Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Resolución de sistemas de ecuaciones en semi banda

Autor: Alejandro Martín Cuevas
Tutor: Teodoro Álamo Cantarero

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Resolución de sistemas de ecuaciones en semi banda

Autor:
Alejandro Martín Cuevas

Tutor:
Teodoro Álamo Cantarero
Catedrático de Universidad

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo de Fin de Grado: Resolución de sistemas de ecuaciones en semi banda

Autor: Alejandro Martín Cuevas

Tutor: Teodoro Álamo Cantarero

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

Agradecimientos

Por una parte, me gustaría dedicar este proyecto a mi familia, que siempre me ha ayudado en todo y cuidado de mí. A pesar de que nuestras opiniones no siempre coinciden, sus consejos y apoyo han sido fundamentales a lo largo de toda mi vida y espero que lo sigan siendo. Quiero agradecer a mi madre por su paciencia y tesón, a mi padre por su bondad y motivación, a mi hermana por soportarme, a mi abuela por su amor y a mis tíos por su cariño.

Por otra parte, agradecer a mi tutor, D. Teodoro Álamo Cantarero, por su dedicación, por su buen hacer y por su amabilidad. Así como por sus conocimientos y ayuda, no olvidaré aquella ocasión en la que resolvió mis dudas sobre una asignatura ajena a su docencia, cuando no estaba en la obligación de hacerlo.

Me gustaría también hacer una mención especial a Jose Ramón Salvador Ortiz por la ayuda prestada en los comienzos del proyecto en materia de programación en C.

Por último y no por ello menos importante, quiero dar las gracias a Lucía por aguantarme todos estos años y enseñarme que las cosas siempre se pueden mirar desde más de un punto de vista.

Mi más sincero agradeciemiento a todos.

Alejandro Martín Cuevas

Sevilla, 2017

Resumen

El objetivo de este proyecto es la resolución de sistemas de ecuaciones cuya matriz tenga la particularidad de ser el resultado de la unión de una en banda *sparse* y de otra de bajo rango.

Para ello, se ha diseñado un determiado conjunto de funciones en C, que sirvan ha modo de herramienta para poder aplicar la filosofía de matrices *sparse* a la resolución de sistemas de ecuaciones de esta clase.

La base del método escogido para llevar a cabo nuestro objetivo ha sido la utilización de la descomposición de Cholesky para determinar la solución al sistema.

Por tanto, ha sido necesario establecer una codificación *sparse* propia, fundamentada en las ya desarrolladas por otros autores, que permita la adaptación requerida a las especificaciones, así como diversas herramientas de manipulación de la información contenida en dicho tipo de estructuras de datos. Entre las posibles operaciones están la suma o resta de matrices, su multiplicación (tanto entre sí como con un vector), la trasposición, la simplificación de la codificación y la descodificación (muestra de la información contenida por pantalla). Por otro lado, tenemos funciones más avanzadas, como son la que permite calcular la descomposición de Cholesky y la que posibilita su utilización para resolver sistemas de ecuaciones en banda. Por último, tenemos programas de mayor nivel, que gestionan las funciones anteriores para implementar funcionalidades más avanzadas, como el método de resolución de sistemas de ecuaciones en semi banda, que nos permite alcanzar nuestro objetivo.

La utilidad final del código desarrollado se enfocó hacia su aplicación a optimización y control basado en MPC (Model Predictive Control), aunque puede ser utilizado en cualquier otro caso que cumpla las hipótesis de partida.

Índice

1 INTRODUCCIÓN

El principio es la mitad del todo.

- Pitágoras de Samos -

A lo largo de este documento se hará referencia a diversa terminología y conceptos variados con los cuales conviene estar familiarizado para la mejor comprensión del texto que compone este documento. Por consiguiente, se expondrán a continuación una serie de conceptos básicos para ser consultados si fuese necesario.

1.1. Sistema de ecuaciones lineales

Se define un sistema de ecuaciones lineales como un conjunto de n ecuaciones con k incógnitas, cuya solución es un conjunto de valores a obtener que satisface todas las ecuaciones al mismo tiempo, aunque el hecho de que exista una solución no está asegurado.

No se va a entrar en detalles en cuanto a los diversos tipos (lineal, en derivadas parciales, etc.) o a las diferentes formas de abordarlos (analíticos o numéricos) en esta sección, aunque sí serán objeto de nuestro estudio más adelante.

Por otro lado, cabe mencionar que su grado de aparición y utilización es más que considerable, pues suelen ser un paso obligado durante el desarrollo de la mayor parte de razonamientos matemáticos aplicados a problemas reales.

Un posible ejemplo de sitema podría ser el siguiente:

$$\begin{cases} x_1 + 2x_2 - x_3 = 3 \\ -4x_1 + x_2 + 7x_3 = 2 \\ x_1 - 5x_2 + 8x_3 = -6 \end{cases}$$

siendo habitual representarlo en forma matricial $Ax = b$:

$$\begin{bmatrix} 1 & 2 & -1 \\ -4 & 1 & 7 \\ 1 & -5 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ -6 \end{bmatrix}$$

Como puede verse, se hace uso de una matriz que representa los coeficientes de las incógnitas en cada ecuación, un vector de incógnitas y un vector de términos independientes.

La matriz de coeficientes A puede tener diferentes propiedades que pueden ser aprovechadas para la resolución del sistema de ecuaciones, algunas de las cuales serán expuestas más adelante.

1.2. Matriz triangular

Una matriz triangular es aquella que la que se cumple que $\forall i,j \ / \ i < j \rightarrow a_{ij} = 0$ (triangular superior) ó $\forall i,j \ / \ i > j \rightarrow a_{ij} = 0$ (triangular inferior). Ejemplos de esto pueden ser las siguientes matrices:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 5 & 2 & 0 & 0 & 0 \\ 5 & 9 & 3 & 0 & 0 \\ 3 & 8 & 8 & 4 & 0 \\ 2 & 8 & 7 & 1 & 5 \end{bmatrix} \quad ; \quad U = \begin{bmatrix} 1 & 3 & 1 & 2 & 2 \\ 0 & 2 & 6 & 3 & 8 \\ 0 & 0 & 3 & 9 & 6 \\ 0 & 0 & 8 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.3. Matriz simétrica

Una matriz es simétrica si cumple que $\forall i,j \rightarrow a_{ij} = a_{ji}$ que viene a traducirse en que cada elemento de la fila *i*, columna *j* es igual al ubicado en la fila *j*, columna *i* (simetría respecto a la diagonal de la matriz).

Esta propiedad es útil tanto a la hora de almacenar los datos (sólo necesitamos la mitad de espacio) como para aprovecharse en algoritmos de manera que se ahorren cálculos.

1.4. Matriz definida positiva

Una matriz $A \in \mathbb{R}^{nxn}$ (real y cuadrada de *n* fila por *n* columnas) es definida positiva si es simétrica y si se cumple que $x^T Ax > 0, \forall x \in \mathbb{R}^n - \{0\}$ (para todo *x* real distinto de cero). Este tipo de matrices nunca son singulares (su determinante es no nulo, por lo que son invertibles), admiten la descomposición de Cholesky.

Teorema: si A es una matriz cuadrada de orden n, es decir,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Entonces es condición necesaria y suficiente para que A sea definida positiva el que se cumplan lo siguiente:

$$a_{11} > 0 \quad ; \quad \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} > 0 \quad ; \quad \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} > 0 \quad ; \quad \cdots \quad ; \quad \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} > 0$$

Teorema: si se multiplica una matriz cuadrada A de orden n por su traspuesta, entonces la matriz resultante $A \cdot A^T$ es simétrica y definida positiva.

2 MATRICES *SPARSE*

Supuestamente el cerebro humano es algo parecido a una libreta que se adquiere en la papelería: muy poco mecanismo y muchas hojas en blanco.

- Alan Turing -

En álgebra lineal numérica se define una matriz *sparse* como aquella cuyos elementos son mayoritariamente nulos. En caso contrario, si la mayor parte de sus elementos son no nulos, será considerada una matriz densa. El número de elementos nulos dividido entre el total de elementos es lo que se denomina como dispersión o sparsity de la matriz. Definición dual de la densidad de la matriz, la cual viene dada por el número de elementos no nulos divididos entre el total de elementos. Por consiguiente, la relación existente entre la dispersión y la densidad de una matriz es que su suma es igual a la unidad.

A nivel conceptual, la dispersión se corresponde con sistemas con bajo acoplamiento, lo cual quiere decir que las relaciones entre las diversas variables involucradas son laxas o excasas. Sin embargo, si dichas variables están interconectadas entre si en gran medida, el resultado es un sistema denso.

Es habitual en ciencia o ingeniería el encontrarse con matrices *sparse* de gran tamaño durante el proceso de resolución de sistemas de ecuaciones de derivadas parciales.

Los recursos necesarios para el almacenamiento (memoria necesaria) y tratamiento (tiempo de procesamiento) de este tipo de información son generalmente considerables, aumentando exponencialmente con el tañamo de los datos. Por consiguiente, se han diseñado algoritmos específicos para tratar este tipo de problemas que aprovechan la ventaja principal de un sistema disperso, su compacidad o compactibilidad, que requiere una cosiderablemente menor cantidad de memoria. Tal es es caso, que este procedimiento se antoja obligatorio conforme el tañamo de la matriz crece, dado que llega el momento en el que ésta se vuelve inabarcable para los algoritmos habituales para matrices densas. Para hacerse una idea más clara, pensemos en el caso en que se desea multiplicar dos matrices. Si ambas son densas, el número de operaciones a realizar crecerá proporcionalmente al número de elementos de las mismas, de manera que serán necesarios una gran cantidad de cálculos conforme el tamaño crezca sea cual sea el algoritmo empleado para llevar a cabo la operación. No obstante, si en lugar de ser densas son *sparse*, el tamaño de las matrices no será tan determinante dado que el número de cálculos necesarios dependerá de la cuantía de los elementos no nulos, reduciéndose así el uso de memoria y de tiempo de ejecución.

2.1. Almacenamiento de una matriz *sparse*:

Existen diversas maneras de almacenar matrices *sparse* que emplean diferentes enfoques y planteamientos, sin embargo, todas ellas tienen en común el objetivo: la obtención de una estructura de datos compacta (que ocupe la menor memoria posible), simple (que sea de fácil acceso, lectura/escritura) y flexible (que pueda adaptarse a un amplío rango de operaciones matriciales). Todo esto se fundamenta en la intención de evitar guardar en memoria los ceros presentes en la matriz, de manera que se conserve únicamente la información esencial.

Una matriz se almacena típicamente en memoria empleando un array bidimensional, en el que se accede a cada elemento a_{ij} del mismo empleando dos índices *i,j*. Es habitual que i se emplee como índice de filas, ordenado de arriba abajo, mientras que *j* se usa como índice de columnas, ordenado de izquierda a derecha. Para una matriz *m x n*, la cantidad de memoria requerida para su almacenamiento es proporcional a su tamaño, más la necesaria para almacenar sus dimensiones *m*, *n*.

En el caso de una matriz *sparse*, el modo de reducir la memoria de almacenamiento necesaria es almacenar únicamente los elementos no nulos. El modo más simple de hacer esto es emplear una lista de elementos no nulos ordenada arbitrariamente. Este listado consta de dos vectores de números enteros *i,j* y de un vector de números reales *x* de longitud igual al número de entradas de la matriz. Un ejemplo de esto podría ser el siguiente:

$$A = \begin{bmatrix} 1.7 & 0 & 1.4 & 0 & 0 \\ 0 & 2.1 & 0 & 0 & 8.9 \\ 0 & 0 & 9.5 & 0 & 0 \\ 0 & 0 & 0 & 4.2 & 0 \\ 3.6 & 0 & 0 & 5.8 & 7.3 \end{bmatrix} \rightarrow \begin{cases} i = [\ 4 & 4 & 2 & 0 & 1 & 0 & 1 & 3 & 4\ ] \\ j = [\ 3 & 0 & 2 & 0 & 1 & 2 & 4 & 3 & 4\ ] \\ x = [\ 5.8 & 3.6 & 9.5 & 1.7 & 2.1 & 1.4 & 8.9 & 8.9 & 7.3\ ] \end{cases}$$

Como podemos ver la estructura empleada es de tipo *zero-based*, es decir, el rango de valores posibles de los índices *i,j* comienza en el *0* y termina en *m-1* y *n-1* respectivamente. Este tipo de rangos dependen del software empleado para implementarlos, como es el caso del lenguaje C, que emplea este último tipo. Sin embargo, otros programas utilizan otros rangos, como los *one-based*, estructuras análogas pero cuyo rango está comprendido entre *1* y *m* ó *n* según el índice. De este modo, el ejemplo anterior quedaría (Davis, 2006):

$$A = \begin{bmatrix} 1.7 & 0 & 1.4 & 0 & 0 \\ 0 & 2.1 & 0 & 0 & 8.9 \\ 0 & 0 & 9.5 & 0 & 0 \\ 0 & 0 & 0 & 4.2 & 0 \\ 3.6 & 0 & 0 & 5.8 & 7.3 \end{bmatrix} \rightarrow \begin{cases} i = [\ 5 & 5 & 3 & 1 & 2 & 1 & 2 & 4 & 5\ ] \\ j = [\ 4 & 1 & 3 & 1 & 2 & 3 & 5 & 4 & 5\ ] \\ x = [\ 5.8 & 3.6 & 9.5 & 1.7 & 2.1 & 1.4 & 8.9 & 8.9 & 7.3\ ] \end{cases}$$

Dependiendo del número y la distribución de los datos, diferentes tipos de estructuras de codificación pueden ser empleadas para ahorrar una enorme cantidad de espacio comparado con el método básico. No obstante, el acceso a la información se vuelve más complejo y son necesarias nuevas estrategias de recuperación de datos.

Podemos clasificar los formatos de codificación en dos grupos:

- Aquellos orientados a una fácil modificación (empleados generalmente para construir matrices) como son: DOK (*Dictionary of keys*, "diccionario de claves"), LIL (*List of lists*, "lista de listas"), o COO (*Coordinate list*, "lista coordinada").

- Aquellos orientados a un acceso eficiente y a las operaciones matriciales como son: CSR (*Compressed Sparse Row*, "fila *sparse* comprimida"), CSC (*Compressed Sparse Column*, "columna *sparse* comprimida"), Diagonal o MSR (*Modified Sparse Row*, "fila *sparse* modificada").

A continuación, detallaremos mejor cada una de las estructuras anteriormente enunciadas:

DOK: consiste en un diccionario que mapea pares de filas y columnas con el valor del elemento no nulo. Los elementos que no aparecen en el diccionario son considerados nulos. Este formato es recomendable a la hora de construir matrices *sparse* de cualquier orden. Sin embargo, no es aconsejable para iterar entre los elementos no nulos de manera ordenada. Es habitual emplear este formato para ocnstruir la matriz y posteriormente traducirlo a otra estructura para operar.

LIL: la información se almacena en una lista para cada fila, de manera que cada entrada contiene el índice de columna y el valor. Normalmente, estas entradas son ordenadas por índice de columna para permitir búsquedas más rápidas dentro de la codificación. Todo ello la convierte es una buena alternativa a la hora de construir matrices.

COO: los datos son guardados en memoria como una lista de ternas compuestas por los índices de fila y columna, y por los valores no nulos (fila, columna, valor). Suelen ordenarse previamente por número de fila y posteriormente por número de columna para acortar los tiempos de lectura. Es habitual su uso para la construcción de matrices (Jaramillo-Vidal-Correa, 2006).

CSR o **CRS** o **formato de Yale:** lleva usándose desde la década de los 60s, apareciendo su primera descripción completa en 1967.Representa la matriz mediante tres arrays monodimensionales que contienen respectivamente los valores no nulos, el número de elementos no números de cada fila y los índices de columna. Es similar a COO, pero comprime los índices de fila, de ahí su nombre. La diferencia con COO reside en el vector de filas que almacena no los índices de las mismas, si no la posición en donde empieza una nueva fila en el vector columna. Este formato permite un acceso rápido por filas y multiplicaciones de matriz por vector con menor coste computacional (Mejía-Solarte-Muñoz, 2013).

CSC o **CCS:** formato análogo al anterior salvo que se emplean las columnas en lugar de las columnas para comprimir la estructura COO. Este formato es eficiente a la hora de realizar operaciones aritméticas, deslizamiento de columnas y productos de matriz por vector. Este es el formato tradicional que emplea MatLab para almacenar matrices *sparse* (mediante la función *sparse*).

Diagonal: los elementos de la matriz se almacenan por diagonales dentro de un único *array* bidimensional que tiene tantas columnas como diagonales posea la matriz y tantas filas como valores numéricos tenga la diagonal principal. Para distinguir una diagonal de otra se emplea otro vector para almacenar los índices de diagonal. Este formato guarda las diagonales enteras, incluyendo algún posible elemento nulo. Sin embargo, es útil a la hora de tratar con matrices que presenten una estructura muy focalizada sobre la diagonal principal, como son el caso de las matrices en banda.

MSR: es una modificación del formato CSR, en el cual la diagonal principal de la matriz se almacena entera en un espacio aparte de memoria para facilitar su acceso. Esto se hace así dado que existen diversas operaciones que emplean la diagonal en su totalidad y, además, es habitual encontrarse con sistemas cuyos elementos de la diagonal son todos no nulos. El hecho de que se almacenen de manera correlativa en memoria facilita su acceso y los cálculos correspondientes (Moltó, 2008).

MatLab emplea el formato de columnas comprimidas, CSC que requiere el índice de filas para cada columna ordenado de manera ascendente y los elementos no nulos. De este modo, el acceso a los datos por columna, es decir, ejecutar el comando $c = A(:, j)$, es bastante rápido, del orden de la propia información extraída, $O(|c|)$. No obstante, para el caso de las filas, al extraer los datos con $r = A(i, :)$ MatLab tiene que recorrer todas las columnas buscando el índice i, proceso considerablemente costoso comparado con el anterior. Para solventar este inconveniente, es preferible trasponer la matriz codificada y filtrar por columnas en lugar de por filas. Además, en cuanto a los valores no nulos de la lista, añadir o eliminar entradas no es sencillo desde el punto de vista computacional, dado que se requiere un tiempo $O(|A|)$ para evitar huecos en la tabla de datos.

Por otro lado, en lenguaje C la estructura correspondiente al formato CSC sería la siguiente:

```
struct csc_sparse{
        int nzmax;    // número máximo de entradas de la lista
        int m;      // número de filas de la matriz
        int n;      // número de columnas de la matriz
        int *p;    // punteros columna de tamaño n+1 o índices de columna de tamaño nzmax
        int *i;    // índices de fila de tamaño nzmax
        doublé *x;    // elementos no nulos de tamaño nzmax
        int nz;   // número de ternas (toma valor -1 para formato CSC
} cs;
```

Como podemos ver, la estructura anterior sirve tanto para lista simple tipo COO como para formato CSC según se emplee (Davis, 2006).

2.2. Estructuras de datos

El método escogido para almacenar los datos de nuestro sistema ha sido la utilización de estructuras de datos que contienen la información correspondiente a una matriz concreta. Puede decirse que son como paquetes de datos asociados a cierta matriz de nuestro sistema. En concreto, nuestras estructuras son del siguiente tipo:

```
struct datos{
        int Nr;         // Número de filas (rows)
        int Nc;         // Número de columnas (columns)
        int Nd;         // Número de diadas
        int M;          // Tamaño de los datos codificados
        int* R;         // Vector de filas (rows)
        int* C;         // Vector de columnas (columns)
        double* Z;      // Vector de valores no nulos
}
```

Donde puede verse que se almacenan en memoria las dimensiones de la matriz, de la diada, de los datos codificados y la propia codificación en forma de vectores. Ésta última será detallada en el siguiente apartado.

2.3. Codificación RCZ

Para poder aplicar la filosofía *sparse* a las matrices de nuestro sistema, supondremos que las matrices involucradas presentan la siguiente forma:

$$A = H + \sum_{i=1}^{N_d} u_i \cdot v_i^T$$

siendo $u_i \in \mathbb{R}^{N_r}$ y $v_i \in \mathbb{R}^{N_c}$. En principio, se presupone que todos los elementos de la matriz H y $u_i, v_i, i = 1, \dots, N_d$ son nulos, siendo las únicas excepciones los codificados del modo expuesto a continuación.

Se emplea una codificación basada en tres vectores R, C, Z que contienen la información correspondiente a los elementos no nulos de la matriz en cuestión. De este modo, se organiza la información en forma de tabla, agrupando cada terna de valores R_k, C_k, Z_k (elementos k-ésimos de cada uno de los vectores):

R	R_1	R_2	...	R_M
C	C_1	C_2	...	C_M
Z	Z_1	Z_2	...	Z_M

Siendo los valores de *R*, *C* números enteros positivos y los de *Z* números reales.

Cada una de estas ternas de valores contiene principalmente la información del número de fila (del inglés *row*), el número de columna (del inglés *column*) y el valor del elemento, respectivamente. De esta manera, tenemos la codificación básica punto a punto de nuestra matriz.

No obstante, una funcionalidad más avanzada de dicha codificación permite modelar sucesiones o repeticiones de elementos e incluso diadas asociadas a la matriz. Para ello se distinguen los siguientes casos:

- $R_k \in [1, N_r]\ ;\ C_k \in [1, N_c] \rightarrow$ Valores de R y C dentro de los límites, codificación de un único elemento no nulo de valor Z_k de manera aditiva, es decir, Z_k es la cantidad que debe ser sumada al elemento (R_k, C_k) de la matriz, generalmente nulo. El concepto aditivo se incluye para permitir la suma de matrices, de manera que ternas con los mismos valores de R y C, sumen sus valores de Z en lugar de sobreescribirlos.

- $R_k > N_r\ ;\ C_k \in [1, N_c] \rightarrow$ Valores de R fuera de los límites con valores de C dentro de los límites, codificación de un único elemento no nulo de valor Z_k perteneciente al vector vertical asociado a la diada $(R_k - N_r)$. El valor Z_k deberá ser añadido al elemento C_k-ésimo del vector vertical de la diada.

- $R_k \in [1, N_r]\ ;\ C_k > N_c \rightarrow$ Valores de R dentro de los límites con valores de C fuera de los límites, codificación de un único elemento no nulo de valor Z_k perteneciente al vector horizontal asociado a la diada $(C_k - N_c)$. El valor Z_k deberá ser añadido al elemento R_k-ésimo del vector horizontal de la diada.

- $R_k > N_r\ ;\ C_k > N_c \rightarrow$ Valores de R y C fuera de los límites, codificación de elementos en flecha, es decir, de una sucesión/repetición de valores dentro de la matriz. El elemento $\left(R_{k-N_r}, C_{k-N_c}\right)$ marca el punto de partida de la sucesión y el valor Z_k determina el valor inicial de la sucesión/repetición. La siguiente terna, la (k+1)-ésima, estará asociada a la k-ésima en este caso y contendrá la información de la dirección y la "pendiente" de la flecha. Es decir, los valores (R_{k+1}, C_{k+1}) apuntan al siguiente elemento que debe ser rellenado con la sucesión/repetición dentro de la matriz, mientras que el valor Z_{k+1} indica el incremento existente entre los elementos de la sucesión. Si este valor es nulo, estaremos modelando una repetición, en la cual todos los elementos tienen el mismo valor que el elemento inicial Z_k.

Por otro lado, cabe decir que la codificación no tiene porqué estar ordenada de manera alguna, pero

en la mayoría de los casos se supondrá ordenada para facilitar los cálculos y las operaciones.

A continuación, mostraremos una serie de ejemplos cada vez más complejos para ilustrar el funcionamiento de la codificación descrita:

Ejemplo 2.3.1: *matriz de 5x5 con 3 elementos no nulos codificados de manera ordenada. La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*
R	*1*	*4*	*3*
C	*3*	*2*	*5*
Z	*0.1*	*0.2*	*0.3*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.3} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Ejemplo 2.3.2: *matriz anterior más 1 entrada con índices repetidos (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*
R	*1*	*4*	*3*	*3*
C	*3*	*2*	*5*	*5*
Z	*0.1*	*0.2*	*0.3*	*0.4*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Como puede verse, los dos últimos elementos de la tabla tienen sus índices R,C repetidos, por lo que hacen referencia a la misma posición dentro de la matriz, lo que provoca que se sumen dando como resultado 0.7 en (3,5).

Ejemplo 2.3.3: *matriz anterior más 1 diada (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
R	*1*	*4*	*3*	*3*	*1*	*4*	*6*	*6*
C	*3*	*2*	*5*	*5*	*6*	*6*	*2*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*1.1*	*1.2*	*1.3*	*1.4*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} \mathbf{1.1} \\ 0.0 \\ 0.0 \\ \mathbf{1.2} \\ 0.0 \end{bmatrix} \cdot \begin{bmatrix} 0.0 & \mathbf{1.3} & 0.0 & \mathbf{1.4} & 0.0 \end{bmatrix}$$

Ejemplo 2.3.4: *matriz anterior más 1 entrada de diada con índices repetidos (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
R	*1*	*4*	*3*	*3*	*1*	*4*	*6*	*6*	*6*
C	*3*	*2*	*5*	*5*	*6*	*6*	*2*	*4*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*1.1*	*1.2*	*1.3*	*1.4*	*-1.5*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} \mathbf{1.1} \\ 0.0 \\ 0.0 \\ \mathbf{1.2} \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; \mathbf{1.3} \;\; 0.0 \;\; \mathbf{-0.1} \;\; 0.0 \,]$$

Al igual que en el ejemplo 2, se repiten los índices de las dos últimas entradas de la tabla, lo que provoca que se sumen del siguiente modo:

$$1.4 + (-1.5) = -0.1$$

Ejemplo 2.3.5: *matriz anterior más 1 diada (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
R	*1*	*4*	*3*	*3*	*1*	*4*	*6*	*6*	*6*	*2*	*7*
C	*3*	*2*	*5*	*5*	*6*	*6*	*2*	*4*	*4*	*7*	*5*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*1.1*	*1.2*	*1.3*	*1.4*	*-1.5*	*1.6*	*1.7*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} \mathbf{1.1} \\ 0.0 \\ 0.0 \\ \mathbf{1.2} \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; \mathbf{1.3} \;\; 0.0 \;\; \mathbf{-0.1} \;\; 0.0 \,] + \begin{bmatrix} 0.0 \\ \mathbf{1.6} \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; 0.0 \;\; 0.0 \;\; 0.0 \;\; \mathbf{1.7} \,]$$

Ejemplo 2.3.6: *matriz anterior más 1 flecha (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
R	*1*	*4*	*3*	*3*	*1*	*4*	*6*	*6*	*6*	*2*	*7*	*6*	*1*
C	*3*	*2*	*5*	*5*	*6*	*6*	*2*	*4*	*4*	*7*	*5*	*6*	*1*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*1.1*	*1.2*	*1.3*	*1.4*	*-1.5*	*1.6*	*1.7*	*2.1*	*0.1*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} \mathbf{2.1} & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & \mathbf{2.2} & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{2.3} & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & \mathbf{2.4} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{2.5} \end{bmatrix} + \begin{bmatrix} \mathbf{1.1} \\ 0.0 \\ 0.0 \\ \mathbf{1.2} \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; \mathbf{1.3} \;\; 0.0 \;\; \mathbf{-0.1} \;\; 0.0 \,] + \begin{bmatrix} 0.0 \\ \mathbf{1.6} \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; 0.0 \;\; 0.0 \;\; 0.0 \;\; \mathbf{1.7} \,]$$

Como puede observarse, se han añadido elementos en la diagonal de la matriz H son un incremento de 0.1 entre ellos. Si en lugar de elegir el elemento (1,1) se hubiera escogido otro, la sucesión habría comenzado en el mismo, así como si en lugar de escoger la dirección de la diagonal se hubiera elegido otra, las posiciones habrían cambiado. Para ilustrar esto último, se modificará la tabla anterior del siguiente modo:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
R	*1*	*4*	*3*	*3*	*1*	*4*	*6*	*6*	*6*	*2*	*7*	*7*	*1*
C	*3*	*2*	*5*	*5*	*6*	*6*	*2*	*4*	*4*	*7*	*5*	*6*	*2*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*1.1*	*1.2*	*1.3*	*1.4*	*-1.5*	*1.6*	*1.7*	*2.1*	*0.1*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.0 & 0.0 & \mathbf{0.1} & 0.0 & 0.0 \\ \mathbf{2.1} & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{2.2} & 0.0 & \mathbf{0.7} \\ 0.0 & \mathbf{0.2} & 0.0 & 0.0 & \mathbf{2.3} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} \mathbf{1.1} \\ 0.0 \\ 0.0 \\ \mathbf{1.2} \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; \mathbf{1.3} \;\; 0.0 \;\; \mathbf{-0.1} \;\; 0.0 \,] + \begin{bmatrix} 0.0 \\ \mathbf{1.6} \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \cdot [\, 0.0 \;\; 0.0 \;\; 0.0 \;\; 0.0 \;\; \mathbf{1.7} \,]$$

Ahora puede apreciarse que la flecha comienza en el elemeto (2,1) y que tiene una dirección tal que el siguiente elemento se sitúa una fila más abajo y dos columnas a la derecha, resultando ser el (3,3). Siguiendo esta progresión, el siguiente es el (4,5) y así seguiría si la matriz tuviera una mayor dimensión.

Por último, cabe añadir que al igual que para los elementos simples y las diadas, si los índices de dos flechas se repiten tanto en origen como dirección, los elementos de sus sucesiones se sumarán de la misma forma anteriormente expuesta para los otros casos (ver ejemplos 2 y 4).

Ejemplo 2.3.7: *matriz 4x4 densa (codificación ordenada). La siguiente tabla recoge la información codificada:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
R	*1*	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*1*	*2*	*3*	*4*	*4*	*4*
C	*1*	*2*	*3*	*1*	*2*	*3*	*1*	*2*	*3*	*4*	*4*	*4*	*1*	*2*	*3*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*	*1.1*	*1.2*	*1.3*	*1.4*	*1.5*

de tal manera que la matriz resultante sería:

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 1.1 \\ 1.2 \end{bmatrix} \cdot \begin{bmatrix} 1.3 & 1.4 & 1.5 \end{bmatrix} = \begin{bmatrix} 1.40 & 1.60 & 1.80 \\ 1.83 & 2.04 & 2.25 \\ 2.26 & 2.48 & 2.70 \end{bmatrix}$$

Como puede verse, nuestra codificación permite almacenar matrices densas cualesquiera tanto mediante elementos individuales como con diadas e incluso flechas, aunque ese no sea el objetivo de la misma.

2.4. Matrices *sparse* en MatLab

Como es de esperar, MatLab distingue entre datos de entrada densos y *sparse*, de manera que ganemos en compacidad de los datos almacenados y en agilidad de los cálculos.

Para ello, se puede emplear la función *sparse*, que transforma la información en forma de matriz normal (*array* de datos) en datos *sparse*, es decir, almacena una lista de valores no nulos y sus correspondientes índices para situarlos en la matriz. La sintaxis a utilizar es sencilla, pues basta con hacer $Asparse = sparse(A)$ y obtendremos en $Asparse$ los datos codificados.

Un ejemplo de esto podría ser el siguiente:

```
n = 7;
A = diag(rand(1,n))+diag(rand(1,n-1),1);
A = A*A' + eye(n)
As = sparse(A)
```

Donde un posible resultado por pantalla es el siguiente:

A =

2.1077	0.14097	0	0	0	0	0
0.14097	1.5121	0.49058	0	0	0	0
0	0.49058	1.5991	0.010094	0	0	0
0	0	0.010094	1.9039	0.26314	0	0
0	0	0	0.26314	1.0779	0.0015904	0
0	0	0	0	0.0015904	1.1946	0.042616
0	0	0	0	0	0.042616	1.0094

As =

(1,1)	2.1077
(2,1)	0.14097
(1,2)	0.14097
(2,2)	1.5121
(3,2)	0.49058
(2,3)	0.49058
(3,3)	1.5991
(4,3)	0.010094
(3,4)	0.010094
(4,4)	1.9039
(5,4)	0.26314
(4,5)	0.26314
(5,5)	1.0779
(6,5)	0.0015904
(5,6)	0.0015904
(6,6)	1.1946
(7,6)	0.042616
(6,7)	0.042616
(7,7)	1.0094

Como puede verse, tras generar A, ésta queda almacenada de manera convencional. Sin embargo, una vez empleamos la función *sparse*, tan sólo se guardan los elementos no nulos.

Otras funciones de interés son:

- $[i, j, x] = find(A)$ que devuelve los índices y los valores no nulos de la matriz A dada.

- $nz = nnz(A)$ que devuelve el número de elementos no nulos en la matriz *A* dada.

- $nz = nzmax(A)$ que devuelve el número espacios de memoria para los elementos no nulos en la matriz *A* dada.

- $spy(A)$ que nos muestra en una figura de MatLab la silueta de la estructura de elementos no nulos de *A*.

No obstante, no serán empleadas en un principio a lo largo de esta memoria (Gilbert- Molery-Schreiberz).

3 OPERACIONES *SPARSE*

Las matemáticas ofrecen a las ciencias naturales exactas un cierto grado de seguridad que sin ella no podrían alcanzar.

- Albert Einstein -

Una vez hemos definido la codificación *sparse* a utilizar, es preciso exponer las operaciones y cálculos que han sido implementados en este proyecto para poder emplear la misma. Es preciso aclarar que todos los pasos se realizan por completo sin pasar en ningún momento por cálculos intermedios que incluyan matrices densas o sin codificar. Las funciones que permiten estas aplicaciones vienen detalladas junto con sus respectivos códigos en el anexo A al final de este documento.

3.1. Suma y resta de matrices *sparse*

En primer lugar, hablaremos de la suma y resta de matrices *sparse* empleando la codificación *RCZ* expuesta en el capítulo anterior.

En nuestro caso, la idea principal tras esta operación consiste en aunar las tablas de ternas de las matrices que se desean sumar en una sola simplemente, lo que hace este cálculo muy rápido y eficiente.

El siguiente pseudocódigo muestra los pasos que nuestro algoritmo lleva a cabo, con el fin de obtener el resultado de $C = A \pm B$:

1. Rellenamos la estructura de datos de la codificación resultado con la información conocida: tamaño de la matriz, número nulo de diadas y reserva de memoria por exceso.

2. En primer lugar, copiamos la codificación de *A* en la de *C* (los vectores *R,C,Z*).

3. En segundo lugar, hacemos lo mismo con la de *B* para los vectores *R* y *C*. Sin embargo, para Z debemos distinguir dos casos:

3.1. Si queremos realizar la suma, copiamos tal cual el vector *Z*.

3.2. Si deseamos hacer la resta, copiamos el vector *Z* pero con el signo cambiado.

4. De este modo, como las ternas de valores hacen referencia a los mismos elementos, se sumarán o restarán sus valores según el caso.

Si se desea simplificar la codificación de *C* resultante, bastará con emplear la función de posprocesado que puede verse en el anexo *A*, al final de este documento.

