomamos la matriz de dimensión 5109×5019 ya utilizada en el Cuadro 10, y utilizamos el Algoritmo 2 descrito en este capítulo para hallar el vector estacionario del PageRank con diferentes valores de α, todos ellos pequeños, y una vez calculado lo ordenaremos en oreden decreciente del valor de sus componentes para encontrar los índices de las 10 páginas más importantes en cada caso. Una vez obtenidos estos datos debería ser claro la poca sensibilidad del vector del PageRank cuando α es pequeño (los índices de las 10 entradas deben ser iguales o muy similares en los diferentes casos).

α	1	2	3	4	5	6	7	8	9	10
0.01	1443	1977	1388	2	1731	1494	3705	4585	1	50
0.05	1443	1977	1388	2	1731	1494	3705	1	4585	1616
0.08	1443	1977	1388	2	1731	1494	3705	1	4585	1616
0.1	1443	1977	1388	2	1731	1494	1	3705	4585	1616
0.2	1443	1977	1388	2	1731	1494	1	1616	1022	3705
0.3	1443	2	1977	1388	1	1616	1022	1494	1731	4585

Cuadro 11: Índices de las primeras 10 páginas del vector PageRank cuando varía α

Se observa en el Cuadro 11 que cuanto menor es el valor de α menor es la sensibilidad respecto de α. De hecho, para $\alpha < 0.2$ las páginas que se sitúan en las 6 primeras posiciones son las mismas y aparecen en el mismo orden. Solo cuando $\alpha = 0.3$ se observan diferencias notables, aunque en el resultado de la búsqueda 4 de las 6 primeras páginas mostradas serían las mismas pero en distinto orden.

Referencias

[1] I. S. Duff,A. M. Erisman J. K. Reid, John Ker, *Direct methods for sparse matrices*, Oxford University Press (2017).

[2] A. George J. W. H. Liu, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice Hall Inc. (1981)

[3] A. George J. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Rev. 31(1989), pp. 1-19.

[4] J. R. Gilbert, C. Moler R. Schreiber, *Sparse matrices in Matlab: Design and implementation*, SIAM J. Matrix Anal. Appl. 13(1992), pp. 333-356.

[5] J. R. Gilbert T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput. 9(1988), pp.862-874.

[6] A. N. Langville C. D. Meyer, *Google's PageRank and beyond: the science of search engine rankings*, Princeton University Press (2012).

[7] A. N. Langville C. D. Meyer, *A reordering for the PageRank problem*, SIAM J. Sci. Comput.,27, pp. 2112-2120.

[8] Suite Sparse Matrix Collection, `https://sparse.tamu.edu/`.

[9] `http://ww.cs.cornell.edu/Courses/cs685/2002fa/`

4. Apéndice

4.1. Matrices dispersas en Matlab

En Matlab una matriz real densa $n \times n$ se almacena como un array lleno de n^2 números en coma flotante. Una matriz real dispersa se almacena como concatenación de los vectores dispersos que corresponden a sus columnas. Cada vector disperso asociado a una columna consiste en un array en coma flotante de los elementos no nulos de dicha columna, junto con un vector de enteros que indican los índices de las filas en las que están dichos elementos. Un segundo vector de enteros, da la ubicación en los otros dos vectores del primer elemento no nulo de cada columna. Consecuentemente, las necesidades de almacenamiento de una matriz real dispersa $n \times n$ con k entradas no nulas se reducen a k reales (los coeficientes no nulos de la matriz) y $k+n+1$ enteros (k para los índices de las filas en las que están las entradas no nulas, n las posiciones del primer elemento de cada columna y 1 para guardar el valor de k).

En Matlab encontramos dos funciones ya integradas `full` y `sparse`, que permiten declarar las matrices como densas o dispersas, respectivamente.

Para una matriz A, `full(A)` devuelve A almacenada como densa. Si A es densa no se produce ningún cambio, pero si A es dispersa se insertan ceros en las entradas apropiadas de la matriz. Por otro lado, `sparse(A)` elimina cualquier cero que tenga la matriz y devuelve la matriz A almacenada como dispersa.

En Matlab, podemos crear una matriz dispersa directamente con el comando `A=sparse(i,j,a,m,n,nzmax)`.

