Metodos Numericos

Trabajo Práctico 1

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Practico 1

Eliminacion Gaussiana

Integrante	LU	Correo electrónico
Sanchez, Facundo	97/22	facundosanchez636@gmail.com.ar
Esposito, Camilo	725/22	espositocamilo@gmail.com
Sessarego, Santiago	693/22	ssessarego72@gmail.com

Índice

1. Resumen

El problema abordado en este trabajo fue la resolución computacional de sistemas de ecuaciones lineales mediante el uso del algoritmo de Eliminación Gaussiana (EG). Para ello, construimos diversas implementaciones basadas en este método y analizamos cuales son más convenientes y/u óptimas según el contexto y las necesidades de uso. Estas son: EG sin pivoteo, con pivoteo, y para sistemas tridiagonales con y sin precómputo.

Se llevaron a cabo diversos experimentos que exhibieron las límitaciones de ciertos algoritmos, la diferencia de performances al aplicar optimizaciones para sistemas tridiagonales, las implicaciones de los errores numéricos en las operaciones debidos al rango de representación de los números en la computadora, y las aplicaciones en el modelado y simulación de problemas de difusión.

A modo de conclusión, llegamos a que los algoritmos de Eliminación Gaussiana para la resolución de sistemas de ecuaciones son herramientas muy poderosas para la resolución de problemas matemáticos y para la simulación de fenómenos naturales como la difusión de calor a lo largo del tiempo, y que poseen una gran flexibilidad al poder ser adaptados a sistemas especiales, aprovechando sus características para incrementar la eficiencia de la resolución.

2. Introducción Teórica

El algoritmo de Eliminación Gaussiana, desarrollado para resolver sistemas de ecuaciones con un mismo número de ecuaciones que de incógnitas y con una única solución, consiste en, dada una matriz cuadrada A que representa los coeficientes de las incógnitas y un vector columna d que representa los términos independientes, en la triangulación de A y la posterior obtención de los resultados a través del despeje de los valores empleando "backward substitution".

Nuestros objetivos son además lograr en lo posible que nuestros algoritmos produzcan el menor grado de error numérico en las soluciones, el cual es producto del rango de representación de los números reales que poseen los tipos de datos de 32 y 64 bits, y crear variaciones de los algoritmos que para casos como el de matrices tridiagonales, donde aplicando el algoritmo común se producen operaciones redundantes con 0, mejore la eficiencia. Posteriormente, aplicaremos las soluciones que desarrollamos en problemas de difusión mediante la resolución de ecuaciones diferenciales expresadas de manera discreta para $\frac{d^2}{dx^2}u = d$.

3. Desarrollo

Para el desarrollo de los algoritmos implementaremos las matrices mediante la librería NumPy[1] ayudándonos de la clase Array para representar vectores y matrices. Para los gráficos, usaremos la librería PyPlot de MatPlotLib[2].

3.1. Eliminacion Gaussiana Sin Pivoteo

El primer enfoque que tomaremos para resolver un sistema de ecuaciones matricial de la forma $Ax = b$, con $A \in \mathbb{R}^{n \times n}$ una matriz cuadrada, $x \in \mathbb{R}^n$ un vector columna con las incógnitas y $b \in \mathbb{R}^n$ un vector columna con los términos independientes, mediante una aplicación del algoritmo de Eliminación Gaussiana, es la implementación en su versión más simple, donde no se lleva a cabo pivoteo, es decir, no se intercambian filas en ningún momento. Comenzamos por construir la matriz extendida $A' \in \mathbb{R}^{n \times n+1}$ que utilizaremos en el procedimiento, que resulta de agregar el vector de términos independientes como una nueva columna de la matriz A a la derecha. Luego, continuamos con la implementación del algoritmo, dividiéndolo en dos partes: la *triangulación* y la sustitución hacia atrás (*backward substitution*).

En la etapa de *triangulación*, nos encargamos de convertir sistemáticamente a la matriz A' en una matriz triangular superior. Para ello, siguiendo con el método propuesto, recorremos los elementos de las diagonales de la matriz A' y en cada iteración k buscamos anular los elementos que se encuentran directamente por debajo de a_{kk} (denominado pivote) mediante la resta entre la fila j $(k < j \leq n)$ que contiene al elemento a anular y la fila k multiplicada por $\frac{a_{jk}}{a_{kk}}$, es decir $F_j = F_j - \frac{a_{jk}}{a_{kk}}F_k$. De esta forma, logramos poner en cero a a_{jk} a través de esta resta, pues:

$$a_{jk} - \frac{a_{jk}}{a_{kk}}a_{kk} = a_{jk} - a_{jk} = 0$$

[1] https://numpy.org/
[2] https://matplotlib.org/

Observamos que existe la posibilidad de una división por cero si el elemento en la diagonal en este paso es nulo, por lo que agregamos un chequeo previo y cortamos la ejecución del programa en caso de que esto ocurra, afirmando que no fue posible encontrar una solución. Veremos una forma de evitarlo más adelante utilizando el pivoteo.

Una vez obtenida la matriz triangulada, realizamos el procedimiento de *backward substitution* especificado en el algoritmo, donde para cada fila, comenzando desde la última, vamos despejando y obteniendo los valores de la variable correspondiente, utilizando los resultados de los pasos anteriores. Cabe destacar que si bien se realiza una división, al ser por un elemento de la diagonal no puede ocurrir que el mismo sea cero, pues si no no se hubiera podido hacer la triangulación previa.