Para que todo lo expuesto sea más fácil de entender, emplearemos los siguientes ejemplos para ilustrar este concepto:

Ejemplo 3.1.1: *supóngase que tenemos dos matrices sparse A y B de 4x4 que queremos sumar de tal forma que A + B = C. Las tablas de ternas son las siguientes:*

Para la matriz A:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*

Lo que se traduce en:

$$A = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.2} & 0.0 & 0.0 \\ \mathbf{0.3} & \mathbf{0.4} & \mathbf{0.5} & 0.0 \\ 0.0 & \mathbf{0.6} & \mathbf{0.7} & \mathbf{0.8} \\ 0.0 & 0.0 & \mathbf{0.9} & \mathbf{1.0} \end{bmatrix}$$

Para la matriz B:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*1.1*	*1.2*	*1.3*	*1.4*	*1.5*	*1.6*	*1.7*	*1.8*	*1.9*	*2.0*

Lo que se traduce en:

$$B = \begin{bmatrix} \mathbf{1.1} & \mathbf{1.2} & 0.0 & 0.0 \\ \mathbf{1.3} & \mathbf{1.4} & \mathbf{1.5} & 0.0 \\ 0.0 & \mathbf{1.6} & \mathbf{1.7} & \mathbf{1.8} \\ 0.0 & 0.0 & \mathbf{1.9} & \mathbf{2.0} \end{bmatrix}$$

de tal manera que la matriz C resultante sería:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*...*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*	*1*	*1*	*2*	*2*	*2*	*...*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*	*1*	*2*	*1*	*2*	*3*	*...*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*	*1.1*	*1.2*	*1.3*	*1.4*	*1.5*	*...*

...	*16*	*17*	*18*	*19*	*20*
...	*3*	*3*	*3*	*4*	*4*
...	*2*	*3*	*4*	*3*	*4*
...	*1.6*	*1.7*	*1.8*	*1.9*	*2.0*

quedando la matriz C:

$$C = \begin{bmatrix} \mathbf{1.2} & \mathbf{1.4} & 0.0 & 0.0 \\ \mathbf{1.6} & \mathbf{1.8} & \mathbf{2.0} & 0.0 \\ 0.0 & \mathbf{2.2} & \mathbf{2.4} & \mathbf{2.6} \\ 0.0 & 0.0 & \mathbf{2.8} & \mathbf{3.0} \end{bmatrix}$$

Como puede verse, la codificación de C se ha obtenido uniendo las de A y B directamente. Este resultado es completamente válido, sin embargo, no es óptimo, dado que la tabla tiene el doble del tamaño necesario y puede causar problemas de cálculo si se deseara realizar más cálculos con esta matriz.

Por tanto, es conveniente realizar un posprocesado que permita simplificar la tabla de C, de manera que obtengamos la siguiente:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*1.2*	*1.4*	*1.6*	*1.8*	*2.0*	*2.2*	*2.4*	*2.6*	*2.8*	*3.0*

Una tabla de ternas del mismo corte que las de A y B (la función que realiza esta simplificación puede verse en detalle en el anexo A al final de este documento).

Ejemplo 3.1.2: *ahora supongamos que tenemos las mismas dos matrices, pero queramos restarlas de tal forma que A - B = C. Las tablas de ternas son las siguientes:*

Para la matriz A:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*

Lo que se traduce en:

$$A = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.2} & 0.0 & 0.0 \\ \mathbf{0.3} & \mathbf{0.4} & \mathbf{0.5} & 0.0 \\ 0.0 & \mathbf{0.6} & \mathbf{0.7} & \mathbf{0.8} \\ 0.0 & 0.0 & \mathbf{0.9} & \mathbf{1.0} \end{bmatrix}$$

Para la matriz B:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*1.1*	*1.2*	*1.3*	*1.4*	*1.5*	*1.6*	*1.7*	*1.8*	*1.9*	*2.0*

Lo que se traduce en:

$$B = \begin{bmatrix} \mathbf{1.1} & \mathbf{1.2} & 0.0 & 0.0 \\ \mathbf{1.3} & \mathbf{1.4} & \mathbf{1.5} & 0.0 \\ 0.0 & \mathbf{1.6} & \mathbf{1.7} & \mathbf{1.8} \\ 0.0 & 0.0 & \mathbf{1.9} & \mathbf{2.0} \end{bmatrix}$$

de tal manera que la tabla de la C resultante sería:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	...
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*	*1*	*1*	*2*	*2*	*2*	...
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*	*1*	*2*	*1*	*2*	*3*	...
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*-1.0*	*-1.1*	*-1.2*	*1.3*	*-1.4*	*-1.5*	...

...	*16*	*17*	*18*	*19*	*20*
...	*3*	*3*	*3*	*4*	*4*
...	*2*	*3*	*4*	*3*	*4*
...	*-1.6*	*-1.7*	*-1.8*	*-1.9*	*-2.0*

quedando la matriz C:

$$C = \begin{bmatrix} \mathbf{-1.0} & \mathbf{-1.0} & 0.0 & 0.0 \\ \mathbf{-1.0} & \mathbf{-1.0} & \mathbf{-1.0} & 0.0 \\ 0.0 & \mathbf{-1.0} & \mathbf{-1.0} & \mathbf{-1.0} \\ 0.0 & 0.0 & \mathbf{-1.0} & \mathbf{-1.0} \end{bmatrix}$$

Como puede verse, la codificación de C se ha obtenido uniendo las de A y B directamente, cambiando únicamente el signo de los elementos de B. Este resultado es completamente válido, sin embargo, no es óptimo, dado que la tabla tiene el doble del tamaño necesario y puede causar problemas de cálculo si se deseara realizar más cálculos con esta matriz.

Por tanto, es conveniente realizar un posprocesado que permita simplificar la tabla de C, de manera que obtengamos la siguiente:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*	*-1.0*

Una tabla de ternas del mismo corte que las de A y B (la función que realiza esta simplificación puede verse en detalle en el anexo A al final de este documento).

3.2. Multiplicación de una matriz *sparse* por un vector

En segundo lugar, expondremos otra posible operación, que es la de multiplicar una matriz *sparse* codificada usando *RCZ* por un vector cualquiera de dimensiones compatibles evidentemente.

La función que se ha creado para este fin puede verse en detalle en el anexo A junto al resto. No obstante, explicaremos en este apartado el modo en que funciona.

Primeramente, es preciso decir que consta de dos funcionalidades a elegir, dado que puede especificarse si se desea multiplicar la matriz tal cual por el vector o, por el contrario, su traspuesta. Una vez elegido esto, procede a recorrer la codificación de la matriz *sparse* y va multiplicando uno a uno los elementos no nulos correspondientes por la componente asociada del vector, de manera que se vayan almacenando los resultados en el vector que se quiere calcular.

El procedimiento antes descrito es común para las dos opciones de las que se dispone, sin embargo, la diferencia estriba únicamente en que es preciso trasponer la matriz antes de realizar las operaciones si ese es el caso. Para ello, con el fin de evitar la necesidad de reservar memoria de manera externa a la función, esta trasposición se realiza localmente, invirtiendo los vectores *R* y *C* de la codificación, pero sin alterar la matriz original (este paso se lleva a cabo local y temporalmente).

El pseudocódigo que describe la ejecución de esta función podría ser el siguiente:

1. Reservamos memoria e inicializamos el vector resultado.

2. Comprobamos el modo elegido, dos posibilidades:

    2.1. Recorremos la tabla de ternas de la matriz a multiplicar y, tras comprobar que la codificación es simple, multiplicamos elemento a elemento la tabla por el vector.

    2.2. Hacemos lo mismo, pero intercambiando previamente los vectores *R* y *C* de la matriz codificada de manera local, dentro de esta función temporalmente, no modificamos los datos de partida.

3. Una vez hemos terminado, devolvemos el vector resultado.

De este modo, los cálculos son cómodos y sencillos y no precisan de posprocesado de los mismos, dado que el resultado es un vector normal devuelto por referencia.

Para ilustrar este procedimiento, se expone el ejemplo siguiente:

Ejemplo 3.2.1: *supóngase que tenemos una matriz sparse A de 4x4 que deseamos multiplicar por un vector v compatible. La tabla de ternas de A es la siguiente:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*

Lo que se traduce en:

$$A = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.2} & 0.0 & 0.0 \\ \mathbf{0.3} & \mathbf{0.4} & \mathbf{0.5} & 0.0 \\ 0.0 & \mathbf{0.6} & \mathbf{0.7} & \mathbf{0.8} \\ 0.0 & 0.0 & \mathbf{0.9} & \mathbf{1.0} \end{bmatrix}$$

Mientras que el vector v viene dado por:

$$v = \begin{bmatrix} \mathbf{1.1} \\ \mathbf{1.2} \\ \mathbf{1.3} \\ \mathbf{1.4} \end{bmatrix}$$

El resultado de la multiplicación $u = A \cdot v$ *sería el siguiente:*

$$u = \begin{bmatrix} \mathbf{0.35} \\ \mathbf{1.46} \\ \mathbf{2.75} \\ \mathbf{2.57} \end{bmatrix}$$

Mientras que el resultado de $u = A^T \cdot v$ *sería el siguiente:*

$$u = \begin{bmatrix} \mathbf{0.47} \\ \mathbf{1.48} \\ \mathbf{2.77} \\ \mathbf{2.44} \end{bmatrix}$$

La única diferencia entre ambas operaciones es la tabla de ternas de A que se multiplica por v. Si se

indica el modo 1, se obtiene el primer vector u, mientras que, si se indica el modo 2, se consigue el segundo vector u. La trasposición de la matriz se lleva a cabo de manera local dentro de la función, por lo que si se desea extraer la codificación de la traspuesta de A es preciso emplear la función correspondiente (véase el apartado 3.4 de este mismo capítulo).

Al tratarse el resultado de un vector, no de una matriz sparse, no es necesario posprocesar.

3.3. Multiplicación de dos matrices *sparse*

En tercer lugar, analizaremos otra función creada con el objeto de realizar multiplicaciones de matrices empleando nuestra codificación *sparse*.

Para comprender el algoritmo utilizado, supondremos que tenemos dos matrices *A* y *B* codificadas *sparse* y de dimensiones compatibles (nuestra función comprueba esta condición antes de proceder a calcular), que queremos multiplicar entre sí, de manera que obtengamos una tercera matriz $C = A * B$ es recorrer la tabla de ternas de *A* para ir multiplicándolas por el elemento de *B* correspondiente. Para ello, es preciso buscar en la tabla de *B* para encontrar las ternas que cumplan que $R_B = C_A$, almacenando el resultado de la multiplicación en una nueva entrada de la tabla de *C*. Esto se hace así para evitar la reserva dinámica de memoria por cada nueva terna resultado, lo que agiliza considerablemente la ejecución del código. Una vez hemos acabado de recorrer toda la codificación de *A*, actualizamos el tamaño de los datos contenidos en *C* y terminamos.

Un posible pseudocódigo de esta función sería el siguiente:

1. Reservamos memoria por exceso para la matriz resultado y rellenamos la información conocida: tamaño de la matriz y número nulo de diadas.

2. Inicializamos un contador de multiplicaciones y recorremos la matriz *A*.

3. Recorremos a su vez la matriz *B* para encontrar las ternas que cumplan que $R_B = C_A$

4. Multiplicamos las ternas correspondientes de *A* y *B*, y aumentamos el contador de multiplicaciones.

5. Una vez hallamos acabado, actualizamos el tamaño de la matriz *C* resultante igualándolo al contador de multiplicaciones y ya habremos terminado.

El detalle de esta función puede apreciarse en su código, el cual se encuentra para referencia en el anexo A, al final de este documento.

Para que todo lo expuesto sea más fácil de entender, se muestra a continuación el siguients ejemplo:

Ejemplo 3.3.1: *supóngase que tenemos dos matrices sparse A y B de 4x4 que queremos multiplicar de tal forma que* $C = A \cdot B$*. Las tablas de ternas son las siguientes:*

Para la matriz A:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.1*	*0.2*	*0.2*	*0.3*	*0.4*	*0.4*	*0.5*	*0.6*	*0.6*	*0.7*

Lo que se traduce en:

$$A = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.2} & 0.0 & 0.0 \\ \mathbf{0.2} & \mathbf{0.3} & \mathbf{0.4} & 0.0 \\ 0.0 & \mathbf{0.4} & \mathbf{0.5} & \mathbf{0.6} \\ 0.0 & 0.0 & \mathbf{0.6} & \mathbf{0.7} \end{bmatrix}$$

Para la matriz B:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.8*	*0.9*	*0.9*	*1.0*	*1.1*	*1.1*	*1.2*	*1.3*	*1.3*	*1.4*

Lo que se traduce en:

$$B = \begin{bmatrix} \mathbf{0.8} & \mathbf{0.9} & 0.0 & 0.0 \\ \mathbf{0.9} & \mathbf{1.0} & \mathbf{1.1} & 0.0 \\ 0.0 & \mathbf{1.1} & \mathbf{1.2} & \mathbf{1.3} \\ 0.0 & 0.0 & \mathbf{1.3} & \mathbf{1.4} \end{bmatrix}$$

de tal manera que la matriz C resultante sería:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
R	*1*	*1*	*1*	*2*	*2*	*2*	*2*	*3*	*3*	*3*	*3*	*4*	*4*	*4*
C	*1*	*2*	*3*	*1*	*2*	*3*	*4*	*1*	*2*	*3*	*4*	*2*	*3*	*4*
Z	*0.26*	*0.29*	*0.22*	*0.43*	*0.92*	*0.81*	*0.52*	*0.36*	*0.95*	*1.82*	*1.49*	*0.66*	*1.63*	*1.76*

quedando la matriz C:

$$C = \begin{bmatrix} \mathbf{0.26} & \mathbf{0.29} & \mathbf{0.22} & 0.0 \\ \mathbf{0.43} & \mathbf{0.92} & \mathbf{0.81} & \mathbf{0.52} \\ \mathbf{0.36} & \mathbf{0.95} & \mathbf{1.82} & \mathbf{1.49} \\ 0.0 & \mathbf{0.66} & \mathbf{1.63} & \mathbf{1.76} \end{bmatrix}$$

Como puede verse, la codificación de C se ha obtenido multiplicando las de A y B, para luego posprocesar la tabla resultado y obtener la mostrada anteriormente. El resultado de la multiplicación sin posprocesado supone una tabla de mayor tamaño, con ternas que hacen referencia a la misma posición en la matriz, aunque con distinto valor no nulo. Por consiguiente, tanto si se desea conseguir la tabla óptima como si se va a utilizar la misma para posteriores operaciones, es preciso posprocesar.

No se ha expuesto la tabla resultante de la multiplicación sin posprocesado por la dificultad que supone dado su tamaño y puesto que es fácil de imaginar su estructura tras ver el caso de la suma/resta de matrices sparse del apartado anterior.

3.4. Trasposición de una matriz *sparse*

En cuarto y último lugar, mostraremos el algoritmo implementado para trasponer una matriz *sparse* del modo más eficiente posible.

Para llevar esto a cabo, la primera posibilidad planteada fue la de reservar memoria para una versión traspuesta completa de la matriz original, sin embargo, esto aumentaba el grado de complejidad de manera innecesaria. Por tanto, se optó por compartir la memoria de la matriz original, en concreto, aquella correspondiente a los vetores *R* y *C*. De este modo bastaba con con asociarel vector *R* original al *C* de la versión traspuesta y el *C* original al *R* de la traspuesta, de forma que se ahorre tiempo y memoria.

Este procedimiento tiene otra ventaja añadida e inherente al mismo, que es la de trasladar cuanquier modificación en la matriz original a su versión traspuesta. Esto se debe al hecho de compartir los vectores *R* y *C*, sólo que invertidos (los punteros apuntan a las mismas direcciones de memoria, por lo que, si se modifican los valores almacenados en un lado, también se cambian en el otro sin necesidad de más operaciones).

El pseudocódigo de esta función podría ser el siguiente:

1. Rellenamos la información conocida: las dimensiones de la matriz traspuesta y el tamaño de los datos codificados.

2. Por último, asignamos el vector *R* de la matriz dato al *C* de la traspuesta, y el *C* al *R*. De este modo los intercambiamos, trasponiendo la matriz.

El detalle de esta función puede encontrarse en el anexo A, al final de este documento.

Para ilustrar este procedimiento, se muestra el siguiente ejemplo:

Ejemplo 3.4.1: *supóngase que tenemos una matriz sparse A de 4x4 que deseamos trasponer. La tabla de ternas es la siguiente:*

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
C	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*

Lo que se traduce en:

$$A = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.2} & 0.0 & 0.0 \\ \mathbf{0.3} & \mathbf{0.4} & \mathbf{0.5} & 0.0 \\ 0.0 & \mathbf{0.6} & \mathbf{0.7} & \mathbf{0.8} \\ 0.0 & 0.0 & \mathbf{0.9} & \mathbf{1.0} \end{bmatrix}$$

de tal manera que la matriz A^T resultante sería:

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
R	*1*	*2*	*1*	*2*	*3*	*2*	*3*	*4*	*3*	*4*
C	*1*	*1*	*2*	*2*	*2*	*3*	*3*	*3*	*4*	*4*
Z	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*

quedando la matriz A^T:

$$A^T = \begin{bmatrix} \mathbf{0.1} & \mathbf{0.3} & 0.0 & 0.0 \\ \mathbf{0.2} & \mathbf{0.4} & \mathbf{0.6} & 0.0 \\ 0.0 & \mathbf{0.5} & \mathbf{0.7} & \mathbf{0.9} \\ 0.0 & 0.0 & \mathbf{0.8} & \mathbf{1.0} \end{bmatrix}$$

Como puede verse, la codificación de A^T resultante es igual a la de A, pero con los vectores R y C intercambiados, lo que provoca que los elementos fuera de la diagonal principal de la matriz intercambien sus posiciones según ésta. Esta tabla de ternas resultante no precisa de prospocesado.

4 DESCOMPOSICIÓN DE CHOLESKY

Divide et impera.

- Julio César -

Una matriz A simétrica y positiva definida puede ser factorizada de manera eficiente por medio de una matriz triangular inferior y una matriz triangular superior. Para una matriz no singular la descomposición $L \cdot U$ nos lleva a considerar una descomposición de tal tipo $A = L \cdot U$. Sin embargo, dadas las condiciones de A, simétrica y definida positiva, no es necesario hacer pivoteo, por lo que la factorización de Cholesky supone la mitad de operaciones que si fuera $L \cdot U$, tomando la forma $A = R_c \cdot R_c^T$, donde R_c ("raíz cuadrada" de A) es una matriz triangular inferior donde los elementos de la diagonal son positivos.

Para encontrar la factorización $A = R_c \cdot R_c^T$, basta con desarrollar la expresión matricial e igualar términos:

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & 0 & \cdots & 0 \\ r_{21} & r_{22} & 0 & \cdots & 0 \\ r_{31} & r_{32} & r_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \cdots & r_{nn} \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{21} & r_{31} & \cdots & r_{n1} \\ 0 & r_{22} & r_{32} & \cdots & r_{n2} \\ 0 & 0 & r_{33} & \cdots & r_{n3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & r_{nn} \end{bmatrix}$$

así obtendríamos que:

$$a_{11} = r_{11}^2 \rightarrow r_{11} = \sqrt{a_{11}}$$

$$a_{21} = r_{21} \cdot r_{11} \rightarrow r_{21} = \frac{a_{21}}{r_{11}} \rightarrow r_{n1} = \frac{a_{n1}}{r_{11}}$$

$$a_{22} = r_{21}^2 + r_{22}^2 \rightarrow r_{22} = \sqrt{(a_{22} - r_{21}^2)}$$

$$a_{32} = r_{31} \cdot r_{21} + r_{32} \cdot r_{22} \rightarrow r_{32} = \frac{(a_{32} - r_{31} \cdot r_{21})}{r_{22}}$$

y de manera general, para $i = 1, \dots, n$ y $j = i + 1, \dots, n$:

$$r_{ii} = \sqrt{\left(a_{ii} - \sum_{k=1}^{i-1} r_{ik}^2 \right)}$$

$$r_{ji} = \left[a_{ji} - \sum_{k=1}^{i-1} (r_{jk} \cdot r_{ik}) \right] \Big/ r_{ii}$$

En nuestro caso, nos encontramos con matrices que cumplen ciertos requisitos, además de los requeridos para poder aplicar esta descomposición, que son:

- La codificación de la matriz *A* viene ordenada, para facilitar la búsqueda de elementos y agilizar las operaciones.

- Consideramos que las posibles diadas y flechas han sido desarrolladas y añadidas a la matriz, de manera que la tabla de ternas sólo contenga elementos simples (existe una función de simplificación, *conversor_sparse*, que permite hacer esto).

- Si se desean resolver sistemas que tengan diadas, emplearemos el método de resolución de sistemas de ecuaciones en semi banda, que será expuesto en el capítulo siguiente.

Por tanto, el pseudocódigo que permite implementar este algoritmo de descomposición es el siguiente (ver anexo A - Cholesky *sparse* para más detalles):

1. Rellenamos la estructura de datos de la descomposición con la información conocida: tamaño de la matriz, número nulo de diadas y reserva de memoria por exceso.

2. Generamos la plantilla a recorrer para agilizar las operaciones, ésta se obtiene usando la codificación de A como patrón, pero con los elementos del vector Z nulos.

3. Recorremos la plantilla y en cada iteración:

   3.1. Actualizamos el valor de una variable auxiliar/intermedia, para lo cual debemos previamente buscar dentro de la codificación de la descomposición de Cholesky, Rc, los elementos correspondientes a Rc_{in} y Rc_{jn}.

   3.2. Comprobamos si estamos en la diagonal de Rc o fuera de la misma, para saber qué

expresión (ver más arriba) aplicar. Según el caso, deberemos encontrar los elementos A_{ii} ó Rc_{jj} para poder realizar los cálculos.

4. Una vez hemos acabado las operaciones numéricas, habremos obtenido una matriz triangular inferior Rc, que será la descomposición de Cholesky deseada.

A lo largo de la ejecución de esta función, todas las operaciones y cálculos realizados se llevan a cabo empleando nuestra filosofía *sparse*, por lo que en ningún momento nos encontramos con datos intermedios densos. Esta es la principal razón por la cual se obtiene agilidad en los cálculos frente a la resolución convencional que emplea matrices densas de manera directa.

4.1. Descomposición de Cholesky en MatLab

Si deseamos realizar la descomposición de Cholesky empleando MatLab debemos recurrir a la función *chol* incluida en el *toolbox* del programa. Las instrucciones que nos da el propio software para la utilización de esta función son las siguientes:

- $R_c = chol(A)$ ó $R_c = chol(A,'upper')$ producen una matriz triangular superior R_c a partir de la diagonal y del triángulo superior de la matriz A, satisfaciendo la ecuación $A = R_c^T \cdot R_c$. La función $chol$ asume que A es simétrica (Hermitiana o Hermítica: cuadrada con elementos complejos cuya característica es ser igual a su propia traspuesta conjugada). Si no es así, $chol$ utiliza la transposición (complejo conjugado) del triángulo superior como el triángulo inferior. La matriz A debe ser definida positiva.

- $R_c^T = chol(A,'lower')$ genera una matriz triangular inferior R_c^T en lugar de la superior del caso anterior. Cuando A es *sparse*, esta sintaxis requiere generalmente menor tiempo de computación.

- $[R_c, p] = chol(A)$ ó $[R_c, p] = chol(A,'upper')$ devuelven lo mismo que anteriormente con $p = 0$ si la matriz A es definida positiva. Si no lo es, p se convierte en un entero positivo y MatLab no nos da error. Cuando A es densa, R_c es una matriz triangular superior de orden $q = p - 1$ tal que $R_c^T \cdot R_c = A(1{:}\,q, 1{:}\,q)$. Cuando A es *sparse*, R_c es una matriz triangular superior de tamaño $q\ x\ n$ de tal manera que la región en forma de L formada por las primeras q filas y las primeras q columnas de $R_c^T \cdot R_c$ coincide con la correspondiente en A.

- $[R_c^T, p] = chol(A,'lower')$ genera una matriz triangular inferior R_c^T en lugar de la superior del caso anterior.

Las siguientes sintaxis de tres salidas requieren que A venga definida en formato *sparse* de MatLab:

- $[R_c, p, S] = chol(A)$, cuando A es *sparse*, devuelve una matriz de permutación S. Cuando $p = 0$, R_c es una matriz triangular superior tal que $R_c^T \cdot R_c = S^T \cdot A \cdot S$. Cuando $p \neq 0$, R_c es una matriz triangular superior de tamaño $q\ x\ n$ de tal manera que la región en forma de L formada por las primeras q filas y las primeras q columnas de $R_c^T \cdot R_c$ coincide con la correspondiente en $S^T \cdot A \cdot S$. Los resultados que se obtienen con $S^T \cdot A \cdot S$ tienden a ser más *sparse* que los conseguidos empleando tan sólo A.

- $[R_c, p, s] = chol(A, 'vector')$ ó $[R_c, p, s] = chol(A, 'upper', 'vector')$, cuando A es *sparse* y $p = 0$, devuelven la información de la permutación como un vector s tal que $A(s, s) = R_c^T \cdot R_c$. Podemos utilizar la opción de 'matrix' en lugar de 'vector' para obtener el comportamiento predeterminado.

- $[R_c^T, p, s] = chol(A, 'lower', 'vector')$ funciona análogamente, pero devuelve la matriz triangular inferior en lugar de la superior (Gilbert- Molery- Schreiberz).

4.2. Escritura/lectura de ficheros en MatLab

El manejo de ficheros en MatLab se basa en las funciones fopen, fclose, fprintf, fscanf, etc. La primera de ellas, fopen, se utiliza para permitirnos el acceso al fichero del siguiente modo:

- $fileID = fopen(nombre\_archivo)$ permite abrir el fichero de nombre $nombre\_archivo$, devolviendo un entero ≥ 3 como identificador. Los valores 0 (entrada estándar), 1 (salida estándar, pantalla) y 2 (error estándar) están reservados. Si *fopen* no puede abrir el archivo devuelve un -1.

- $fileID = fopen(nombre\_archivo, permiso)$ permite realizar lo mismo, pero especificando el tipo de permiso de acceso que deseemos, que puede ser: *'r'* (abre un fichero existente para lectura), *'r +'* (abre un fichero existente para lectura y escritura), *'w'* (crea un fichero nuevo para escritura) y *'w +'* (crea un fichero nuevo para lectura y escritura).

- $fileID = fopen(nombre\_archivo, permiso, machinefmt, encodingIn)$ con *machinefmt* podemos además especificar el orden de lectura/escritura de bytes o bits en el fichero, mientras que con *encodingIn* determinamos el método de codificación del archivo.

- $[fileID, errmsg] = fopen(nombre\_archivo, permiso, machinefmt, encodingIn)$ así nos devolverá el mensaje de error correspondiente si se produce algún fallo den la apertura del fichero. En caso contrario, devuelve una cadena de caracteres vacía.

- $fIDs = fopen('all')$ de este modo obtenemos en *fIDs* (vector fila) todos los identificadores de los ficheros que estén abiertos, a excepción de los reservados. El número de elementos en

el vector es igual al de archivos abiertos.

- $nombre\_archivo = fopen(fileID)$ nos devuelve el nombre de un fichero que haya sido abierto previamente, operación inversa de las empleadas anteriormente. Esta operación no lee datos contenidos en el archivo para determinar el nombre del fichero.

- $[nombre\_archivo, permiso, machinefmt, encodingOut] = fopen(fileID)$ es la operación inversa de la expuesta antes, dado que nos devuelve los datos correspondientes al fichero abierto cuyo identificador es *fileID*. Si el fichero ha sido abierto en formato binario, la letra 'b' estará incluida en *permiso*. Esta operación no lee datos contenidos en el archivo para determinar la información de salida. Un identificador no válido se traducirá en cadenas de caracteres vacías para todos los argumentos de salida.

La segunda función a tener en cuenta es la que nos permite realizar la operación contraria a la anterior, cerrar el archivo:

- $fclose(fileID)$ permite cerrar el fichero correspondiente al identificador *fileID* obtenido previamente con *fopen*.

- $fclose('all')$ cierra todos los ficheros que estén abiertos.

- $status = fclose(\dots)$ devuelve un 0 si la operación ha sido realizada con éxito, si no devuelve un -1.

El uso de esta función es obligatorio para liberar el fichero, de manera que otro software pueda tener acceso al mismo.

Por otro lado, tenemos las funciones que nos permiten escribir y leer datos de un fichero, la primera de ellas es:

- $fprintf(fileID, formato, a_1, \dots, a_n)$ permite escribir los datos $a_1, \dots, a_n$ en ese orden con el formato definido en el fichero correspondiente al identificador *fileID*, para ello la función emplea la misma codificación especificada con *fopen*. Si no se especifica el *fileID* se mostrarán los datos por pantalla (en MatLab).

- $nbytes = fprintf(fileID, formato, a_1, \dots, a_n)$ devuelve el n´mero de bytes que fprintf escribe en el fichero.

Mientras que la segunda es:

- $A = fscanf(fileID, formato)$ lee datos del fichero cuyo identificador es fileID para lo cual emplea el formato especificado. Por consiguiente, es necesario conocer el modo en el que está escrita la información. Si el formato dado no se corresponde o coincide parcialmente con el del fichero, la función únicamente extrae la parte de los datos con el formato dado a *fscanf*.

- $A = fscanf(fileID, formato, size)$ de esta manera podemos especificar el tamaño de los datos a leer.

- $[A, count] = fscanf(fileID, formato, size)$ devuelve además el número de campos leídos y almacenados.

Un ejemplo de lo anterior podría ser el que se emplea para copiar una matriz A en un fichero de texto (Gilbert- Molery- Schreiberz):

```
archivoID = fopen('datos.txt','w'); % Abrimos el archivo para
sobrescritura
for i = 1:n
  for j = 1:n
    fprintf(archivoID,' %f',A(i,j)); % Escribimos los datos en
el archivo
  end
    fprintf(archivoID,'\r\n'); % Pasamos a la siguiente línea
end
fclose(archivoID); % Cerramos el archivo
type datos.txt % Mostramos por pantalla el contenido del archivo
```

4.3. Resolución de sistemas de ecuaciones mediante descomposición de Cholesky

La factorización de Cholesky se puede emplear para resolver sistemas de ecuaciones matriciales. Si llamamos A a la matriz de coeficientes de un sistema de ecuaciones dado, es condición necesaria y suficiente para que dicha matriz admita factorización de Cholesky, el que sea simétrica y definida positiva. Si esto se cumple, podemos factorizar esta matriz de manera que $A = R_c \cdot R_c^T$. Esta descomposición, llamada de Cholesky, nos permitirá resolver el sistema de ecuaciones.

Sea el sistema de ecuaciones lineales $A \cdot x = b$, donde A es simétrica y definida positiva, el método de Cholesky para la resolución del mismo está basado en la descomposición de la matriz A del siguiente modo:

$$A = R_c \cdot R_c^T \qquad (4.3.1)$$

donde Rc es una matriz triangular inferior de orden n, es decir, R_c tiene la siguiente forma:

$$R_c = \begin{bmatrix} r_{11} & 0 & 0 & \cdots & 0 \\ r_{21} & r_{22} & 0 & \cdots & 0 \\ r_{31} & r_{32} & r_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \cdots & r_{nn} \end{bmatrix}$$

Descompuesta de esta forma la matriz A, la resolución del sistema $A \cdot x = b$ viene dada por la de dos sistemas triangulares. En efecto,

$$A \cdot x = b \quad \rightarrow \quad (R_c \cdot R_c^T) \cdot x = b \quad \rightarrow \quad R_c \cdot (R_c^T \cdot x) = b$$

Si definimos y como:

$$y = R_c^T \cdot x \qquad (4.3.2)$$

y sustituimos, tenemos que:

$$R_c \cdot y = b \qquad (4.3.3)$$

un sistema triangular inferior en y con:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad ; \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Que puede resolverse empleando la siguiente expresión:

$$y_1 = \frac{b_1}{r_{11}} \quad \rightarrow \quad y_k = \frac{1}{r_{kk}} \left[b_k - \sum_{j=1}^{k-1} r_{kj} \cdot y_j \right] \quad con \quad k = 2, 3, \dots, n$$

Una vez que se resuelve (4.3.3), procedemos a resolver el sistema (4.3.2), que es un sistema triangular superior:

$$R_c \rightarrow triangular\ inferior \quad ; \quad R_c^T \rightarrow triangular\ superior$$

Es decir, los valores de x_k vienen dados por:

$$x_n = \frac{y_n}{r_{nn}} \quad ; \quad x_k = \frac{1}{r_{kk}} \left[y_k - \sum_{j=k+1}^{n} r_{jk} \cdot x_j \right] \quad con \quad k = n-1, n-2, \dots, 1$$

De (4.3.1) se obtienen las siguientes ecuaciones para el cálculo de los elementos de la matriz $R_c(r_{ij})$:

$$r_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} r_{ik}^2} \quad ; \quad r_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} r_{ik} \cdot r_{jk}}{r_{jj}} \quad con \quad j = 1, 2, \dots, i-1$$

A continuación, se enuncian un teorema relacionado con el hecho de que la matriz *A* deba ser simétrica y definida positiva:

Teorema: si $A = R_c \cdot R_c^T$, con R_c triangular inferior, real y no singular (determinante no nulo), entonces A es simétrica y definida positiva. Observación: es necesario verificar que ningún elemento de la diagonal principal sea igual a cero.

Todo lo desarrollado en este apartado es extensible a matrices en banda, que son a las que aplicaremos estas herramientas matemáticas (Davis, 2006).

Una vez expuesto todo lo anterior, ilustraremos el algoritmo con un ejemplo de su utilización, de manera que se facilite su comprensión:

Ejemplo 4.3.4: *supongamos que tenemos el siguiente sistema de ecuaciones $Ax = b$ a resolver:*

$$\begin{bmatrix} 1.67 & 0.09 & 0 & 0 & 0 \\ 0.09 & 1.90 & 0.04 & 0 & 0 \\ 0 & 0.04 & 1.32 & 0.50 & 0 \\ 0 & 0 & 0.50 & 2.75 & 0.61 \\ 0 & 0 & 0 & 0.61 & 1.40 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Si determinamos la descomposición de Cholesky de la matriz A y sustituimos obtenemos que:

$$\left(\begin{bmatrix} 1.29 & 0 & 0 & 0 & 0 \\ 0.07 & 1.38 & 0 & 0 & 0 \\ 0 & 0.03 & 1.15 & 0 & 0 \\ 0 & 0 & 0.44 & 1.60 & 0 \\ 0 & 0 & 0 & 0.38 & 1.12 \end{bmatrix} \cdot \begin{bmatrix} 1.29 & 0.07 & 0 & 0 & 0 \\ 0 & 1.38 & 0.03 & 0 & 0 \\ 0 & 0 & 1.15 & 0.44 & 0 \\ 0 & 0 & 0 & 1.60 & 0.38 \\ 0 & 0 & 0 & 0 & 1.12 \end{bmatrix} \right) \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

aplicando la propiedad asociativa de la multiplicación de matrices:

$$\begin{bmatrix} 1.29 & 0 & 0 & 0 & 0 \\ 0.07 & 1.38 & 0 & 0 & 0 \\ 0 & 0.03 & 1.15 & 0 & 0 \\ 0 & 0 & 0.44 & 1.60 & 0 \\ 0 & 0 & 0 & 0.38 & 1.12 \end{bmatrix} \cdot \left(\begin{bmatrix} 1.29 & 0.07 & 0 & 0 & 0 \\ 0 & 1.38 & 0.03 & 0 & 0 \\ 0 & 0 & 1.15 & 0.44 & 0 \\ 0 & 0 & 0 & 1.60 & 0.38 \\ 0 & 0 & 0 & 0 & 1.12 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

si ahora hacemos el cambio de variable:

$$\begin{bmatrix} 1.29 & 0 & 0 & 0 & 0 \\ 0.07 & 1.38 & 0 & 0 & 0 \\ 0 & 0.03 & 1.15 & 0 & 0 \\ 0 & 0 & 0.44 & 1.60 & 0 \\ 0 & 0 & 0 & 0.38 & 1.12 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

por lo que podemos determinar y resolviendo el sistema. Una vez tengamos este vector intermedio, nos bastará con resolver el sistema:

$$\begin{bmatrix} 1.29 & 0.07 & 0 & 0 & 0 \\ 0 & 1.38 & 0.03 & 0 & 0 \\ 0 & 0 & 1.15 & 0.44 & 0 \\ 0 & 0 & 0 & 1.60 & 0.38 \\ 0 & 0 & 0 & 0 & 1.12 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

para calcular el vector x deseado.

