Algebra Lineal Numérica

Laboratorio 2

Análisis de Matlab

Integrantes

Nombre	CI	Correo
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se detallan los puntos pedidos en la consigna de la **Tarea 2**. A continuación, se destacan ciertos aspectos a tener en cuenta del análisis y de los archivos entregables:

1. La implementación de los algoritmos se realizó empleando el lenguaje y entorno de **Matlab**. Ejecutando el mismo sobre Windows 10.

2. Las especificaciones de la plataforma de ejecución son **16GB** de RAM DDR4 y un procesador **AMD Ryzen 5 3600**, cuyas especificaciones pueden ser consultadas en el siguiente enlace. De las mismas se destaca lo siguiente:
   - **Cores:** 6 (12 threads)
   - **CPU max MHz:** 3600 MHz
   - **L1i Cache:** 192 KB (6 x 32 KB)
   - **L1d Cache:** 192 KB (6 x 32 KB)
   - **L2 Cache:** 3 MB (6 x 512 KB)
   - **L3 Cache:** 32 MB (2 x 16 MB)
   - **Cache Latency:** 4 (L1); 12 (L2); 40 (L3)

3. Los datos manipulados fueron matrices compuestos por números decimales representados en punto flotante de precisión doble (*double* de **Matlab**). Estas se inicializaron aleatoriamente.

4. Los tiempos registrados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, para medir los mismos se empleó la herramienta de Matlab timeit, la cual realiza un benchkmark para una función, ejecutando múltiples veces la misma y computando su tiempo promedio de ejecución.

5. Debido a trabajar con matrices y vectores aleatorios los resultados capturados (Como los vectores b de la parte b II) al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, sin embargo su variación osciló con un radio lo suficientemente bajo como para que las observaciones y conclusiones presentadas se preserven.

2. Desarrollo

2.1. Ejercicio 1 - Manejo de Matrices

2.1.1. Código Implementado

La implementación pedida para la multiplicación de matrices se proporciona en el archivo `a_mat_mul.m`. Se implementaron las siguientes funciones:

- `matlab_mult`: Multiplicación de matrices empleando la multiplicación estándar de Matlab.
- `matlab_mult`: Multiplicación de matrices iterando en la matriz resultante y aplicando multiplicaciones vectoriales.
- `matlab_mult`: Multiplicación de matrices a nivel de coeficientes individuales (Algoritmo de multiplicación matricial tradicional). Se destaca del mismo que multiplica recorriendo ambas matrices por sus filas.

Se destaca que el código utilizado para evaluar dichas implementaciones, también se encuentra en el archivo anteriormente mencionado.

2.1.2. Experimentación y Resultados

Tras efectuar la experimentación se registraron los siguientes tiempos de ejecución:

Algoritmo	Tamaño Matriz	Tiempo (s)
Matlab	2048×2048	0.0965
Vectores	2048×2048	125.8586
Coeficientes	2048×2048	127.5896
Matlab	4096×4096	0.6379
Vectores	4096×4096	894.8913
Coeficientes	4096×4096	1025.4000
Matlab	8192×8192	4.3552
Vectores	8192×8192	5935.6000
Coeficientes	8192×8192	13772.0000

2.1.3. Observaciones y Conclusiones

Teniendo en cuenta los resultados capturados, se pueden realizar las siguientes observaciones:

- El tiempo registrado utilizando el operador ***** de Matlab fue drasticamente inferior a los otros dos algoritmos en todos los casos.
- El tiempo registrado al utilizar multiplicación vectorial fue inferior al de multiplicación por coeficientes en todos los casos, sin embargo, para matrices de 2048×2048 la diferencia fue muy poca y la diferencia para matrices de 4096×4096 tampoco fue tan alta como con las matrices de 8192×8192.

Se validó que Matlab detecta cuando el equipo en el que se ejecuta dispone de multicore, pudiendo servirse de esto en situaciones donde hay recursos disponibles. Además los algoritmos de Matlab (como la multiplicación matricial y la multiplicación vectorial) se hacen llamando directamente a rutinas altamente optimizadas de la BLAS.