Se tiene que `i` y `j` son vectores de índices enteros, mientras que `s` es un vector de reales o complejos; `m`, `n` y `nzmax` son escalares enteros que denotan las dimensiones de la matriz y el número de elementos no nulos de la matriz, respectivamente. Se debe cumplir que $\texttt{A}(\texttt{i}(\texttt{k}), \texttt{j}(\texttt{k})) = \texttt{a}(\texttt{k})$, para `k` entre 1 y `nnzmax`; `m` y `n` son las dimensiones de la matriz, mientras que `nzmax` es el máximo número de entradas no nulas que tendrá la matriz.
Nótese que tanto `i` como `j` como `s` tienen la misma longitud (`nnz(A)`).
El comando `[i, j, a]=find(A)` nos devuelve tres vectores con las posiciones de los elementos no nulos de A, contando en orden creciente de columnas y `[m, n]=size(A)` proporciona las dimensiones de A. Ilustramos el uso de estas funciones con la matriz

$$A = \begin{pmatrix} 7 & 0 & 1 & 3 \\ 0 & 5 & 0 & 9 \\ 6 & 0 & 1 & 0 \\ 10 & 2 & 0 & 1 \end{pmatrix}$$

```
A = [7,0,1,3;  0,5,0,9;  6,0,1,0;  10,2,0,1];    %matriz densa
A = sparse(A);     %A es ahora matriz dispersa
[i,j,a] = find(A);
i = [1,3,4,2,4,1,3,1,2,4];
j = [1,1,1,2,2,3,3,4,4,4];
s = [7,6,10,5,2,1,1,3,9,1];
[m,n] = size(A);
m = 4,n = 4;
```

Si quisiesemos ver el almacenamiento interno que realiza Matlab, nos encontraríamos con

los siguientes vectores

$$[7,6,10,5,2,1,1,3,9,1]; \quad [1,3,4,2,4,1,3,1,2,4]; \quad [1,2,1,1]; \quad [10];$$

Para más información ver sobre el tratamiento que hace Matlab de las matrices dispersas [4].

4.2. Algunos programas utilizados en este trabajo

A continuación se presentan las principales funciones de Matlab que se han implementados durante este trabajo y que han servido para realizar varias de las tablas expuestas, así como gráficas, que ilustran los resultados teóricos explicados.

4.2.1. Algoritmo de mínimo grado

```
function [p]=MDOmass(A)
[i, j, k]=find(A);
[m, n]=size(A);
k(:)=1;
A1=sparse(i,j,k,m,n);
p=[];
c=1;
while(c<n+1)
    [i ,j ,k]=find(A1);
    s=sum(A1);
    [w, q, z]=find(s);
    [m1 ,i]=min(z);
    i=q(i);
    A2=A1(:,i);
    [i1 ,j1 ,k1]=find(A2);
    y=sparse([],[],[],m,n);
    A3=A1; %matriz donde hacemos mass elimination
    for t=1:length(i1)
        x=(A2-A1(:,i1(t)))==1;
       if((A2-A1(:,i1(t)))==0)  %todos aquellos nodos adyacentes
    indistinguibles se eliminan simultaneamente al de menor grado y se
    acumulan los indices de las columnas
            A3(i1(t),:)=0;
            A3(:,i1(t))=0;
            p=[p, i1(t)];
            c=c+1;
        end
        [i2,j2,k2]=find(x);
        j2(:)=i1(t);
        x=sparse(i2,j2,k2,m,n);
        y=y+x; %actualizacion de grados
    end
    A1=A3+y; %actualizacion matriz
end
end
```