4.4. Multiplicación de matrices inversas mediante Cholesky

Otra posible aplicación de la descomposición de Cholesky, aparte de la de resolver sistemas de ecuaciones, es que nos permite calcular el producto de la inversa de una matriz por otra, sin la necesidad de obtener previamente dicha inversa.

Para ello, supongamos que debemos calcular $A = B^{-1} \cdot U$ donde B y U son dos matrices de dimensiones compatibles y existe la inversa de B. Si ahora multiplicamos a ambos lados de la igualdad por la propia matriz B, el resultado sería el siguiente:

$$B \cdot A = B \cdot B^{-1} \cdot U = I \cdot U = U$$

de manera que finalmente tenemos que:

$$B \cdot A = U$$

sistema matricial que puede descomponerse en tantos subsistemas de ecuaciones como columnas tenga U del siguiente modo:

$$B \cdot A_k = U_k \quad con \quad k = 1,2,3, \dots, N_c$$

Si determinamos la descomposición de Cholesky de la matriz B y aplicamos el método de resolución anteriormente visto, es posible determinar k vectores columna con los que construir la matriz A, de

modo y forma que hallamos calculado la multiplicación de la inversa de B por U sin tener que determinar la inversa previamente.

Cabe decir que todo lo anterior se ha implementado teniendo presente la filosofía *sparse* utilizada en nuesto proyecto, por lo que las matrices se presuponen codificadas de manera simple (ni diadas ni flechas) y ordenada.

El pseudocódigo que modela este algoritmo de multiplicación es el siguiente:

1. Inicializamos un índice para recorrer la matriz A resultado.

2. Procedemos por columnas empleando otro índice, e inicializamos a cero dos vectores auxiliares, U_j y A_j, para los subsistemas a resolver.

3. Recorremos la codificación de U por medio de un tercer índice, de manera que busquemos las ternas que cumplan que sus índices $C_U = j$.

4. Asignamos el valor no nulo Z_U de las ternas anteriores a las componentes del vector auxiliar U_j previamente inicializado.

5. Montamos el subsistema $B \cdot A_j = U_j$ y lo resolvemos mediante el método expuesto en el apartado anterior, de forma que calculemos A_j.

6. Una vez resulto el subsistema, guardamos las componentes no nulas del vector A_j recién obtenido en la codificación de la matriz A que queremos calcular (para ello usamos el índice del punto 1 de este pseudocódigo).

7. Repetimos el proceso de los puntos 2 a 6 hasta haber acabado con todas las columnas, de modo que hallamos obtenido la codificación completa de A que se perseguía.

8. Una vez hecho esto, actualizamos el tamaño de los datos codificados en A igualándolo al índice del punto 1 y habremos terminado.

Este procedimiento nos será de gran utilidad para implementar el método de resolución de sistemas de ecuaciones en semi banda, que será expuesto en el capítulo siguiente.

5 SISTEMAS DE ECUACIONES EN SEMI BANDA

La felicidad ininterrumpida aburre. Debe tener alternativas.

- Jean-Baptiste Poquelin (Molière) -

A lo largo de este trabajo se han presupuesto una serie de condiciones necesarias para poder aplicar nuestras funciones al sistema de ecuaciones a resolver. Una de las hipótesis fundamentales es que la matriz A del sistema $Ax = b$ es una matriz *sparse* en banda estrictamente hablando, es decir, ningún elemento ajeno a la banda es no nulo. Sin embargo, en la mayoría de los casos esto no es así, salvo en contadas ocasiones. Por ello, se ha implementado una función que permite resolver un sistema cuya matriz incluya elementos no nulos fuera de la banda.

Un ejemplo posible de esto podría ser el siguiente (matriz ejemplo, no es relevante si no es semidefinida positiva):

$$\begin{bmatrix} \mathbf{1.0} & \mathbf{0.1} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \mathbf{0.1} & \mathbf{2.0} & \mathbf{0.2} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{0.2} & \mathbf{3.0} & \mathbf{0.3} & 0.0 & 0.0 & 0.0 & \mathbf{1.1} & 0.0 & \mathbf{2.2} \\ 0.0 & 0.0 & \mathbf{0.3} & \mathbf{4.0} & \mathbf{0.4} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & \mathbf{0.4} & \mathbf{5.0} & \mathbf{0.5} & 0.0 & 0.0 & \mathbf{3.3} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.5} & \mathbf{6.0} & \mathbf{0.6} & 0.0 & 0.0 & \mathbf{4.4} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{0.6} & \mathbf{7.0} & \mathbf{0.7} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{1.1} & 0.0 & 0.0 & 0.0 & \mathbf{0.7} & \mathbf{8.0} & \mathbf{0.8} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{3.3} & 0.0 & 0.0 & \mathbf{0.8} & \mathbf{9.0} & \mathbf{0.9} \\ 0.0 & 0.0 & \mathbf{2.2} & 0.0 & 0.0 & \mathbf{4.4} & 0.0 & 0.0 & \mathbf{0.9} & \mathbf{10.0} \end{bmatrix}$$

Como podemos ver la matriz sigue siendo simétrica y principalmente en banda, pero con elementos no nulos fuera de la misma. Si intentáramos resolver directamente este sistema de la manera explicada anteriormente, es posible que fuéramos capaces de obtener una solución, sin embargo, ésta presentaría imprecisiones como resultado de los términos añadidos. Además de esto, lo más probable es que nuestro programa dé como resultado un error y termine su ejecución sin llegar a calcular la descomposición de Cholesky. Esto se debe a que se ha generado el código para funcionar bajo hipótesis bastante estrictas, de manera que se obtenga la eficiencia requerida.

Por tanto, es necesario modificar el *modus operandi* de nuestro algoritmo si queremos abarcar casos más amplios. Para ello, se ha empleado el método de resolución de sistemas de ecuaciones en semi banda, que será descrito a continuación.

5.1. Fundamentos teóricos

El método de resolución de sistemas de ecuaciones en semi banda se basa en la posibilidad de definir la matriz de nuestro sistema:

$$Wx = h$$

como:

$$W = B + U \cdot V^t$$

donde *B* es una matriz en banda estricta, mientras que *U* y *V* son matrices que contienen diadas que dan como resultado los elementos no nulos ajenos a la banda que hemos visto antes.

Así, empleando esta definición de *W*, siendo tanto ésta como *B* invertibles, si aplicamos el teorema del determinante de Sylvester tenemos que:

$$0 \neq det(B + U \cdot V^t) = det(B) \cdot det(I + V^t \cdot B^{-1} \cdot U)$$

y puesto que *B* es invertible, su determinante es no nulo, $det(B) \neq 0$, por lo que podemos deducir que el $det(I + V^t \cdot B^{-1} \cdot U) \neq 0$, lo que implica que $I + V^t \cdot B^{-1} \cdot U$ es invertible.

De este modo, aplicando la igualdad de Woodbury nos encontramos con que:

$$W^{-1} = (B + U \cdot V^t)^{-1} = B^{-1} - B^{-1} \cdot U \cdot (I + V^t \cdot B^{-1} \cdot U)^{-1} \cdot V^t \cdot B^{-1}$$

Si ahora definimos:

- z_1 como solución del subsistema $Bz_1 = h$

- z_2 como solución del subsistema $(I + V^t \cdot B^{-1} \cdot U)z_2 = V^t z_1$

- z_3 como solución del subsistema $Bz_3 = Uz_2$

Sustituyendo en nuestro sistema $Wx = h$, el vector de soluciones puede obtenerse tal que así:

$$\begin{aligned} \boldsymbol{x} &= W^{-1} \cdot h \\ &= (B^{-1} - B^{-1} \cdot U \cdot (I + V^t \cdot B^{-1} \cdot U)^{-1} \cdot V^t \cdot B^{-1}) \cdot h \\ &= z_1 - B^{-1} \cdot U \cdot (I + V^t \cdot B^{-1} \cdot U)^{-1} \cdot V^t \cdot z_1 \\ &= z_1 - B^{-1} \cdot U \cdot z_2 \\ &= \boldsymbol{z_1 - z_3} \end{aligned}$$

Por lo que podemos hallar *x* indirectamente si resolvemos los subsistemas de z_1, z_2, z_3, siendo posibles de obtener aplicando nuestras funciones, dado que z_1 y z_3 tienen asociados subsistemas *sparse* en banda estricta, mientras que el de z_2 es denso, pero de tamaño igual al número de diadas que tengamos (supuesto pequeño respecto al tamaño de *B*) (Alamo, 2017).

Para facilitar la comprensión de este método, expondremos a continuación un ejemplo que simule un caso genérico:

Ejemplo 5.1.1: *supongamos que tenemos el siguiente sistema de ecuaciones $Wx = h$ a resolver:*

$$\begin{bmatrix} 2.09 & 0.95 & 0.67 & 0.86 & 0.92 \\ 0.55 & 1.70 & 0.36 & 0.30 & 0.33 \\ 0.15 & 0.42 & 1.99 & 0.55 & 0.56 \\ 0.06 & 0.11 & 0.19 & 1.52 & 0.55 \\ 0.19 & 0.38 & 0.53 & 1.01 & 2.18 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Como puede verse, la matriz W no es ni en banda ni sparse, por lo que no podemos aplicar el procedimiento de resolución de sistemas de ecuaciones mediante Cholesky que se vió en el capítulo anterior.

Por tanto, nos aprovecharemos de que la matriz W se ha obtenido como $B + U \cdot V^T$ como se ha visto en teoría, quedando el sistema anterior con la siguiente estructura:

$$\left(\begin{bmatrix} 1.85 & 0.46 & 0 & 0 & 0 \\ 0.46 & 1.53 & 0.12 & 0 & 0 \\ 0 & 0.12 & 1.58 & 0.03 & 0 \\ 0 & 0 & 0.03 & 1.32 & 0.34 \\ 0 & 0 & 0 & 0.34 & 1.46 \end{bmatrix} + \begin{bmatrix} 0.96 \\ 0.34 \\ 0.59 \\ 0.22 \\ 0.75 \end{bmatrix} \cdot [\, 0.26 \;\; 0.51 \;\; 0.70 \;\; 0.89 \;\; 0.96 \,]\right) \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Así, podemos aplicar el método de resolución de sistemas de ecuaciones en semi banda de forma que obtengamos x como $z_1 - z_3$ para lo que tenemos que determinar z_1, z_1, z_3 resolviendo los siguientes subsistemas:

$$Bz_1 = h \rightarrow \begin{bmatrix} 1.85 & 0.46 & 0 & 0 & 0 \\ 0.46 & 1.53 & 0.12 & 0 & 0 \\ 0 & 0.12 & 1.58 & 0.03 & 0 \\ 0 & 0 & 0.03 & 1.32 & 0.34 \\ 0 & 0 & 0 & 0.34 & 1.46 \end{bmatrix} \cdot \begin{bmatrix} z_{11} \\ z_{12} \\ z_{13} \\ z_{14} \\ z_{15} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$(I + V^t \cdot B^{-1} \cdot U) \cdot z_2 = V^t z_1 \rightarrow 1.92 \cdot z_2 = [\, 0.26 \;\; 0.51 \;\; 0.70 \;\; 0.89 \;\; 0.96 \,] \cdot \begin{bmatrix} z_{11} \\ z_{12} \\ z_{13} \\ z_{14} \\ z_{15} \end{bmatrix}$$

$$Bz_3 = Uz_2 \rightarrow \begin{bmatrix} 1.85 & 0.46 & 0 & 0 & 0 \\ 0.46 & 1.53 & 0.12 & 0 & 0 \\ 0 & 0.12 & 1.58 & 0.03 & 0 \\ 0 & 0 & 0.03 & 1.32 & 0.34 \\ 0 & 0 & 0 & 0.34 & 1.46 \end{bmatrix} \cdot \begin{bmatrix} z_{31} \\ z_{32} \\ z_{33} \\ z_{34} \\ z_{35} \end{bmatrix} = \begin{bmatrix} 0.96 \\ 0.34 \\ 0.59 \\ 0.22 \\ 0.75 \end{bmatrix} \cdot z_2$$

Del primer subsistema obtenemos z_1, del segundo z_2 y del tercero z_3 como puede verse. A la vista de esto, es preciso recalcar que en este caso z_2 es un escalar dado de que sus dimensiones vienen fijadas por el número de diadas que tenga nuestro sistema, de haber constado de dos en lugar de una z_2 habría tenido dimensiones 2x1 y la matriz $(I + V^t \cdot B^{-1} \cdot U)$ habría sido de 2x2 y densa. Esto último supone el mayor esfuerzo de computación dado que debemos resolver un sistema de ecuaciones que no es sparse, sin embargo, a priori, sus dimensiones siempre serán mucho menores que las del sistema original, pues una de las hipótesis de partida es que el número de diadas es mucho menor que el tamaño del sistema.

6 TIEMPOS DE EJECUCIÓN

Se dice que el tiempo es un gran maestro,
lo malo es que va matando a sus discípulos.

- Louis Hector Berlioz -

A continuación, se analizan los tiempos de ejecución de las funciones más relevantes de entre las desarrolladas en este proyecto. El objetivo es analizar la carga computacional y mostrar de manera visual e intuitiva la eficiencia de las mismas para diferentes datos de entrada. Por último, al final de este capítulo, se expone el modo en que se han generado los datos mediante MatLab en base a las especificaciones pertinentes (tamaño del sistema, número de diadas, densidad de la matriz, etc.), así como todo lo necesario para crear los gráficos incluidos en este capítulo.

6.1. Tiempos de ejecución esperados

En primer lugar, determinaremos los tiempos de ejecución teóricos de cada función principal, de manera que puedan tomarse como referencia para comparar con los resultados experimentales de las simulaciones.

Para llevar a cabo esto, se han analizado las funciones C principales de manera que pueda determinarse el número de iteraciones y cálculos realizados por las mismas. La finalidad de esto es obtener una expresión dependiente de los datos de entrada, que nos dé una cota superior del número de operaciones llevadas a cabo por cada código.

6.1.1 Lectordatos

La primera de las funciones a analizar es la de toma de datos de un fichero .txt externo. Dado que este código recorrer en su totalidad el archivo de texto, es lógico pensar que el número de pasos dados está directamente relacionado con el tamaño de los mismos y, al tratarse de la lectura de matrices, el resultado no es más que:

$$\boldsymbol{Lectordatos \rightarrow N_r \cdot N_c \rightarrow N^2} \ (para\ el\ caso\ de\ matrices\ cuadradas)$$

siendo N_r y N_c el número de filas y columnas de la matriz a leer respectivamente.

6.1.2 Descomposición de Cholesky

La segunda función es la que permite obtener la descomposición de Cholesky de una matriz cuadrada cualquiera. En realidad, se han creado dos versiones que implementan esto, teniendo para la primera de estas lo siguiente:

$$Cholesky\ v1 \rightarrow N \cdot (1 + 2 + 3 + \cdots + N) \cdot \big(1 + 2 + 3 + \cdots + (N - 1)\big)$$

donde N es el tamaño de la matriz de la que se desea obtener la descomposición.

Si ahora sumamos las series empleando la expresión:

$$\sum_{k=1}^{N} k = \frac{(1 + N) \cdot N}{2}$$

podemos simplificar la expresión del siguiente modo:

$$Cholesky\ v1 \rightarrow N \cdot \frac{(1 + N) \cdot N}{2} \cdot \frac{\big(1 + (N - 1)\big) \cdot (N - 1)}{2}$$

y agrupando términos llegamos a:

$$Cholesky\ v1 \rightarrow N \cdot \frac{(1 + N) \cdot N}{2} \cdot \frac{N \cdot (N - 1)}{2}$$

$$Cholesky\ v1 \rightarrow \frac{-N^3}{4} \cdot (1+N) \cdot (1-N)$$

$$Cholesky\ v1 \rightarrow \frac{-N^3}{4} \cdot (1-N^2)$$

$$\boldsymbol{Cholesky\ v1 \rightarrow \frac{N^3}{4} \cdot (N^2-1)}$$

expresión que depende únicamente de la dimensión de la matriz de entrada.

Mientras que para la segunda versión tenemos:

$$Cholesky\ v2 \rightarrow N \cdot \left[N + \left(1+2+3+\cdots+(N-1)\right) + N \cdot N\right]$$

donde N es el tamaño de la matriz de la que se desea obtener la descomposición.

Si ahora sumamos las series empleando la expresión:

$$\sum_{k=1}^{N} k = \frac{(1+N) \cdot N}{2}$$

podemos simplificar la expresión del siguiente modo:

$$Cholesky\ v2 \rightarrow N \cdot \left[N + \frac{\left(1+(N-1)\right) \cdot (N-1)}{2} + N \cdot N\right]$$

y agrupando términos llegamos a:

$$Cholesky\ v2 \rightarrow N \cdot \left[N + \frac{N \cdot (N-1)}{2} + N^2\right]$$

$$Cholesky\ v2 \rightarrow N \cdot \left[N + \frac{N^2 - N}{2} + N^2\right]$$

$$Cholesky\ v2 \rightarrow N \cdot \left[\frac{2 \cdot N + N^2 - N + 2 \cdot N^2}{2}\right]$$

$$\boldsymbol{Cholesky\ v2} \rightarrow N \cdot \left[\frac{3 \cdot N^2 + N}{2}\right] = \boldsymbol{N^2 \cdot \left[\frac{3 \cdot N + 1}{2}\right]}$$

expresión que depende únicamente de la dimensión de la matriz de entrada.

La primera de las versiones es la que se ha usado como base para construir la versión *sparse*, mostrada a continuación.

6.1.3 Descomposición de Cholesky *sparse*

La tercera función es la que permite obtener la descomposición de Cholesky de una matriz *sparse*, para las dos versiones desarrolladas. Así, para la primera de ellas tenemos que:

$$Cholesky\ sparse\ v1 \rightarrow N \cdot (1 + 2 + 3 + \cdots + N) \cdot \big(1 + 2 + 3 + \cdots + (N-1)\big) \cdot (M_{Rc} + M_A + M_{Rc})$$

donde N es el tamaño de la matriz A de la que deseamos obtener la descomposición, M_{Rc} es el tamaño de los datos codificados (número de elementos no nulos) en la matriz R_c resultado de la descomposición y M_A el tamaño de la tabla de ternas de A (número de elementos no nulos).

Si ahora sumamos las series empleando la expresión:

$$\sum_{k=1}^{N} k = \frac{(1+N)\cdot N}{2}$$

podemos simplificar la expresión del siguiente modo:

$$Cholesky\ sparse\ v1 \rightarrow\ N \cdot \frac{(1+N)\cdot N}{2} \cdot \frac{\big(1+(N-1)\big)\cdot (N-1)}{2} \cdot (M_{Rc} + M_A + M_{Rc})$$

y agrupando términos llegamos a:

$$Cholesky\ sparse\ v1 \rightarrow\ N\ \cdot \frac{(1+N)\cdot N}{2}\ \cdot \frac{N\cdot (N-1)}{2}\ \cdot (M_A + 2\cdot M_{Rc})$$

$$Cholesky\ sparse\ v1 \rightarrow \frac{-N^3}{4} \cdot (1+N)\cdot (1-N)\cdot (M_A + 2\cdot M_{Rc})$$

$$Cholesky\ sparse\ v1 \rightarrow \frac{-N^3}{4} \cdot (1-N^2)\cdot (M_A + 2\cdot M_{Rc})$$

$$Cholesky\ sparse\ v1 \rightarrow \frac{N^3}{4} \cdot (N^2-1)\cdot (M_A + 2\cdot M_{Rc})$$

y si ahora sustituimos $M_A < L\cdot N$ y $M_{Rc} < \left(\frac{(L-1)}{2}+1\right)\cdot N$ siendo L el ancho de la banda, obtenemos que:

$$Cholesky\ sparse\ v1 \rightarrow \frac{N^3}{4} \cdot (N^2-1)\cdot \left(L\cdot N + 2\cdot \left(\frac{(L-1)}{2}+1\right)\cdot N\right)$$

$$Cholesky\ sparse\ v1 \rightarrow \frac{N^3}{4} \cdot (N^2-1)\cdot N\cdot \Big(L + \big((L-1)+2\big)\Big)$$

$$\boldsymbol{Cholesky\ sparse\ v1 \rightarrow \frac{N^4}{4} \cdot (N^2 - 1) \cdot (2 \cdot L + 1)}$$

expresión que depende únicamente de los datos de entrada: dimensiones de la matriz y ancho de la banda.

Mientras que para la segunda versión tenemos:

$$Cholesky\ sparse\ v2 \rightarrow M_A + \left[M_A \cdot \left((N-1) \cdot (M_{Rc} + M_{Rc}) + M_{Rc}\right)\right]$$

donde N es el tamaño de la matriz A de la que deseamos obtener la descomposición, M_{Rc} es el tamaño de los datos codificados (número de elementos no nulos) en la matriz R_c resultado de la descomposición y M_A el tamaño de la tabla de ternas de A (número de elementos no nulos).

Agrupando términos llegamos a:

$$Cholesky\ sparse\ v2 \rightarrow M_A + \left[M_A \cdot \left((N-1) \cdot 2 \cdot M_{Rc} + M_{Rc}\right)\right]$$

$$Cholesky\ sparse\ v2 \rightarrow M_A + [M_A \cdot M_{Rc} \cdot (2 \cdot (N-1) + 1)]$$

$$Cholesky\ sparse\ v2 \rightarrow M_A + [M_A \cdot M_{Rc} \cdot (2 \cdot N - 2 + 1)]$$

$$Cholesky\ sparse\ v2 \rightarrow M_A \cdot [1 + M_{Rc} \cdot (2 \cdot N - 1)]$$

y si ahora sustituimos $M_A < L \cdot N$ y $M_{Rc} < \left(\frac{(L-1)}{2} + 1\right) \cdot N$ siendo L el ancho de la banda, obtenemos que:

$$Cholesky\ sparse\ v2 \rightarrow L \cdot N \cdot \left[1 + \left(\frac{(L-1)}{2} + 1\right) \cdot N \cdot (2 \cdot N - 1)\right]$$

$$Cholesky\ sparse\ v2 \rightarrow \frac{L \cdot N}{2} \cdot \left[2 + \left((L-1)+2\right) \cdot N \cdot (2 \cdot N - 1)\right]$$

$$Cholesky\ sparse\ v2 \rightarrow \left[L \cdot N + \frac{L \cdot N}{2} \cdot (L+1) \cdot N \cdot (2 \cdot N - 1)\right]$$

$$Cholesky\ sparse\ v2 \rightarrow \left[L \cdot N + \left(\frac{L^2 \cdot N}{2} + \frac{L \cdot N}{2}\right) \cdot N \cdot (2 \cdot N - 1)\right]$$

$$Cholesky\ sparse\ v2 \rightarrow L \cdot N + \frac{L^2 \cdot N}{2} \cdot (2 \cdot N^2 - N) + \frac{L \cdot N}{2} \cdot (2 \cdot N^2 - N)$$

$$Cholesky\ sparse\ v2 \rightarrow L \cdot N + \frac{L^2}{2} \cdot (2 \cdot N^3 - N^2) + \frac{L}{2} \cdot (2 \cdot N^3 - N^2)$$

$$Cholesky\ sparse\ v2 \rightarrow L \cdot N + 2 \cdot N^3 \cdot \frac{L^2}{2} - N^2 \cdot \frac{L^2}{2} + 2 \cdot N^3 \cdot \frac{L}{2} - N^2 \cdot \frac{L}{2}$$

$$Cholesky\ sparse\ v2 \rightarrow \frac{2 \cdot L \cdot N}{2} + \frac{2 \cdot N^3 \cdot L^2}{2} - \frac{N^2 \cdot L^2}{2} + \frac{2 \cdot N^3 \cdot L}{2} - \frac{N^2 \cdot L}{2}$$

$$Cholesky\ sparse\ v2 \rightarrow \frac{(2 \cdot N^3 - N^2) \cdot L^2 + (2 \cdot N^3 - N^2 + 2 \cdot N) \cdot L}{2}$$

$$Cholesky\ sparse\ v2 \rightarrow \left(N^3 - \frac{N^2}{2}\right) \cdot L^2 + \left(N^3 - \frac{N^2}{2} + N\right) \cdot L$$

$$\boldsymbol{Cholesky\ sparse\ v2 \rightarrow N \cdot L \cdot \left[\left(N - \frac{1}{2}\right) \cdot N \cdot L + \left(N - \frac{1}{2}\right) \cdot N + 1\right]}$$

expresión que depende únicamente de los datos de entrada: dimensiones de la matriz y ancho de la banda.

6.1.4 Resolución de sistemas *sparse*

La tercera función es la que permite obtener la descomposición de Cholesky de una matriz *sparse*, para las dos versiones desarrolladas. Así, para la primera de ellas tenemos que:

$$Sistecchol\ sparse$$
$$\rightarrow N + M_{Rc} + (N-1)\cdot\left[M_{Rc} + \left(1+2+3+\cdots+(N-1)\right)\cdot M_{Rc}\right] + M_{Rc}$$
$$+ (N-1)\cdot\left[M_{Rc} + \left(1+2+3+\cdots+(N-2)\right)\cdot M_{Rc}\right]$$

donde N es el tamaño de la matriz de la que deseamos obtener la descomposición y M_{Rc} es el tamaño de los datos codificados (número de elementos no nulos) en la matriz R_c resultado de la descomposición.

Si ahora sumamos las series empleando la expresión:

$$\sum_{k=1}^{N} k = \frac{(1+N)\cdot N}{2}$$

podemos simplificar la expresión del siguiente modo:

$$Sistecchol\ sparse$$
$$\rightarrow N + M_{Rc} + (N-1)\cdot\left[M_{Rc} + \frac{\left(1+(N-1)\right)\cdot(N-1)}{2}\cdot M_{Rc}\right] + M_{Rc}$$
$$+ (N-1)\cdot\left[M_{Rc} + \frac{\left(1+(N-2)\right)\cdot(N-2)}{2}\cdot M_{Rc}\right]$$

y agrupando términos llegamos a:

$$Sistecchol\ sparse \rightarrow N + M_{Rc} + (N-1)\cdot\left[M_{Rc} + \frac{N\cdot(N-1)}{2}\cdot M_{Rc}\right] + M_{Rc} + (N-1) \cdot\left[M_{Rc} + \frac{(N-1)\cdot(N-2)}{2}\cdot M_{Rc}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot\left[1 + \frac{N\cdot(N-1)}{2}\right] + (N-1)\cdot M_{Rc} \cdot\left[1 + \frac{(N-1)\cdot(N-2)}{2}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot\left[1 + \frac{N\cdot(N-1)}{2} + 1 + \frac{(N-1)\cdot(N-2)}{2}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot\left[\frac{4 + N\cdot(N-1) + (N-1)\cdot(N-2)}{2}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot\left[\frac{4 + N^2 - N + N^2 - 2\cdot N - N + 2}{2}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot\left[\frac{2\cdot N^2 - 4\cdot N + 6}{2}\right]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + (N-1)\cdot M_{Rc}\cdot[N^2 - 2\cdot N + 3]$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + M_{Rc}\cdot(N^3 - 2\cdot N^2 + 3\cdot N - N^2 + 2\cdot N - 3)$$

$$Sistecchol\ sparse \rightarrow N + 2\cdot M_{Rc} + M_{Rc}\cdot(N^3 - 3\cdot N^2 + 5\cdot N - 3)$$

$$Sistecchol\ sparse \rightarrow N + M_{Rc}\cdot(N^3 - 3\cdot N^2 + 5\cdot N - 1)$$

y si ahora sustituimos $M_{Rc} < \left(\frac{(L-1)}{2} + 1\right) \cdot N$ siendo L el ancho de la banda, obtenemos que:

$$Sistecchol\ sparse \rightarrow N + \left(\frac{(L-1)}{2} + 1\right) \cdot N \cdot (N^3 - 3 \cdot N^2 + 5 \cdot N - 1)$$

$$Sistecchol\ sparse \rightarrow N + \left(\frac{L+1}{2}\right) \cdot (N^4 - 3 \cdot N^3 + 5 \cdot N^2 - N)$$

$$Sistecchol\ sparse \rightarrow N + \frac{L}{2} \cdot (N^4 - 3 \cdot N^3 + 5 \cdot N^2 - N) + \frac{1}{2} \cdot (N^4 - 3 \cdot N^3 + 5 \cdot N^2 - N)$$

$$Sistecchol\ sparse \rightarrow N + L \cdot N \cdot \left(\frac{N^3}{2} - \frac{3}{2} \cdot N^2 + \frac{5}{2} \cdot N - \frac{1}{2}\right) + N \cdot \left(\frac{N^3}{2} - \frac{3}{2} \cdot N^2 + \frac{5}{2} \cdot N - \frac{1}{2}\right)$$

$$Sistecchol\ sparse \rightarrow L \cdot N \cdot \left(\frac{N^3}{2} - \frac{3}{2} \cdot N^2 + \frac{5}{2} \cdot N - \frac{1}{2}\right) + N \cdot \left(\frac{N^3}{2} - \frac{3}{2} \cdot N^2 + \frac{5}{2} \cdot N + \frac{1}{2}\right)$$

expresión que depende únicamente de los datos de entrada: dimensiones de la matriz y ancho de la banda (Boy, 2009).

6.2. Lectura de datos

En segundo lugar, analizaremos el tiempo que nuestro programa tarda en cargar los datos desde un fichero externo .txt que previamente ha sido generado con MatLab (ver últimos apartados de este capítulo). En la figura 6.2.1, puede verse cómo el tiempo empleado en leer los datos del fichero depende exclusivamente del tamaño de la matriz, lo que es de esperar dado el código de la misma (recorre todo el fichero independientemente de que haya ceros o no). Así, tenemos una relación de tipo exponencial entre el tiempo de ejecución de esta función y el tamaño de la matriz (cuadrada en el ensayo) contenida en el fichero de texto. No obstante, el ancho de la banda no parece influir por muchos elementos no nulos que ésta aporte.

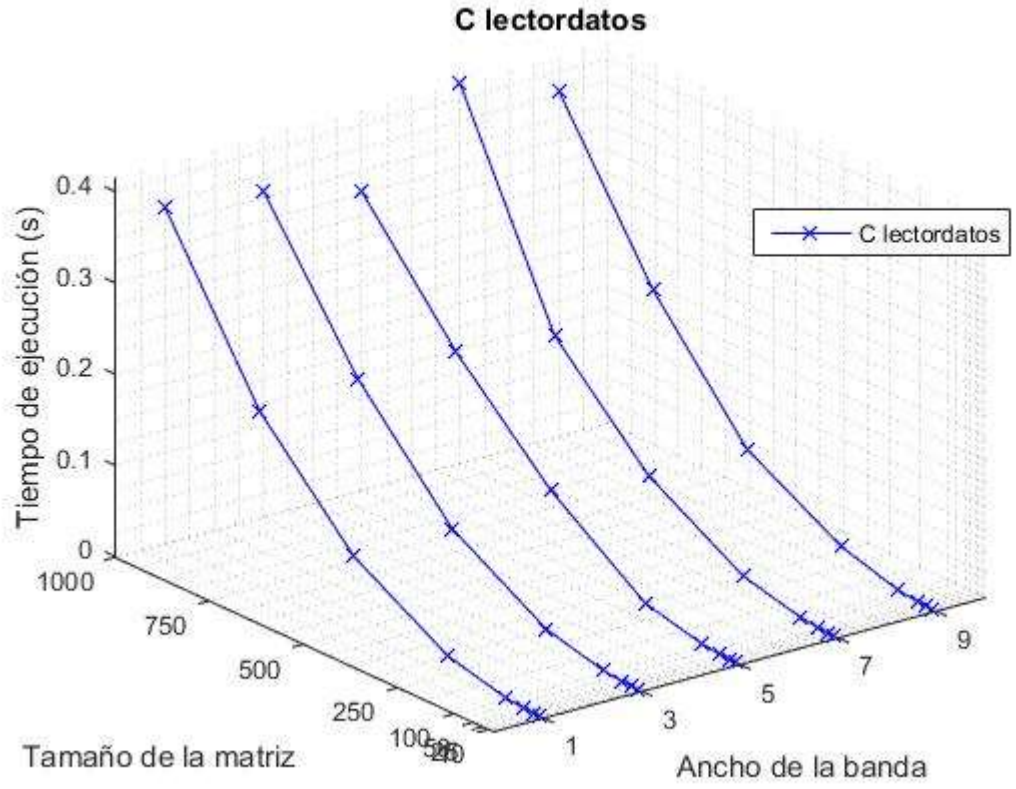

Figura 6.2.1

Para poder apreciar con mayor facilidad la dependencia del tiempo con el tamaño de la matriz, se empleará la expresión vista en el punto 6.1.1, que nos decía que el número de operaciones llevadas a cabo era del orden de N^2. Por consiguiente, los tiempos de ejecución deberán ser proporcionales a esta expresión.

Para poder mostrar esto gráficamente, se dividirán los tiempos de ejecución entre el tamaño de la matriz al cuadrado de modo que obtengamos la figura 6.2.2 ($Relación = \sqrt{tiempos}$).

Como puede apreciarse en la misma, el eje de ordenadas muestra los tiempos de ejecución divididos entre N^2, de manera que la forma de las curvas se asemeja considerablemente a la de una recta, quedando así demostrada la relación proporcional entre el tiempo y el cuadrado del tamaño de la matriz.

En cuanto a la eficiencia, cabe decir que, comparado con el tiempo que precisa MatLab para escribir los datos, nuestra función de lectura de datos (*lectordatos_sparse*) es considerablemente más rápida, como puede apreciarse en la figura 6.2.3.