Con estos pasos en mente, implementaremos la Eliminación Gaussiana Sin Pivoteo guiandonos del siguiente pseudocodigo (1), dada $Ax = d$ la ecuacion matricial a resolver.

(1) - EG Sin Pivoteo

```
1:  M ← matriz_extendida(A, d)
2:  n ← A.size
3:  para j = 0 to n − 1 hacer
4:      para i = j + 1 to n − 1 hacer
5:          si M_jj == 0 entonces
6:              CORTAR EJECUCIÓN
7:          factor ← M_ij / M_jj
8:          para k = j to n hacer
9:              M_ik ← M_ik − factor · M_jk
10: para i = n − 1 to 0 hacer
11:     x_i ← (M_in − Σ_{j=i+1}^{n} M_ij · x_j) / M_ii
12: devolver x
```

Al realizar casos de test, vimos que los sistemas en los cuales no existen elementos nulos en diagonal en ninguna iteración son resueltos correctamente. En cambio, por ej en el sistema A=[[1,2,4],[1,2,-2],[-2,-8,4]] b=[2,2,4] en la segunda iteración tiene un cero en la diagonal lo cual provoca que el algoritmo no halle solución, cuando la misma existe y es [6, -2, 0]:

$$\left(\begin{array}{ccc|c} 1 & 2 & 4 & 2 \\ 1 & 2 & -2 & 2 \\ -2 & -8 & 4 & 4 \end{array}\right) \Rightarrow \left(\begin{array}{ccc|c} 1 & 2 & 4 & 2 \\ 0 & \mathbf{0} & -6 & 0 \\ -2 & -8 & 4 & 4 \end{array}\right)$$

La implementación realizada en el inciso anterior no asegura hallar la solución siempre que exista (y sea única), pues falla si en alguna iteración durante la triangulación el pivote es cero. Para solucionarlo vamos a aplicar una estrategia de pivoteo a la hora de triangular la matriz extendida, donde al hallar un pivote a_{kk} nulo en la diagonal intercambiaremos su fila con una fila j $(k < j \leq n)$ con a_{jk} que no sea nulo. Esto nos permite asegurarnos de encontrar la solución, siempre y cuando sea única.

Otra cosa que observamos es que la división por cero no es el único problema que genera esta operación: al realizar una división por un pivote con valor cercano a cero podría producirse un error numérico que afecte al resultado. Para minimizar esta posibilidad de inestabilidad numérica es conveniente realizar una selección del pivote a utilizar a la hora de triangular la matriz extendida aun cuando el mismo no sea cero, tomando aquel que tenga el mayor valor absoluto respecto de los demás candidatos.

3.2. Eliminacion Gaussiana Con Pivoteo

Para implementar la EG Con Pivoteo, partimos de la primera solución que no usa pivoteo y le hicimos pequeñas modificaciones. Primero, añadimos un parámetro adicional llamado *tolerancia*, con el cual si en algun momento dividimos una valor por otro menor a *tolerancia* aparecera una advertencia en pantalla que indique que se esta dividiendo por un valor cercano a 0, lo que podría generar un error numérico. Luego, añadiremos previo al calculo del factor, el pivoteo mencionado previamente. Debemos asegurarnos que el pivote tomado en cada iteración luego del intercambio de filas sea no nulo, pues en los casos de sistemas con cero o infinitas soluciones, durante la triangulacion se genera una columna con solo ceros por debajo del pivote. De esta manera, implementaremos la Eliminación Gaussiana Con Pivoteo en base al pseudocódigo (2), dada $Ax = d$ el sistema de ecuaciones a resolver.

(2) - EG Con Pivoteo

```
1:  M ← matriz_extendida(A, d)
2:  n ← A.size
3:  para j = 0 to n − 1 hacer
4:      m ← j
5:      para k = j + 1 to n hacer
6:          si |M_ki| > |M_mj| entonces
7:              m ← k
8:      M_j ↔ M_m
9:      para i = j + 1 to n − 1 hacer
10:         si M_jj == 0 entonces
11:             CORTAR EJECUCIÓN
12:         factor ← M_ij / M_jj
13:         para k = j to n hacer
14:             M_ik ← M_ik − factor · M_jk
15: para i = n − 1 to 0 hacer
16:     x_i ← (M_in − Σ_{j=i+1}^{n} M_ij · x_j) / M_ii
17: devolver x
```

Aunque hayamos disminuido el margen de error mediante la elección adecuada de pivotes, puede ocurrir que debido al límite de representación de los números reales en la computadora se provoquen pequeños redondeos númericos que, al acumularse, causen resultados que no son precisamente correctos. Para medir el error numérico, realizamos el experimento detallado en la sección 4.1: Tomaremos una ecuación matricial $Ax = d$ ya resuelta y vamos a sumar un valor ϵ de la siguiente manera:

$$A = \begin{pmatrix} 1 & 2+\epsilon & 3-\epsilon \\ 1-\epsilon & 2 & 3+\epsilon \\ 1+\epsilon & 2-\epsilon & 3 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 6 \\ 6 \end{pmatrix}$$

donde $10^{-6} \leq \epsilon \leq 10^{0}$.

Mediremos el margen de error para números de punto flotante en 32 y 64 bits. Definiremos el error dada la solución $\hat{x}$ (la solución obtenida) y x (la solucion esperada) como $\max_i |\hat{x}_i - x_i|$. Veremos cómo varía este error con cada valor de ϵ más adelante en la sección de Resultados y Discusión.