Tanto el algoritmo de multiplicación de Matlab como el de multiplicación vectorial se benefician de ser paralelizados, sin embargo se destaca como el algoritmo de multiplicación con vectores sigue siendo secuencial en naturaleza (paralalizandose una multiplicación vectorial en cada una de las N*N iteraciones). El algoritmo de multiplicación de Matlab es altamente paralelizado, orquestando la resolución del mismo por bloques. Por otra parte, el algoritmo de multiplicación por coeficientes no se beneficia en nada de esto, dado que Matlab realiza multiplicaciones normales sobre C (No detectando que se está deseando multiplicar vectores o matrices).

Teniendo esto en cuenta, se pueden extraer las siguientes interpretaciones y conclusiones:

- Debido a lo observado al monitorear los recursos del sistema durante ejecución, se cree que disponer de un procesador multicore en la plataforma de ejecución fue altamente beneficioso tanto para el algoritmo de multiplicación de Matlab como para la implementación que usa multiplicación vectorial, dado que Matlab hizo correcto uso de estos recursos.
- Se interpreta que usar multiplicación vectorial respecto a multiplicar coeficientes ofrece una mejora de performance (al paralelizar operaciones de mayor grado de optimización), sin embargo esta también tiene un overhead asociado, lo que podría explicar porque la ganancia es muy poca para matrices de 2048×2048, y sustancialmente alta para matrices de 192×8192. Esto se debe a que que el overhead es muy alto para el poco tiempo que demora la operación.

Suponiendo el caso de ejecución en el equipo **HWALN** que dispone de un procesador Intel Xeon Gold 6138 y 128GB de RAM, se estima que la característica principal a tener en cuenta es el procesador, el cual parece ser superior al de la plataforma de ejecución en todo aspecto posible.

Se realizan las siguientes presunciones:

- Todos los algoritmos obtendrían una mejora de tiempo de ejecución, debido a la inherente naturaleza de la plataforma (Siendo altamente superior en poder de computo al de la plataforma de ejecución)
- Se destaca que para la experimentación que fue realizada (Matrices de tamaño $2048, 4096, 8192$ como nunca se llegó a usar al 100 % la RAM, tener 128GB de RAM no presentará una mejora para los tiempos de ejecución de los algoritmos.
- Se anticipa que para el equipo HWALN la multiplicación de Matlab será la que mejor haga uso de las características del mismo, sin embargo para los tiempos de ejecución capturados con matrices de tamaño 2048 y 4096 esto no representará una mejora sustancial para un humano. Si ocurriría que para tamaño 8192 el algoritmo probablemente demore un tiempo sustancialmente menor, lo mismo se cree para matrices de mayor tamaño.
- El algoritmo de multiplicación por vectores podría ver una mejora significativa, dado que hay más nucleos para paralelizar, sin embargo, cada operación paraleliza tan pocas operaciones que probablemente el overhead de orquestar la paralelización sea significativamente más alto que el tiempo de ejecución en la multiplicación vectorial.
- Se estima que el algoritmo de multiplicación por coeficientes obtendría una mejora lineal, dado el mayor tamaño de caché y mayor velocidad de procesamiento. Esta ganancia sería significante, pero no no sería tan significativo en comparación a la mejora no lineal que obtendrían los otros dos algoritmos (Siendo paralelos en su naturaleza).

2.2. Ejercicio 2, Parte I - Estrategias de Ordenamiento

2.2.1. Marco Teórico

Existen múltiples técnicas de almacenamiento de matrices dispersas, no obstante, Matlab emplea la denominada **CSC** *(Compressed Sparse Column)*. Normalmente, este formato cuenta con 4 estructuras:

- *Values* - Vector con los valores no nulos de la matriz en cuestión.
- *Columns* - Vector del mismo largo que *values*, donde el i-ésimo elemento indica en que columna se encuentra el i-ésimo elemento de *values*.
- *PointerB* - Vector donde el j-ésimo elemento da el índice en *values* del primer elemento no nulo de la columna j.
- *PointerE* - Vector donde el j-ésimo elemento da el índice en *values* del último elemento no nulo de la columna j.