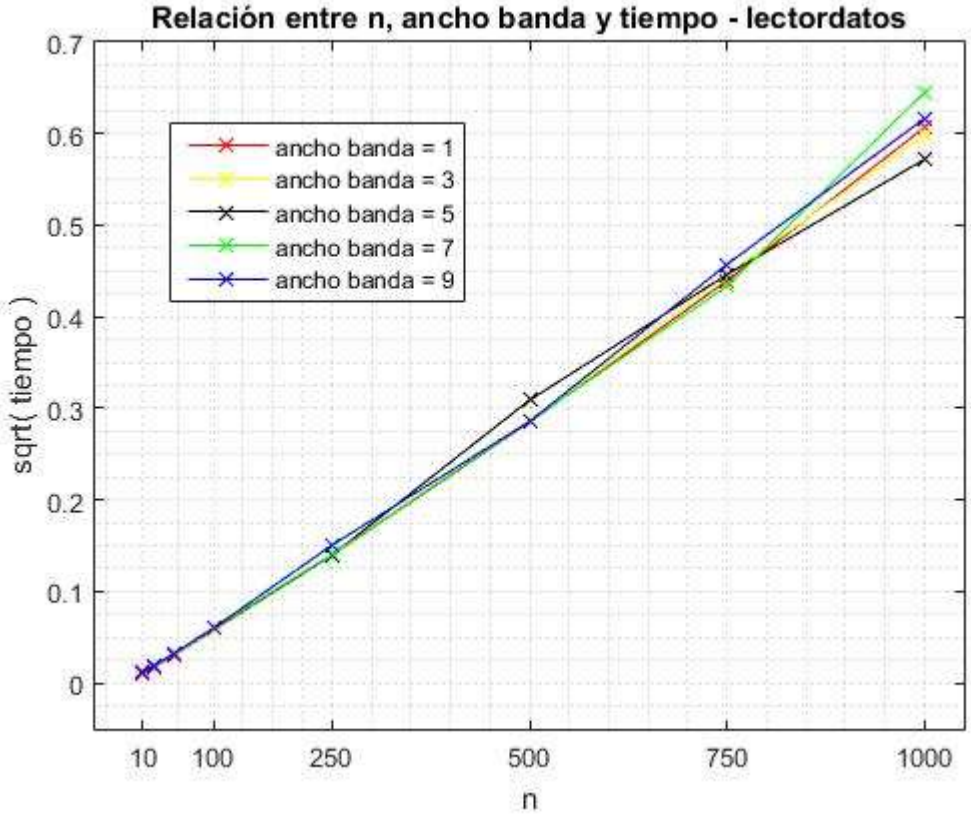

Figura 6.2.2

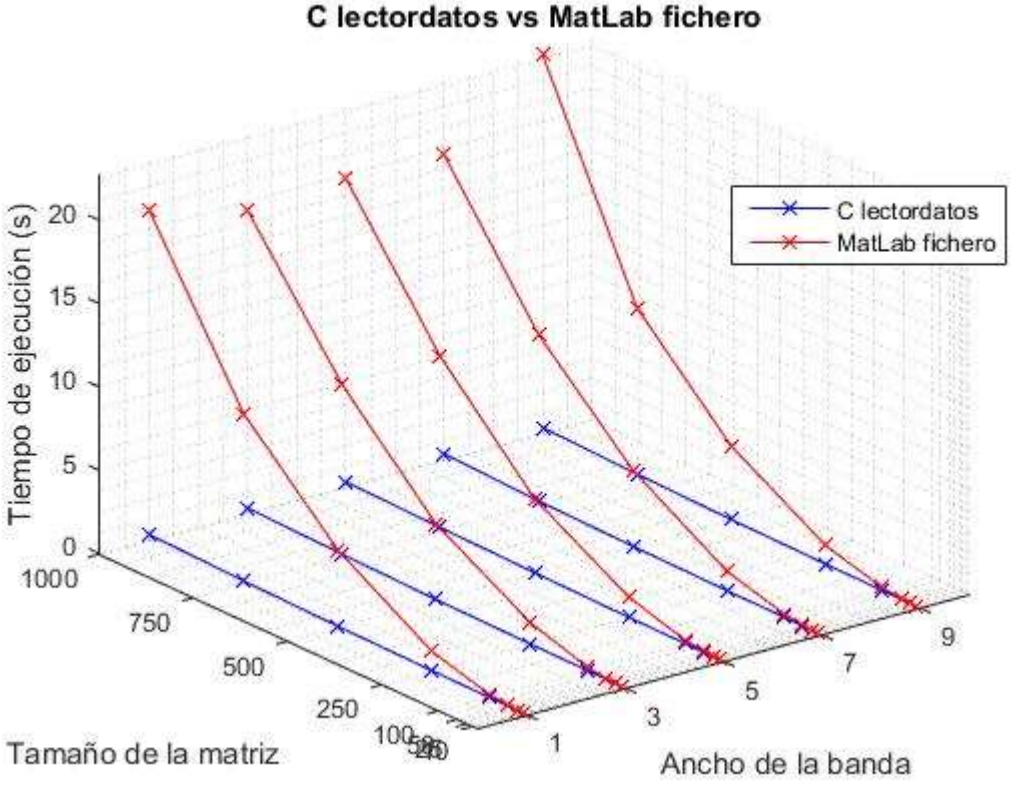

Figura 6.2.3

6.3. Tiempos de ejecución de Cholesky

En tercer lugar, abordaremos una de las funciones de mayor peso en el proyecto, aquella que nos permite calcular la descomposición de Cholesky de la matriz que le indiquemos. No obstante, deberemos distinguir entre las dos versiones desarrolladas en nuestro proyecto y cuyos códigos se exponen en el apéndice A de este documento.

De este modo, pasamos a visualizar los tiempos de la primera versión disponible:

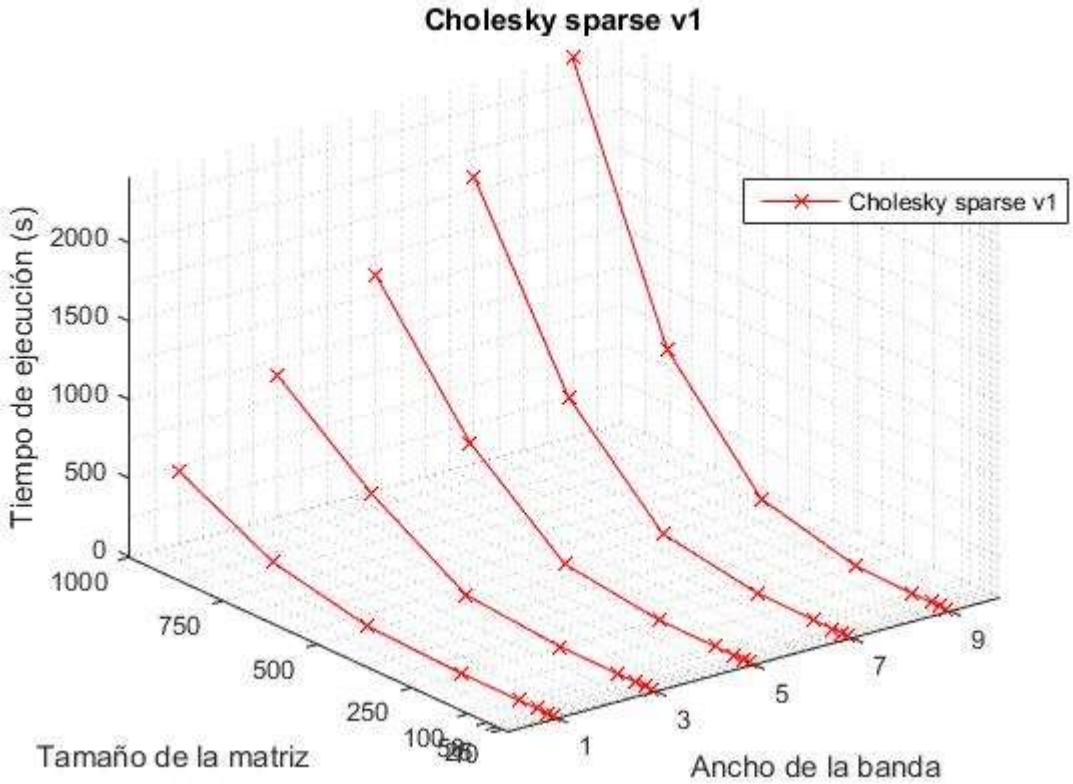

Figura 6.3.1

Como puede apreciarse, existe aparentemente una relación exponencial con el tamaño de la matriz, que es acentuada a medida que el ancho de la banda de la misma crece. Si recordamos lo desarrollado en el punto 6.6.2, llegamos a una expresión que relacionaba una cota superior del número de operaciones llevadas a cabo con el tamaño de la matriz y el ancho de la banda. No obstante, dada la complejidad de esta relación, obtaremos por llevar a cabo el proceso análogo al del apartado anterior empleando $L^2 \cdot N$ como expresión (esto es así dado que es sabido que el número de operaciones de una descomposición de Cholesky de una matriz en banda es de este orden).

Asimismo, la figura 6.3.2 muestra las siguientes curvas $\left(Relación = \sqrt{(tiempos/N)} \right)$:

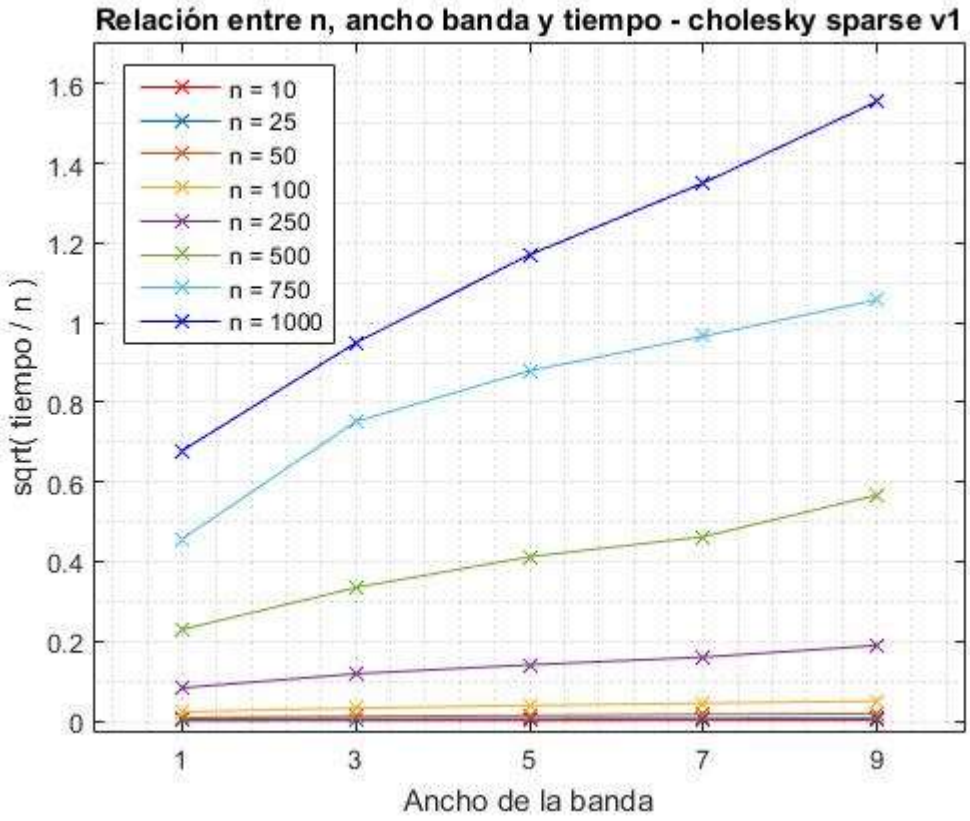

Figura 6.3.2

donde pueden verse una serie de rectas correspondiente a cada tamaño de matriz calculado. Esto prueba la proporcionalidad entre el tiempo de ejecución y el número de operaciones estimado.

Del mismo modo, en cuanto a la relación entre el tamaño de la matriz y el tiempo, tenemos las curvas obtenidas en la figura 6.3.3 ($Relación = tiempos/L^2$).

Estas curvas, tienen una forma exponencial como ya se intuyó al comienzo del apartado, si embargo, dicha tendencia decrece conforme aumenta el tamaño de la matriz, lo que implica una mayor influencia del ancho de la banda frente al tamaño de la matriz (a medida que el ancho de bada aumenta, la tendencia exponencial con el tamaño disminuye).

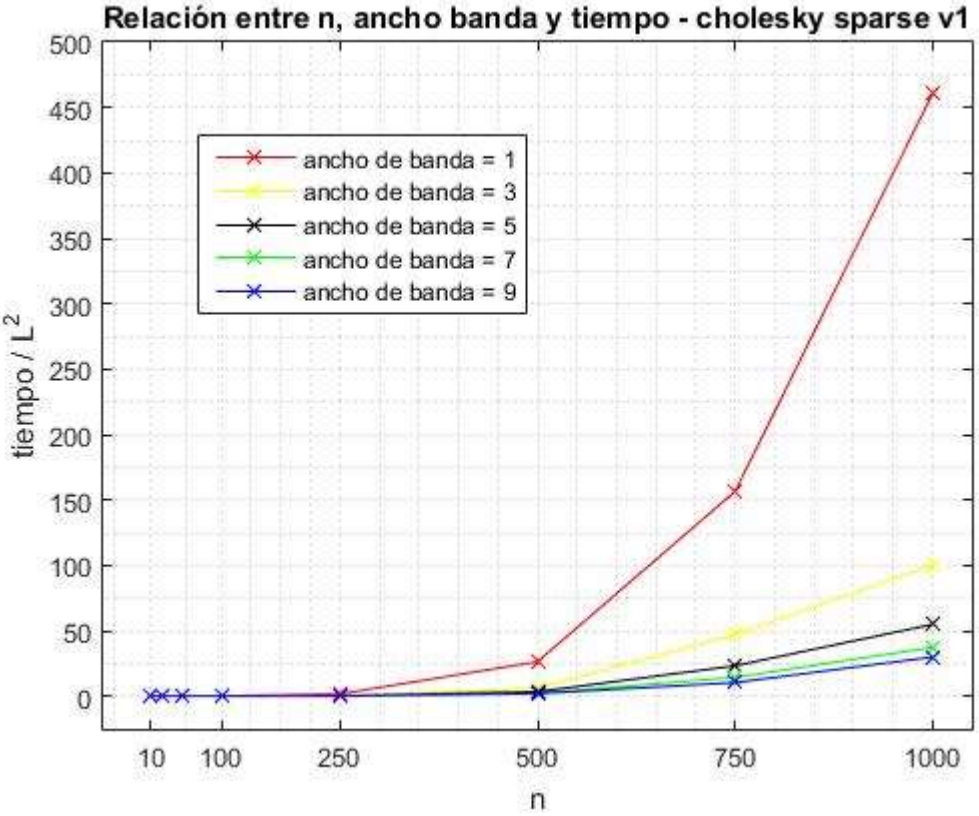

Figura 6.3.3

En cambio, nuestra segunda versión presenta unos tiempos tal que así:

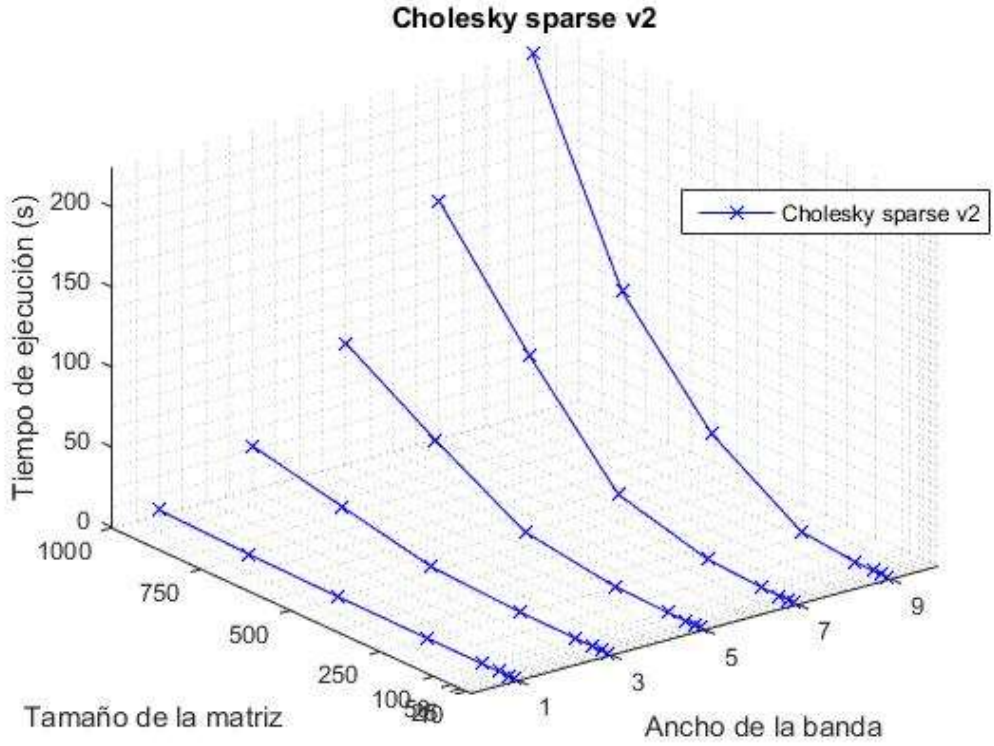

Figura 6.3.4

Donde puede observarse que las relaciones existentes entre los tiempos y el tamaño o el ancho de banda son similares. Sin embargo, si se presta atención al orden de magnitud, es fácil percatarse de que esta segunda versión es un orden más rápida que la primera. Esto es el resultado de las mejoras de rendimiento aplicadas como puede verse en el anexo A.

En cuanto a las relaciones tanto con el ancho de la banda como con el tamaño de la matriz, siguiendo el mismo procedimiento que para la primera versión, obtenemos las dos figuras siguientes:

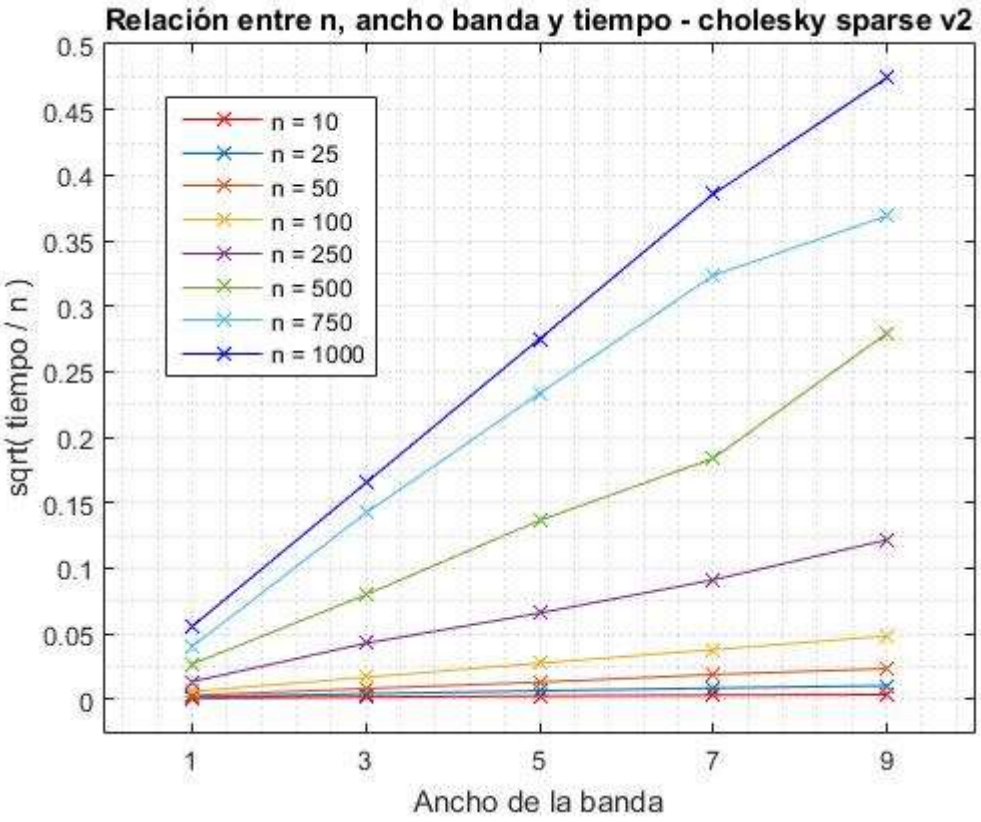

Figura 6.3.5

donde se aprecia un comportamiento análogo al de la figura 6.3.2, eso sí, de un orden cosiderablemente inferior al mismo (los valores del eje de ordenadas son tres veces menores).

En cuanto a la figura 6.3.6, las curvas son de tipo exponencial como ya ocurría en el gráfico 6.3.3, no obstante, esta dependencia exponencial, no parece depender senciblemente del ancho de banda, al estar las curvas aproximadamente superpuestas.

Finalmente, para que sea más sencillo comparar ambas versiones, la figura 6.3.7 muestra ambas combinadas.

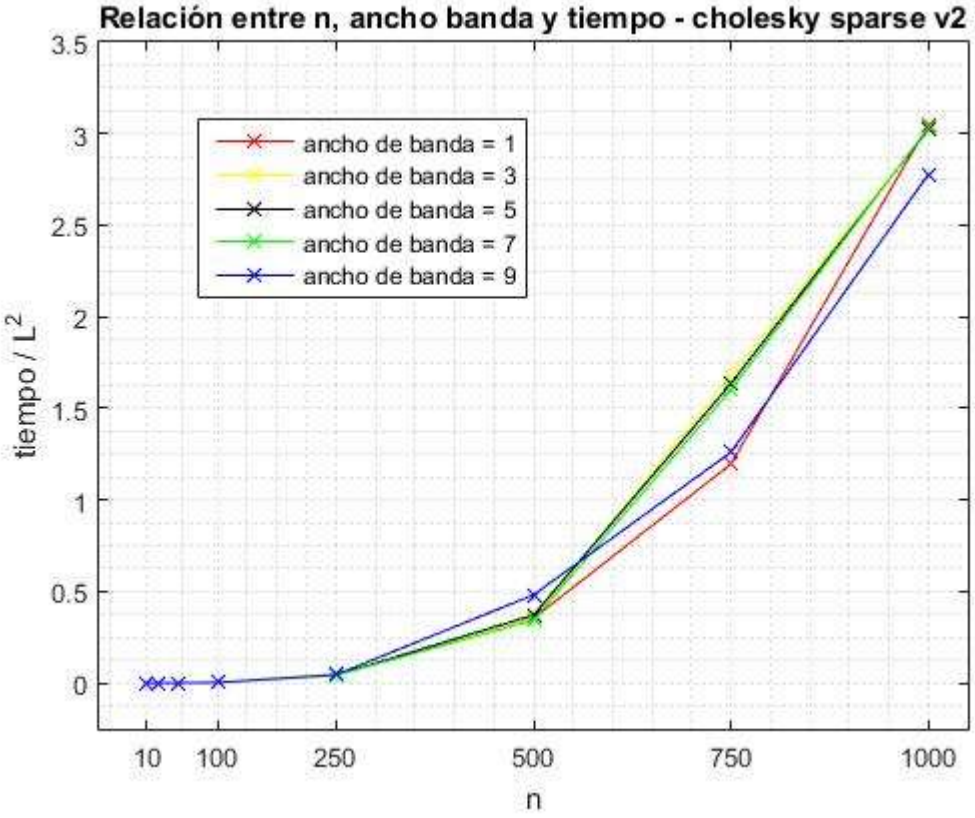

Figura 6.3.6

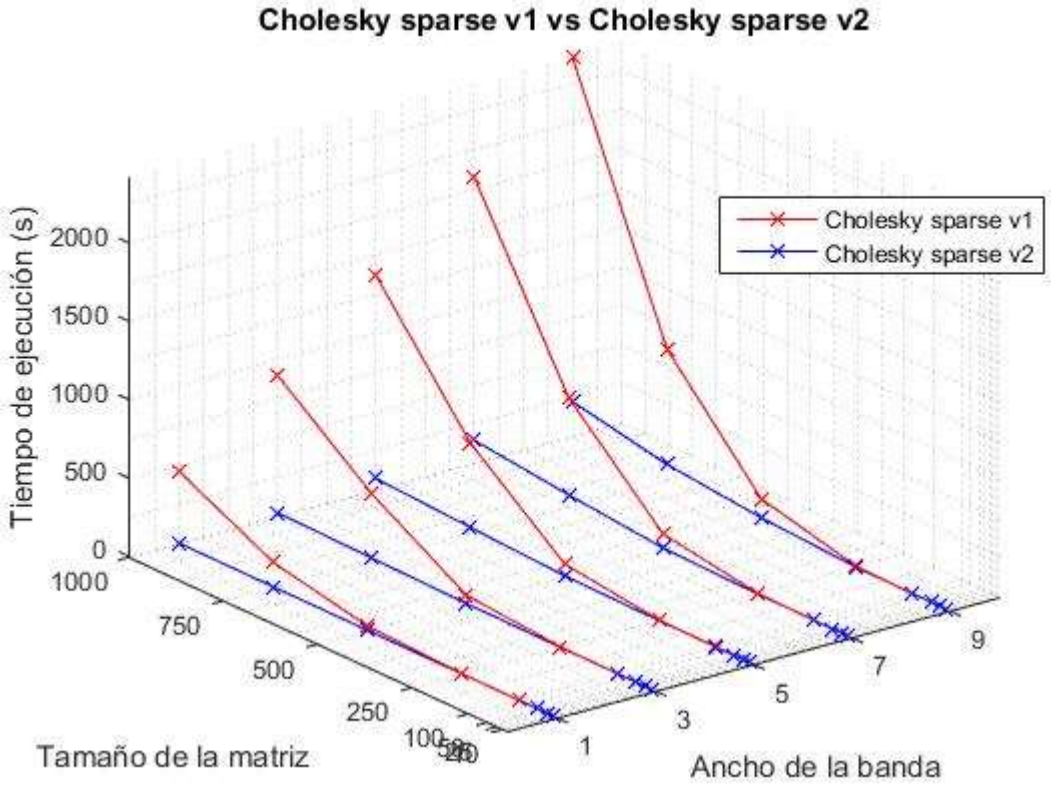

Figura 6.3.7

6.4. Resolver sistemas mediante Cholesky

En cuarto lugar, nos centraremos en otra función fundamental del trabajo como es *sistecchol_sparse*, la cual nos permite resolver sistemas de ecuaciones empleando la descomposición de Cholesky obtenida anteriormente.

Así, los tiempos de ejecución que hemos obtenido han sido los siguientes:

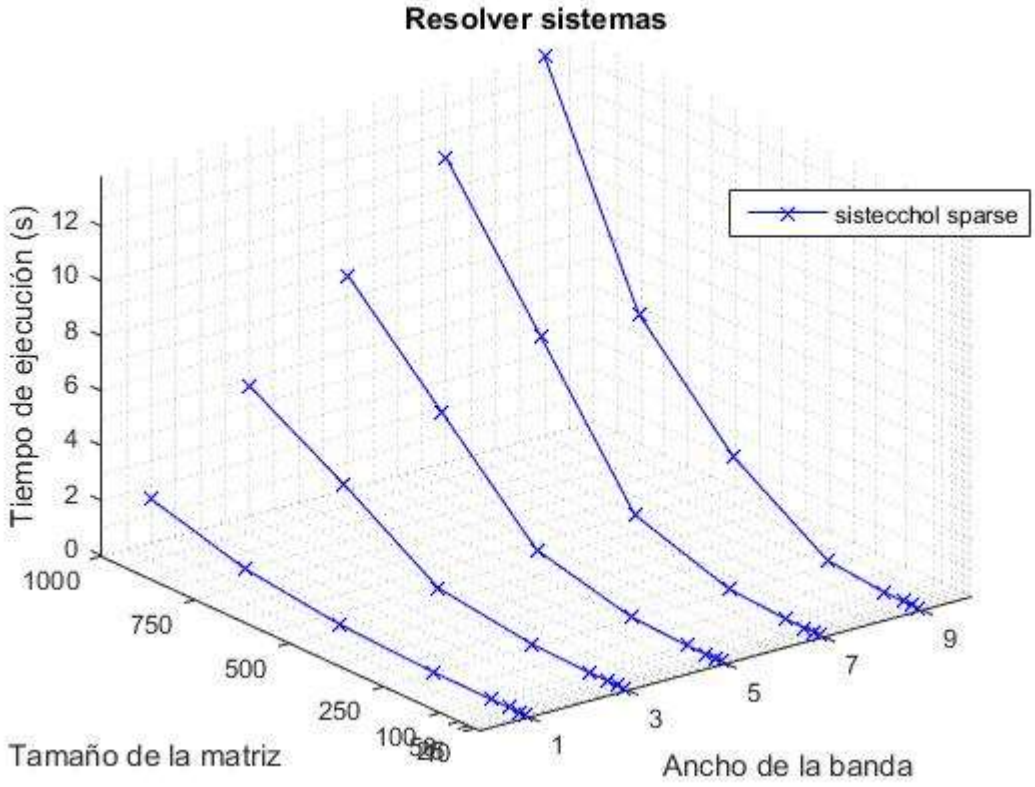

Figura 6.4.1

De nuevo pueden verse las relaciones exponenciales ya encontradas en otras funciones, si bien los tiempos son de un orden de magnitud inferior.

Para probar esto, sometemos a los datos temporales al mismo tratamiento del apartado anterior, de forma que se consiguen las figuras 6.4.2 y 6.4.3. donde se aprecia un comportamiento análogo a los correspondientes a las funciones de Cholesky, aunque de un orden de magnitud menor.

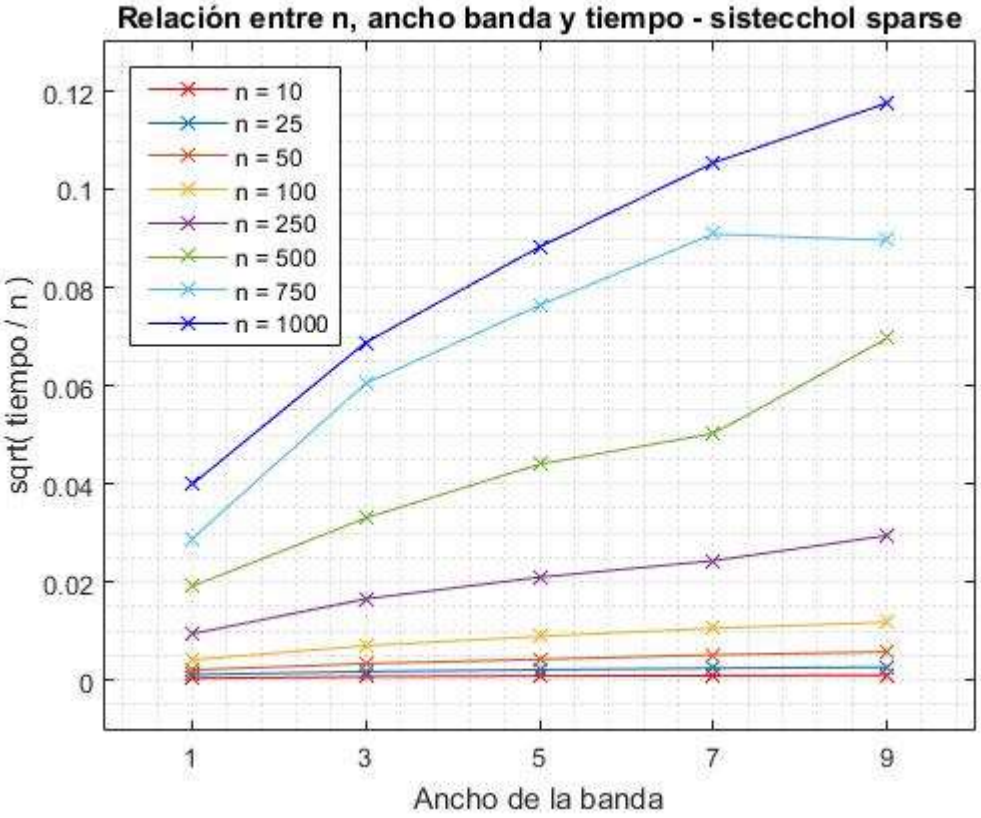

Figura 6.4.2

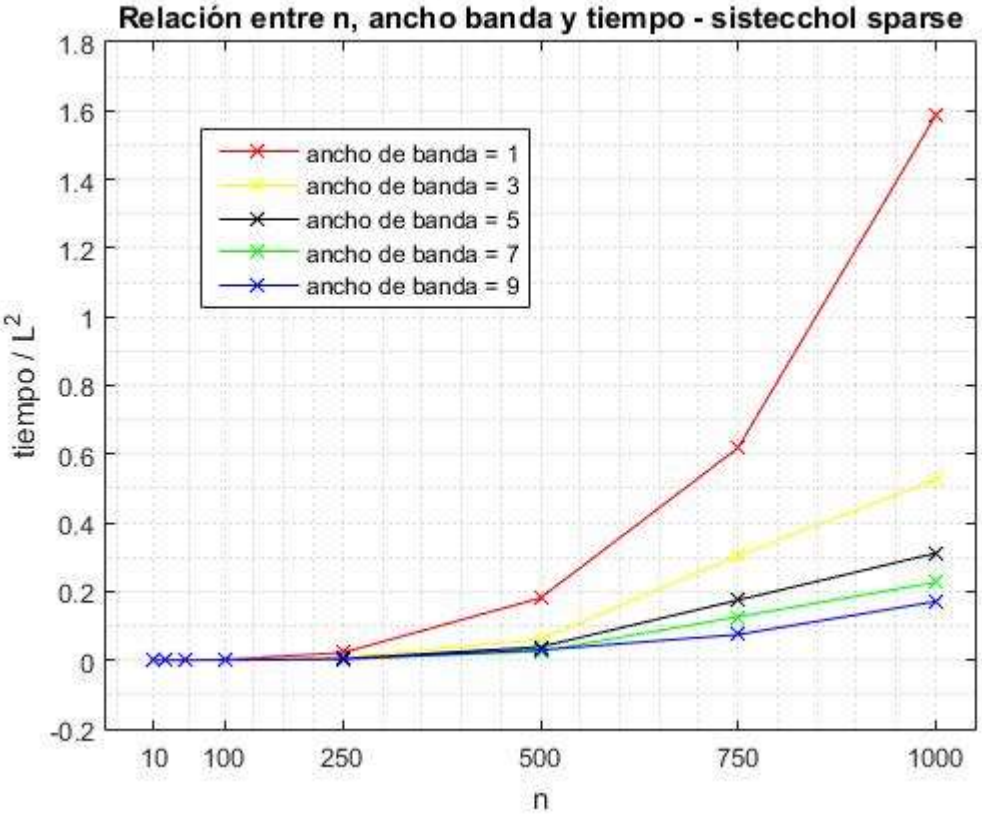

Figura 6.4.3

6.5. Tiempos para sistemas de ecuaciones en semi banda

A continuación, se muestran las gráficas de tiempo correspondientes al método de resolución de sistemas de ecuaciones en semi banda. Se ha simulado para dos escenarios diferentes según el número de diadas del sistema:

El primer escenario contempla que nuestro sistema consta de tres diadas únicamente, siendo el resultado el correspondiente a la figura 6.5.1. Mientras que el segundo considera que nos encontramos con diez diadas en el sistema (figura 6.5.2). Como puede verse, ambos gráficos muestran curvas análogas, a excepción del orden de magnitud, que en el segundo caso es superior (el hecho de añadir diadas a nuestro sistema provoca un aumento proporcional de los tiempos). Un modo de ver esto con mayor facilidad es enfrentar ambas figuras en una sola (figura 6.5.3).

Por otro lado, para comparar con los apartados anteriores, se ha empleado el mismo procedimiento de análisis con las mismas expresiones para las relaciones, de manera que quede evidenciado que el comportamiento de la función para sistemas de ecuaciones en semi banda es más complejo. Esto se aprecia en las figuras 6.5.4, 6.5.5, 6.5.6 y 6.5.7, en las que puede verse que las curvas no son rectas ni exponenciales definidas como en los casos anteriores.