3.3. Eliminacion Gaussiana para Sistemas Tridiagonales

Ya habiendo tratado los casos para matrices cuadradas generales, ahora nos ocuparemos de los sistemas tridiagonales de ecuaciones, que son un caso particular de las mismas con la forma:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

Observamos que su representación matricial tiene una forma de 3 diagonales (a, b, y c). Se los puede resolver también mediante los algoritmos previamente implementados, pero sabiendo ya de antemano la forma de la matriz, se puede sacar provecho y crear una variante del algoritmo mucho más simple y eficiente en términos de tiempo y memoria.

Como primera implementación, vamos a asumir que la matriz tridiagonal no posee valores nulos en sus diagonales en ningún paso de la triangulación, haciendo un algoritmo que no use pivoteo.

Al utilizar el algoritmo de EG Sin Pivoteo en sistemas tridiagonales, estamos realizando muchas operaciones redundantes sobre valores nulos al hacer el paso k de triangulación, restando la fila k multiplicada por el escalar $\frac{a_{kk+j}}{a_{kk}}$ a cada fila j ($k < j < n$), pues a partir de la fila $k+2$ los valores a_{kk+j} son 0. Además, cuando restamos la fila k a las filas inferiores, sabemos que los valores en las columnas entre $k+2$ y n en la fila k también son 0, por lo cual también es redundante realizar las operaciones para estos elementos.

Nos gustaría únicamente restar la k-ésima fila a la fila inmediata de abajo por el escalar adecuado, haciendo algo

de la forma:

$$a_i = a_i - factor * b_{i-1}$$
$$b_i = b_i - factor * c_{i-1}$$
$$d_i = d_i - factor * d_{i-1}$$

Aunque esto reducirá el tiempo de cómputo de manera significativa, aún tenemos un costo grande de memoria al almacenar 0s en la matriz fuera de la diagonal, los cuales no son usados en el algoritmo y solo ocupan espacio. Lo mas óptimo seria trabajar con 3 vectores a, b, y c que representen la diagonal. De esta forma tenemos el algoritmo de Thomas [3]

En nuestra implementación usamos el siguiente pseudocódigo (3), dadas las diagonales a, b, y c, y el termino independiente d, todos de tamaño n:

(3) - EG Tridiagonal - Algoritmo de Thomas

```
1: n ← size(b)
2: para i = 1 hasta n − 1 hacer
3:     factor ← a_i / b_{i−1}
4:     b_i ← b_i − factor · c_{i−1}
5:     d_i ← d_i − factor · d_{i−1}
6: x ← array(n)
7: x_{n−1} ← d_{n−1} / b_{n−1}
8: para i = n − 2 hasta 0 hacer
9:     x_i ← (d_i − c_i · x_{i+1}) / b_i
10: devolver x
```

En nuestra implementación, añadimos adicionalmente una advertencia en consola cada vez que se realiza una división por un valor muy cercano a 0 (el cual es definido por el usuario mediante el parametro *tolerancia*).

Se realizaron ensayos y pruebas del algoritmo con distintas matrices tridiagonales. El mismo logró triangular y resolver varios sistemas tridiagonales de manera correcta. Aun así, el algoritmo falló y frenó su ejecución en los casos donde la matriz simulada poseía un 0 en la diagonal al igual que la triangulación sin pivoteo. Para los sistemas lineales que buscaremos resolver más adelante en este informe esto no será problema, pues resolveremos sistemas en los que no deberían aparecer elementos nulos en las diagonales.

3.4. Eliminacion Gaussiana para Sistemas Tridiagonales Con Precomputo

El algoritmo es de naturaleza muy simple, pues consiste en ir restando a cada fila de manera sucesiva la multiplicación de la anterior por un escalar ($\frac{a_k}{b_{k-1}}$). En la matriz extendida con la que se está trabajando en el proceso, las operaciones realizadas sobre d (término independiente) en la triangulación son las mismas para todo d independientemente de su valor. Si sacamos ventaja de esto, podemos construir una mejora del algoritmo basada en dos partes:

- **Pre-cómputo**: En esta etapa, triangulamos la matriz de diagonales a, b, y c, guardando los factores usados en cada iteración k y devolviendolos al final.
- **Post-cómputo**: En esta etapa, aplicamos a un termino independiente d las operaciones con esos factores y realizamos el posterior despeje de variables mediante *backward substitution*.

Para implementar esta variante de la EG Tridiagonal, nos basamos en el siguiente pseudocodigo (4) y (5):

La ventaja de esta variante consiste en que el cálculo de la triangulación solo debe realizarse una sola vez, y ahora para cualquier d solo tenemos que hacer los despejes. Esto significa una ganancia en eficiencia cuando queramos calcular la solución de un sistema tridiagonal $Ax = d$ para diferentes valores de d.