En Matlab, este tipo de almacenamiento cambia un poco, utilizando una única estructura para almacenar *pointerB* y *pointerE*. No obstante, la idea detrás es la misma.

Respecto a los algoritmos de ordenamiento, se proporcionan cuatro bien diferenciados:

- **Conteo de Columnas** (`colperm`) - Algoritmo que ordena la matriz de forma tal que los elementos no nulos más grandes quedan cerca del final de la matriz.
- **Cuthill-McKee Inverso** (`symrcm`) - Algoritmo que ordena la matriz de forma tal que los elementos no nulos queden lo más cerca posible de la diagonal, reduciendo así el ancho de banda de la matriz en cuestión.
- **Grado Mínimo** (`amd`) - Algoritmo que busca generar grandes bloques de elementos nulos, agrupando los elementos no nulos en patrones que permitan esto.
- **Disección Anidada** (`dissect`) - Algoritmo que trata a la matriz como una matriz de adyacencia de un grafo dado. Colapsando conjuntos de vértices y aristas, reordenando el grafo resultante y realizando el proceso inverso, se obtiene una matriz reordenada.

Según la documentación de Matlab, el algoritmo de disección anidada es el que suele tener mejores resultados en relación al resto (aunque se mencions que las diferencias con el de grado mínimo son pocas a nivel resultados).