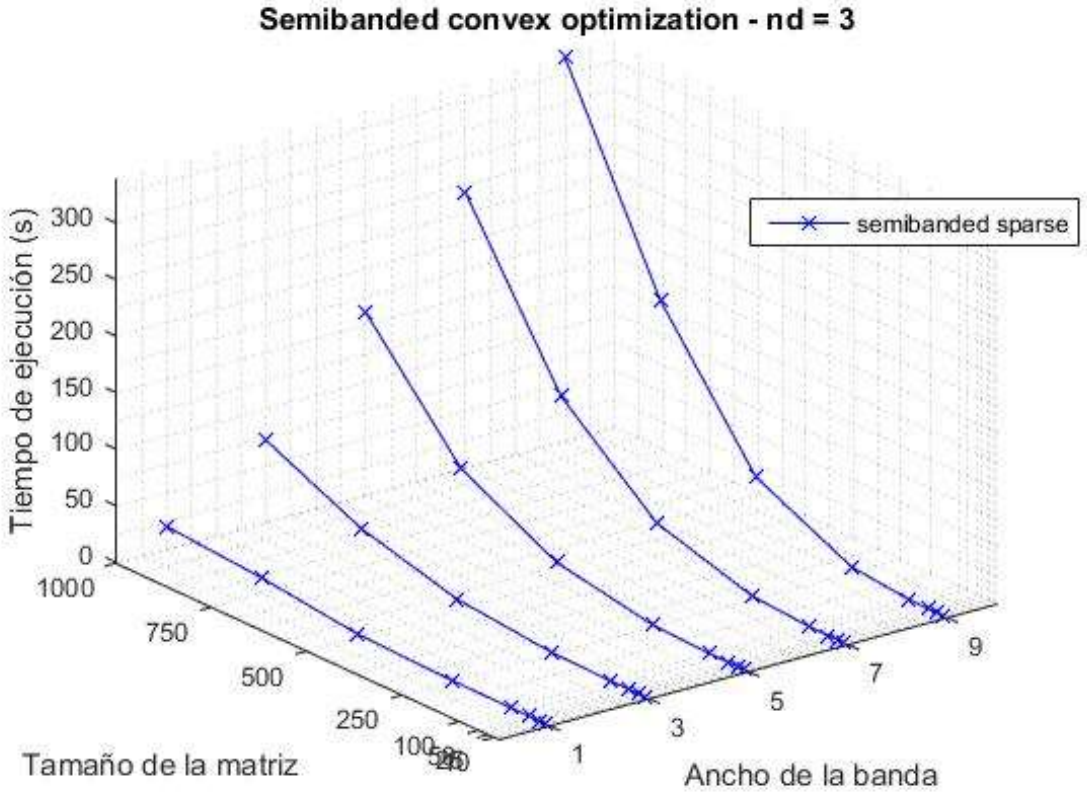

Figura 6.5.1

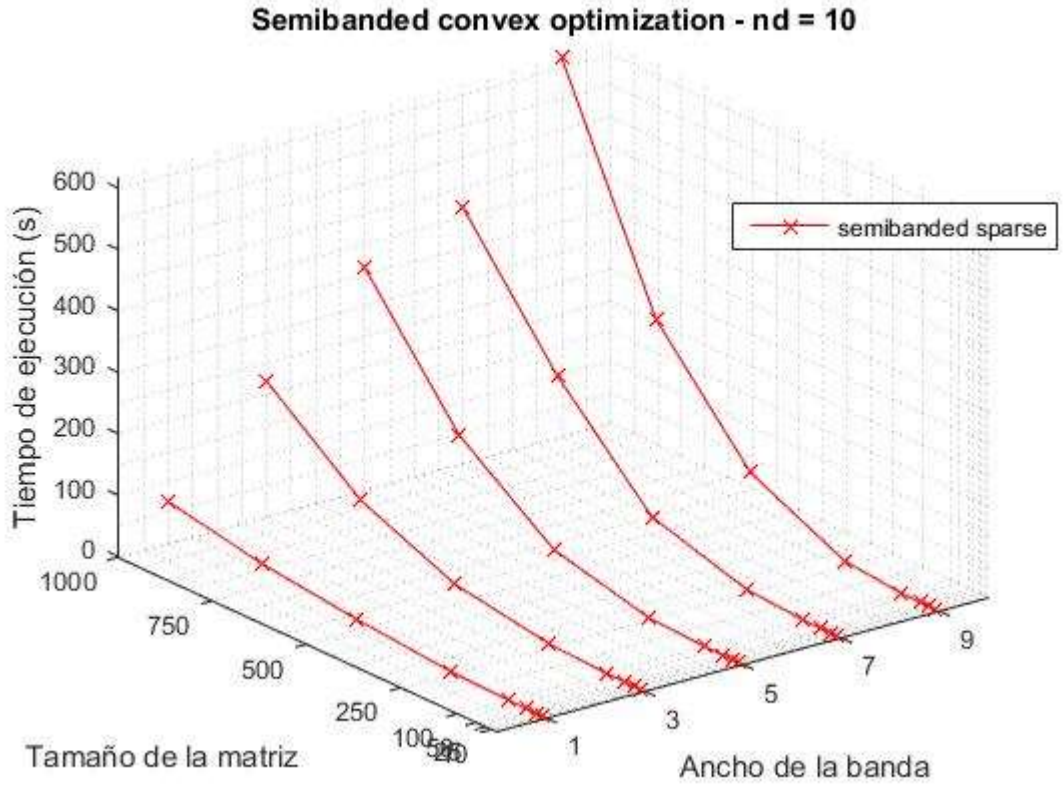

Figura 6.5.2

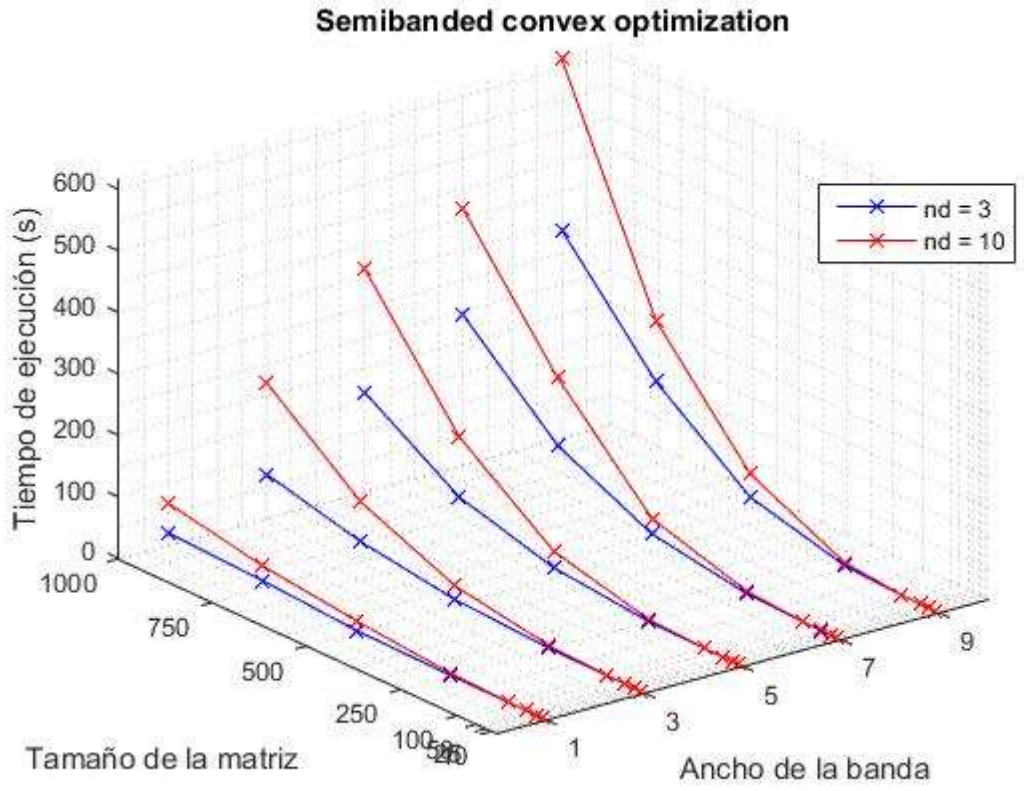

Figura 6.5.3

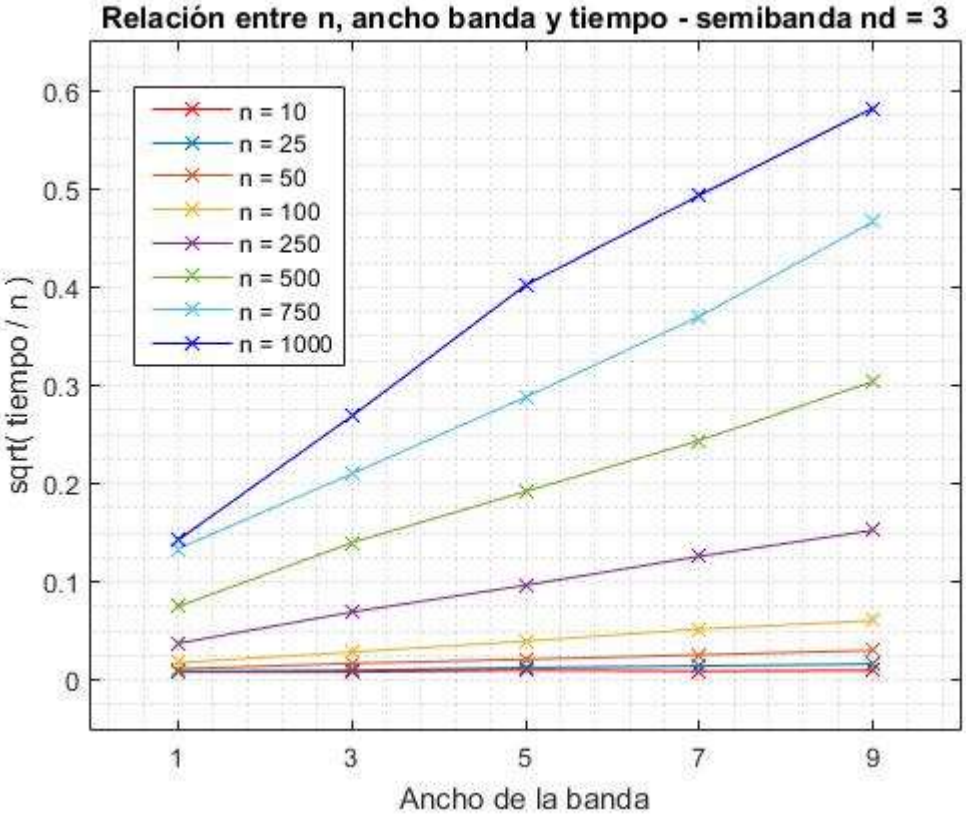

Figura 6.5.4

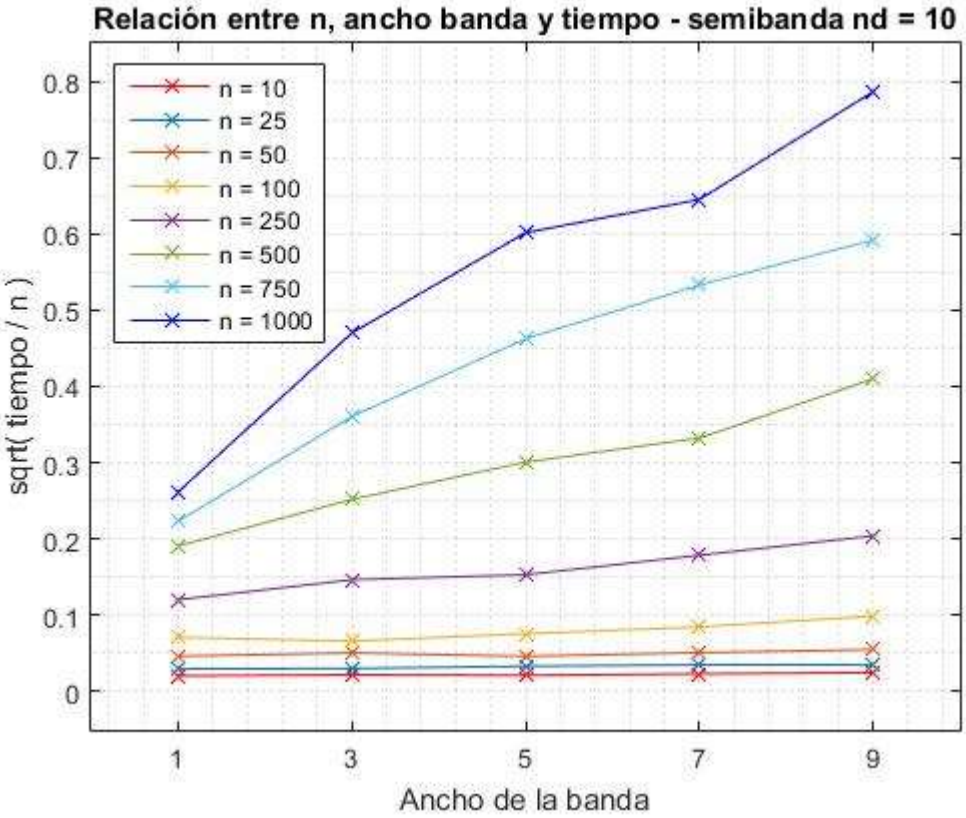

Figura 6.5.5

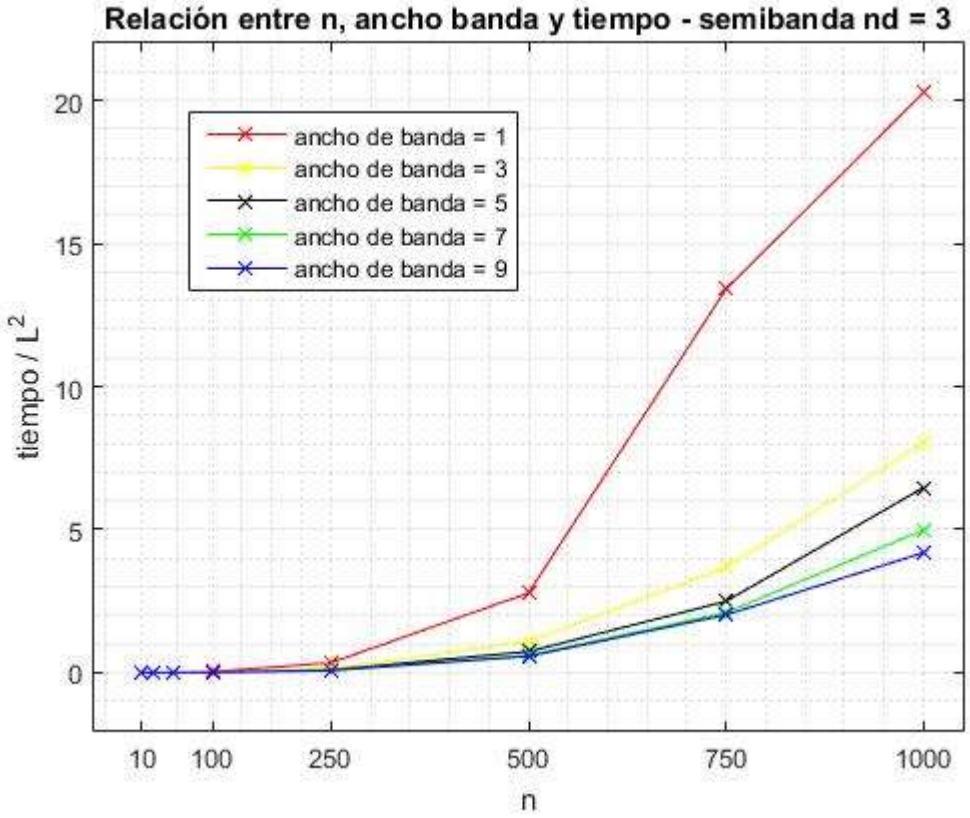

Figura 6.5.6

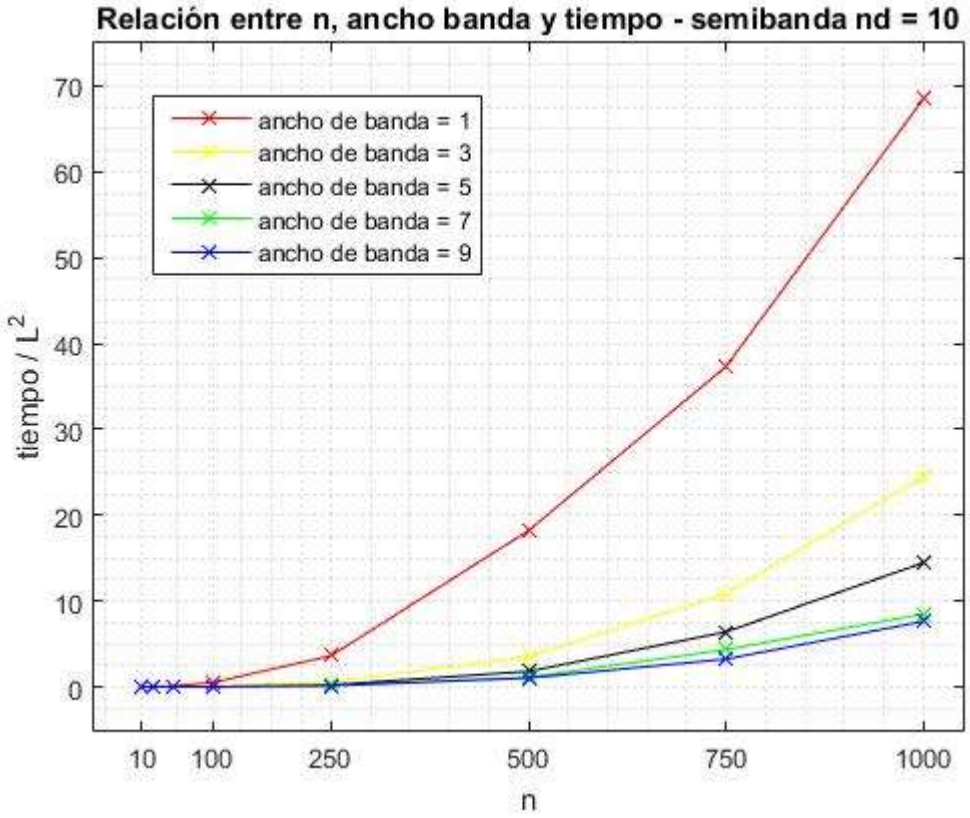

Figura 6.5.7

6.6. Generar datos con MatLab

Por último, es preciso exponer el modo en el que se han generado los datos y las gráficas antes expuestas, para lo cual mostraremos y comentaremos los códigos MatLab empleados.

Para crear la matriz A del sitema $Ax = b$ se ha empleado el siguiente código:

```
format
format compact
format short g
clear all,close all,clc
% --------------------------------------------------------------
for n = [ 10 , 25 , 50 , 100 , 250 , 500 , 750 , 1000 ]
  % Ancho de banda 1:
    % A = diag(rand(1,n));

  % Ancho de banda 3:
      A = diag(rand(1,n))+diag(rand(1,n-1),1);

  % Ancho de banda 5:
    % A = diag(rand(1,n))+diag(rand(1,n-1),1)+...
    %     diag(rand(1,n-2),2);

  % Ancho de banda 7:
    % A = diag(rand(1,n))+diag(rand(1,n-1),1)+...
    % diag(rand(1,n-2),2)+diag(rand(1,n-3),3);

  % Ancho de banda 9:
    % A = diag(rand(1,n))+diag(rand(1,n-1),1)+...
    % diag(rand(1,n-2),2)+diag(rand(1,n-3),3)+...
    % diag(rand(1,n-4),4);
% --------------------------------------------------------------
  % La multiplicamos por su traspuesta para hacerla simétrica:
  A = A*A' + eye(n);
% --------------------------------------------------------------
  % Escribimos en un fichero los datos:
  % Abrimos el archivo para escritura:
  archivoID = fopen('matriz_A.txt','w');
  for i = 1:n
    for j = 1:n
      % Escribimos los datos en el archivo:
      fprintf(archivoID,' %f',A(i,j));
    end
    fprintf(archivoID,'\r\n'); % Pasamos a la siguiente línea
  end
  fclose(archivoID); % Cerramos el archivo
% --------------------------------------------------------------
  % Generamos el vector b del sistema Ax=b
```

```
  b = ones(n,1);
% ------------------------------------------------------------
  % Versión normal:
    Rc = chol(A,'lower');
    aux = Rc\b;
    x = (Rc')\aux;
  % Versión sparse:
    As = sparse(A);
    Rcs = chol(As,'lower');
    aux = Rcs\b;
    xs = (Rcs')\aux;
% ------------------------------------------------------------
  n
  pause
end
```

Donde podemos ver que el ancho de banda se indica comentando/descomentando la línea de código correspondiente. Además de esto, se ha incluido un pequeño código que permite obtener tanto la descomposición de Cholesky de A, como la resolución del sistema $Ax = b$ usando lo anterior. La finalidad de esto es la de poder comprobar fácilmente si la soluciones obtenidas en C y en MatLab son equivalentes.

Por otro lado, para generar la matriz $B + U \cdot V^T$ se ha empleado el siguiente código:

```
format
format compact
format short
clear all,close all,clc

n = 10; % Tamaño de las matrices W y B
nd = 3; % Número de diadas
% ------------------------------------------------------------
% Ancho banda 1:
B = diag(rand(1,n));
% ------------------------------------------------------------
% Ancho banda 3:
% B = diag(rand(1,n)) + diag(rand(1,n-1),1);
% ------------------------------------------------------------
% Ancho banda 5:
% B = diag(rand(1,n)) + diag(rand(1,n-1),1) + ...
%     diag(rand(1,n-2),2);
% ------------------------------------------------------------
% Ancho banda 7:
% B = diag(rand(1,n)) + diag(rand(1,n-1),1) + ...
%     diag(rand(1,n-2),2) + diag(rand(1,n-3),3);
% ------------------------------------------------------------
% Ancho banda 9:
```

```
% B = diag(rand(1,n)) + diag(rand(1,n-1),1) + ...
%     diag(rand(1,n-2),2) + diag(rand(1,n-3),3) + ...
%     diag(rand(1,n-4),4);
% ------------------------------------------------------------
% La multiplicamos por su traspuesta para hacerla simétrica:
B = B*B' + eye(n);
% Escribimos en un fichero los datos:
% Abrimos el archivo para sobrescritura:
archivoID = fopen('matriz_B.txt','w');
for i = 1:n
    for j = 1:n
        % Escribimos los datos en el archivo:
        fprintf(archivoID,' %1.20f',B(i,j));
    end
    fprintf(archivoID,'\r\n'); % Pasamos a la siguiente línea
end
fclose(archivoID); % Cerramos el archivo
% ------------------------------------------------------------
% Matriz de vectores verticales de las diadas (n x nd):
U = rand(n,nd);
% Escribimos en un fichero los datos:
% Abrimos el archivo para sobrescritura:
archivoID = fopen('matriz_U.txt','w');
for i = 1:n
    for j = 1:nd
        % Escribimos los datos en el archivo:
        fprintf(archivoID,' %1.20f',U(i,j));
    end
    fprintf(archivoID,'\r\n'); % Pasamos a la siguiente línea
end
fclose(archivoID); % Cerramos el archivo
% ------------------------------------------------------------
% Matriz de vectores horizontales (una vez traspuesta)
% de las diadas (n x nd):
% V = rand(n,nd);
V = U; % Suponemos esto para simplificar (n x nd)
% Escribimos en un fichero los datos:
% Abrimos el archivo para sobrescritura:
archivoID = fopen('matriz_V.txt','w');
for i = 1:n
    for j = 1:nd
        % Escribimos los datos en el archivo:
        fprintf(archivoID,' %1.20f',V(i,j));
    end
    fprintf(archivoID,'\r\n'); % Pasamos a la siguiente línea
end
fclose(archivoID); % Cerramos el archivo
```

Por otra parte, para representar las gráficas mostradas en este capítulo, se ha utilizado el siguiente código:

```
format
format compact
format short g
clear all,close all,clc
% ------------------------------------------------------------------
% Vector de tamaños de la matriz:
n = [   10     10     10     10     10 ;
        25     25     25     25     25 ;
        50     50     50     50     50 ;
       100    100    100    100    100 ;
       250    250    250    250    250 ;
       500    500    500    500    500 ;
       750    750    750    750    750 ;
      1000   1000   1000   1000   1000 ];
% ------------------------------------------------------------------
% Matriz de anchos de banda:
ancho_banda = [ 1*ones(length(n),1) , ...
                3*ones(length(n),1) , ...
                5*ones(length(n),1) , ...
                7*ones(length(n),1) , ...
                9*ones(length(n),1) ];
% ancho_banda =
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
%      1     3     5     7     9
% ------------------------------------------------------------------
% Matrices de tiempos de ejecución de las funciones C:
Lectordatos = [ 0.000123 0.000142 0.000152 0.000160 0.000145 ;
                0.000315 0.000362 0.000383 0.000328 0.000366 ;
                0.000920 0.000958 0.000979 0.000971 0.001067 ;
                0.003342 0.003417 0.003528 0.003629 0.003613 ;
                0.019540 0.019690 0.019530 0.019760 0.022690 ;
                0.081100 0.082400 0.095500 0.081700 0.081600 ;
                0.193000 0.199000 0.199000 0.188000 0.209000 ;
                0.368000 0.357000 0.327000 0.416000 0.380000 ];
% ------------------------------------------------------------------
Chol_1 = [ 0.000007   0.000018    0.000026    0.000032
0.000039 ;
           0.000258   0.000512    0.000721    0.000893
0.001086 ;
           0.003348   0.006579    0.009416    0.013259
0.016628 ;
```

```
           0.052376   0.104035    0.152027    0.197556
0.245440 ;
           1.718240   3.503580    4.961290    6.413580
8.967820 ;
          26.295900  56.045900   84.996400  107.039900
160.855100 ;
         156.445000 423.727000  579.896000  699.727000
839.212000 ;
         460.237000 901.917000 1370.997000 1825.390000
2420.580000 ];
% ------------------------------------------------------------
Chol_2 = [ 0.000003  0.000023  0.000054   0.000082   0.000110 ;
           0.000048  0.000393  0.000985   0.001739   0.002595 ;
           0.000387  0.003157  0.008189   0.017516   0.026786 ;
           0.003275  0.027533  0.074670   0.141554   0.231612 ;
           0.043600  0.452460  1.076550   2.068180   3.692420 ;
           0.353800  3.167100  9.288600  16.879000  39.015000 ;
           1.196000 15.117000 40.881000  78.413000 102.081000 ;
           3.050000 27.192000 75.598000 148.584000 224.996000 ];
% ------------------------------------------------------------
Sistema = [ 0.000001 0.000004 0.000006  0.000007  0.000009 ;
            0.000027 0.000070 0.000105  0.000138  0.000166 ;
            0.000202 0.000542 0.000867  0.001291  0.001611 ;
            0.001676 0.004814 0.007825  0.011026  0.013740 ;
            0.022030 0.067980 0.109240  0.147210  0.216100 ;
            0.181100 0.544000 0.965400  1.260300  2.422200 ;
            0.618000 2.738000 4.371000  6.192000  6.022000 ;
            1.585000 4.730000 7.785000 11.102000 13.827000 ];
% ------------------------------------------------------------
% Cargamos los datos de tiempo de MatLab:
load Matrices_tiempos_MatLab
% ------------------------------------------------------------
% Tiempos de semibanda C en función de n, del ancho de banda y
% del número de diadas:
% nd = 3:
Semi_3 = [  0.000645  0.000624   0.000944   0.000790   0.000910 ;
            0.001857  0.002334   0.003797   0.005010   0.006400 ;
            0.006133  0.013899   0.021709   0.032320   0.044800 ;
            0.030110  0.079750   0.155170   0.266900   0.364100 ;
            0.345000  1.191000   2.322820   3.955700   5.800700 ;
            2.784300  9.763000  18.410700  29.733333  46.222000 ;
           13.398333 33.052000  62.258333 102.817000 163.444000 ;
           20.268000 72.220000 161.657667 243.863000 339.391000 ];
% nd = 10:
Semi_10 = [  0.004270   0.004910   0.004590   0.005500   0.006200 ;
             0.022230   0.023100   0.027960   0.030970   0.030800 ;
             0.108150   0.129030   0.107500   0.130640   0.152200 ;
             0.506650   0.439300   0.575850   0.719900   0.982000 ;
             3.631200   5.377800   5.872400   7.991800  10.418000 ;
            18.216400  31.726000  45.266000  55.242000  84.033000 ;
            37.315667  97.622000 160.350000 212.933000 262.748000 ;
            68.475000 220.920000 361.754000 415.993000 617.979000 ];
% ------------------------------------------------------------
```

```
% Representación:
% ------------------------------------------------------------------
% Lectordatos:
figure,
lectordatos = plot3(ancho_banda,n,Lectordatos,'-xb');
grid minor; title('C lectordatos');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('C lectordatos');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------------
% Lectordatos vs MatLab_fichero:
figure,
lectordatos = plot3(ancho_banda,n,Lectordatos,'-xb');hold all;
fichero = plot3(ancho_banda,n,MatLab_fichero,'-xr'); hold off;
grid minor; title('C lectordatos vs MatLab fichero');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend([lectordatos(1),fichero(1)],'C lectordatos',...
       'MatLab fichero');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------------
% Chol_1:
figure,
chol_1 = plot3(ancho_banda,n,Chol_1,'-xr');
grid minor; title('Cholesky sparse v1');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('Cholesky sparse v1');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------------
% Chol_2:
figure,
chol_2 = plot3(ancho_banda,n,Chol_2,'-xb');
grid minor; title('Cholesky sparse v2');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('Cholesky sparse v2');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------------
% Chol_1 vs Chol_2:
figure,
chol_1 = plot3(ancho_banda,n,Chol_1,'-xr');hold all;
chol_2 = plot3(ancho_banda,n,Chol_2,'-xb');hold off;
grid minor; title('Cholesky sparse v1 vs Cholesky sparse v2');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend([chol_1(1),chol_2(1)],'Cholesky sparse v1',...
```

```
       'Cholesky sparse v2');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------
% Sistema:
figure,
sistema = plot3(ancho_banda,n,Sistema,'-xb');
grid minor; title('Resolver sistemas');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('sistecchol sparse');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------
% Semi_3:
figure,
semibanda_3 = plot3(ancho_banda,n,Semi_3,'-xb');
grid minor; title('Sistemas de ecuaciones en semi banda - nd =
3');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('semibanda sparse');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------
% Semi_10:
figure,
semibanda_10 = plot3(ancho_banda,n,Semi_10,'-xr');
grid minor; title('Sistemas de ecuaciones en semi banda - nd =
10');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend('semibanda sparse');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------
% Semi_3 vs Semi_10:
figure,
semiband_3 = plot3(ancho_banda,n,Semi_3,'-xb');hold all;
semiband_10 = plot3(ancho_banda,n,Semi_10,'-xr');hold off;
grid minor; title('Sistemas de ecuaciones en semi banda');
xlabel('Ancho de la banda'); ylabel('Tamaño de la matriz');
zlabel('Tiempo de ejecución (s)');
legend([semiband_3(1),semiband_10(1)],'nd = 3',...
       'nd = 10');
axis([0 10 0 1000 0 inf]); ax = gca; ax.XTick = [1 3 5 7 9];
ax.YTick = n(:,1);
% ------------------------------------------------------------
% Relación proporcional Lectordatos entre n, ancho banda y
tiempo:
Rel_1 = sqrt( Lectordatos(:,1) );
Rel_3 = sqrt( Lectordatos(:,2) );
```

```
Rel_5 = sqrt( Lectordatos(:,3) );
Rel_7 = sqrt( Lectordatos(:,4) );
Rel_9 = sqrt( Lectordatos(:,5) );
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - lectordatos');
xlabel('n'); ylabel('sqrt( tiempo )');
legend('ancho banda = 1','ancho banda = 3',...
       'ancho banda = 5','ancho banda = 7','ancho banda = 9');
axis([-50 1050 -0.05 0.7]);
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
% Relación proporcional Chol_1 entre n, ancho banda y tiempo:
Rel_10 = sqrt( Chol_1(1,:) / 10 );
Rel_25 = sqrt( Chol_1(2,:) / 25 );
Rel_50 = sqrt( Chol_1(3,:) / 50 );
Rel_100 = sqrt( Chol_1(4,:) / 100 );
Rel_250 = sqrt( Chol_1(5,:) / 250 );
Rel_500 = sqrt( Chol_1(6,:) / 500 );
Rel_750 = sqrt( Chol_1(7,:) / 750 );
Rel_1000 = sqrt( Chol_1(8,:) / 1000 );
figure,
rel_10 = plot(ancho_banda(1,:),Rel_10,'-xr');hold all;
rel_25 = plot(ancho_banda(2,:),Rel_25,'-x');
rel_50 = plot(ancho_banda(3,:),Rel_50,'-x');
rel_100 = plot(ancho_banda(4,:),Rel_100,'-x');
rel_250 = plot(ancho_banda(5,:),Rel_250,'-x');
rel_500 = plot(ancho_banda(6,:),Rel_500,'-x');
rel_750 = plot(ancho_banda(7,:),Rel_750,'-x');
rel_1000 = plot(ancho_banda(8,:),Rel_1000,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - cholesky sparse
v1');
xlabel('Ancho de la banda'); ylabel('sqrt( tiempo / n )');
legend('n = 10','n = 25','n = 50','n = 100','n = 250',...
       'n = 500','n = 750','n = 1000');
axis([0 10 -0.025 1.7]);
ax = gca; ax.XTick = [1 3 5 7 9];
% ------------------------------------------------------------
% Relación proporcional Chol_2 entre n, ancho banda y tiempo:
Rel_10 = sqrt( Chol_2(1,:) / 10 );
Rel_25 = sqrt( Chol_2(2,:) / 25 );
Rel_50 = sqrt( Chol_2(3,:) / 50 );
Rel_100 = sqrt( Chol_2(4,:) / 100 );
Rel_250 = sqrt( Chol_2(5,:) / 250 );
Rel_500 = sqrt( Chol_2(6,:) / 500 );
Rel_750 = sqrt( Chol_2(7,:) / 750 );
```

```
Rel_1000 = sqrt( Chol_2(8,:) / 1000 );
figure,
rel_10 = plot(ancho_banda(1,:),Rel_10,'-xr');hold all;
rel_25 = plot(ancho_banda(2,:),Rel_25,'-x');
rel_50 = plot(ancho_banda(3,:),Rel_50,'-x');
rel_100 = plot(ancho_banda(4,:),Rel_100,'-x');
rel_250 = plot(ancho_banda(5,:),Rel_250,'-x');
rel_500 = plot(ancho_banda(6,:),Rel_500,'-x');
rel_750 = plot(ancho_banda(7,:),Rel_750,'-x');
rel_1000 = plot(ancho_banda(8,:),Rel_1000,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - cholesky sparse
v2');
xlabel('Ancho de la banda'); ylabel('sqrt( tiempo / n )');
legend('n = 10','n = 25','n = 50','n = 100','n = 250',...
       'n = 500','n = 750','n = 1000');
axis([0 10 -0.025 0.5]);
ax = gca; ax.XTick = [1 3 5 7 9];
% ------------------------------------------------------------
% Relación proporcional Sistema entre n, ancho banda y tiempo:
Rel_10 = sqrt( Sistema(1,:) / 10 );
Rel_25 = sqrt( Sistema(2,:) / 25 );
Rel_50 = sqrt( Sistema(3,:) / 50 );
Rel_100 = sqrt( Sistema(4,:) / 100 );
Rel_250 = sqrt( Sistema(5,:) / 250 );
Rel_500 = sqrt( Sistema(6,:) / 500 );
Rel_750 = sqrt( Sistema(7,:) / 750 );
Rel_1000 = sqrt( Sistema(8,:) / 1000 );
figure,
rel_10 = plot(ancho_banda(1,:),Rel_10,'-xr');hold all;
rel_25 = plot(ancho_banda(2,:),Rel_25,'-x');
rel_50 = plot(ancho_banda(3,:),Rel_50,'-x');
rel_100 = plot(ancho_banda(4,:),Rel_100,'-x');
rel_250 = plot(ancho_banda(5,:),Rel_250,'-x');
rel_500 = plot(ancho_banda(6,:),Rel_500,'-x');
rel_750 = plot(ancho_banda(7,:),Rel_750,'-x');
rel_1000 = plot(ancho_banda(8,:),Rel_1000,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - sistecchol
sparse');
xlabel('Ancho de la banda'); ylabel('sqrt( tiempo / n )');
legend('n = 10','n = 25','n = 50','n = 100','n = 250',...
       'n = 500','n = 750','n = 1000');
axis([0 10 -0.01 0.13]);
ax = gca; ax.XTick = [1 3 5 7 9];
% ------------------------------------------------------------
% Relación proporcional Semi_3 entre n, ancho banda y tiempo:
Rel_10 = sqrt( Semi_3(1,:) / 10 );
Rel_25 = sqrt( Semi_3(2,:) / 25 );
Rel_50 = sqrt( Semi_3(3,:) / 50 );
Rel_100 = sqrt( Semi_3(4,:) / 100 );
Rel_250 = sqrt( Semi_3(5,:) / 250 );
```

```
Rel_500 = sqrt( Semi_3(6,:) / 500 );
Rel_750 = sqrt( Semi_3(7,:) / 750 );
Rel_1000 = sqrt( Semi_3(8,:) / 1000 );
figure,
rel_10 = plot(ancho_banda(1,:),Rel_10,'-xr');hold all;
rel_25 = plot(ancho_banda(2,:),Rel_25,'-x');
rel_50 = plot(ancho_banda(3,:),Rel_50,'-x');
rel_100 = plot(ancho_banda(4,:),Rel_100,'-x');
rel_250 = plot(ancho_banda(5,:),Rel_250,'-x');
rel_500 = plot(ancho_banda(6,:),Rel_500,'-x');
rel_750 = plot(ancho_banda(7,:),Rel_750,'-x');
rel_1000 = plot(ancho_banda(8,:),Rel_1000,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - semibanda nd =
3');
xlabel('Ancho de la banda'); ylabel('sqrt( tiempo / n )');
legend('n = 10','n = 25','n = 50','n = 100','n = 250',...
       'n = 500','n = 750','n = 1000');
axis([0 10 -0.05 0.65]);
ax = gca; ax.XTick = [1 3 5 7 9];
% ------------------------------------------------------------------
% Relación proporcional Semi_10 entre n, ancho banda y tiempo:
Rel_10 = sqrt( Semi_10(1,:) / 10 );
Rel_25 = sqrt( Semi_10(2,:) / 25 );
Rel_50 = sqrt( Semi_10(3,:) / 50 );
Rel_100 = sqrt( Semi_10(4,:) / 100 );
Rel_250 = sqrt( Semi_10(5,:) / 250 );
Rel_500 = sqrt( Semi_10(6,:) / 500 );
Rel_750 = sqrt( Semi_10(7,:) / 750 );
Rel_1000 = sqrt( Semi_10(8,:) / 1000 );
figure,
rel_10 = plot(ancho_banda(1,:),Rel_10,'-xr');hold all;
rel_25 = plot(ancho_banda(2,:),Rel_25,'-x');
rel_50 = plot(ancho_banda(3,:),Rel_50,'-x');
rel_100 = plot(ancho_banda(4,:),Rel_100,'-x');
rel_250 = plot(ancho_banda(5,:),Rel_250,'-x');
rel_500 = plot(ancho_banda(6,:),Rel_500,'-x');
rel_750 = plot(ancho_banda(7,:),Rel_750,'-x');
rel_1000 = plot(ancho_banda(8,:),Rel_1000,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - semibanda nd =
10');
xlabel('Ancho de la banda'); ylabel('sqrt( tiempo / n )');
legend('n = 10','n = 25','n = 50','n = 100','n = 250',...
       'n = 500','n = 750','n = 1000');
axis([0 10 -0.05 0.85]);
ax = gca; ax.XTick = [1 3 5 7 9];
% ------------------------------------------------------------------
% Relación proporcional Chol_1 entre n, ancho banda y tiempo:
Rel_1 = Chol_1(:,1) / (ancho_banda(1,1)^2);
Rel_3 = Chol_1(:,2) / (ancho_banda(1,2)^2);
Rel_5 = Chol_1(:,3) / (ancho_banda(1,3)^2);
```

```
Rel_7 = Chol_1(:,4) / (ancho_banda(1,4)^2);
Rel_9 = Chol_1(:,5) / (ancho_banda(1,5)^2);
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - cholesky sparse
v1');
xlabel('n'); ylabel('tiempo / L^2');
legend('ancho de banda = 1','ancho de banda = 3',...
       'ancho de banda = 5','ancho de banda = 7',...
       'ancho de banda = 9');
axis([-50 1050 -25 500]);
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
% Relación proporcional Chol_2 entre n, ancho banda y tiempo:
Rel_1 = Chol_2(:,1) / (ancho_banda(1,1)^2);
Rel_3 = Chol_2(:,2) / (ancho_banda(1,2)^2);
Rel_5 = Chol_2(:,3) / (ancho_banda(1,3)^2);
Rel_7 = Chol_2(:,4) / (ancho_banda(1,4)^2);
Rel_9 = Chol_2(:,5) / (ancho_banda(1,5)^2);
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - cholesky sparse
v2');
xlabel('n'); ylabel('tiempo / L^2');
legend('ancho de banda = 1','ancho de banda = 3',...
       'ancho de banda = 5','ancho de banda = 7',...
       'ancho de banda = 9');
axis([-50 1050 -0.25 3.5]);
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
% Relación proporcional Sistema entre n, ancho banda y tiempo:
Rel_1 = Sistema(:,1) / (ancho_banda(1,1)^2);
Rel_3 = Sistema(:,2) / (ancho_banda(1,2)^2);
Rel_5 = Sistema(:,3) / (ancho_banda(1,3)^2);
Rel_7 = Sistema(:,4) / (ancho_banda(1,4)^2);
Rel_9 = Sistema(:,5) / (ancho_banda(1,5)^2);
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
```

```
grid minor;
title('Relación entre n, ancho banda y tiempo - sistecchol
sparse');
xlabel('n'); ylabel('tiempo / L^2');
legend('ancho de banda = 1','ancho de banda = 3',...
       'ancho de banda = 5','ancho de banda = 7',...
       'ancho de banda = 9');
axis([-50 1050 -0.2 1.8]);
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
% Relación proporcional Semi_3 entre n, ancho banda y tiempo:
Rel_1 = Semi_3(:,1) / (ancho_banda(1,1)^2);
Rel_3 = Semi_3(:,2) / (ancho_banda(1,2)^2);
Rel_5 = Semi_3(:,3) / (ancho_banda(1,3)^2);
Rel_7 = Semi_3(:,4) / (ancho_banda(1,4)^2);
Rel_9 = Semi_3(:,5) / (ancho_banda(1,5)^2);
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - semibanda nd =
3');
xlabel('n'); ylabel('tiempo / L^2');
legend('ancho de banda = 1','ancho de banda = 3',...
       'ancho de banda = 5','ancho de banda = 7',...
       'ancho de banda = 9');
axis([-50 1050 -2 22]);
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
% Relación proporcional Semi_10 entre n, ancho banda y tiempo:
Rel_1 = Semi_10(:,1) / (ancho_banda(1,1)^2);
Rel_3 = Semi_10(:,2) / (ancho_banda(1,2)^2);
Rel_5 = Semi_10(:,3) / (ancho_banda(1,3)^2);
Rel_7 = Semi_10(:,4) / (ancho_banda(1,4)^2);
Rel_9 = Semi_10(:,5) / (ancho_banda(1,5)^2);
figure,
rel_1 = plot(n(:,1),Rel_1,'-xr');hold all;
rel_3 = plot(n(:,2),Rel_3,'-xy');
rel_5 = plot(n(:,3),Rel_5,'-xk');
rel_7 = plot(n(:,4),Rel_7,'-xg');
rel_9 = plot(n(:,5),Rel_9,'-xb');hold off;
grid minor;
title('Relación entre n, ancho banda y tiempo - semibanda nd =
10');
xlabel('n'); ylabel('tiempo / L^2');
legend('ancho de banda = 1','ancho de banda = 3',...
       'ancho de banda = 5','ancho de banda = 7',...
       'ancho de banda = 9');
axis([-50 1050 -5 75]);
```

```
ax = gca; ax.XTick = [10 100 250 500 750 1000];
% ------------------------------------------------------------
```

