Algebra Lineal Numérica

Laboratorio 3

Valores Propios y Singulares

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 3**. 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**). Dichas matrices fueron cargadas de archivos ya existentes o en su defecto, 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 tic, la cual almacena el horario actual en el momento de ejecutar la instruccion `tic` y hace lo mismo al ejecutar la instrucción `toc`. Luego de eso, la diferencia entre ambos registros devuelve el tiempo que tomó ejecutar las instrucciones del medio, permitiendo medir en forma general el tiempo de ejecución de una o varias rutinas.

5. Debido a que se trabajó con matrices y vectores aleatorios, los resultados capturados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes. No obstante, 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 - Valores Propios en Matrices Densas

2.1.1. Solución General

La implementación pedida para el cálculo de valores propios en matrices densas se proporciona en el archivo `parte_a.m`. De dicha implementación, se destacan los siguientes aspectos:

- Como datos de entrada, se generaron dos matrices reales no simétricas con números aleatorios, siendo las mismas de dimensiones 10×10 y 1024×1024 respectivamente.
- Para el cálculo de valores y vectores propios de dichas matrices, se emplearon tres algoritmos implementados en Matlab:
  - `eig`
  - `eig(nobalance)`
  - `eigs`
- Al momento de evaluar la precisión de cada algoritmo, se tomaron como *baseline* los valores retornados por el algoritmo `eig`, calculando el error relativo de los resultados del resto de algoritmos comparando directamente. El calculo del error se realizó tomando la norm de restarle el vector de valores propios o la matriz de vectores propios al baseline (eig).
- Al momento de evaluar el tiempo de ejecución de cada algoritmo, se ejecutó la función `tic` antes y la función `toc` al finalizar cada uno de los mismos.

Poniendo el foco en los algoritmos implementados, se destacan los siguientes aspectos de cada uno:

- `eig`:
  - Algoritmo estándar de Matlab para calcular valores y vectores propios de una matriz.
  - Funciona con matrices de diversos formatos, pero no funciona con matrices dispersas.
  - Es particularmente útil cuando se buscan todos los valores propios.
  - Dependiendo del formato de la matriz, ejecuta distintas rutinas de *LAPACK* para computar los valores y vectores propios en cuestión. A continuación, algunos ejemplos basados en la estructura de la matriz de entrada A:
    - A real simétrica: `DSYEV (Double - Symmetric - Eigen Values)`.
    - A real no simétrica: `DGEEV (Double - General - Eigen Values)`.
    - A hermítica: `ZHEEV (Complex - Hermitian - Eigen Values)`.
    - A no hermítica: `ZGEEV (Complex - General - Eigen Values)`.
- `eig(nobalance)`:
  - Versión no balanceada del algoritmo `eig`.
  - Sigue la misma idea pero, a diferencia de su versión balanceada, no prebalancea la matriz de entrada, por lo que no afecta su condicionamiento.
  - En muchos casos, el balanceo mejora la precisión de los valores computados. En otros casos, el balanceo hace crecer valores pequeños resultantes de errores de redondeo, lo cual afecta al resultado final de forma negativa. En estos escenarios, la versión `eig(nobalance)` es particularmente buena.
  - Dependiendo del formato de la matriz, ejecuta distintas rutinas de *LAPACK* para computar los valores y vectores propios en cuestión. A continuación, algunos ejemplos basados en la estructura de la matriz de entrada A:
    - A real no simétrica:
      `DGEHRD (Double - General - Hessenberg Reduction)`.
      `DHSEQR (Double - Hessenberg - Schur Reduction)`.

- A real no simétrica (pero computando vectores propios):
  `DGEHRD (Double - General - Hessenberg Reduction).`
  `DORGHR (Double - Orthogonal - Hessenberg Reduction, Explicit Q).`
  `DHSEQR (Double - Hessenberg - Schur Reduction).`
  `DTREVC (Double - Triangular - Eigen Vectors).`

- `eigs:`
  - Algoritmo iterativo de Matlab para calcular valores y vectores propios de una matriz.
  - Hace llamados a ARPACK.
  - Funciona con matrices todo tipo pero su performance es mejor para matrices dispersas, grandes y no simétricas.
  - Es particularmente útil cuando se busca un subconjunto de los valores propios. Este sin embargo no es el caso, y es necesario explicitar vía parámetros que queremos extraer todos los valores y vectores propios.