4.2.2. Algoritmo de Cuthill-Mckee inverso

```
function [p]=RCM(A)
r=0; %contador para recorrer toda matriz
if(issymmetric(A)==0) %si la matriz no es simetrica trabajamos con A+A'
```

```
    A=A+A';
end
[i, j, k]=find(A);
[m, n]=size(A);
k(:)=1; %ponemos valor de todas las entradas de la matriz igual a 1 para
    que sea como una matriz de adyacencia
A=sparse(i,j,k);
s=sum(A); %suma de las columnas
G=graph(A); %grafo de A
v1=conncomp(G); %componentes conexas del grafo, es decir conjuntos
    independientes unos de otros de nodos
r1=1; %indica componente conexa en la que estamos
p=[]; %secuencia de nodos total
while(r<n)
    p2=[]; %va a ser secuencia de nodos de la componente conexa
    v2=v1==r1; %buscamos conjunto independiente nodos
    v2=sparse(v2);
    [i7, j7, k7]=find(v2);%j7 son los nodos de una componente conexa
    m1=length(j7);
    r=r+m1; %r aumenta la longitud de esta componente conexa
    r1=r1+1; %r1 apunta ahora a la siguiente comp conexa
    x=Gibbs(A,j7); %hallamos nodo pseudo-periferico de la componente
    p3=[x]; %p va a contener secuencia de nodos parcial, comienza con x
    y=A(:,x); %adyacentes del nodo
    z=y;%nodos visitados
    y(x,1)=0; %nos olvidamos del nodo
    [i1, j1, k1]=find(y); %son solo los adyacentes del nodo x
    s1=s(i1); %grado de estos adyacentes
    [v, q]=sort(s1); %los ordenamos de menor a mayor
    i2=i1(q); %ordenamos los adyacentes de esa forma
    p3=[p3, i2']; %los adjuntamos a la secuencia de forma ordenadat=1; %
    contador
    c1=length(i1); %numero de nodos
    i4=[i2']; %nodos adyacentes ordenados
    while(t<c1+1)
       y1=A(:,i4(t)); %para cada adyacencente sacamos la informacion de su
     columa
       g=y1-z;
       g(x)=0; %el nodo x siempre esta visitado
       g1=g==1; %buscamos nodos no visitados dentro de la componente
       [a, b, c]=find(g1); %a son aquellos no visitados
       s2=s(a); %suma columnas
       [v, q]=sort(s2); %ordenacion de esta secuencia
       i3=a(q); %ordenacion nodos
       i4=[i4, i3']; %adyacentes mas adyacentes de adyacentes
       p3=[p3, i3']; %adjuntamos nodos ordenados
       c1=c1+length(a); %se aumenta c1 para poder recorrer todo i4 que ha
    aumentado
       t=t+1; %un paso mas
       g1=sparse(a, b, c, m, 1);  %nodos nuevos
       z=z+g1; %nodos visitados
    end
    p2=[p2, p3];%actualizamos p2
    p2=p2(length(p2):-1:1);%es Cuthill-Mckee inverso
    p=[p, p2]; %se aaden las secuencias de las componentes conexas
    ordenadas
end
end
```

```
%buscamos nodo pseudo-periferico, se trata de una DFS esencialmente
function [x]=Gibbs(A,q) %A matriz, q vector de columnas que forman
    componente conexa del grafo de la matriz
E=A(q,q); %parte de la matriz globar correspondiente a esas columas (y
    filas)
[i, j, k]=find(E); %informacion dispersa
[m, n]=size(E); %tamao E
t=0; %contador para encontrar nodo
s=sum(E); %suma de las columnas(si son unos es el grado del nodo)
[m2, i5]=min(s); %buscamos un nodo de grado minimo para empezar el
    algoritmo
x=q(i5); %nodo con el que comenzamos
c=E(:,i5); %columna de este 'nodo', con la informacion de los adyacentes
[i1, j1, k1]=find(c); %i1 son los nodos adyacentes
marcados=zeros(m,1); %el vector marcados nos va a permitir ir construyendo
     los niveles de ayacentes
marcados(i1)=1;  %marcamos los adyacentes del nodo primero, entre los
    cuales est el mismo
marcados(i5)=1; %el nodo que cogemos esta marcado
y=[i1]; %nivel primero
if(m>1)
    r=0;%lo usamos para salir del while
t2=1;%contador de niveles hasta llegar al ultimo
while (r==0)
    y1=[]; %y1 va a ir siendo cada nivel de adyacentes
    for t1=1:length(y)
        [i3, j3, k3]=find(E(:,y(t1))); %i3 va a ser los adyacentes de los
    adyacentes
        for g=1:length(i3);
            if(marcados(i3(g))==0)%solo marcamos y a adimos al siguiente
    nivel aquellos no marcados
                marcados(i3(g))=1;
                y1=[y1, i3(g)];
            end
        end
    end
    y=y1; %siguiente nivel
    t2=t2+1;%contador de niveles+1
    if (marcados==ones(m,1)) %salimos del while cuando hemos visitado todo
     el grafo
        r=1;
    end
end
if(isempty(y))
    y=i1;
end
d=s(y); %y es el ultimo nivel, sumamos sus columnas
[m1, ind]=min(d); %encontramos nodo de grado minimo
i5=y(ind);%siguiente nodo
while(t==0)     %este bucle nos da el nodo que queremos
c=E(:,i5);    %hacemos lo mismo que con el primer nodo (lineas 38-61)
[i1, j1, k1]=find(c);
marcados=zeros(m,1);
marcados(i1)=1;
y=[i1];
r=0;
t3=1;
while (r==0)
    y1=[];
```

```
    for t1=1:length(y)
        [i3, j3, k3]=find(E(:,y(t1)));
        for g=1:length(i3)
            if(marcados(i3(g))==0)
                marcados(i3(g))=1;
                y1=[y1, i3(g)];
            end
        end
    end
    y=y1;
    t3=t3+1;
    if (marcados==ones(m,1))
        r=1;
    end
end
d=s(y); %y es ultimo nivel del nodo
[m7, ind]=min(d);%buscamos nodo grado minimo
if(t3>t2)  %si el numero de niveles de adyacencia de este nodo es mayor
    que el de antes cambiamos nodos y repetimos
    t2=t3;
    i5=y(ind);
else    %si es menor o igual ya hemos encontrado el nodo que queriamos
    t=1;
    i5=y(ind);
end
end
end
x=q(i5); %nodo pseudo-periferico
end
```