Los datos contenidos en las matrices de tiempos se han obtenido generando los datos en MatLab como se ha indicado anteriormente y cargándolos en el fichero C correspondiente (ver anexo A - programas principales).

7 CONCLUSIONES

De razones vive el hombre, de sueños sobrevive.

- Miguel de Unamuno -

Para cerrar este proyecto, es necesario reflexionar sobre el proceso llevado a cabo durante la realización del mismo, así como acerca de los resultados, tanto numéricos como analíticos, obtenidos o desarrollados. Asimismo, es necesario decir que existen diversas formas de ampliar las fronteras de este trabajo, algunas de las cuales veremos en este capítulo.

7.1. Desarrollo del proyecto

En primer lugar, hablaremos sobre la ejecución del trabajo llevado a cabo, el cual ha costado de diferentes etapas, como han sido: planteamiento, documentación, primeros códigos, funciones, simulaciones y ensayos o redacción de la memoria.

En cuanto a los primeros pasos dados para la puesta en marcha del proyecto, decir que tanto la base teórica, como los algoritmos, así como los problemas a resolver eran desconocidos en un primer momento. Por lo tanto, establecer el escenario a abordar y sus límites fue el primer reto a enfrentar.

Una vez establecidas las bases, la siguiente tarea y la segunda en peso del desarrollo fue la fase de documentación. La complejidad de este punto radicó principalmente en el desconocimiento del ámbito matemático y teórico del problema a abordar. Por lo que se tuvo que compenzar desde cero para ganar en profundidad a lo largo de toda la ejecución del trabrajo. De hecho, pasada la primera toma de contacto, esta fase de documentación y justificación teórica ha continuado hasta la finalización misma del proyecto. Ahí es donde puede verse el grado de importancia de la misma. Además de esto, ha supuesto el mayor desarrollo de los conocimientos del adquiridos y una de las partes de mayor interés a nivel personal.

A lo largo de todo el proyecto, ha sido necesaria la creación de infinidad de líneas de código, tanto C como MatLab, correspondientes a un gran número de programas. Decir que, de cada una de las funciones generadas, se han desarrollado un considerable número de versiones sucesivas cuyo fin era desde corregir deficiencias de funcionamiento a añadir nuevas funcionalidades. Este proceso ha formado parte de casi la totalidad de la ejecución del proyecto y ha supuesto la mayor carga de trabajo del mismo, Esto se debe a que casi la totatlidad del código programado es original, siendo una pequeña fracción adaptaciones mejoradas basadas en otros autores.

En lo referente a los resultados de los numerosos ensayos y simulaciones llevados a cabo, sería preciso recalcar que no se han llevado a cabo exclusivamente al final del proyecto, si no que han sido compañeros constantes a lo largo del mismo. Han permitido la depuración y mejora de los programas desarrollados, así como una herramienta para permitir una mejor y más profunda comprensión de los conceptos teóricos. En última instancia, han servido a su vez a modo de demostración de los algoritmos propuestos.

Para cerrar este apartado, sería conveniente añadir que la redacción de este documento comenzó tras la obtención de las primeras funciones y resultados fiables, habiendo sido llevada a cabo, hasta la finalización de la misma, a lo largo de toda la ejecución del trabajo. Al igual que el código y los resultados, ha sufrido un considerable número de modificaciones y constado de diversas versiones hasta poder ser completada.

7.2. Análisis de los resultados obtenidos

En segundo lugar, nos centraremos en los resultados conseguidos al final del proyecto: códigos C y MatLab generados, así como valores empíricos de rendimiento y eficacia de los mismos.

En cuanto a la librería de funciones y programas principales en C, cabría decir que contituyen el grueso de los resultados y aportan el mayor valor. Esto se debe a su grado de robustez y eficacia, tanto en la precisión de los cálculos como en la agilidad de los mismos, habiendo sido más que satisfactorios. Además de esto, han supuesto el mayor escollo a superar y la actividad más enriquecedora para el desarrollo de las habilidades personales de lógica, programación, predicción y análisis entre otras. Por otro lado, en lo referente a MatLab, su papel ha sido el de complemento o herramienta auxiliar, siendo, por ejemplo, empleado para la generación de datos aleatorios de entrada, la realización de gráficos o la comprobación de resultados. Además de esto, decir que el código desarrollado permite su utilización en escenarios de diversa índole, pudiendo ser empleado para la resolución de gran cantidad de problemas multidisciplinares (control, matemáticos, análisis estadísticos o de gran envergadura, etc.).

En referencia a los valores numéricos obtenidos en las simulaciones, decir que han superado enormemente las expectativas iniciales, habiendo resultado en tiempos de ejecución menores de lo esperado y precisión de los mismos comparable a la obtenible en MatLab. Añadir además que han permitido demostrar el comportamiento esperado de las funciones con una desviación aceptable debida a imprecisiones de medida y/o a la aleatoriedad y diversidad de los datos de entrada.

7.3. Posibles vías de desarrollo

En tercer y último lugar, expondremos algunas de las posibilidades de mejora o caminos alternativos que sería viable tomar en un hipotético futuro o continuación de este proyecto.

Primeramente, en lo referente a la mejora del trabajo realizado, decir que sería posible implementar nuevas funcionalidades, tanto sobre las funciones existentes como sobre otras nuevas, que permitan añadir complejidad al conjunto. Pensemos, por ejemplo, en la posibilidad de realizar cálculos directamente con diadas o flechas codificadas, si la necesidad de simplificación. Esto supondría no sólo una mejora en el rendimiento, si no la posibilidad de tomar datos de entrada más complejos. No obstante, no ha sido posible efectuar estas mejoras por falta de tiempo y puesto que añadían un grado de complejidad demasido elevado inicialmente.

Acto seguido, en cuanto a las vías alternativas, sería beneficioso ampliar el tipo de herramientas de las que disponemos. Nos referimos principalmente a la utilización de otro tipo de factorizaciones matriciales como alternativa a la descomposición de Cholesky, como son lso casos de la *LU* o la *QR*, cada una con sus propiedades y limitaciones. En el caso de que se deseara desarrollar esto, se deja a modo de referencia breve las siguientes explicaciones sobre estas factorizaciones matriciales.

Factorización *LU*

La factorización LU es un tipo de descomposición que puede emplearse para dividir una matriz cuadrada A en el producto de otras dos: una triangular inferior L y otra triangular superior U. De este modo, tenemos que $A = LU$ (Jaramillo-Vidal-Correa, 2006).

La utilidad principal de este proceso es la de resolver sistemas de ecuaciones de un modo más eficiente, para lo cual convertimos un sistema del tipo $Ax = b$ en dos subsistemas triangulares equivalentes como son:

$$Ax = b \rightarrow LUx = b \rightarrow \begin{cases} Ux = y \\ Ly = b \end{cases} \quad donde\ y = L^{-1}b$$

Por tanto, primeramente, se recuelve el sistema $Ly = b$ para determinar y (incógnitas añadidas), para luego hallar la solución final mediante $Ux = y$, donde y es conocida y x contiene las incógnitas. La ventaja que supone emplear este procedimiento es que resolver dos sistemas de ecuaciones triangulares es más sencillo que enfrentarse a uno complejo.

Las matrices triangulares L y U se obtienen aplicando eliminación Gaussiana sobre A, aunque también es posible utilizar otro procedimiento muy habitual en álgebra lineal, como es la separación por bloques, expuesta a continuación:

Se la descomposición LU de A, desarrollada en forma matricial:

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{21} & u_{31} & \cdots & u_{n1} \\ 0 & u_{22} & u_{32} & \cdots & u_{n2} \\ 0 & 0 & u_{33} & \cdots & u_{n3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

Se han de hallar las siguientes ecuaciones:

$$a_{11} = l_{11}u_{11}$$

$$a_{12} = l_{11}u_{12}$$

$$a_{21} = l_{21}u_{11}$$

$$a_{22} = l_{21}u_{12} + l_{22}u_{22}$$

Por lo tanto, el primer paso es determinar de manera directa (eliminación gaussiana) u_{11} y l_{11}, para lo cual se convierten en cero todos los elementos por debajo de a_{11}. Luego, dado que l_{11} es una matriz triangular inferior, se puede aplicar el método de resolución de sistema de ecuaciones triangulares y, empleando la segunda ecuación ($a_{12} = l_{11}u_{12}$), se obtiene u_{12}. De igual modo, usando la tercera ecuación ($a_{21} = l_{21}u_{11}$) se determina l_{21}. Por último, con la cuarta ecuación ($a_{22} = l_{21}u_{12} + l_{22}u_{22}$) se consigue despejar l_{22} y u_{22} reescribiendola así:

$$l_{22}u_{22} = a_{22} - l_{21}u_{12}$$

Lo que permite seguir aplicando el método anteriormente descrito sobre la submatriz a_{11}

actualizada, es decir, sobre la submatriz $a_{11} - l_{22}u_{22}$.

Si se analiza lo visto hasta ahora, se puede observar que la resolución de la factorización *LU* por bloques se fundamenta en: algunas eliminaciones gaussianas, resolución de sistemas de ecuaciones triangulares, multiplicación y resta de matrices, etc. Todo ello conlleva problemas de estabilidad numérica indeseados en el algoritmo, por lo que se puede optar por una modificación del mismo empleando la técnica del pivoteo, en la que se busca el elemento de mayor valor absoluto de cada fila y se permutan las columnas, antes de aplicar la factorización a cada elemento. A cambio de ganar en estabilidad, se sacrifica rendimiento, pero por lo general merecer la pena cargar con dicha penalización (Tinetti- Denhamy, 2003).

Factorización *QR*

Sea *A* una matriz cuadrada de dimensión *n* x *n*, de números reales e invertible, existen dos matrices: *Q* cuadrada, de dimensión *n* x *n*, de números reales y ortogonal ($Q^T Q = I$); y *R* cuadrada, de dimensión *n* x *n*, de números reales, triangular superior e invertible; tales que:

$$A = QR$$

Esta descomposición se conoce como factorización *QR* de *A*. Para determinar dicha descomposición, es necesario aplicar le método de Gram-Schmidt sobre *A* de manera que obtengamos *Q*. Una vez hecho esto, se despeja *R* empleando:

$$R = Q^T A$$

La utilidad fundamental de esta descomposición es la de permitir la resolución de sistemas de ecuaciones del siguiente modo:

Sea un sistema del tipo:

$$Ax = b$$

aplicamos la factorización *QR* sobre *A* y obtenemos ambas matrices. Acto seguido, la aplicamos al sistema anterior:

$$QRx = b$$

y multiplicando a ambos lados por Q^T tenemos que:

$$Rx = Q^T b$$

un sistema triangular superior, que puede resolverse fácilmente por sustitución recursiva al ser R una matriz triangular superior, tras realicar el producto $Q^T b$ (Ojeda-Gago, 2008).

ANEXO A: CÓDIGO C CREADO

A continuación, se detalla el código creado durante el desarrollo de este proyecto. Las funciones vienen ordenadas alfabéticamente para mayor comodidad. Finalmente, se exponen varios programas completos que ilustran el modo de emplear las funciones descritas.

A1. Cholesky *sparse*

Debido a la necesidad de resolver sistemas de ecuaciones complejos se ha creado una función, llamada *cholesky_sparse*, que permite obtener la descomposición de Cholesky de una determinada matriz. La verdadera potencia de este código reside en que puede ser aplicado a matrices *sparse* codificadas, devolviendo como resultado la codificación de la descomposición requerida.

De esta función se han desarrollado dos versiones: *cholesky_sparse_v1* y *cholesky_sparse_v2* que implementan lo anteriormente expuesto. La segunda de ellas es considerablemente más rápida que la primera al asumir que los datos codificados vienen dados de manera ordenada, entre otras mejoras menores.

El código correspondiente a *cholesky_sparse_v1* es el siguiente:

```
// Función que realiza la descomposición de Cholesky de una matriz sparse
cuadrada semidefinida positiva:
int cholesky_sparse_v1(struct datos* infoA, struct datos* infoRc){
  int i, j, k, n, cont;
  double aux, Rcik, Rcjk, Aij, Rcjj, res;
  // Rellenamos infoRc:
  infoRc->Nr = infoA->Nr;
  infoRc->Nc = infoRc->Nr;
  infoRc->Nd = 0;
  infoRc->M = infoA->M; // Reservamos memoria por exceso
  infoRc->R = (int*)malloc(infoRc->M*sizeof(int));
  infoRc->C = (int*)malloc(infoRc->M*sizeof(int));
  infoRc->Z = (double*)malloc(infoRc->M*sizeof(double));
```

```
// Ponemos a cero el contador de resultados y comenzamos:
cont = 0;
for(i=0;i<infoRc->Nc;i++){
  for(j=0;j<(i+1);j++){
    aux = 0;
    for(k=0;k<j;k++){
      // Buscamos dentro de infoRc:
      Rcik = 0;
      Rcjk = 0;
      for(n=0;n<cont;n++){
        // Si encontramos R=i y C=k entonces nos quedamos con la Z:
        if(infoRc->R[n]==i && infoRc->C[n]==k){
          Rcik = infoRc->Z[n];
        }
        // Si encontramos R=j y C=k entonces nos quedamos con la Z:
        if(infoRc->R[n]==j && infoRc->C[n]==k){
          Rcjk = infoRc->Z[n];
        }
      }
      aux = aux + Rcik * Rcjk;
    }
    if(i==j){
      // Buscamos dentro de infoA:
      for(n=0;n<infoA->M;n++){
        // Si encontramos R=i y C=i entonces nos quedamos con la Z:
        if(infoA->R[n]==i && infoA->C[n]==i){
          res = sqrt(infoA->Z[n]-aux);
          // Si el resultado de la operación es no nulo:
          if(res>0.0 || res<0.0){
            cont = cont + 1; // Incrementamos el número de ternas resultado
            // Rellenamos la terna en infoRc:
            infoRc->R[cont-1] = i;
            infoRc->C[cont-1] = j;
            infoRc->Z[cont-1] = res;
          }
        }
      }
    }
    else{
      // Buscamos dentro de infoA:
      Aij = 0;
      Rcjj = 0;
      for(n=0;n<infoA->M;n++){
        // Si encontramos R=i y C=j entonces nos quedamos con la Z:
        if(infoA->R[n]==i && infoA->C[n]==j){
          Aij = infoA->Z[n];
        }
      }
      // Buscamos dentro de infoRc:
      for(n=0;n<cont;n++){
        // Si encontramos R=j y C=j entonces nos quedamos con la Z:
        if(infoRc->R[n]==j && infoRc->C[n]==j){
          Rcjj = infoRc->Z[n];
        }
      }
```

```
        res = (Aij-aux) / Rcjj;
        // Si el resultado de la operación es no nulo:
        if(res>0.0 || res<0.0){
          cont = cont + 1; // Incrementamos el número de ternas resultado
          // Rellenamos la terna en infoRc:
          infoRc->R[cont-1] = i;
          infoRc->C[cont-1] = j;
          infoRc->Z[cont-1] = res;
        }
      }
    }
  }
  // Una vez hemos acabado, sustituimos el tamaño de los datos por el
contador obtenido (actualizamos):
  infoRc->M = cont;
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

El código correspondiente a *cholesky_sparse_v2* es el siguiente:

```
// Función que realiza la descomposición de Cholesky de una matriz sparse
cuadrada semidefinida positiva cuya codificación viene ordenada por filas:
int cholesky_sparse_v2(struct datos* infoA, struct datos* infoRc){
  int k, n, i, j, encontrado, m;
  double aux, Rcin, Rcjn, Aii, Rcjj;
  // Rellenamos infoRc:
  infoRc->Nr = infoA->Nr;
  infoRc->Nc = infoRc->Nr;
  infoRc->Nd = 0;
  infoRc->M = infoA->M; // Reservamos memoria por exceso
  infoRc->R = (int*)malloc(infoRc->M*sizeof(int));
  infoRc->C = (int*)malloc(infoRc->M*sizeof(int));
  infoRc->Z = (double*)malloc(infoRc->M*sizeof(double));
  // Generamos la plantilla de Rc:
  for(k=0;k<infoA->M;k++){
    infoRc->R[k] = infoA->R[k];
    infoRc->C[k] = infoA->C[k];
    infoRc->Z[k] = 0.0;
  }
  // Recorremos la plantilla de Rc:
  for(k=0;k<infoRc->M;k++){
    i = infoRc->R[k];
    j = infoRc->C[k];
    aux = 0;
    for(n=0;n<j;n++){
      // Buscamos m tal que R[m] = i y C[m] = n
      encontrado = 0;
      m = 0;
      Rcin = 0.0;
      while(encontrado == 0 && m < infoRc->M){
```

```
      if(infoRc->R[m] == i && infoRc->C[m] == n){
        encontrado = 1;
        Rcin = infoRc->Z[m];
      }
      else{
        m = m + 1;
      }
    }
    // Buscamos m tal que R[m] = j y C[m] = n
    encontrado = 0;
    m = 0;
    Rcjn = 0.0;
    while(encontrado == 0 && m < infoRc->M){
      if(infoRc->R[m] == j && infoRc->C[m] == n){
        encontrado = 1;
        Rcjn = infoRc->Z[m];
      }
      else{
        m = m + 1;
      }
    }
    aux = aux + Rcin * Rcjn;
  }
  if(i==j){
    // Buscamos el elemento de A que tiene R=i y C=i:
    encontrado = 0;
    m = 0;
    Aii = 0.0;
    while(encontrado == 0 && m < infoA->M){
      if(infoA->R[m] == i && infoA->C[m] == i){
        encontrado = 1;
        Aii = infoA->Z[m];
      }
      else{
        m = m + 1;
      }
    }
    // Si no vamos a hacer la raíz cuadrada de un elemento negativo:
    if( (Aii - aux) >0.0){
      infoRc->Z[k] = sqrt( Aii - aux );
    }
  }
  else{
    // Buscamos m tal que R[m] = j y C[m] = j:
    encontrado = 0;
    m = 0;
    Rcjj = 0.0;
    while(encontrado == 0 && m < infoRc->M){
      if(infoRc->R[m] == j && infoRc->C[m] == j){
        encontrado = 1;
        Rcjj = infoRc->Z[m];
      }
      else{
        m = m + 1;
      }
```

```
      }
      // Si no vamos a dividir por cero:
      if(Rcjj>0.0 || Rcjj<0.0){
        infoRc->Z[k] = ( infoA->Z[k] - aux ) / Rcjj;
      }
    }
  }
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

Como precursoras de estas dos funciones *sparse*, se generaron otras más simples que permiten calcular, de dos formas diferentes, la descomposición de Cholesky de una matriz cualquiera (no tiene que ser *sparse*, ni en banda ni nada similar, salvo ser cuadrada y semidefinida positiva). Los códigos desarrollados se muestran a continuación a modo de referencia:

Código de *cholesky_v1*:

```
// Función que realiza la descomposición de Cholesky de una matriz cuadrada
semidefinida positiva:
double* cholesky_v1(double* A, int n){
  int i, j, k;
  double aux;
  double* Rc = (double*)calloc(n*n,sizeof(double));
  // Recorremos la matriz usando los índices i,j:
  for(i=0;i<n;i++){
    for(j=0;j<(i+1);j++){
      // Inicializamos aux a cero para cada elemento:
      aux = 0;
      // Actualizamos aux para cada columna:
      for(k=0;k<j;k++){
        aux = aux + Rc[i*n+k] * Rc[j*n+k];
      }
      // Si i = j
      if(i==j){
        Rc[i*n+j] = sqrt(A[i*n+i]-aux);
      }
      // Si i != j
      else{
        Rc[i*n+j] = (A[i*n+j]-aux) / Rc[j*n+j];
      }
    }
  }
  // Devolvemos Rc al terminar los cálculos:
  return Rc;
}
```

Código de *cholesky_v2*:

```
// Función que realiza la descomposición de Cholesky de una matriz cuadrada
semidefinida positiva:
double* cholesky_v2(double* A, int n){
  int k, i, j;
  double* G = A;
```

```
  double* v = (double*)calloc(n,sizeof(double));
  double* Rc = (double*)calloc(n*n,sizeof(double));
  // Recorremos la matriz usando los índices k,i:
  for(k=0;k<n;k++){
    // Inicializamos v a 0.0 para cada k:
    for(i=0;i<n;i++){
      v[i] = 0.0;
    }
    v[k] = sqrt(G[k*n+k]);
    Rc[k*n+k] = v[k];
    for(i=k+1;i<n;i++){
      v[i] = ( 1/sqrt( G[k*n+k] ) )*G[i*n+k];
      Rc[i*n+k] = v[i];
    }
    // Restamos G - v*v' sobrescribiendo/actualizando G:
    for(i=0;i<n;i++){
      for(j=0;j<n;j++){
        G[i*n+j] = G[i*n+j] - v[i]*v[j];
      }
    }
  }
  // Liberamos el espacio reservado:
  free(G);
  free(v);
  // Devolvemos Rc al terminar los cálculos:
  return Rc;
}
```

A2. Conversor *sparse*

Con la finalidad de simplificar las estructuras de datos para poder realizar cálculos con las mismas, se ha creado una función, llamada *conversor_sparse*, que permite obtener la codificación simple (sólo elementos simples) de una determinada matriz a partir de su versión compleja (elementos simples, diadas y flechas).

El código correspondiente a esta función es el siguiente:

```
// Función para convertir codificaciones complejas en simples:
int conversor_sparse(struct datos* info, struct datos* infoSimple){
  // Rellenamos infoSimple:
  infoSimple->Nr = info->Nr;
  infoSimple->Nc = info->Nc;
  infoSimple->Nd = 0;  // Eliminamos las diadas
  infoSimple->M = 10*info->M; // Reservamos memoria por exceso
  infoSimple->R = (int*)malloc(infoSimple->M*sizeof(int));
  infoSimple->C = (int*)malloc(infoSimple->M*sizeof(int));
  infoSimple->Z = (double*)malloc(infoSimple->M*sizeof(double));
  // Índices:
```

```
int i, j, cont;
// Rellenamos la codificación simple en función de los datos codificados:
cont = 0; // Inicializamos el contador de ternas a cero
for(i=0;i<info->M;i++){
  // Si estamos dentro de los límites de la matriz H:
  if(info->R[i]>=0 && info->R[i]<info->Nr && info->C[i]>=0 &&
     info->C[i]<info->Nc){
    cont = cont + 1; // Incrementamos el número de ternas resultado
    // Rellenamos la terna en infoSimple:
    infoSimple->R[cont-1] = info->R[i];
    infoSimple->C[cont-1] = info->C[i];
    infoSimple->Z[cont-1] = info->Z[i];
  }
  // Si estamos en la diada, concretamente en el vector u (vertical):
  else if(info->R[i]>=0 && info->R[i]<info->Nr && info->C[i]>=info->Nc &&
          info->C[i]<(info->Nc+info->Nd)){
    // Buscamos los elementos no nulos del vector v de la diada horizontal:
    for(j=0;j<info->M;j++){
      // Tenemos que detectar las flechas para no confundir una terna de
      dirección con el vector v (horizontal):
      if(info->R[j]>=info->Nr && info->R[j]<(info->Nr+info->Nd) &&
         info->C[j]>=info->Nc && info->C[j]<(info->Nc+info->Nd)){
        // Nos saltamos la terna de la dirección, incrementamos j:
        j = j + 1;
      }
      // Si el elemento pertenece al vector v de la diada horizontal:
      else if(info->R[j]>=info->Nr && info->R[j]<(info->Nr+info->Nd) &&
              info->C[j]>=0 && info->C[j]<info->Nc){
        // Si R[j] = C[i]:
        if(info->R[j] == info->C[i]){
          // Rellenamos codificación simple así: R[cont-1]=R[j],
          C[cont-1]=C[i], Z[cont-1]=Z[i]*Z[j]
          cont = cont + 1; // Incrementamos el número de ternas resultado
          // Rellenamos la terna en infoSimple:
          infoSimple->R[cont-1] = info->R[i];
          infoSimple->C[cont-1] = info->C[j];
          infoSimple->Z[cont-1] = info->Z[i]*info->Z[j];
        }
      }
    }
  }
  // Si estamos en la diada, concretamente en el vector v (horizontal):
  else if(info->R[i]>=info->Nr && info->R[i]<(info->Nr+info->Nd)
          && info->C[i]>=0 && info->C[i]<info->Nc){
    // No hacemos nada porque ya hemos considerado esta info en el Caso 2 -
    fuera de los limites - diada - u
  }
  // Si estamos en caso de repetición:
  else if(info->R[i]>=info->Nr && info->R[i]<(info->Nr+info->Nd) &&
  info->C[i]>=info->Nc && info->C[i]<(info->Nc+info->Nd)){
    i = i + 1;  // Incrementamos i
    // Guardamos en infoSimple el primer elemento de la flecha:
    cont = cont + 1; // Incrementamos el número de ternas resultado
    // Rellenamos la terna en infoSimple:
```

```
      infoSimple->R[cont-1] = info->R[i-1]-info->Nr;
      infoSimple->C[cont-1] = info->C[i-1]-info->Nc;
      infoSimple->Z[cont-1] = info->Z[i-1];
      // Guardamos la sucesión según la terna de dirección:
      for(j=1; (info->R[i-1]-info->Nr+j*info->R[i])<info->Nr
          && (info->C[i-1]- info->Nc+j*info->C[i])<info->Nc; j++){
        cont = cont + 1; // Incrementamos el número de ternas resultado
        // Rellenamos la terna en infoSimple:
        infoSimple->R[cont-1] = info->R[i-1]-info->Nr+j*info->R[i];
        infoSimple->C[cont-1] = info->C[i-1]-info->Nc+j*info->C[i];
        infoSimple->Z[cont-1] = info->Z[i-1] + info->Z[i]*j;
      }
    }
    else{
      puts("\n\tconversor_sparse - terna erronea");
    }
  }
  // Una vez hemos acabado, sustituimos el tamaño de los datos por el
  contador obtenido (actualizamos):
  infoSimple->M = cont;
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

A3. Descodificador *sparse*

Con el objetivo de poder visualizar las matrices codificadas, se ha creado una función llamada *descodificador_sparse* que lee la codificación dada y devuelve la matriz correspondiente. Su funcionamiento consiste en leer una a una las ternas de codificación y rellenar una matriz previamente inicializada con la información codificada. Si detecta fallos en la codificación, avisa por pantalla y continúa con el siguiente grupo de datos.

El código c que permite hacer esto es el siguiente:

```
// Función para descodificar los datos:
int descodificador_sparse(struct datos* info){
  // Reserva de memoria:
  // Matriz para codifición sin diadas (Nr*Nc):
  double* H = (double*)malloc(info->Nr*info->Nc*sizeof(double));
  // Matriz de vectores verticales de cada diada (Nr*Nd):
  double* u = (double*)malloc(info->Nr*info->Nd*sizeof(double));
  // Matriz de vectores horizontales de cada diada (Nd*Nc):
  double* v = (double*)malloc(info->Nc*info->Nd*sizeof(double));
  // Matriz resultado (Nr*Nc):
  double* A = (double*)malloc(info->Nr*info->Nc*sizeof(double));
  // Índices para recorrer los vectores de datos codificados:
  int i, j, k;
  // Rellenamos la matriz H de ceros:
```

```
for(i=0;i<info->Nr;i++){
  for(j=0;j<info->Nc;j++){
    H[i*info->Nc+j] = 0.0;
  }
}
// Rellenamos las diadas de ceros:
// u:
for(i=0;i<info->Nr;i++){
  for(j=0;j<info->Nd;j++){
    u[i*info->Nd+j] = 0.0;
  }
}
// v:
for(i=0;i<info->Nd;i++){
  for(j=0;j<info->Nc;j++){
    v[i*info->Nc+j] = 0.0;
  }
}
// Rellenamos la matriz en función de los datos codificados:
for(i=0;i<info->M;i++){
  // Si estamos dentro de los límites de la matriz H:
  if(info->R[i]>=0 && info->R[i]<info->Nr && info->C[i]>=0 &&
     info->C[i]<info->Nc){
    // Sumamos al elemento concreto el valor no nulo que debe ser:
    H[info->R[i]*info->Nc+info->C[i]]=H[info->R[i]*info->Nc+
      info->C[i]]+info->Z[i];
  }
  // Si estamos en la diada, concretamente en el vector u (vertical):
  else if(info->R[i]>=0 && info->R[i]<info->Nr && info->C[i]>=info->Nc &&
          info->C[i]<(info->Nc+info->Nd)){
    // Sumamos al elemento concreto el valor no nulo que debe ser:
    u[info->R[i]*info->Nd+(info->C[i]-info->Nc)]=u[info->R[i]*
    info->Nd+(info->C[i]-info->Nc)]+info->Z[i];
  }
  // Si estamos en la diada, concretamente en el vector v (horizontal):
  else if(info->R[i]>=info->Nr && info->R[i]<(info->Nr+info->Nd) &&
          info->C[i]>=0 && info->C[i]<info->Nc){
    // Sumamos al elemento concreto el valor no nulo que debe ser:
    v[(info->R[i]-info->Nr)*info->Nc+info->C[i]]=v[(info->R[i]-
    info->Nr)*info->Nc+info->C[i]]+info->Z[i];
  }
  // Si estamos en caso de repetición, es decir, en la matriz A:
  else if(info->R[i]>=info->Nr && info->R[i]<(info->Nr+info->Nd) &&
          info->C[i]>=info->Nc && info->C[i]<(info->Nc+info->Nd)){
    i = i + 1;  // Incrementamos i
    // Sumamos el primer valor de la secuencia repetitiva:
    H[(info->R[i-1]-info->Nr)*info->Nc+(info->C[i-1]-info->Nc)]=
    H[(info->R[i-1]-info->Nr)*info->Nc+(info->C[i-1]-info->Nc)]+
    info->Z[i-1];
    // Sumamos las repeticiones y las vamos mostrando por pantalla:
    for(j=1;(info->R[i-1]-info->Nr+j*info->R[i])<info->Nr && (info->C[i-1]-
        info->Nc+j*info->C[i])<info->Nc;j++){
      H[(info->R[i-1]-info->Nr+j*info->R[i])*info->Nc+info->C[i-1]-
      info->Nc+j*info->C[i]]=H[(info->R[i-1]-info->Nr+j*info->R[i])*
```

```
        info->Nc+info->C[i-1]-info->Nc+j*info->C[i]]+info->Z[i-1]+
        info->Z[i]*j;
      }
    }
    else{
      puts("\n\tCaso desconocido - Terna erronea");
    }
  }
  // Desarrollamos la diada multiplicando u*v y obteniendo la matriz D:
  double* D = (double*)malloc(info->Nr*info->Nc*sizeof(double));
  // Rellenamos la matriz D de ceros:
  for(i=0;i<info->Nr;i++){
    for(j=0;j<info->Nc;j++){
      D[i*info->Nc+j] = 0.0;
    }
  }
  // Realizamos la multiplicación de las matrices u y v:
  double aux;
  for(i=0;i<info->Nr;i++){
    for(j=0;j<info->Nc;j++){
      aux = 0;
      for(k=0;k<info->Nd;k++){
        aux = aux + u[i*info->Nd+k] * v[k*info->Nc+j];
      }
      D[i*info->Nc+j] = aux;
    }
  }
  // Rellenamos la matriz A de ceros:
  for(i=0;i<info->Nr;i++){
    for(j=0;j<info->Nc;j++){
      A[i*info->Nc+j] = 0.0;
    }
  }
  // Sumamos la matriz H con la D para obtener A:
  for(i=0;i<info->Nr;i++){
    for(j=0;j<info->Nc;j++){
      A[i*info->Nc+j] = H[i*info->Nc+j] + D[i*info->Nc+j];
      printf("\t%f  ", A[i*info->Nc+j]);
    }printf("\n");
  }
  // Liberamos espacio reservado:
  free(H); free(u); free(v); free(D); free(A);
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

A4. Leer datos externos

Para poder importar datos a nuestro programa generados por otro programa (en nuestro caso MatLab) se ha creado una función, llamada *lectordatos_sparse*, que permite extraer de un fichero .txt la codificación *sparse* simple de una matriz.