2.1.2. Experimentación y Resultados

A continuación, se adjunta una tabla con los tiempos de ejecución y los errores de precisión retornados por cada algoritmo:

Algoritmo	Matriz	Tiempo (s)	Error (valores)	Error (vectores)
`eig`	10×10	3.506e-04	-	-
`eig(nobalance)`	10×10	2.271e-04	0	0
`eigs`	10×10	0.0137	1.423	2.342
`eig`	1024×1024	0.842	-	-
`eig(nobalance)`	1024×1024	0.802	0	0
`eigs`	1024×1024	0.794	256.672	5.673

2.1.3. Observaciones y Conclusiones

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

1. La ejecución de `eig` empleó la rutina `DGEEV` de *LAPACK*, mientras que la ejecución de `eig(nobalance)` empleó las rutinas `DGEHRD` y `DHSEQR`. Esto se debe a que las matrices de entrada son reales no simétricas.
2. El tiempo empleado por ambas versiones de `eig` fue similar en ambas situaciones.
3. El error obtenido por la versión sin balanceo de `eig` fue 0, es decir, dió los exactos mismos resultados.
4. El tiempo empleado por `eigs` fue notablemente mayor al empleado por los otros dos algoritmos para la matriz de dimensiones 10×10. No obstante, para la matriz de dimensiones 1024×1024 el tiempo empleado por `eigs` no solo fue muy cercano al resto sino que fue un poco menor.
5. El error obtenido por `eigs` fue relativamente pequeño para ambas matrices tanto en valores como en vectores propios, no obstante, para los valores de la matriz de dimensiones 1024×1024, el error escaló notoriamente.

A partir de dichos datos y observaciones, se extraen las siguientes conclusiones:

1. El tiempo de la versión `eig(nobalance)` fue apenas menor al de `eig`, por lo que se estima que la diferencia se encuentra en el paso extra de balanceo que el algoritmo `eig` ejecuta.
2. El resultado de la versión `eig(nobalance)` fue exactamente igual al de `eig`, por lo que se concluye que en este escenario, no existieron errores de redondeo que afectaran la performance general de `eig`.
3. Se concluye que el tiempo de `eigs` fue mayor en comparación a `eig` para la matriz de dimensiones 10×10 porque está diseñado para matrices de gran tamaño. Para la matriz de dimensiones 1024×1024, el tiempo no fue lo suficientemente menor, como para reafirmar esta teoría (oscilando entre valores menores y mayores para cada ejecución).

2.2. Ejercicio 2 - Valores Propios en Matrices Dispersas

2.2.1. Solución General

La implementación pedida para el cálculo de valores propios en matrices dispersas se proporciona en el archivo `parte_b.m`. De dicha implementación, se destacan los siguientes aspectos:

- Como datos de entrada, se cargaron en memoria dos matrices dispersas provistas por el cuerpo docente:
  - `bcsstk01`: Matriz real simétrica de dimensiones 48×48.
  - `bcsstk15`: Matriz real simétrica de dimensiones 3948×3948.
- Para el cálculo de valores y vectores propios de dichas matrices, se emplearon tres algoritmos implementados en Matlab:
  - `eig`
  - `eig(nobalance)`
  - `eigs`
- Dado que tanto `eig` como `eig(nobalance)` sólo funcionan para matrices densas, las matrices de entrada debieron ser adaptadas a este formato previo a la ejecución de ambos algoritmos, perdiendo la propiedad de dispersión. Para lograrlo, se ejecutó la función full.
- Al momento de evaluar la precisión de cada algoritmo, se tomaron como *baseline* los valores retornados por el algoritmo `eig`, calculando el error relativo de los resultados del resto de algoritmos comparando directamente.
- Al momento de evaluar el tiempo de ejecución de cada algoritmo, se ejecutó la función `tic` antes y la función `toc` al finalizar cada uno de los mismos.

2.2.2. Experimentación y Resultados

A continuación, se adjunta una tabla con los tiempos de ejecución y los errores de precisión retornados por cada algoritmo:

Algoritmo	Matriz	Tiempo (s)	Error (valores)	Error (vectores)
`eig`	`bcsstk01`	3.515e-04	-	-
`eig(nobalance)`	`bcsstk01`	3.956e-04	0	0
`eigs`	`bcsstk01`	8.430e-04	1.063e+10	2
`eig`	`bcsstk15`	4.380	-	-
`eig(nobalance)`	`bcsstk15`	4.379	0	0
`eigs`	`bcsstk15`	4.224	8.061e+10	2

2.2.3. Observaciones y Conclusiones

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

1. La ejecución de `eig` empleó la rutina `DSYEV` de *LAPACK*. Esto se debe a que las matrices de entrada son reales simétricas.
2. El tiempo empleado por ambas versiones de `eig` fue similiar en ambas situaciones.
3. El error obtenido por la versión sin balanceo de `eig` fue 0, es decir, dió los exactos mismos resultados.
4. El tiempo empleado por `eigs` fue apenas mayor al empleado por los otros dos algoritmos para la matriz `bcsstk01`. No obstante, para la matriz `bcsstk15` el tiempo empleado por `eigs` no solo fue muy cercano al resto sino que fue un poco menor.