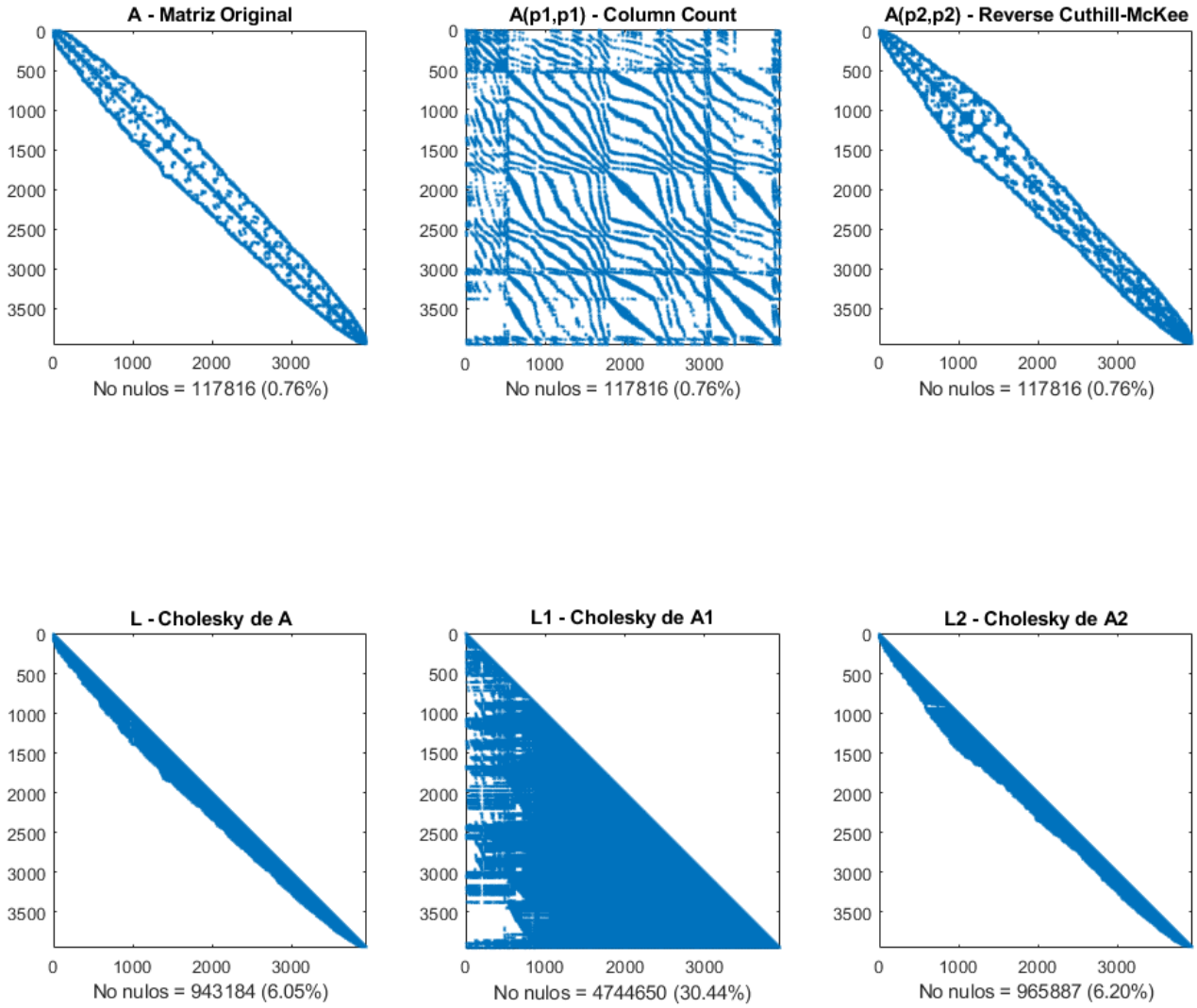
2.2.2. Código implementado

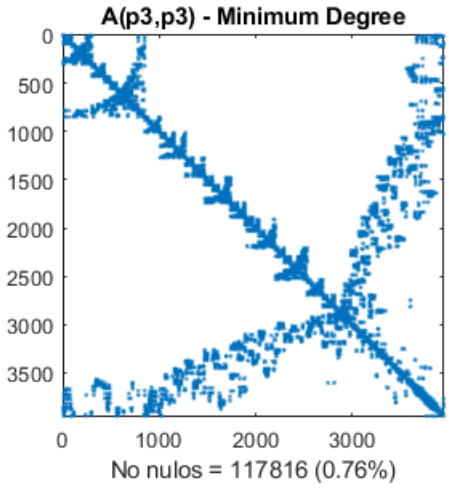
La implementación pedida para el análisis del ordenamiento de matrices dispersas se proporciona en el archivo `b_sparses.m`. En dicho script se llevan a cabo las siguientes acciones:

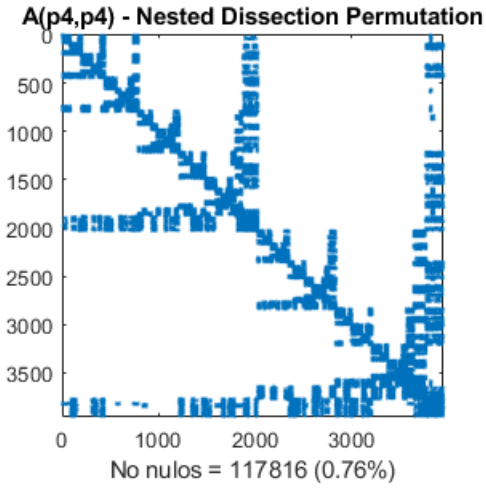
1. Se carga la matriz `bcsstk15 (A)`.
2. Se realiza una Factorización de Cholesky de dicha matriz `L = chol(A)`.
3. Se cuenta la cantidad de elementos no nulos en ambas versiones.
4. Se reordenan las matrices `A` y `L` utilizando las 4 técnicas de ordenamiento anteriormente mencionadas.
5. Se grafica la distribución de cada matriz de forma comparativa.
6. Se monitorea el espacio utilizado en memoria por cada una de ellas.