4.2.3. Algoritmos de ordenación para el cálculo del PageRank

```
function[Pi,z]=PageRank1(H,alpha,v) %Pi es el PageRank vector, z la
    permutacion
[i, j, k]=find(H);
[m, n]=size(H);
k(:)=1;
v=sparse(v);
H1=sparse(i,j,k,m,n);
s=sum(H1');  %suma de las filas para ver filas nulas
z1=s~=0;
z2=s==0;
[i1, j1, k1]=find(z1);
[i2, j2, k2]=find(z2);
z=[j1, j2]; %filas no nulas, filas nulas, es la permutacion
m1=length(j1);
m2=length(j2);
H11=H(j1,j1); %submatriz para el sistema lineal
H12=H(j1,j2); %submatriz para el computo de x2
v1=v(j1);    %particionamos v
v2=v(j2);
G=speye(m1)-alpha*H11;
x1=v1'/G;   %sistema lineal
x2=alpha*x1*H12+v2'; %computo de x2
Pi=[x1, x2];
Pi=Pi./norm(Pi,1); %normalizamos PageRank
end
```

```
function [Pi,s]=PageRank2(H,alpha,v) %Pi es el PageRank vector, s la
    permutacion
[i, j, k]=find(H);
[m, n]=size(H);
k(:)=1;
v=sparse(v);
H1=sparse(i,j,k,m,n);
finish=0;     %nos va a indicar cuando ya no halla mas filas de 0's
r=[];         %va aser el vector que nos indique la separacin de bloques
s1=sum(H1');%suma de las filas para ver filas nulas
z1=s1~=0;
z2=s1==0;
[i1, j1, k1]=find(z1);
[i2, j2, k2]=find(z2);
m2=length(j2);
r=[r, m2];    %el primer separador es el N de filas nulas al principio
s=[j2];       %la permutacion, primero ponemos filas nulas
H1=H1(j1,j1);%submatriz con filas no nulas
z3=j1;        %va a ser el vector original de filas
while(finish==0)
    s1=sum(H1');   %repetimos proceso para submatriz
    z1=s1~=0;
    z2=s1==0;
    if(z2==0)
        finish=1; %acabamos si no hay filas nulas
    else
    [i3, j3, k3]=find(z1);
    [i4, j4, k4]=find(z2);
    z2=z3(j4);
    z3=z3(j3);
    m2=length(z2);
    r=[r, m2];
    s=[z2, s];
    H1=H(z3,z3);
    end
end
s=[z3, s];   % a adimos el bloque de filas no nulas final
r=[r, length(z3)]; % a adimos longitud filas no nulas final
r=r(length(r):-1:1);%lo invertimos ya que esta al reves
for t=2:length(r)
    r(t)=r(t-1)+r(t); %tranformamos el vector separador para que tenga las
     'frecuencias acumuladas'
end
H2=H(s,s); %matriz reordenada
v=v(s);
H11=H2(1:r(1),1:r(1));
x1=v(1:r(1))'/(speye(r(1))-alpha*H11); %resolvemos sistema lineal inicial
Pi=[x1]; %PageRank
for t=2:length(r) %hacemos el computo de los siguientes xi que solo
    involucra multiplicaciones y sumas
    Htk=H2(1:r(t-1),r(t-1)+1:r(t));
    xt=Pi*Htk;
    xt=alpha*xt;
    xt=xt+v(r(t-1)+1:r(t))';
    Pi=[Pi, xt];
end
Pi=Pi./norm(Pi,1);%normalizamos el Page Rank
end
```