5. El error obtenido por `eigs` fue el mismo para el cálculo de vectores propios de ambas matrices. Por otra parte, el error con respecto a los valores propios fue muy alto en ambos casos.

A partir de dichos datos y observaciones, se extraen las siguientes conclusiones:

1. Al igual que en el caso anterior, el tiempo de la versión `eig(nobalance)` fue apenas menor al de `eig`, se cree que en este caso, al tratarse de matrices dispersas es posible afirmar que a mayor tamaño mejor será la performance de `eigs` que la de `eig` en cuanto a tiempo.

2. De nuevo, al igual que en el caso anterior, el resultado de la versión `eig(nobalance)` fue exactamente igual al de `eig`, por lo que se concluye que en este escenario, no existieron errores de redondeo que afectaran la performance general de `eig`.

3. En relación al tiempo de `eigs`, en este escenario fue mayor en comparación a `eig` para la matriz `bcsstk01` pero con menor diferencia de tiempo. Por otra parte, para `bcsstk15` fue menor, igual que en el caso anterior, reafirmando una vez más que el algoritmo está diseñado para matrices de gran tamaño.

4. Respecto al enorme error de precisión al calcular los valores propios, se estima que el algoritmo no logró converger de ninguna forma.

2.3. Ejercicio 3 - Compresión con SVD

2.3.1. Código Implementado

La implementación pedida para la compresión de imagen a través de valores singulares se proporciona en el archivo `parte_c.m`. De dicha implementación, se destacan los siguientes aspectos:

- Se experimentó comprimiendo la imagen `lena.pgm`, empleando los N valores y vectores singulares más significativos de la misma. Se realizaron compresiones para $192, 128, 96, 64, 32$ y 16. La filosofía de comprimir imágenes generándolas con su descomposición SVD es que, para una representación SVD ordenada (por magnitud de valores singulares), los primeros elementos tienden a contener mayor información sobre los elementos de la matriz que los últimos, dado que los valores singulares de pequeña magnitud no suelen almacenar mucha información de la transformación lineal en sus vectores asociados. Esta filosofía es similar a la que el algoritmo PCA emplea para reducir dimensionalidad de datos.
- Según la cantidad de vectores empleados, varía el almacenamiento requerido para la imagen en formato "svd", esto es, guardar las matrices U, V y Σ necesarias para generar la imagen (en lugar de guardar la imagen como una matriz de **65536 elementos**).
- A modo de simplificar el análisis de la compresión, se evaluó el ratio de compresión, contando la cantidad de elementos a representar,v pese a que los elementos para almacenar una matriz `pgm` son de tipo `uint8` y los elementos de la descomposición SVD son de tipo `double`.
- A modo de evaluar calidad de compresión, para cada matriz resultante se comparó la misma con la imagen original, calculando entre estas su *Peak Signal-to-Noise Ratio* (**PSNR**), su *Signal-to-Noise Ratio* (**SNR**), y su *Root Mean Squared Error* (**RMSE**).
- También se calcula el *Data Compression Ratio Percentage* como $100 * \frac{Uncompressed\ Size}{Compressed\ Size}$. Siendo por ende un *Compression Ratio* de 100 que ambas representaciones pesan lo mismo.

La imagen con la que se experimentó comprimir fue `lena.pgm`. La misma se adjunta a continuación, a modo de comparación visual:

2.3.2. Experimentación y Resultados

El **rango de la imagen proporcionada** es de 256. Esto se resuelve contando la cantidad de valores singulares no nulos extraídos de la descomposición SVD de la imagen en cuestión.

Como era también esperable, el **rango del primer cuadrante** de la matriz es de 128, la resolución del mismo se resolvió de igual manera (contando los valores singulares no nulos para la descomposición de ese cuadrante).