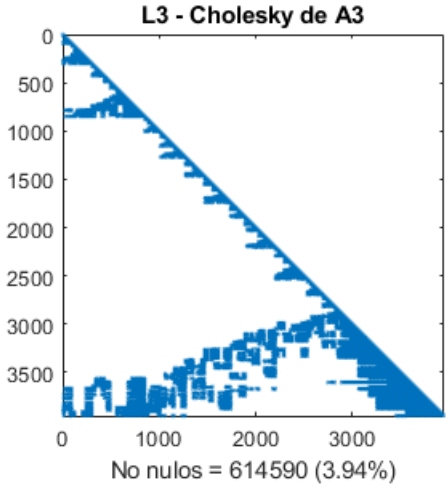
2.2.3. Experimentación y resultados obtenidos

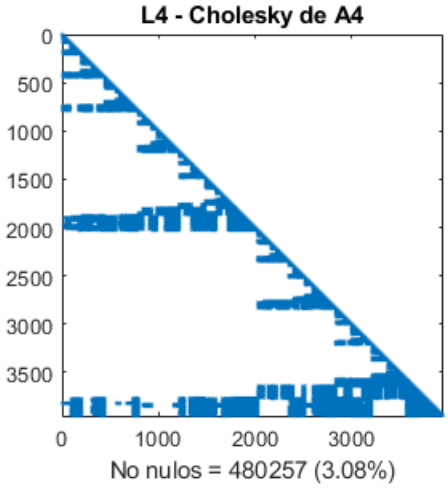
A continuación se adjunta la comparativa de gráficas para la distribución de los elementos no nulos en la matriz dispersa y sus respectivos ordenamientos:











2.2.4. Observaciones y Conclusiones

Teniendo en cuenta los resultados capturados, se pueden realizar las siguientes observaciones:

- La factorización de Cholesky siempre aumenta la cantidad de elementos no nulos.
- La factorización de Cholesky del ordenamiento de conteo de columnas sufre de gran fill-in, aumentando sustancialmente la cantidad de elementos no nulos.
- La factorización de Cholesky del ordenamiento de cuthill-mckee tiene resultados similares a la factorización sin ordenamiento previo.
- La factorización de Cholesky de los ordenamientos de grado mínimo y disección anidada tienen resultados similares, obteniendo una cantidad de valores no nulos menor al resto de estrategias evaluadas.

2.3. Ejercicio 2, Parte II - Sistemas Lineales

A continuación se presenta una breve experimentación centrada en cómo las matrices computadas anteriormente se comportan como entrada del método de gradiente conjugado para la resolución de sistemas lineales.

Se destaca que el código implementado para esta parte se agregó al script `b_sparses`, generando los vectores b y haciendo las respectivas llamadas a pcg.

2.3.1. Experimentación y resultados obtenidos

La experimentación llevada a cabo busca evaluar dos cosas:

- ¿Cómo se comportan las matrices computadas frente a distintos vectores de solución b?
- ¿Qué tanto afecta el precondicionamiento en la convergencia del método?

Para evaluar el primer punto se decidió capturar resultados de la función `pcg` variando el vector b de la siguiente forma:

- `b_a`: Generado mediante `rand`, compuesto por doubles entre [0,1] uniformemente distribuidos a lo largo del vector.
- `b_b`: Generado mediante `randi`, compuesto por enteros entre [0,10000] uniformemente distribuidos a lo largo del vector.
- `b_c`: Generado mediante `randn`, compuesto por doubles que fueron tomados de una función con distribución normal, con media $\mu = 0$ y desviacion estandar $\sigma = 1$.

Para evaluar el segundo punto se decidió capturar resultados llamando a la función `pcg`:

- Solamente con las matrices $A, A1, A2, A3, A4$, el vector b correspondiente, el maximo de iteraciones y la tolerancia para la solución.
- Pasando además de las matrices A los precocondicionadores `M1, M2`, siendo para cada A las matrices L y L', en donde L es la factorización de Cholesky inferior (lower) calculada en la parte anterior

Por último de la experimentación se destaca también lo siguiente:

- Se fijó la cantidad de iteraciones máximas para un método como `maxit = 100000`. Esta alta cifra se decidió porque se quería evaluar cada ejecución sin acotar el número potencial de iteraciones que el mismo necesitase para converger.
- Se fijó una tolerancia al error de `tol = 1e-14`. Si bien ninguna de las corridas llegó a cumplir con dicha tolerancia (debido a que el método numérico se estancaba en un resultado) es de interés ver qué tanto llegaron a acercar su error residual relativo al valor de tolerancia exigido.