La función *pre-cómputo* que implementamos devuelve 3 arrays: *b_triangulado*, *c_triagulado*, y *factores*, donde en cada posición de $factores_i$ se guarda el *factor* por el cual se hizo la operación $M_{i+1} - factor.M_i$, siendo M la matriz

[3] https://serc.carleton.edu/teaching_computation/virtual_workshop_2023/activities/279427.html

(4) - EG Tridiagonal - Pre-Computo

1: $n \leftarrow \text{size}(b)$
2: $factores \leftarrow \text{array}(n)$
3: **para** $i = 1$ **hasta** $n - 1$ **hacer**
4: $\quad factor \leftarrow \frac{a_i}{b_{i-1}}$
5: $\quad factores_i \leftarrow factor$
6: $\quad b_i \leftarrow b_i - factor \cdot c_{i-1}$
7: **devolver** $b, c, factores$

(5) - EG Tridiagonal - Post-Computo

1: **PRECONDICION:** a y b ya triagulados con los $factores$
2: $n \leftarrow \text{size}(b)$
3: **para** $i = 1$ **hasta** $n - 1$ **hacer**
4: $\quad d_i \leftarrow d_i - factores_i \cdot d_{i-1}$
5: $x \leftarrow \text{array}(n)$
6: $x_{n-1} \leftarrow \frac{d_{n-1}}{b_{n-1}}$
7: **para** $i = n - 2$ **hasta** 0 **hacer**
8: $\quad x_i \leftarrow \frac{d_i - c_i \cdot x_{i+1}}{b_i}$
9: **devolver** x

extendida del sistema. Con estos tres datos, en la función *Post-computo* hacemos la sustitución hacia atras, pero únicamente con dos elementos, como puede observarse en el pseudocódigo.

Más adelante, en el **EXPERIMENTO 4**, dado un sistema de ecuaciones a resolver de tamaño fijo, compararemos los tiempos de cómputo entre el algoritmo EG Tridiagonal y EG Tridiagonal Con Precómputo cuando se quiere resolver el sistema una cantidad n de veces.

3.5. Verificacion de la Implementacion

3.5.1. Solucion de Ecuaciones Diferenciales

Ahora que tenemos desarrollados nuestros algoritmos, vamos a verificar su implementación mediante el experimento de la sección 4.2 en el cual buscaremos la solución de problemas de ecuaciones diferenciales simples en el caso unidimensional de la forma:

$$\frac{d^2}{dx^2}u = d$$

En este caso, podemos discretizar la operación mediante la fórmula:

$$u_{i-1} - 2u_i + u_{i+1} = d_i$$

Donde el valor de la derivada de una constante será 0 gracias a que $1 - 2 + 1 = 0$. Si de esta ecuación hacemos un sistema de n ecuaciones, para hallar u tendríamos que resolver el siguiente sistema de ecuaciones matriciales $Ax = d$:

$$\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & \cdots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{pmatrix}$$

Gracias a que la matriz A es tridiagonal, podemos hacer uso del algoritmo de EG tridiagonal para resolver el sistema de forma más eficiente. Representaremos a A mediante los siguientes vectores de tamaño n:

$$a = [0, 1, \ldots, 1, 1], \quad b = [-2, -2, \ldots, -2, -2], \quad c = [1, 1, \ldots, 1, 0]$$

Finalmente, representaremos al u obtenido en un gráfico para luego sacar conclusiones más adelante.

Aprovechamos lo estudiado sobre la matriz laplaciana para llevar a cabo comparaciones entre los tiempos de cómputo de los diferentes algoritmos propuestos a lo largo del trabajo.

Por un lado, en la sección 4.3 analizamos el comportamiento entre el algoritmo de EG Con Pivoteo y la versión para sistemas tridiagonales. Para ello vamos a tomar una matriz tridiagonal basada en la matriz laplaciana y su versión en vectores, y vamos a calcular los diferentes tiempos de ejecución de los algoritmos a medida que la matriz aumenta su tamaño. En un primer intento, ejecutamos cada algoritmo una única vez para cada tamaño $n \times n$ de matriz, y luego comparamos cada tiempo con el tamaño, respectivamente. El resultado fue un gráfico anómalo y ruidoso. Supusimos que fue debido al scheduling del sistema, por lo que para solucionarlo, en cada iteración para cada n resolvimos el sistema una cierta cantidad de veces y tomamos la que empleó menos tiempo. Esta modificación dio por resultado un gráfico más claro y con mediciones más precisas.

En el experimento de la sección 4.4 estudiamos la diferencia del tiempo de ejecución entre el algoritmo de Thomas, optimizado para matrices tridiagonales, y su versión con precómputo. Para hacerlo, vamos a aplicar ambos algoritmos r veces consecutivas para un mismo sistema y observar si, como esperamos, el segundo procedimiento es más eficiente una vez realizado y guardado el precómputo, que luego es utilizado en las demás iteraciones.

3.5.2. Solucion de Problemas de Difusion 1D

Ahora usaremos nuestros algoritmos para modelar problemas de difusión. El caso de la Difusión en una dimensión, lo plantearemos en la sección 4.5.

Lo que haremos será representar la difusión en una dimensión a lo largo del tiempo a través de una matriz $u \in \mathbb{R}^{m \times n}$, (m: cantidad de pasos, n: tamaño de ancho), en la cual en la u_i se guardarán los valores de la difusión en el instante de tiempo i.

¿Cómo obtenemos u? Aprovechando lo estudiado anteriormente, mediante la eliminación gaussiana de un sistema tridiagonal resolveremos la ecuación de difusión de forma discreta. Para cada punto discreto i, u_i se dispersará y atenuará hacia u_{i-1} y u_{i+1}. Como queremos que la magnitud se conserve utilizamos el operador laplaciano, que nos asegura esta conservación ya que la suma de sus coeficientes es $1 - 2 + 1 = 0$, garantizando que el ritmo de atenuación y dispersión sean iguales. Entonces, considerando que el incremento para el paso k dependerá del laplaciano aplicado en la magnitud en este paso $u^{(k)}$, expresamos la ecuación de difusión de la siguiente manera:

$$u_i^{(k)} - u_i^{(k-1)} = \alpha(u_{i-1}^{(k)} - 2u_i^{(k)} + u_{i+1}^{(k)})$$

donde α es el coeficiente de difusión. A partir de un reorden de términos, logramos expresar el sistema de ecuaciones matricial $Au^k = u^{k-1}$, obteniendo la solución de forma implícita. Lo hacemos así:

$$\begin{aligned}
U_i^{(k)} - U_i^{(k-1)} &= \alpha\left(U_{i-1}^{(k)} - 2U_i^{(k)} + U_{i+1}^{(k)}\right) \\
U_i^{(k)} - \alpha\left(U_{i-1}^{(k)} - 2U_i^{(k)} + U_{i+1}^{(k)}\right) &= U_i^{(k-1)} \\
U_i^{(k)} - \alpha U_{i-1}^{(k)} + 2\alpha U_i^{(k)} - \alpha U_{i+1}^{(k)} &= U_i^{(k-1)} \\
\\
-\alpha U_{i-1}^{(k)} + (1+2\alpha)U_i^{(k)} - \alpha U_{i+1}^{(k)} &= U_i^{(k-1)}
\end{aligned}$$

Notemos que nos queda una matriz A tridiagonal, con diagonales $a = [-\alpha, \ldots, -\alpha]$, $b = [1 + 2\alpha, \ldots, 1 + 2\alpha]$, y $c = [-\alpha, \ldots, -\alpha]$, que son los coeficientes obtenidos mediante el método implícito. A tendrá esta pinta:

$$\begin{pmatrix}
1+2\alpha & -\alpha & 0 & 0 & 0 & 0 \\
-\alpha & 1+2\alpha & -\alpha & 0 & 0 & 0 \\
0 & -\alpha & 1+2\alpha & -\alpha & 0 & 0 \\
0 & 0 & -\alpha & 1+2\alpha & -\alpha & 0 \\
0 & 0 & 0 & -\alpha & 1+2\alpha & -\alpha \\
0 & 0 & 0 & 0 & -\alpha & 1+2\alpha
\end{pmatrix}$$

Entonces, resolviendo el sistema de ecuaciones $Ax = d$ para cada paso k, logramos generar la evolución del sistema, donde $x = u^{(k)}$ y $d = u^{(k-1)}$. La matriz final que tendremos representará la evolución de la difusión a lo largo del tiempo mediante un mapa de calor.

3.5.3. Solucion de Problemas de Difusion 2D

Finalmente en la sección 4.6, vamos a realizar una difusión en 2D, en la cual representaremos la difusión en una placa de 15x15 a lo largo del tiempo. Análogamente a la difusión unidimensional, para cada punto discreto (i, j), u_{ij} se dispersará y atenuará hacia $u_{(i-1)j}$, $u_{(i+1)j}$, $u_{i(j-1)}$ y $u_{i(j+1)}$. Para garantizar la conservación, podemos expresar la ecuación de difusión para dos dimensiones considerando nuevamente que el incremento dependerá del laplaciano aplicado en la magnitud en el paso k:.

$$u_{i,j}^{(k)} - u_{i,j}^{(k-1)} = \alpha(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} - 4u_{i,j}^{(k)})$$

Si reordenamos esta ecuación de forma similar a como hicimos con la difusión en una dimensión, podemos expresar u^k en base a u^{k-1}:

$$U_{i,j}^{(k)} - U_{i,j}^{(k-1)} = \alpha(U_{i-1,j}^{(k)} + U_{i+1,j}^{(k)} + U_{i,j-1}^{(k)} + U_{i,j+1}^{(k)} - 4U_{i,j}^{(k)})$$

$$U_{i,j}^{(k)} - \alpha(U_{i-1,j}^{(k)} + U_{i+1,j}^{(k)} + U_{i,j-1}^{(k)} + U_{i,j+1}^{(k)} - 4U_{i,j}^{(k)}) = U_{i,j}^{(k-1)}$$

$$U_{i,j}^{(k)} - \alpha U_{i-1,j}^{(k)} - \alpha U_{i+1,j}^{(k)} - \alpha U_{i,j-1}^{(k)} - \alpha U_{i,j+1}^{(k)} + 4\alpha U_{i,j}^{(k)} = U_{i,j}^{(k-1)}$$

$$\alpha U_{i-1,j}^{(k)} - \alpha U_{i+1,j}^{(k)} + (1+4\alpha)U_{i,j}^{(k)} - \alpha U_{i,j-1}^{(k)} - \alpha U_{i,j+1}^{(k)} = U_{i,j}^{(k-1)}$$

Con lo cual podemos expresar el sistema de ecuaciones de n^2 fórmulas mediante un sistema de ecuaciones matricial de la forma $Au^k = u^{k-1}$:

$$\begin{bmatrix} 1+4\alpha & -\alpha & 0 & -\alpha & 0 & 0 & 0 & 0 & 0 \\ -\alpha & 1+4\alpha & -\alpha & 0 & -\alpha & 0 & 0 & 0 & 0 \\ 0 & -\alpha & 1+4\alpha & 0 & 0 & -\alpha & 0 & 0 & 0 \\ -\alpha & 0 & 0 & 1+4\alpha & -\alpha & 0 & -\alpha & 0 & 0 \\ 0 & -\alpha & 0 & -\alpha & 1+4\alpha & -\alpha & 0 & -\alpha & 0 \\ 0 & 0 & -\alpha & 0 & -\alpha & 1+4\alpha & 0 & 0 & -\alpha \\ 0 & 0 & 0 & -\alpha & 0 & 0 & 1+4\alpha & -\alpha & 0 \\ 0 & 0 & 0 & 0 & -\alpha & 0 & -\alpha & 1+4\alpha & -\alpha \\ 0 & 0 & 0 & 0 & 0 & -\alpha & 0 & -\alpha & 1+4\alpha \end{bmatrix} \begin{bmatrix} u_{1,1}^{(k)} \\ u_{1,2}^{(k)} \\ u_{1,3}^{(k)} \\ u_{2,1}^{(k)} \\ u_{2,2}^{(k)} \\ u_{2,3}^{(k)} \\ u_{3,1}^{(k)} \\ u_{3,2}^{(k)} \\ u_{3,3}^{(k)} \end{bmatrix} = \begin{bmatrix} u_{1,1}^{(k-1)} \\ u_{1,2}^{(k-1)} \\ u_{1,3}^{(k-1)} \\ u_{2,1}^{(k-1)} \\ u_{2,2}^{(k-1)} \\ u_{2,3}^{(k-1)} \\ u_{3,1}^{(k-1)} \\ u_{3,2}^{(k-1)} \\ u_{3,3}^{(k-1)} \end{bmatrix}$$

donde $u^{(k)}$ será un vector de largo n^2, y A tendrá dimensión $n^2 \times n^2$. A diferencia de la difusión 1D, aquí necesitaremos una matriz de coeficientes NO tridiagonal, sino una pentadiagonal, por lo que ya no podremos usar el algoritmo de matrices tridiagonales. En cambio, usaremos el algoritmo EG con Pivoteo. Para construir la matriz A, la diagonal principal estará compuesta de $1 + 4\alpha$ en su totalidad, al igual que las dos diagonales que arrancan en A_{0n} y en A_{n0} (y finalizan en $A_{(n^2-1-n),(n^2-1)}$ y $A_{(n^2-1),(n^2-1-n)}$ respectivamente), las cuales estarán conformadas de principio a fin por $-\alpha$. Luego, las diagonales contiguas a la principal tendrán un comportamiento similar. La superior, estará compuesta del valor $-\alpha$, excepto en las columnas cuyos índices sean múltiplos de n, donde tendrán valor 0. Análogamente, la inferior estará conformada por $-\alpha$, a excepción de las filas cuyos índices sean múltiplos de n, donde también tendremos 0. Estos 0 en las diagonales contiguas que aparecen cada n posiciones, se deben a los casos bordes, ya que para calcular la difusión en esos puntos debemos considerar que la evolución no es en 4 direcciones, si no en 3 o en 2 dependiendo el caso.

Aplicando el mismo procedimiento de antes, mediante el método implícito resolveremos m (instantes de tiempo) sistemas de ecuaciones $Ax = d$, donde $x = u^{(k)}$ y $d = u^{(k-1)}$. Representaremos todos los $u^{(i)}$ en una matriz de dimensiones $m \times n^2$, donde en cada fila representaremos cada placa en el instante i. El caso inicial $u^{(0)}$ será con una fuente de calor en el centro de valor 100 y valor 0 en las demás, y α (coeficiente de difusión) sera 0.1. Posteriormente, ya habiendo calculado $u^{(m)}$ y pasos anteriores, graficaremos el vector en forma de matriz y lo representaremos mediante un mapa de calor mostrando la evolución a partir del tiempo.

Luego, como dato final, mediremos la performance en tiempo de esta simulacion si aplicamos el EG Con Pivoteo y el EG Sin Pivoteo. Guardaremos el tiempo que tarda en calcular la difusión en cada instante de tiempo para cada algoritmo y luego compararemos cada tiempo mediante un boxplot.

4. Resultados y Discusión

4.1. Evaluación de resultados con error numérico

Para este experimento, se resolvió el sistema de ecuaciones $Ax = b$ planteado 1000 veces con 1000 valores distintos de ϵ entre 10^{-6} y 10^{0}. Tras explorar el error numérico de nuestros resultados, al computar el mayor valor de las diferencias absolutas entre la solución esperada y la obtenida, lo comparamos con cada ϵ, obteniendo los siguientes gráficos (Figura 1 y Figura 2):

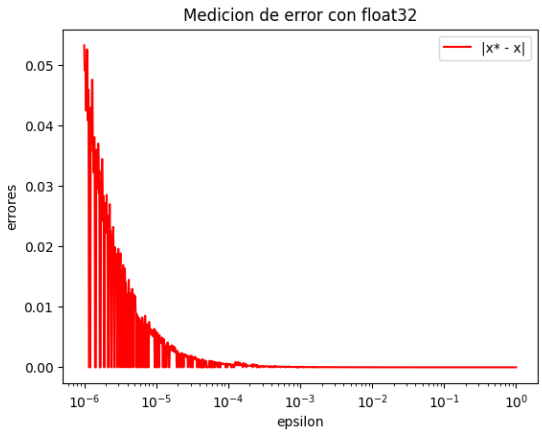

Figura 1: Medición de error con float 32

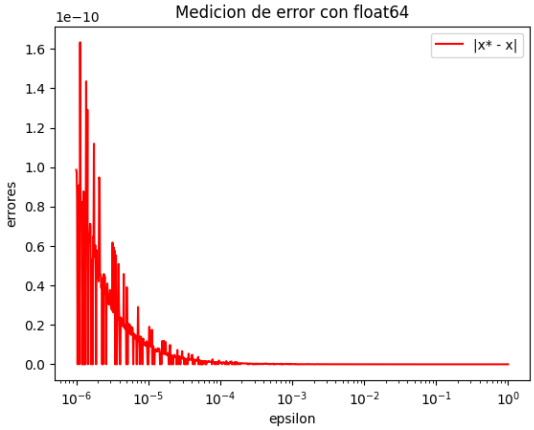

Figura 2: Medición de error con float64

Comparando ambas figuras se puede ver que mientras más pequeño sea el epsilon ϵ, más crecerá el error, es decir, habrá más diferencia entre la solución esperada y la obtenida. Por otra parte, notamos que el grado de error con float64 resulta menor que el de float32. Podemos concluir que esto es debido a una cuestión de precisión del tipo de dato.