El código correspondiente a esta función es el siguiente:

```
// Función para importar datos R,C,Z de un fichero de texto:
int lectordatos_sparse(struct datos* info,char* nombre){
  int i, j, k;
  FILE* archivoID;
  char aux[10];
  double dato;
  // Abrimos el archivo de datos:
  archivoID = fopen(nombre,"r");
  // Inicializamos el contador de ternas no nulas a cero:
  k = 0;
  // Recorremos el fichero buscando:
  for(i=0;i<info->Nr;i++){
    for(j=0;j<info->Nc;j++){
      fscanf(archivoID," %s ",&aux);
      // Convertimos el dato extraído del fichero de cadena de caracteres a
      número double:
      dato = atof(aux);
      // Si el dato es distinto de cero:
      if(dato>0.0 || dato<0.0){
        info->R[k] = i;
        info->C[k] = j;
        info->Z[k] = dato;
        k++;
      }
    }
  }
  // Cerramos el archivo para dejarlo libre de acceso:
  fclose(archivoID);
  // Actualizamos el tamaño de la tabla de datos:
  info->M = k;
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

A5. Multiplicador de matrices *sparse*

Como herramienta para facilitar los cálculos, se ha creado una función, llamada *multiMxM_sparse*, que permite multiplicar dos matrices *sparse* empleando sus respectivas codificaciones directamente.

El código correspondiente a esta función es el siguiente:

```
// Función para multiplicar dos matrices sparse de codificación simple (usar
previamente conversor_sparse si no es así):
int multiMxM_sparse(struct datos* infoA, struct datos* infoB,
                    struct datos* infoC){
  // Índices:
  int i, j, k, x, y, cont;
  // Comprobamos que A y B puedan multiplcarse (Nc_A = Nr_B):
  if(infoA->Nc==infoB->Nr){
  // Rellenamos y reservamos memoria:
    infoC->Nr = infoA->Nr;
    infoC->Nc = infoB->Nc;
    infoC->Nd = 0;
    infoC->M = infoA->M*infoB->M;  // Inicialización por exceso
    infoC->R = (int*)malloc(infoC->M*sizeof(int));
    infoC->C = (int*)malloc(infoC->M*sizeof(int));
    infoC->Z = (double*)malloc(infoC->M*sizeof(double));
    // Multiplicación:
    cont = 0; // Inicializamos el contador de ternas a cero
    // Código que multiplica dos matrices sparse:
    for(i=0;i<infoA->M;i++){
      // Si estamos dentro de los límites de la matriz A:
      if(infoA->R[i]>=0 && infoA->R[i]<infoA->Nr && infoA->C[i]>=0 &&
         infoA->C[i]<infoA->Nc){
        // Recorremos la codificación de B para encontar ternas con RB = CA:
        for(j=0;j<infoB->M;j++){
          // Si RB[j] = CA[i]:
          if(infoB->R[j]==infoA->C[i]){
            // Buscamos el elemento correspondiente de B:
            // Nos aseguramos de seguir dentro de los límites de B:
            if(infoB->R[j]>=0 && infoB->R[j]<infoB->Nr && infoB->C[j]>=0 &&
               infoB->C[j]<infoB->Nc){
              cont = cont + 1; // Incrementamos el número de ternas resultado
              // Multiplicamos y almacenamos en el elemento RC[k]=RA[i],
              CC[k]=CB[j], ZC[k]=ZA[i]*ZB[j]:
              infoC->R[cont-1] = infoA->R[i];
              infoC->C[cont-1] = infoB->C[j];
              infoC->Z[cont-1] = infoA->Z[i]*infoB->Z[j];
            }
          }
        }
      }
      // Si estamos en la diada, concretamente en el vector u (vertical):
      else if(infoA->R[i] >= 0 && infoA->R[i]<infoA->Nr && infoA->C[i] >=
              infoA->Nc && infoA->C[i] < (infoA->Nc+infoA->Nd)){
```

```
        // Sólo consideramos el caso de operaciones simples
      }
      // Si estamos en la diada, concretamente en el vector v (horizontal):
      else if(infoA->R[i] >= infoA->Nr && infoA->R[i] < (infoA->Nr+infoA->Nd)
              && infoA->C[i] >= 0 && infoA->C[i]<infoA->Nc){
        // Sólo consideramos el caso de operaciones simples
      }
      // Si estamos en caso de repetición, es decir, en la matriz A:
      else if(infoA->R[i]>=infoA->Nr && infoA->R[i]<(infoA->Nr+infoA->Nd) &&
              infoA->C[i]>=infoA->Nc && infoA->C[i]<(infoA->Nc+infoA->Nd)){
        i = i + 1;  // Incrementamos i para saltarnos la flecha
        // Sólo consideramos el caso de operaciones simples
      }
      else{
        printf("\n\tmultiMxM_Sparse: fallo en la codificación, terna
               errónea");
        // Sólo consideramos el caso de operaciones simples
      }
    }
    // Una vez hemos acabado, actualizamos el tamaño de los datos por el
    contador obtenido:
    infoC->M = cont;
    // Si la función termina correctamente devuelve 0:
    return 0;
  }
  else{
    printf("\n\tmultiMxM_Sparse: las matrices no tienen dimensiones
    compatibles");
    // Si la función termina por este motivo devuelve un uno:
    return 1;
  }
}
```

Cabe decir que, tras el uso de esta función, es convenienete emplear *posprocesador_sparse* para simplificar la codificación resultante.

A6. Multiplicar una matriz *sparse* por un vector

A modo de instrumento para facilitar los cálculos, se ha creado una función, llamada *multMxV_sparse*, que permite multiplicar una matriz *sparse* con un vector o su traspuesta por el mismo vector, empleando su codificación directamente.

El código correspondiente a esta función es el siguiente:

```
// Función para multiplicar una matriz sparse (codificación simple) o su
traspuesta por un vector:
double* multMxV_sparse(struct datos* info,double* x, int modo){
  // Índice para recorrer vectores:
  int i = 0;
  // Reservamos memoria para el vector y de soluciones:
  double* y = (double*)malloc(info->Nr*sizeof(double));
  // Ponemos a cero todos los elementos del vector:
  for(i=0;i<info->Nr;i++){
    y[i] = 0;
  }
  // Comprobamos el modo:
  if(modo == 1){
    // Dejamos la codificación como está y multiplicamos tal cual:
    for(i=0;i<info->M;i++){
      // Si estamos dentro de los límites de la matriz:
      if(info->R[i]>=0 && info->R[i]<info->Nr && info->C[i]>=0 &&
         info->C[i]<info->Nc){
        y[info->R[i]] = y[info->R[i]] + info->Z[i] * x[info->C[i]];
      }
      else{
        puts("\n\tmultMxV - Caso no considerado");
      }
    }
  }
  else if(modo == 2){
    // Tenemos que trasponer la matriz -> cambiamos la codificación
    localmente (sólo dentro de esta función, no globalmente) y después
    multiplicamos:
    int* R = (int*)malloc(info->M*sizeof(int));
    int* C = (int*)malloc(info->M*sizeof(int));
    R = info->C;
    C = info->R;
    // Multiplicación:
    for(i=0;i<info->M;i++){
      // Si estamos dentro de los límites de la matriz:
      if(R[i]>=0 && R[i]<info->Nr && C[i]>=0 && C[i]<info->Nc){
        y[R[i]] = y[R[i]] + info->Z[i] * x[C[i]];
      }
      else{
        puts("\n\tmultMxV - Caso no considerado");
      }
```

```
    }
    // Liberamos espacio reservado:
    free(R); free(C);
  }
  else{
    puts("\n\tmultMxV - Error: modo incorrecto");
  }
  // Si la función termina correctamente devuelve un puntero al vector
  resultante:
  return y;
  // Liberamos espacio reservado:
  free(y);
}
```

A7. Posprocesador *sparse*

Debido a los cálculos y operaciones llevados a cabo por otras funciones, pueden aparecer en las codificaciones resultantes ceros o ternas que hacen referencia al mismo elemento de la matriz, lo cual es necesario eliminar o simplificar para que no se produzcan errores. Para ello, se ha desarrollado una función, llamada *posprocesador_sparse*, que realiza los cambios necesarios.

El código correspondiente a esta función es el siguiente:

```
// Función para ordenar, simplificar y eliminar ceros de codificaciones
simples:
int posprocesador_sparse(struct datos* infoI, struct datos* infoF){
  // Índices:
  int i, j, repetido, cont;
  double aux;
  // Rellenamos los datos de la codificación final:
  infoF->Nr = infoI->Nr;
  infoF->Nc = infoI->Nc;
  infoF->Nd = infoI->Nd;
  infoF->M = infoI->M; // Inicialización por exceso
  infoF->R = (int*)malloc(infoF->M*sizeof(int));
  infoF->C = (int*)malloc(infoF->M*sizeof(int));
  infoF->Z = (double*)malloc(infoF->M*sizeof(double));
  // Copiamos la codificación inicial en tres vectores auxiliares para no
  modificar infoI:
  int* usado = (int*)malloc(infoI->M*sizeof(int));
  // Inicializamos usado a uno, no hemos utilizado ninguna terna:
  for(i=0;i<infoI->M;i++){
    usado[i] = 1;
  }
  // Recorremos la codificación inicial:
  cont = 0; // Inicializamos el contador de ternas a cero
  for(i=0;i<infoI->M;i++){
    // Recorremos la codificación inicial para buscar coincidencias:
```

```
    repetido = 0; // Inicializamos, no hay repetición
    aux = 0; // Inicializamos para cada i
    for(j=i+1;j<infoI->M;j++){
      // Si encontramos una terna repetida:
      if(infoI->R[i]==infoI->R[j] && infoI->C[i]==infoI->C[j] &&
         usado[j]==1){
        repetido = 1; // Hemos encontrado una terna repetida, con que se
                      repita una vez se activa
        // La sumamos y almacenamos temporalmente:
        aux = aux + infoI->Z[j];
        // Ponemos a cero usado para eliminar las ternas de la lista que ya
        han sido usadas/sumadas:
        usado[j] = 0;
      }
    }
    // Si la terna no está repetida y es no nula:
    if(repetido==0 && usado[i]==1 && ( infoI->Z[i]>0.0 || infoI->Z[i]<0.0 )){
      // La almacenamos:
      cont = cont + 1; // Incrementamos el número de ternas resultado
      // Rellenamos la terna en infoSimple:
      infoF->R[cont-1] = infoI->R[i];
      infoF->C[cont-1] = infoI->C[i];
      infoF->Z[cont-1] = infoI->Z[i];
    }
    // Si la terna está repetida:
    else if(repetido==1 && usado[i]==1){
      cont = cont + 1; // Incrementamos el número de ternas resultado
      // Rellenamos la terna en infoSimple:
      infoF->R[cont-1] = infoI->R[i];
      infoF->C[cont-1] = infoI->C[i];
      infoF->Z[cont-1] = infoI->Z[i] + aux;
    }
  }
  //Liberamos el espacio reservado:
  free(usado);
  // Una vez hemos acabado, sustituimos el tamaño de los datos por el
  contador obtenido (actualizamos):
  infoF->M = cont;
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

A8. Sistemas de ecuaciones en semi banda *sparse*

En un caso real, puede darse la posibilidad de que nos encontremos con un sistema de ecuaciones a resolver, cuya matriz no sea estrictamente en banda. Para poder calcular la solución del mismo empleando nuestras funciones, recurriremos al método de resolución de sistemas de ecuaciones en semi banda expuesto en capítulos anteriores.

La función que permite implementar este procedimiento ha sido llamada *semibanda_sparse*, siendo su código el siguiente:

```
// Función que implementa el método para sistemas de ecuaciones en semi banda
int semibanda_sparse(int n,int nd){
  // Definimos todas las estructuras de datos necesarias:
  struct datos infoB;
  struct datos infoBSimple;
  struct datos infoBPos;
  struct datos infoRcB;
  struct datos infoRcBPos;
  //-------------------------------------------------------------------------
  struct datos infoU;
  struct datos infoV;
  struct datos infoVt;
  //-------------------------------------------------------------------------
  struct datos infoC;
  struct datos infoVtC;
  struct datos infoVtCPos;
  //-------------------------------------------------------------------------
  struct datos infoI;
  //-------------------------------------------------------------------------
  struct datos infoA;
  struct datos infoAPos;
  struct datos infoRcA;
  struct datos infoRcAPos;
  //-------------------------------------------------------------------------
  int k, res, i, j;
  //-------------------------------------------------------------------------
  // Rellenamos infoB con la codificación:
  infoB.Nr = n;
  infoB.Nc = infoB.Nr;
  infoB.Nd = 0;
  infoB.M = infoB.Nr*infoB.Nc;
  infoB.R = (int*)malloc(infoB.M*sizeof(int));
  infoB.C = (int*)malloc(infoB.M*sizeof(int));
  infoB.Z = (double*)malloc(infoB.M*sizeof(double));
  for(k=0;k<infoB.M;k++){
    infoB.R[k] = 0;
    infoB.C[k] = 0;
```

```
    infoB.Z[k] = 0.0;
  }
  // Importamos datos del fichero externo llamando a la función lectordatos:
  res = lectordatos_sparse(&infoB,"matriz_B.txt");
  //-------------------------------------------------------------------------
  // Rellenamos infoU con la codificación:
  infoU.Nr = infoB.Nr;
  infoU.Nc = nd;
  infoU.Nd = 0;
  infoU.M = infoU.Nr*infoU.Nc;
  infoU.R = (int*)malloc(infoU.M*sizeof(int));
  infoU.C = (int*)malloc(infoU.M*sizeof(int));
  infoU.Z = (double*)malloc(infoU.M*sizeof(double));
  for(k=0;k<infoU.M;k++){
    infoU.R[k] = 0;
    infoU.C[k] = 0;
    infoU.Z[k] = 0.0;
  }
  // Importamos datos del fichero externo llamando a la función lectordatos:
  res = lectordatos_sparse(&infoU,"matriz_U.txt");
  //-------------------------------------------------------------------------
--------
  // Rellenamos infoV con la codificación:
  infoV.Nr = infoB.Nr;
  infoV.Nc = nd;
  infoV.Nd = 0;
  infoV.M = infoV.Nr*infoV.Nc;
  infoV.R = (int*)malloc(infoV.M*sizeof(int));
  infoV.C = (int*)malloc(infoV.M*sizeof(int));
  infoV.Z = (double*)malloc(infoV.M*sizeof(double));
  for(k=0;k<infoV.M;k++){
    infoV.R[k] = 0;
    infoV.C[k] = 0;
    infoV.Z[k] = 0.0;
  }
  // Importamos datos del fichero externo llamando a la función lectordatos:
  res = lectordatos_sparse(&infoV,"matriz_V.txt");
  //-------------------------------------------------------------------------
  // Tenemos que obtener V':
  res = traspuesta_sparse(&infoV,&infoVt);
  //-------------------------------------------------------------------------
  // Ahora necesitamos generar una matriz identidad de tamaño nd x nd:
  // NOTA: la generamos directamente en codificación simple, en lugar de usar
  una flecha, para no tener que usar conversor_sparse
  infoI.Nr = nd;
  infoI.Nc = nd;
  infoI.Nd = 0;
  infoI.M = nd;
  infoI.R = (int*)malloc(infoI.M*sizeof(int));
  infoI.C = (int*)malloc(infoI.M*sizeof(int));
  infoI.Z = (double*)malloc(infoI.M*sizeof(double));
  for(k=0;k<infoI.M;k++){
    infoI.R[k] = k;
    infoI.C[k] = k;
    infoI.Z[k] = 1.0;
```

```
}
//---------------------------------------------------------------------------
// Definimos los vectores z1, z3, x, y h:
double* z1 = (double*)malloc(infoB.Nr*sizeof(double));
double* z3 = (double*)malloc(infoB.Nr*sizeof(double));
double* x = (double*)malloc(infoB.Nr*sizeof(double));
double* h = (double*)malloc(infoB.Nr*sizeof(double));
// Inicializamos los vectores z1, z3, x, xd, y h:
for(k=0;k<infoB.Nr;k++){
  z1[k] = 0.0;
  z3[k] = 0.0;
  x[k] = 0.0;
  h[k] = 1;
}
//---------------------------------------------------------------------------
// Definimos los vectores z2 y z2_aux:
double* z2 = (double*)malloc(infoU.Nc*sizeof(double));
//printf("\nInicializamos el vector z2:");
for(k=0;k<infoU.Nc;k++){
  z2[k] = 0.0;
}
//---------------------------------------------------------------------------
// Tenemos que resolver 3 sistemas de ecuaciones:
//
//    B*z1 = h     ;    (I+V'*(B^-1)*U)*z2 = (V'*z1)    ;    B*z3 = (U*z2)
//
//---------------------------------------------------------------------------
// Hacemos la descomposición de Cholesky de B para resolver los sitemas
res = conversor_sparse(&infoB,&infoBSimple);
res = posprocesador_sparse(&infoBSimple,&infoBPos);
res = cholesky_sparse_v2(&infoBPos,&infoRcB);
// Posprocesamos RcB para eliminar posibles ceros en la codificación:
res = posprocesador_sparse(&infoRcB,&infoRcBPos);
//---------------------------------------------------------------------------
// Resolvemos el sitema B*z1 = h usando Cholesky --> (RcB*RcB')*z1 = h -->
// RcB*(RcB'*z1) = h --> RcB*z1_aux = h --> z1_aux = RcB\h --> RcB'*z1 =
// z1_aux --> = RcB'\z1_aux   para lo cual usamos sistecchol_sparse:
//---------------------------------------------------------------------------
// Resolvemos sistema B*z1 = h:
res = sistecchol_sparse(&infoRcBPos,z1,h);
//---------------------------------------------------------------------------
// Definimos el vector Vtz1:
double* Vtz1 = (double*)malloc(infoVt.Nr*sizeof(double));
// Inicializamos el vector a cero:
for(k=0;k<infoVt.Nr;k++){
  Vtz1[k] = 0.0;
}
// Definimos los vectores Cj y Uj::
double* Cj = (double*)malloc(infoU.Nr*sizeof(double));
double* Uj = (double*)malloc(infoU.Nr*sizeof(double));
// Y los inicializamos a cero:
for(k=0;k<infoU.Nr;k++){
  Cj[k] = 0.0;
  Uj[k] = 0.0;
```

```
}
// Rellenamos la codificación de C:
infoC.Nr = infoU.Nr;
infoC.Nc = infoU.Nc;
infoC.Nd = 0;
infoC.M = infoC.Nr*infoC.Nc;  // Reservamos por exceso
infoC.R = (int*)malloc(infoC.M*sizeof(int));
infoC.C = (int*)malloc(infoC.M*sizeof(int));
infoC.Z = (double*)malloc(infoC.M*sizeof(double));
for(k=0;k<infoC.M;k++){
  infoC.R[k] = 0;
  infoC.C[k] = 0;
  infoC.Z[k] = 0.0;
}
//---------------------------------------------------------------------------
// Resolvemos el sitema (I+V'*(B^-1)*U)*z2 = (V'*z1) usando
sistecchol_sparse:
// Necesitamos calcular (I+V'*(B^-1)*U) y (V'*z1) antes de resolver:
// Obtenemos Vtz1 = (V'*z1):
// Calculamos Vtz1 multiplicando Vt por z1:
Vtz1 = multMxV_sparse(&infoVt,z1,1);
//---------------------------------------------------------------------------
// Obtenemos A = (I+V'*(B^-1)*U), por lo que emplearemos que C = (B^-1)*U
// de tal manera que B*C = B*(B^-1)*U = U, que puede divirse en Ui = B*Ci
// (tantos subsistemas vector = matriz*vector como columnas (diadas) tenga
// U), que a su vez pueden resolverse empleando la descomposición de
// Cholesky de B. Para ello tengo que extraer las columnas de U, calcular
// los vectores Ci, con ellos montar la matriz C.
//---------------------------------------------------------------------------
i = 0; // Inicializamos a cero el índice para rellenar infoC
// Nos movemos por columnas:
for(j=0;j<infoU.Nc;j++){
  // Ponemos a cero los vectores Cj y Uj:
  for(k=0;k<infoU.Nr;k++){
      Cj[k] = 0.0;
      Uj[k] = 0.0;
  }
  // Recorremos la codificación de U:
  for(k=0;k<infoU.M;k++){
    // Si encontramos elementos con C = j:
    if(infoU.C[k] == j){
      // Guardamos en la posición R el vector columna Uj el valor Z
      asociado a C:
      Uj[ infoU.R[k] ] = infoU.Z[k];
    }
  }
  // Una vez hemos extraído el vector Uj, resolvemos el subsistema B*Cj =
  Uj para obtener Cj:
  res = sistecchol_sparse(&infoRcB,Cj,Uj);
  // Copiamos el contenido de Cj en la codificación de C:
  for(k=0;k<infoU.Nr;k++){
    // Si el elemento k de Cj es igual a 0.0:
    if(Cj[k] == 0.0){
      // No lo copiamos
    }
```

```
    // Pero si es distinto de cero sí lo almacenamos:
    else{
      infoC.R[i] = k;
      infoC.C[i] = j;
      infoC.Z[i] = Cj[k];
      i = i + 1;
    }
  }
}
// Una vez hemos calculado C, acualizamos el tamaño de la tabla de datos:
infoC.M = i;
//---------------------------------------------------------------------------
// Ahora determinamos VtC = V'*C empleando multiMxM_sparse:
res = multiMxM_sparse(&infoVt,&infoC,&infoVtC);
// Posprocesamos VtC para simplificar y eliminar posibles ceros en la
codificación:
res = posprocesador_sparse(&infoVtC,&infoVtCPos);
// Ahora sumamons A = I + VtC con sumres_sparse:
res = sumres_sparse(&infoI,&infoVtC,&infoA,1);
// Posprocesamos A para simplificar y eliminar posibles ceros:
res = posprocesador_sparse(&infoA,&infoAPos);
// Calculamos la Cholesky de A:
res = cholesky_sparse_v2(&infoAPos,&infoRcA);
// Posprocesamos RcA para eliminar posibles ceros en la codificación:
res = posprocesador_sparse(&infoRcA,&infoRcAPos);
// Resolvemos sistema A*z2 = Vtz1:
res = sistecchol_sparse(&infoRcAPos,z2,Vtz1);
//---------------------------------------------------------------------------
// Resolvemos el sitema B*z3 = (U*z2) usando sistecchol_sparse:
// Definimos el vector Uz2:
double* Uz2 = (double*)malloc(infoU.Nr*sizeof(double));
// Inicializamos el vector a cero:
for(k=0;k<infoU.Nr;k++) Uz2[k] = 0.0;
// Calculamos Uz2 = U*z2:
Uz2 = multMxV_sparse(&infoU,z2,1);
// Resolvemos sistema B*z3 = Uz2:
res = sistecchol_sparse(&infoRcB,z3,Uz2);
//---------------------------------------------------------------------------
// Obtenemos x como z1 - z3:
for(k=0;k<infoB.Nr;k++){
  x[k] = z1[k] - z3[k];
}
//---------------------------------------------------------------------------
// Liberamos espacio reservado:
free(infoB.R); free(infoB.C); free(infoB.Z);
free(infoBSimple.R); free(infoBSimple.C); free(infoBSimple.Z);
free(infoBPos.R); free(infoBPos.C); free(infoBPos.Z);
free(infoRcB.R); free(infoRcB.C); free(infoRcB.Z);
free(infoRcBPos.R); free(infoRcBPos.C); free(infoRcBPos.Z);
free(infoU.R); free(infoU.C); free(infoU.Z);
free(infoV.R); free(infoV.C); free(infoV.Z);
// NOTA: no hace falta liberar memoria de Vt porque ya está almacenada en V
(ver función traspuesta_sparse)
//free(infoVt.R); free(infoVt.C); free(infoVt.Z);
```

```
  free(infoC.R); free(infoC.C); free(infoC.Z);
  free(infoVtC.R); free(infoVtC.C); free(infoVtC.Z);
  free(infoVtCPos.R); free(infoVtCPos.C); free(infoVtCPos.Z);
  free(infoI.R); free(infoI.C); free(infoI.Z);
  free(infoA.R); free(infoA.C); free(infoA.Z);
  free(infoAPos.R); free(infoAPos.C); free(infoAPos.Z);
  free(infoRcA.R); free(infoRcA.C); free(infoRcA.Z);
  free(infoRcAPos.R); free(infoRcAPos.C); free(infoRcAPos.Z);
  free(z1); free(z3); free(x); free(h); free(z2);
  free(Vtz1); free(Cj); free(Uj);
  //---------------------------------------------------------------------
  // Si el programa principal termina correctamente devuelve 0:
  return 0;
}
```

A9. Resolver sistemas triangulares de ecuaciones *sparse*

Debido a la necesidad de resolver sistemas de ecuaciones complejos se ha creado una función, llamada *sistecchol_sparse*, que emplea la descomposición de Cholesky *sparse* obtenida con la función *cholesky_sparse* para obtener la solución del sistema.

El código correspondiente a esta función es el siguiente:

```
// Función para sistemas de ecuaciones usando Cholesky - Halla vector x
usando matriz de Cholesky Rc y vector dato b
// Resolvemos el sitema A*x = b usando Cholesky --> (Rc*Rc')*x = b -->
Rc*(Rc'*x) = b --> Rc*y = b --> y = Rc\b --> Rc'*x = y --> x = Rc'\y:
int sistecchol_sparse(struct datos* infoRc, double* x, double* b){
  // Declaraciones:
  int i, j, k;
  double aux, Rcij, Rcji, Rcii;
  // Definimos el vector y:
  double* y = (double*)malloc(infoRc->Nr*sizeof(double));
  // Inicializamos el vector y a cero:
  for(k=0;k<infoRc->Nr;k++){
    y[k] = 0.0;
  }
  //---------------------------------------------------------------------
  // Algoritmo recursivo de resolución para y = Rc\b:
  // y(0) = b(0) / Rc(0,0)  --> el primer elemento se calcula aparte
  // y(i) = [ b(i) - Sumatorio_de_j=1_hasta_j=i-1_de(Rc(i,j)*y(j)) ] /
     Rc(i,i)  con  i = 2,3,4,...,n
  //---------------------------------------------------------------------
  // Primer elemento: y(0) = b(0) / Rc(0,0)
  // Buscamos el elemento Rc(0,0):
  Rcii = 0.0; // Inicializamos a cero
  for(k=0;k<infoRc->M && Rcii==0.0;k++){
    // Si R(k) = 0 y C(k) = 0 lo hemos encontrado:
```

```
  if(infoRc->R[k] == 0 && infoRc->C[k] == 0){
    Rcii = infoRc->Z[k]; // Guardamos el valor encontrado
  }
}
y[0] = b[0] / Rcii;
//-------------------------------------------------------------------------
// Resto de elementos:
// y(i) = [ b(i) - Sumatorio_de_j=1_hasta_j=i-1_de(Rc(i,j)*y(j)) ] /
          Rc(i,i)  con i = 2,3,4,...,n
for(i=1;i<infoRc->Nr;i++){
  aux = 0.0; // Inicializamos para cada i
  Rcii = 0.0; // Inicializamos para cada i
  for(k=0;k<infoRc->M && Rcii==0.0;k++){
    // Si R(k) = i y C(k) = i lo hemos encontrado:
    if(infoRc->R[k] == i && infoRc->C[k] == i){
      Rcii = infoRc->Z[k]; // Guardamos el valor encontrado
    }
  }
  for(j=0;j<i;j++){
    // Localizamos Rc(i,j) y Rc(i,i) - recorremos infoRc:
    Rcij = 0.0; // Inicializamos para cada j
    for(k=0;k<infoRc->M && Rcij==0.0;k++){
      // Si R(k) = i y C(k) = j lo hemos encontrado:
      if(infoRc->R[k] == i && infoRc->C[k] == j){
        Rcij = infoRc->Z[k]; // Guardamos el valor encontrado
      }
    }
    aux = aux + Rcij * y[j];
  }
  // Si no vamos a dividir por cero:
  if(Rcii>0.0 || Rcii<0.0){
    y[i] = ( b[i] - aux ) / Rcii;
  }
}
//-------------------------------------------------------------------------
// Algoritmo recursivo de resolución para x = Rc'\y:
// x(n) = b(n) / Rc(n,n)  --> el último elemento se calcula aparte
// x(i) = [ y(i) - Sumatorio_de_j=i+1_hasta_j=n_de(Rc(j,i)*x(j)) ] /
          Rc(i,i)  con  i = n-1,n-2,...,1
//-------------------------------------------------------------------------
// Último elemento: x(n) = b(n) / Rc(n,n)
// Buscamos el elemento Rc(n,n):
Rcii = 0.0; // Inicializamos a cero
for(k=0;k<infoRc->M && Rcii==0.0;k++){
  // Si R(k) = n = (infoRc->Nr - 1) y C(k) = n = (infoRc->Nr - 1) lo hemos
  encontrado:
  if(infoRc->R[k] == (infoRc->Nr - 1) && infoRc->C[k] == (infoRc->Nr - 1)){
    Rcii = infoRc->Z[k]; // Guardamos el valor encontrado
  }
}
x[infoRc->Nr - 1] = y[infoRc->Nr - 1] / Rcii;
//-------------------------------------------------------------------------
// Resto de elementos:
// x(i) = [ y(i) - Sumatorio_de_j=i+1_hasta_j=n_de(Rc(j,i)*x(j)) ] /
```

```
            Rc(i,i)  con  i = n-1,n-2,...,1
  for(i=(infoRc->Nr - 2);i>=0;i--){
    aux = 0.0; // Inicializamos para cada i
    Rcii = 0.0; // Inicializamos para cada i
    for(k=0;k<infoRc->M && Rcii==0.0;k++){
      // Si R(k) = i y C(k) = i lo hemos encontrado:
      if(infoRc->R[k] == i && infoRc->C[k] == i){
        Rcii = infoRc->Z[k]; // Guardamos el valor encontrado
      }
    }
    for(j=i+1;j<(infoRc->Nr - 1);j++){
      // Localizamos Rc(j,i) y Rc(i,i) - recorremos infoRc:
      Rcji = 0.0; // Inicializamos para cada j
      for(k=0;k<infoRc->M && Rcji==0.0;k++){
        // Si R(k) = j y C(k) = i lo hemos encontrado:
        if(infoRc->R[k] == j && infoRc->C[k] == i){
          Rcji = infoRc->Z[k]; // Guardamos el valor encontrado
        }
      }
      aux = aux + Rcji * x[j];
    }
    // Si no vamos a dividir por cero:
    if(Rcii>0.0 || Rcii<0.0){
      x[i] = ( y[i] - aux ) / Rcii;
    }
  }
  //---------------------------------------------------------------------------
  // Liberamos espacio reservado:
  free(y);
  //---------------------------------------------------------------------------
  // Si el programa principal termina correctamente devuelve 0:
  return 0;
}
```

A10. Suma/resta de matrices *sparse*

Como herramienta para facilitar los cálculos, se ha creado una función, llamada *sumres_sparse*, que permite sumar o restar dos matrices *sparse* empleando sus respectivas codificaciones directamente.