A continuación se presentan los resultados obtenidos, siendo estos la cantidad de iteraciones y el error residual relativo asociado a la solución encontrada por el método.

Los resultados obtenidos **sin usar** matrices precondicionadas son los siguientes:

	$b = b\_a$		$b = b\_b$		$b = b\_c$	
	Iters	Relres	Iters	Relres	Iters	Relres
Original Matrix (A)	26449	$7.0213e^{-10}$	26440	$8.7148e^{-10}$	26273	$5.9673e^{-10}$
Column Count	26431	$7.9008e^{-10}$	26424	$7.9491e^{-10}$	26453	$5.0255e^{-10}$
Reverse Cuthill-McKee	26209	$1.0224e^{-09}$	26434	$7.8949e^{-10}$	26257	$3.7015e^{-10}$
Minimum Degree	26391	$7.0551e^{-10}$	26380	$9.0070e^{-10}$	26249	$5.0478e^{-10}$
Nested Dissection Permutation	26422	$8.5384e^{-10}$	26610	$7.4090e^{-10}$	26224	$5.2540e^{-10}$

Los resultados obtenidos **usando** matrices precondicionadas son los siguientes:

	$b = b\_a$		$b = b\_b$		$b = b\_c$	
	Iters	Relres	Iters	Relres	Iters	Relres
Original Matrix (A)	2	$8.9441e^{-12}$	2	$8.3577e^{-12}$	2	$3.8826e^{-13}$
Column Count	2	$8.8832e^{-12}$	2	$9.7465e^{-12}$	2	$6.7352e^{-13}$
Reverse Cuthill-McKee	2	$7.8233e^{-12}$	2	$1.0355e^{-11}$	2	$6.0793e^{-14}$
Minimum Degree	2	$8.8973e^{-12}$	2	$8.9170e^{-12}$	2	$9.0013e^{-14}$
Nested Dissection Permutation	2	$9.0836e^{-12}$	2	$9.4274e^{-12}$	2	$8.0192e^{-14}$

2.3.2. Observaciones y Conclusiones

Teniendo en cuenta los resultados capturados, se pueden realizar las siguientes observaciones:

- Para matrices no precondicionadas, el algoritmo convergió a un punto estacionario en poco más de 26000 iteraciones para todos los ordenamientos de matrices.
- Para matrices precondicionadas, el algoritmo convergió en solamente 2 iteraciones para todos los ordenamientos de matrices.
- En ambos casos no se alcanzó la solución para la tolerancia especificada, sino que el método se estancó (esto era de esperarse, dado el muy bajo umbral de tolerancia que fue especificado).
- El problema con el vector b_c (distribución normal $\mu = 0, \sigma = 1$) le fue más fácil de resolver al método numérico, logrando siempre obtener un error residual menor al resto de los problemas.

De los datos observados se pueden extraer las siguientes interpretaciones y conclusiones:

- Todos los métodos llegaron a converger a una solución candidata de muy bajo error residual relativo. Si bien ninguno llegó a encontrar la solución exacta esta cifra cumple con un nivel de tolerancia bastante alto para el problema.
- Los ordenamientos de matrices no influyeron de forma perceptible en la ejecución ni performance del método numérico, ya que los resultados y cantidad de iteraciones fueron muy similares dentro de cada b. Se cree que esto pudo pasar porque la matriz A no planteaba un problema muy difícil para el método numérico.
- Precondicionar un problema logra mejorar la eficacia del método numérico sustancialmente, resultando en no sólo una convergencia más rapida (menor cantidad de iteraciones) sinó en que se hayan alcanzado soluciones de menor error residual relativo.