Los resultados obtenidos en la compresión de `lena.pgm` son los siguientes:

73920 elements svd (88.66% compression ratio)
PSNR: 49.3481 dB; SNR: 43.7235 db; RMSE: 0.0034

32896 elements svd (199.22% compression ratio)
PSNR: 38.4301 dB; SNR: 32.8034 dB; RMSE: 0.0120

18528 elements svd (353.71% compression ratio)
PSNR: 34.1916 dB; SNR: 28.5611 dB; RMSE: 0.0195

8256 elements svd (793.80% compression ratio)
PSNR: 30.1708 dB; SNR: 24.5312 dB; RMSE: 0.0310

2080 elements svd (3150.77% compression ratio)
PSNR: 25.7137 dB; SNR: 20.0466 dB; RMSE: 0.0518

528 elements svd (12412.12% compression ratio)
PSNR: 22.5666 dB; SNR: 16.8534 dB; RMSE: 0.0744

2.3.3. Observaciones y Conclusiones

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

- Almacenar los 192 valores y vectores más significativos de la descomposición SVD tiene más costo que almacenar la matriz original, requiriendo almacenar 73920 elementos en lugar de 65536.
- Para el resto de los valores con los que se experimentó comprimir la imagen se obtuvo una mejora sustancial en la cantidad de elementos. Obteniendo aún más compression ratio cuando se representa la imagen con menos valores y vectores singulares.
- Al aumentar el ratio de compresión (empleando menos valores singulares) el PSNR y SNR de la imagen resultante disminuye, mientras que su RMSE aumenta.
- Observando las imágenes resultantes a plena vista se destaca como visualmente las imágenes con $k = 192, 128$ y 96 se notan significantemente visualmente similares entre sí. Para $k = 64$ se empieza a notar la perdida de información significativa, mientras que para $k = 32$ y $k = 16$ la perdida de información es lo suficientemente alta como para causar ruido visual.

A partir de dichos datos y observaciones, se extraen las siguientes conclusiones:

- Almacenar una matriz en formato SVD puede representar un coste en la cantidad de elementos almacenados, recién al almacenar los 180 valores y vectores significativos para las matrices U, V y Σ se obtiene una ganancia en almacenar una menor cantidad de elementos.
- Se logró reducir significativamente la cantidad de elementos necesarios para almacenar la imagen sin comprometer la calidad de imagen (para $k = 96$ y $k = 64$), si bien a plena vista las imágenes no pierden mucha calidad, se destaca como la misma existe al comparar los $PSNR$, NSR y $RMSE$ obtenidos para compresiones con un k mayor.
- Pese a lo dicho anteriormente, la ganancia en compresión no se aplica cuando se evalúa el Data Compression Ratio considerando que cada elemento almacenados en formato SVD requiere 8 veces más espacio de almacenamiento (dado que los doubles ocupan 8 bytes, mientras que uint8 ocupa 1 byte).

2.3.4. Sobre una mejora de la compresión de imagen:

Si se desea mejorar el tamaño total en bytes para el método utilizado (lo cual no afectará en la cantidad de elementos almacenados), se destaca como se podría obtener un buen resultado convirtiendo los elementos de las matrices U, V y Σ a elementos de tipo float o incluso half-precision, necesitando así ocupar 4 y 2 bytes respectivamente (en lugar de 8)

Pese a que esto sea posible para reducir el tamaño total de almacenamiento para la imagen, se destaca como la compresión de imágenes es un área importante de estudio para la Teoría de la Información, y como haciendo uso de las particularidades de las imágenes (más siendo estas imágenes en grayscale) algoritmos pueden alcanzar ratios muy altos de compresión con una perdida despreciable, o inclusive sin perdida.

Para empezar se observa como el transformar la imagen `lena.pgm` a `lena_jpg.jpg`, disminuye sustancialmente el espacio necesario para almacenar dicho archivo, requiriendo 65,652 bytes para almacenar la matriz con su metadata, mientras que tan solo 23,166 bytes para almacenar la misma en formato JPG sin perdida.

A su vez, es posible comprimir esta imagen con algoritmos que la transformen en un JPG con pérdida, a continuación se muestran ejemplos junto a su Data Compression Rate en bytes al compararlos con el tamaño en bytes de `lena.pgm`. Estos mismos fueron generados comprimiendo la imagen empleando el algoritmo que utiliza la página https://compressjpeg.com/ y variando la calidad de la misma.

(a) 22.8 KB, 283,4 % CR (b) 13.8 KB, 490,8 % CR (c) 9.45 KB, 677,9 % CR (d) 7.66 KB, 836,8 % CR

(e) 5.98 KB, 1071,2 % CR (f) 4.69 KB, 1366,6 % CR (g) 3.08 KB, 2081,5 % CR (h) 1.80 KB, 3550,7 % CR

Al comparar con la representación en `pgm`, que consta de un archivo de texto con metadata y el valor para el color de cada píxel en la matriz, tanto la representación JPG sin perdida como imágenes comprimidas y representadas en JPG con pérdida son capaces de disminuir de manera sustancial el tamaño en bytes del archivo que contiene una imagen.