4.2. Resolución de ecuaciones diferenciales simples

Se tomaron las derivadas d 4.1, 4.2, y 4.3 y se las representó de forma discreta mediante una array d_i a cada una para cada valor $i = 1$, 2, y 3. Se resolvió la ecuación diferencial para cada una y se obtuvieron los siguientes resultados (Figura 3):

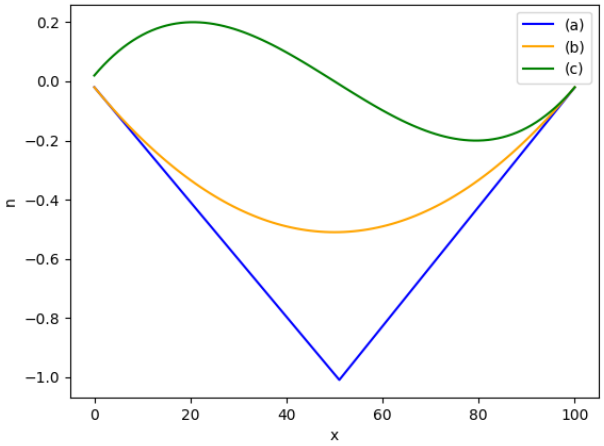

Figura 3: Anti-derivadas obtenidas para los casos 4.1 (a), 4.2 (b), y 4.3 (c)

Las conclusiones de cada resultado fueron las siguientes:

- 4.1 (a): Al tratarse la segunda derivada de una función en forma discreta mediante el concepto de diferencias finitas, en vez de interpretar la antiderivada (a) del gráfico de forma analítica, resulta mejor analizarla por el lado del significado de la segunda derivada. En este caso, cuando la segunda derivada de una función es positiva, esta es cóncava hacia arriba, lo cual se puede ver en este caso en la u (a) obtenida. El salto abrupto de 0 a $\frac{4}{n}$ en d puede explicar el notorio cambio de curvatura en a. Es interesante ver que gracias al metódo utilizado, podemos hallar una antiderivada aproximada de una función que mediante los métodos analíticos convencionales no podríamos.
- 4.2 (b): El valor de la segunda derivada d es constante para todo el dominio, valiendo $\frac{4}{n^2}$. Viendo la u (b) obtenida podemos deducir que se trata de una función cuadrática, ya que la segunda derivada de una cuadrática es un valor constante.
- 4.3 (c): La derivada d que tenemos es una función lineal que depende de i, y por la forma de la u (c) obtenida podemos deducir que se trata de una funcion cúbica, la cual tiene como segunda derivada una función lineal, lo cual explica la forma de d.

Por lo tanto, concluimos que si A es la segunda derivada discreta, u y d son dos funciones expresadas de manera discreta y $Au = d$, entonces podemos afirmar que d es lo que se obtiene tras derivar dos veces u, y que u es la segunda antiderivada de d.

4.3. Comparación de tiempos de cómputo de eliminación gaussiana sobre sistemas tridiagonales

Observamos que en sistemas asociados a una matriz tridiagonal, a medida que aumenta su tamaño, el procedimiento adoptado para estos sistemas tiene una diferencia considerable en tiempos de ejecución. Esta mejora se asocia directamente a la complejidad algorítmica de ambos métodos respecto al tamaño de la entrada n: mientras que el algoritmo EG Con Pivoteo cuesta $\mathcal{O}(n^3)$ operaciones, la optimización EG Tridiagonal tiene un costo lineal $\mathcal{O}(n)$.

Tras haber hecho una comparación entre el tiempo de ejecución de cada algoritmo dado un tamaño de matriz n (Figura 4), vimos que la versión Con Pivoteo nos muestra una recta de pendiente 3, debido a las complejidades mencionadas, ya que tiene un costo cúbico. Lo mismo ocurre con la otra versión, que nos muestra una recta de pendiente 1 (ya que es lineal). Finalmente, podemos concluir que el algoritmo EG para sistemas tridiagonales es mas rápido que el EG Con Pivoteo en estos casos, lo cual hace más ventajoso el uso del EG Tridiagonal siempre que sepamos que tenemos una matriz tridiagonal.