El código correspondiente a esta función es el siguiente:

```
// Función para sumar/restar dos matrices sparse:
int sumres_sparse(struct datos* infoA, struct datos* infoB,
                  struct datos* infoC,int modo){
  // Comprobamos que A y B tengan las mismas dimensiones para poder sumar:
  if(infoA->Nr == infoB->Nr && infoA->Nc == infoB->Nc){
    // Índice:
    int k;
    // Rellenamos y reservamos memoria:
    infoC->Nr = infoA->Nr;
    infoC->Nc = infoA->Nc;
    infoC->Nd = 0;
    infoC->M = infoA->M + infoB->M;
    infoC->R = (int*)malloc(infoC->M*sizeof(int));
    infoC->C = (int*)malloc(infoC->M*sizeof(int));
    infoC->Z = (double*)malloc(infoC->M*sizeof(double));
    // Copiamos R,C,Z de A en C y luego B en C detrás, ampliamos la
    codificación:
    for(k=0;k<infoC->M;k++){
      // Copiamos A en C primero:
      if(k<infoA->M){
        infoC->R[k] = infoA->R[k];
        infoC->C[k] = infoA->C[k];
        infoC->Z[k] = infoA->Z[k];
      }
      // Copiamos B en C a continuación:
      if(k>=infoA->M){
        infoC->R[k] = infoB->R[k-(infoA->M)];
        infoC->C[k] = infoB->C[k-(infoA->M)];
        // Si queremos que C = A + B (modo 1 - suma):
        if(modo == 1){
          // Dejamos el signo de B como esté:
          infoC->Z[k] = infoB->Z[k-(infoA->M)];
        }
        // Si queremos que C = A - B (modo 2 - resta):
        if(modo == 2){
          // Cambiamos el signo de B:
          infoC->Z[k] = -(infoB->Z[k-(infoA->M)]);
        }
      }
    }
    // Si la función termina correctamente devuelve un cero:
    return 0;
  }
```

```
  // Si las dimensiones no coinciden:
  else{
    printf("\n\tsumres_sparse - Las dimensiones de las matrices no
           coinciden.\n\tNo se puede realizar la operación.\n");
    // Si la función termina por este motivo devuelve un uno:
    return 1;
  }
}
```

Cabe decir que, tras el uso de esta función, es convenienete emplear *posprocesador_sparse* para simplificar la codificación resultante.

A11. Calcular la traspuesta *sparse* de una matriz

Para poder llevar a cabo operaciones más complejas, es necesario trasponer matrices *sparse* como paso previo a los cálculos, para lo cual se ha definido una función, llamada *traspuesta_sparse*, que traspone la codificación de la matriz dato.

El código correspondiente a esta función es el siguiente:

```
// Función que permite trasponer una matriz sparse codificada simple (usar
previamente conversor_sparse si no es así):
int traspuesta_sparse(struct datos* infoA, struct datos* infoAt){
  // Rellenamos la codificación de la traspuesta:
  infoAt->Nr = infoA->Nc;
  infoAt->Nc = infoA->Nr;
  infoAt->Nd = 0;
  infoAt->M = infoA->M;
  // Invertimos los vectores R,C para trasponer la matriz:
  infoAt->R = infoA->C;
  infoAt->C = infoA->R;
  infoAt->Z = infoA->Z;
  // NOTA: como no hemos reservado más memoria, la codificación de Vt usa el
  mismo espacio de V, comparten modificaciones además
  // Si la función termina correctamente devuelve 0:
  return 0;
}
```

A12. Programas principales

Para ilustrar el modo de usar las funciones desarrolladas en este proyecto, expondremos diversos programas principales (*main*) que permiten implementar varias aplicaciones.

El primero de ellos es *main_chol*, que muestra un ejemplo sencillo de cómo cargar datos en nuestro programa (matriz A), procesarlos, calcular su descomposición de Cholesky y resolver $Ax = b$ suponiendo que se trata de un sistema *sparse*:

```
//Librería necesarias:
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//-----------------------------------------------------------------------------
// Declaración de la estructura de datos codificados para las matrices sparse
necesarias definida fuera del main para que sea global y no dé problemas
siendo argumento de funciones externas a main)
struct datos{
  int Nr;     // Número de filas (rows)
  int Nc;     // Número de columnas (columns)
  int Nd;     // Número de diadas
  int M;      // Tamaño de datos codificados
  int* R;     // Vector de filas (rows)
  int* C;     // Vector de columnas (columns)
  double* Z;  // Vector de valores no nulos
};
//-----------------------------------------------------------------------------
// Funciones necesarias:
#include "descodificador_sparse.c"
#include "multMxV_sparse.c"
#include "cholesky_v1.c"
#include "cholesky_v2.c"
#include "cholesky_sparse_v1.c"
#include "cholesky_sparse_v2.c"
#include "multiMxM_sparse.c"
#include "sumres_sparse.c"
#include "conversor_sparse.c"
#include "posprocesador_sparse.c"
#include "sistecchol_sparse.c"
#include "lectordatos_sparse.c"
#include "traspuesta_sparse.c"
#include "semibanda_sparse.c"
//-----------------------------------------------------------------------------
// Programa principal:
int main(){
  struct datos infoA, infoASimple, infoAPos, infoRc, infoRcPos;
  int k, res;
  puts("\n ------------------------------------------------------ main");
```

```
// Rellenamos infoA con la codificación:
infoA.Nr = 10;
infoA.Nc = infoA.Nr;
infoA.Nd = 0;
infoA.M = infoA.Nr*infoA.Nr;
infoA.R = (int*)malloc(infoA.M*sizeof(int));
infoA.C = (int*)malloc(infoA.M*sizeof(int));
infoA.Z = (double*)malloc(infoA.M*sizeof(double));
for(k=0;k<infoA.M;k++){
  infoA.R[k] = 0;
  infoA.C[k] = 0;
  infoA.Z[k] = 0.0;
}
// Vectores R,C,Z:
// Importamos datos del fichero externo llamando a la función lectordatos:
puts("\n ------------------------------------------------ lectordatos");
res = lectordatos_sparse(&infoA,"matriz_A.txt");
puts("\n ------------------------------------------------------- main");
// Llamada a la función de descodificación:
printf("\nMatriz A:");
puts("\n ---------------------------------------------- descodificador");
res = descodificador_sparse(&infoA);
puts("\n ------------------------------------------------------- main");
//-------------------------------------------------------------------------
// Convertimos infoA en infoASimple:
puts("\nConvertimos infoA en infoASimple:");
puts("\n ------------------------------------------- conversor_sparse");
res = conversor_sparse(&infoA,&infoASimple);
puts("\n ------------------------------------------------------- main");
// Llamada a la función de descodificación:
printf("\nMatriz A simple:");
puts("\n ---------------------------------------------- descodificador");
res = descodificador_sparse(&infoASimple);
puts("\n ------------------------------------------------------- main");
//-------------------------------------------------------------------------
// Posprocesado de ASimple:
puts("\n ---------------------------------------- posprocesador_sparse");
res = posprocesador_sparse(&infoASimple,&infoAPos);
puts("\n ------------------------------------------------------- main");
// Llamada a la función de descodificación:
printf("\nMatriz A simple posprocesada:");
puts("\n ---------------------------------------------- descodificador");
res = descodificador_sparse(&infoAPos);
puts("\n ------------------------------------------------------- main");
//-------------------------------------------------------------------------
// Realizamos la descomposición de Cholesky de la matriz posprocesada
(suponemos que es definida positiva):
printf("\nDescomposicion de Cholesky sparse:");
puts("\n ---------------------------------------- cholesky_sparse_2");
res = cholesky_sparse_v2(&infoAPos,&infoRc);
puts("\n ------------------------------------------------------- main");
// Llamada a la función de descodificación:
printf("\nMatriz Rc:");
puts("\n ---------------------------------------------- descodificador");
res = descodificador_sparse(&infoRc);
```

```
  // Posprocesado de Rc:
  puts("\n ---------------------------------------- posprocesador_sparse");
  res = posprocesador_sparse(&infoRc,&infoRcPos);
  puts("\n ------------------------------------------------------------ main");
  //---------------------------------------------------------------------------
  // Definimos vector b del sistema Rc*x = b:
  double* b = (double*)malloc(infoRc.Nr*sizeof(double));
  printf("\nVector B:");
  for(k=0;k<infoRc.Nr;k++){
    B[k] = 1;
    printf("\n\tB[%d] = %f",k,B[k]);
  }printf("\n");
  //---------------------------------------------------------------------------
  // Definimos vector x del sistema Rc*x = b:
  double* x = (double*)malloc(infoRc.Nr*sizeof(double));
  //printf("\nInicializamos el vector x:");
  for(k=0;k<infoRc.Nr;k++){
    x[k] = 0.0;
  }
  //---------------------------------------------------------------------------
  // Resolvemos el sitema Rc*x = b usando sistecchol_sparse:
  printf("\nResolvemos sistema Rc*x = B:");
  puts("\n ------------------------------------- sistecchol_sparse");
  res = sistecchol_sparse(&infoRcPos,x,B);
  puts("\n ------------------------------------------------------ main");
  // Mostramos por pantalla los resultados:
  puts("\nEl vector x resultante es:");
  for(k=0;k<infoB.Nr;k++) printf("\n\tx[%d] = %f",k,x[k]);printf("\n");
  //---------------------------------------------------------------------------
  //Liberamos espacio reservado:
  free(infoA.R); free(infoA.C); free(infoA.Z);
  free(infoASimple.R); free(infoASimple.C); free(infoASimple.Z);
  free(infoAPos.R); free(infoAPos.C); free(infoAPos.Z);
  free(infoRc.R); free(infoRc.C); free(infoRc.Z);
  free(infoRcPos.R); free(infoRcPos.C); free(infoRcPos.Z);
  free(B); free(x);
  //---------------------------------------------------------------------------
  // Si el programa principal termina correctamente devuelve 0:
  return 0;
}
```

Durante su ejecución, el código va escribiendo por pantalla diversa información que permite seguir los pasos de la ejecución del programa.

El segundo de los mismos es *main_chol_tiempos*, que implementa el mismo proceso anterior, pero en vez de mostrar los pasos por pantalla, mide los tiempos de ejecución de cada función relevante y los muestra por pantalla.

```
//Librería necesarias:
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
//-----------------------------------------------------------------------------
// Declaración de la estructura de datos codificados para las matrices sparse necesarias
// (definida fuera del main para que sea global y no dé problemas siendo argumento de funciones externas a main)
struct datos{
  int Nr;     // Número de filas (rows)
  int Nc;     // Número de columnas (columns)
  int Nd;     // Número de diadas
  int M;      // Tamaño de datos codificados
  int* R;     // Vector de filas (rows)
  int* C;     // Vector de columnas (columns)
  double* Z;  // Vector de valores no nulos
};
//-----------------------------------------------------------------------------
// Funciones necesarias:
#include "descodificador_sparse.c"
#include "multMxV_sparse.c"
#include "cholesky_v1.c"
#include "cholesky_v2.c"
#include "cholesky_sparse_v1.c"
#include "cholesky_sparse_v2.c"
#include "multiMxM_sparse.c"
#include "sumres_sparse.c"
#include "conversor_sparse.c"
#include "posprocesador_sparse.c"
#include "sistecchol_sparse.c"
#include "lectordatos_sparse.c"
#include "traspuesta_sparse.c"
#include "semibanda_sparse.c"
//-----------------------------------------------------------------------------
// Programa principal:
int main(){
  struct datos infoA, infoASimple, infoAPos, infoRc, infoRcPos;
  int k, res;
  int flops = 100;
  clock_t t_ini, t_fin;  // Para medir el tiempo
  double tiempo;  // Para pasar el tiempo a segundos
  // Rellenamos infoA con la codificación:
  infoA.Nr = 10;
  infoA.Nc = infoA.Nr;
  infoA.Nd = 0;
  infoA.M = infoA.Nr*infoA.Nc;
  infoA.R = (int*)malloc(infoA.M*sizeof(int));
```

```
infoA.C = (int*)malloc(infoA.M*sizeof(int));
infoA.Z = (double*)malloc(infoA.M*sizeof(double));
for(k=0;k<infoA.M;k++){
  infoA.R[k] = 0;
  infoA.C[k] = 0;
  infoA.Z[k] = 0.0;
}
// Vectores R,C,Z:
// Importamos datos del fichero externo llamando a la función lectordatos:
t_ini = clock();  // Tomamos el tiempo inicial
for(k=0;k<flops;k++){
  res = lectordatos_sparse(&infoA,"matriz_A.txt");
}
t_fin = clock();  // Tomamos el tiempo final
// Calculamos el tiempo transcurrido:
tiempo = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
tiempo = tiempo / flops;  // Hacemos la media por flop
// Convertimos a segundos y mostramos por pantalla:
printf("\n\nlectordatos_sparse: %f segundos\n", tiempo);
//---------------------------------------------------------------------
// Convertimos infoA en infoASimple:
res = conversor_sparse(&infoA,&infoASimple);
//---------------------------------------------------------------------
// Posprocesado de ASimple:
res = posprocesador_sparse(&infoASimple,&infoAPos);
//---------------------------------------------------------------------
// Realizamos la descomposición de Cholesky de la matriz posprocesada
(suponemos que es definida positiva):
t_ini = clock();  // Tomamos el tiempo inicial
for(k=0;k<flops;k++){
  res = cholesky_sparse_v1(&infoAPos,&infoRc);
}
t_fin = clock();  // Tomamos el tiempo final
// Calculamos el tiempo transcurrido:
tiempo = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
tiempo = tiempo / flops;  // Hacemos la media por flop
// Convertimos a segundos y mostramos por pantalla:
printf("\n\ncholesky sparse 1: %f segundos\n", tiempo);
//---------------------------------------------------------------------
t_ini = clock();  // Tomamos el tiempo inicial
for(k=0;k<flops;k++){
  res = cholesky_sparse_v2(&infoAPos,&infoRc);
}
t_fin = clock();  // Tomamos el tiempo final
// Calculamos el tiempo transcurrido:
tiempo = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
tiempo = tiempo / flops;  // Hacemos la media por flop
// Convertimos a segundos y mostramos por pantalla:
printf("\n\ncholesky sparse 2: %f segundos\n", tiempo);
//---------------------------------------------------------------------
// Posprocesado de Rc:
res = posprocesador_sparse(&infoRc,&infoRcPos);
//---------------------------------------------------------------------
// Definimos vector B del sistema Rc*x = b:
```

```
  double* b = (double*)malloc(infoRc.Nr*sizeof(double));
  for(k=0;k<infoRc.Nr;k++){
    b[k] = 1;
  }
  //---------------------------------------------------------------------------
  // Definimos vector x del sistema Rc*x = b:
  double* x = (double*)malloc(infoRc.Nr*sizeof(double));
  for(k=0;k<infoRc.Nr;k++){
    x[k] = 0.0;
  }
  //---------------------------------------------------------------------------
  // Resolvemos el sitema Rc*x = b usando sistecchol_sparse:
  t_ini = clock();  // Tomamos el tiempo inicial
  for(k=0;k<flops;k++){
    res = sistecchol_sparse(&infoRcPos,x,b);
  }
  t_fin = clock();  // Tomamos el tiempo final
  // Calculamos el tiempo transcurrido:
  tiempo = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
  tiempo = tiempo / flops;  // Hacemos la media por flop
  // Convertimos a segundos y mostramos por pantalla:
  printf("\n\nsistecchol_sparse: %f segundos\n", tiempo);
  //---------------------------------------------------------------------------
  //Liberamos espacio reservado:
  free(infoA.R); free(infoA.C); free(infoA.Z);
  free(infoASimple.R); free(infoASimple.C); free(infoASimple.Z);
  free(infoAPos.R); free(infoAPos.C); free(infoAPos.Z);
  free(infoRc.R); free(infoRc.C); free(infoRc.Z);
  free(infoRcPos.R); free(infoRcPos.C); free(infoRcPos.Z);
  free(b); free(x);
  //---------------------------------------------------------------------------
  // Si el programa principal termina correctamente devuelve 0:
  return 0;
}
```

El tercer programa es *main_semibanda*, que implementa el método de resolución de sistemas de ecuaciones en semi banda de manera directa, sin el uso de la función *semibanda_sparse*. Esto es así, dado que el es código precursor a dicha función, que nos sirve de ejemplo para mostrar las operaciones complejas que pueden realizarse con nuestras funciones:

```
//Librería necesarias:
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//-----------------------------------------------------------------------
// Declaración de la estructura de datos codificados para las matrices sparse
necesarias (definida fuera del main para que sea global y no dé problemas
siendo argumento de funciones externas a main)
struct datos{
  int Nr;     // Número de filas (rows)
  int Nc;     // Número de columnas (columns)
  int Nd;     // Número de diadas
  int M;      // Tamaño de datos codificados
  int* R;     // Vector de filas (rows)
  int* C;     // Vector de columnas (columns)
  double* Z;  // Vector de valores no nulos
};
//-----------------------------------------------------------------------
// Funciones necesarias:
#include "descodificador_sparse.c"
#include "multMxV_sparse.c"
#include "cholesky_v1.c"
#include "cholesky_v2.c"
#include "cholesky_sparse_v1.c"
#include "cholesky_sparse_v2.c"
#include "multiMxM_sparse.c"
#include "sumres_sparse.c"
#include "conversor_sparse.c"
#include "posprocesador_sparse.c"
#include "sistecchol_sparse.c"
#include "lectordatos_sparse.c"
#include "traspuesta_sparse.c"
#include "semibanda_sparse.c"
//-----------------------------------------------------------------------
// Programa principal:
int main(){
  // Definimos las estructuras de datos a usar:
  struct datos infoB;
  struct datos infoBSimple;
  struct datos infoBPos;
  struct datos infoRcB;
  struct datos infoRcBPos;
  //-----------------------------------------------------------------------
  struct datos infoU;
  struct datos infoV;
  struct datos infoVt;
  //-----------------------------------------------------------------------
```

```
struct datos infoC;
struct datos infoVtC;
struct datos infoVtCPos;
//---------------------------------------------------------------------------
struct datos infoI;
//---------------------------------------------------------------------------
struct datos infoA;
struct datos infoAPos;
struct datos infoRcA;
struct datos infoRcAPos;
//---------------------------------------------------------------------------
int k, res, i, j;
int n = 10;  // Tamaño del sistema
int nd = 2;  // Número de diadas
//---------------------------------------------------------------------------
// Rellenamos infoB con la codificación:
infoB.Nr = n;
infoB.Nc = infoB.Nr;
infoB.Nd = 0;
infoB.M = infoB.Nr*infoB.Nc;
infoB.R = (int*)malloc(infoB.M*sizeof(int));
infoB.C = (int*)malloc(infoB.M*sizeof(int));
infoB.Z = (double*)malloc(infoB.M*sizeof(double));
for(k=0;k<infoB.M;k++){
  infoB.R[k] = 0;
  infoB.C[k] = 0;
  infoB.Z[k] = 0.0;
}
// Importamos datos del fichero externo llamando a la función lectordatos:
res = lectordatos_sparse(&infoB,"matriz_B.txt");
//---------------------------------------------------------------------------
// Rellenamos infoU con la codificación:
infoU.Nr = infoB.Nr;
infoU.Nc = nd;
infoU.Nd = 0;
infoU.M = infoU.Nr*infoU.Nc;
infoU.R = (int*)malloc(infoU.M*sizeof(int));
infoU.C = (int*)malloc(infoU.M*sizeof(int));
infoU.Z = (double*)malloc(infoU.M*sizeof(double));
for(k=0;k<infoU.M;k++){
  infoU.R[k] = 0;
  infoU.C[k] = 0;
  infoU.Z[k] = 0.0;
}
// Importamos datos del fichero externo llamando a la función lectordatos:
res = lectordatos_sparse(&infoU,"matriz_U.txt");
//---------------------------------------------------------------------------
// Rellenamos infoV con la codificación:
infoV.Nr = infoB.Nr;
infoV.Nc = nd;
infoV.Nd = 0;
infoV.M = infoV.Nr*infoV.Nc;
infoV.R = (int*)malloc(infoV.M*sizeof(int));
infoV.C = (int*)malloc(infoV.M*sizeof(int));
infoV.Z = (double*)malloc(infoV.M*sizeof(double));
```

```
for(k=0;k<infoV.M;k++){
  infoV.R[k] = 0;
  infoV.C[k] = 0;
  infoV.Z[k] = 0.0;
}
// Importamos datos del fichero externo llamando a la función lectordatos:
res = lectordatos_sparse(&infoV,"matriz_V.txt");
//---------------------------------------------------------------------------
// Tenemos que obtener V':
res = traspuesta_sparse(&infoV,&infoVt);
//---------------------------------------------------------------------------
// Ahora necesitamos generar una matriz identidad de tamaño nd x nd:
// NOTA: la generamos directamente en codificación simple, en lugar de usar
una flecha, para no tener que usar conversor_sparse
infoI.Nr = nd;
infoI.Nc = nd;
infoI.Nd = 0;
infoI.M = nd;
infoI.R = (int*)malloc(infoI.M*sizeof(int));
infoI.C = (int*)malloc(infoI.M*sizeof(int));
infoI.Z = (double*)malloc(infoI.M*sizeof(double));
for(k=0;k<infoI.M;k++){
  infoI.R[k] = k;
  infoI.C[k] = k;
  infoI.Z[k] = 1.0;
}
//---------------------------------------------------------------------------
// Definimos los vectores z1, z3, x, y h:
double* z1 = (double*)malloc(infoB.Nr*sizeof(double));
double* z3 = (double*)malloc(infoB.Nr*sizeof(double));
double* x = (double*)malloc(infoB.Nr*sizeof(double));
double* h = (double*)malloc(infoB.Nr*sizeof(double));
// Inicializamos los vectores z1, z3, x, xd, y h:
for(k=0;k<infoB.Nr;k++){
  z1[k] = 0.0;
  z3[k] = 0.0;
  x[k] = 0.0;
  h[k] = 1;
}
//---------------------------------------------------------------------------
// Definimos los vectores z2 y z2_aux:
double* z2 = (double*)malloc(infoU.Nc*sizeof(double));
for(k=0;k<infoU.Nc;k++){
  z2[k] = 0.0;
}
//---------------------------------------------------------------------------
// Tenemos que resolver 3 sistemas de ecuaciones:
//
//    B*z1 = h    ;    (I+V'*(B^-1)*U)*z2 = (V'*z1)    ;    B*z3 = (U*z2)
//
//---------------------------------------------------------------------------
// Hacemos la descomposición de Cholesky de B para resolver los sitemas
res = conversor_sparse(&infoB,&infoBSimple);
res = posprocesador_sparse(&infoBSimple,&infoBPos);
```

```
res = cholesky_sparse_v2(&infoBPos,&infoRcB);
// Posprocesamos RcB para eliminar posibles ceros en la codificación:
res = posprocesador_sparse(&infoRcB,&infoRcBPos);
//---------------------------------------------------------------------
// Resolvemos el sitema B*z1 = h usando Cholesky --> (RcB*RcB')*z1 = h -->
// RcB*(RcB'*z1) = h --> RcB*z1_aux = h --> z1_aux = RcB\h --> RcB'*z1 =
// z1_aux --> z1 = RcB'\z1_aux   para lo cual usamos sistecchol_sparse:
//---------------------------------------------------------------------
// Resolvemos sistema B*z1 = h:
res = sistecchol_sparse(&infoRcBPos,z1,h);
//---------------------------------------------------------------------
// Definimos el vector Vtz1:
double* Vtz1 = (double*)malloc(infoVt.Nr*sizeof(double));
// Inicializamos el vector a cero:
for(k=0;k<infoVt.Nr;k++){
  Vtz1[k] = 0.0;
}
//---------------------------------------------------------------------
// Definimos los vectores Cj y Uj::
double* Cj = (double*)malloc(infoU.Nr*sizeof(double));
double* Uj = (double*)malloc(infoU.Nr*sizeof(double));
// Y los inicializamos a cero:
for(k=0;k<infoU.Nr;k++){
  Cj[k] = 0.0;
  Uj[k] = 0.0;
}
//---------------------------------------------------------------------
// Rellenamos la codificación de C:
infoC.Nr = infoU.Nr;
infoC.Nc = infoU.Nc;
infoC.Nd = 0;
infoC.M = infoC.Nr*infoC.Nc;  // Reservamos por exceso
infoC.R = (int*)malloc(infoC.M*sizeof(int));
infoC.C = (int*)malloc(infoC.M*sizeof(int));
infoC.Z = (double*)malloc(infoC.M*sizeof(double));
for(k=0;k<infoC.M;k++){
  infoC.R[k] = 0;
  infoC.C[k] = 0;
  infoC.Z[k] = 0.0;
}
//---------------------------------------------------------------------
// Resolvemos el sitema (I+V'*(B^-1)*U)*z2 = (V'*z1) usando
sistecchol_sparse:
// Necesitamos calcular (I+V'*(B^-1)*U) y (V'*z1) antes de resolver:
// Obtenemos Vtz1 = (V'*z1):
Vtz1 = multMxV_sparse(&infoVt,z1,1);
//---------------------------------------------------------------------
// Obtenemos A = (I+V'*(B^-1)*U), por lo que emplearemos que C = (B^-1)*U
// de tal manera que B*C = B*(B^-1)*U = U, que puede divirse en Ui = B*Ci
// (tantos subsistemas vector = matriz*vector como columnas (diadas) tenga
// U), que a su vez pueden resolverse empleando la descomposición de
// Cholesky de B. Para ello tengo que extraer las columnas de U, calcular
// los vectores Ci, con ellos montar la matriz C.
//---------------------------------------------------------------------
i = 0; // Inicializamos a cero el índice para rellenar infoC
```

```
// Nos movemos por columnas:
for(j=0;j<infoU.Nc;j++){
  // Ponemos a cero los vectores Cj y Uj:
  for(k=0;k<infoU.Nr;k++){
    Cj[k] = 0.0;
    Uj[k] = 0.0;
  }
  // Recorremos la codificación de U:
  for(k=0;k<infoU.M;k++){
    // Si encontramos elementos con C = j:
    if(infoU.C[k] == j){
      // Guardamos en la posición R el vector columna Uj el valor Z
      asociado a C:
      Uj[ infoU.R[k] ] = infoU.Z[k];
    }
  }
  // Una vez hemos extraído el vector Uj, resolvemos el subsistema B*Cj =
  Uj para obtener Cj:
  res = sistecchol_sparse(&infoRcB,Cj,Uj);
  // Copiamos el contenido de Cj en la codificación de C:
  for(k=0;k<infoU.Nr;k++){
    // Si el elemento k de Cj es igual a 0.0:
    if(Cj[k] == 0.0){
      // No lo copiamos
    }
    // Pero si es distinto de cero sí lo almacenamos:
    else{
      infoC.R[i] = k;
      infoC.C[i] = j;
      infoC.Z[i] = Cj[k];
      i = i + 1;
    }
  }
}
// Una vez hemos calculado C, acualizamos el tamaño de la tabla de datos:
infoC.M = i;
//-------------------------------------------------------------------------
// Ahora determinamos VtC = V'*C empleando multiMxM_sparse:
res = multiMxM_sparse(&infoVt,&infoC,&infoVtC);
// Posprocesamos VtC para simplificar y eliminar posibles ceros en la
codificación:
res = posprocesador_sparse(&infoVtC,&infoVtCPos);
// Ahora sumamons A = I + VtC con sumres_sparse:
res = sumres_sparse(&infoI,&infoVtC,&infoA,1);
// Posprocesamos A para simplificar y eliminar posibles ceros:
res = posprocesador_sparse(&infoA,&infoAPos);
// Calculamos la Cholesky de A:
res = cholesky_sparse_v2(&infoAPos,&infoRcA);
// Posprocesamos RcA para eliminar posibles ceros en la codificación:
res = posprocesador_sparse(&infoRcA,&infoRcAPos);
//-------------------------------------------------------------------------
// Resolvemos sistema A*z2 = Vtz1:
res = sistecchol_sparse(&infoRcAPos,z2,Vtz1);
//-------------------------------------------------------------------------
```

```
  // Resolvemos el sitema B*z3 = (U*z2) usando sistecchol_sparse:
  // Definimos el vector Uz2:
  double* Uz2 = (double*)malloc(infoU.Nr*sizeof(double));
  // Inicializamos el vector a cero:
  for(k=0;k<infoU.Nr;k++) Uz2[k] = 0.0;
  //---------------------------------------------------------------------------
  // Calculamos Uz2 = U*z2:
  Uz2 = multMxV_sparse(&infoU,z2,1);
  // Resolvemos sistema B*z3 = Uz2:
  res = sistecchol_sparse(&infoRcB,z3,Uz2);
  //---------------------------------------------------------------------------
  // Obtenemos x como z1 - z3:
  for(k=0;k<infoB.Nr;k++){
    x[k] = z1[k] - z3[k];
  }
  // Mostramos por pantalla los resultados:
  puts("\nEl vector x resultante es:");
  for(k=0;k<infoB.Nr;k++) printf("\n\tx[%d] = %f",k,x[k]);printf("\n");
  //---------------------------------------------------------------------------
  // Liberamos espacio reservado:
  free(infoB.R); free(infoB.C); free(infoB.Z);
  free(infoBSimple.R); free(infoBSimple.C); free(infoBSimple.Z);
  free(infoBPos.R); free(infoBPos.C); free(infoBPos.Z);
  free(infoRcB.R); free(infoRcB.C); free(infoRcB.Z);
  free(infoRcBPos.R); free(infoRcBPos.C); free(infoRcBPos.Z);
  free(infoU.R); free(infoU.C); free(infoU.Z);
  free(infoV.R); free(infoV.C); free(infoV.Z);
  // NOTA: no hace falta liberar memoria de Vt porque ya está almacenada en V
  (ver función traspuesta_sparse)
  //free(infoVt.R); free(infoVt.C); free(infoVt.Z);
  free(infoC.R); free(infoC.C); free(infoC.Z);
  free(infoVtC.R); free(infoVtC.C); free(infoVtC.Z);
  free(infoVtCPos.R); free(infoVtCPos.C); free(infoVtCPos.Z);
  free(infoI.R); free(infoI.C); free(infoI.Z);
  free(infoA.R); free(infoA.C); free(infoA.Z);
  free(infoAPos.R); free(infoAPos.C); free(infoAPos.Z);
  free(infoRcA.R); free(infoRcA.C); free(infoRcA.Z);
  free(infoRcAPos.R); free(infoRcAPos.C); free(infoRcAPos.Z);
  free(z1); free(z3); free(x); free(h); free(z2);
  free(Vtz1); free(Cj); free(Uj);
  //---------------------------------------------------------------------------
  // Si el programa principal termina correctamente devuelve 0:
  return 0;
}
```

Por último, el cuarto código, mucho más escueto que el anterior, *main_semibanda_tiempos*, nos muestra el modo en el que se debe emplear la función *semibanda_sparse*, al mismo tiempo que mide y muestra por pantalla el tiempo de ejecución de la misma:

```
//Librería necesarias:
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
//-----------------------------------------------------------------------------
// Declaración de la estructura de datos codificados para las matrices sparse
necesarias definida fuera del main para que sea global y no dé problemas
siendo argumento de funciones externas a main)
struct datos{
  int Nr;     // Número de filas (rows)
  int Nc;     // Número de columnas (columns)
  int Nd;     // Número de diadas
  int M;      // Tamaño de datos codificados
  int* R;     // Vector de filas (rows)
  int* C;     // Vector de columnas (columns)
  double* Z;  // Vector de valores no nulos
};
//-----------------------------------------------------------------------------
// Funciones necesarias:
#include "descodificador_sparse.c"
#include "multMxV_sparse.c"
#include "cholesky_v1.c"
#include "cholesky_v2.c"
#include "cholesky_sparse_v1.c"
#include "cholesky_sparse_v2.c"
#include "multiMxM_sparse.c"
#include "sumres_sparse.c"
#include "conversor_sparse.c"
#include "posprocesador_sparse.c"
#include "sistecchol_sparse.c"
#include "lectordatos_sparse.c"
#include "traspuesta_sparse.c"
#include "semibanda_sparse.c"
//-----------------------------------------------------------------------------
// Programa principal:
int main(){
  int k, res;
  clock_t t_ini, t_fin;       // Para medir el tiempo
  double tiempo;      // Para pasar el tiempo a segundos
  int flops = 100;  // Número de repeticiones
  int n = 10;  // Tamaño del sistema
  int nd = 2; // Número de diadas
  //-----------------------------------------------------------------------------
  // Medimos el tiempo en calcular x:
  t_ini = clock();  // Tomamos el tiempo inicial
  for(k=0;k<flops;k++){
    res = semibanda_sparse(n,nd);
```

```
    }
    t_fin = clock();  // Tomamos el tiempo final
    // Calculamos el tiempo transcurrido:
    tiempo = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
    tiempo = tiempo / flops;  // Hacemos la media por flop
    // Mostramos por pantalla el tiempo:
    printf("\n\nTiempo en calcular x: %f segundos\n", tiempo);
    //------------------------------------------------------------------------
    // Si el programa principal termina correctamente devuelve 0:
    return 0;
}
```

BIBLIOGRAFÍA

[1] Álamo Cantarero, Teodoro «Semibanda convex optimization» Departamento de Ingeniería de Sistemas y Automática, Universidad de Sevilla, Escuela Técnica Superior de Ingeniería, 2017.

[2] Boyd, Stephen; Vandenberghe, Lieven «Convex optimization» Department of Electrical Engineering, Stanford University & University of California, Los Angeles, 2009.

[3] Davis, Timothy A. «Direct Methods for *Sparse* Linear Systems» Fundamentals of Algorithms, Ed. Siam, 2006.

[4] Gilbert, John R.; Molery, Cleve; Schreiberz, Robert «*Sparse* matrices in matlab: design and implementation» (http://www.mathworks.com/help/pdf_doc/otherdocs/simax.pdf).

[5] Jaramillo, J. D., A. V. Macıa, F. C. Zabala «Métodos directos para la solución de sistemas de ecuaciones lineales simétricos, indefinidos, dispersos y de gran dimensión » Universidad Eafit, 2006.

[6] Mejía Soto, José; Solarte Martínez, Guillermo Roberto; Muñoz Guerrero, Luis Eduardo «Matrices *sparse* descripcion y aplicaciones» Universidad Tecnológica de Pereira, 2013.

[7] Moltó Román, José Enrique «Estructuras de datos para Matrices *Sparse*» Universitat Politècnica de València. Escola Tècnica Superior d'Enginyeria Informàtica, 2008.

[8] Ojeda, Ignacio; Gago, Jesús «Métodos matemáticos para estadística» Manuales UEX 58 - Espacio Europeo Edcación Superior (E.E.E.S.), 2008.

[9] Tinetti, Fernando G.; Denhamy, Mónica «Paralelización de la Factorización LU de Matrices en Clusters Heterogéneos» Facultad de Informática de la Universidad Nacional de La Plata, 2003.