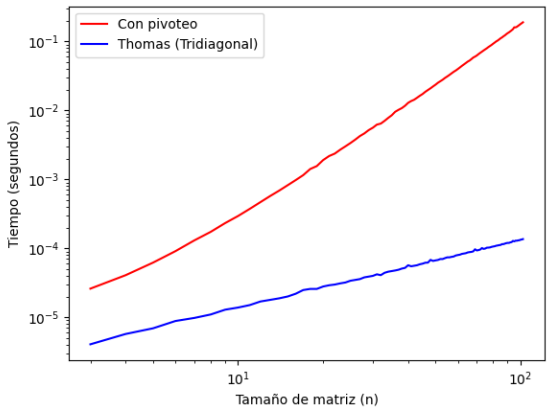

Figura 4: Comparación tiempo de cómputo (en segundos) entre EG Con Pivoteo y EG Tridiagonal

4.4. Análisis de los tiempos de ejecución al emplear eliminación gaussiana con precómputo

Para este experimento se tomó una matriz laplaciana de tamaño 100 (representada mediante 3 vectores a, b, y c) con un término independiente d y se resolvió el sistema desde 10 a 100 veces (tomando en cada iteración el tiempo mas rápido en 20 corridas de cada algoritmo) y se obtuvieron los siguientes resultados:

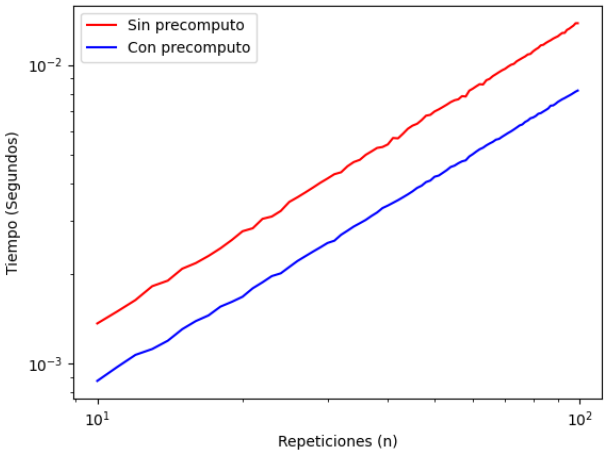

Figura 5: Comparación tiempo de cómputo (en segundos) entre EG Tridiagonal y EG Con Precómputo

Se pueden ver dos rectas pararelas, siendo la inferior el tiempo de computo de EG Con Precomputo en n repeticiones, y la superior lo mismo pero con EG Sin Precomputo. Son paralelas debido a que ambos algoritmos tienen la misma complejidad $\mathcal{O}(n)$, pero se puede apreciar como el EG Tridiagonal Con Precomputo tarda menos en calcular la misma solución una n cantidad de veces a comparacion de la EG Sin Precomputo.

Esto se debe a que la versión con Pre-computo realiza una única vez el paso de triangulación, y luego al ya tener precomputada la matriz triangulada y los factores utilizados, solo tiene que hacer el paso de *backward substitution* n veces (no confundir con la n de la complejidad, nos referimos a n aquí como la cantidad de repeticiones). En cambio, la EG Sin Precomputo en cada una de las n repeticiones tiene que hacer el paso de triangulación y el de *backward substitution*.

Resumidamente, concluimos que si vamos a trabajar con la misma matriz A todo el tiempo y solo vamos a cambiar el término independiente d, la versión con precómputo es más eficiente que hacerlo sin precómputo.

4.5. Simulación de sistemas de difusión de calor en 1D

Para este experimento se tomaron los siguientes parámetros: ($n = 101$, $r = 10$, $m = 1000$, y $\alpha = \{$ 0.1, 0.5, 1, 3, 5 $\}$). Se obtuvieron los siguientes resultados:

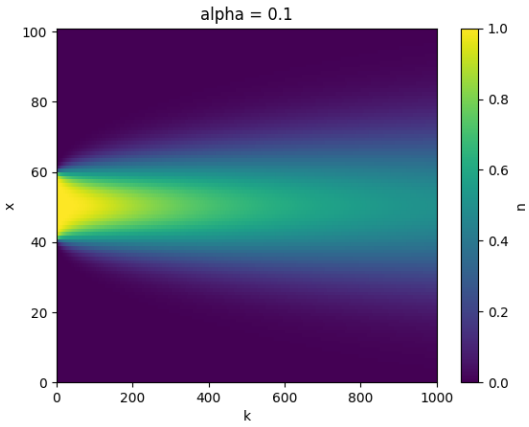

Figura 6: Difusión 1D ($\alpha = 0{,}1$)

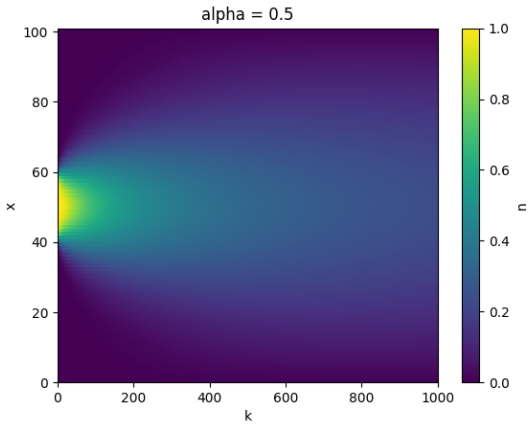

Figura 7: Difusión 1D ($\alpha = 0{,}5$)

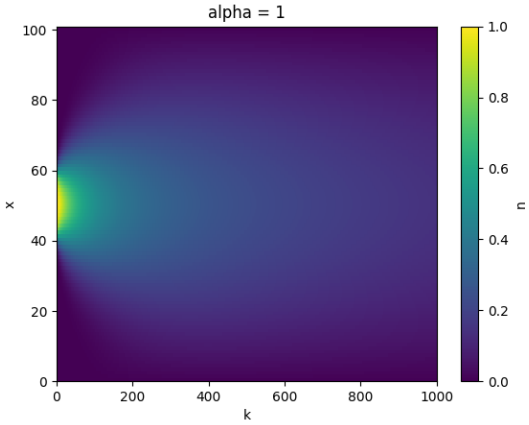

Figura 8: Difusión 1D ($\alpha = 1$)

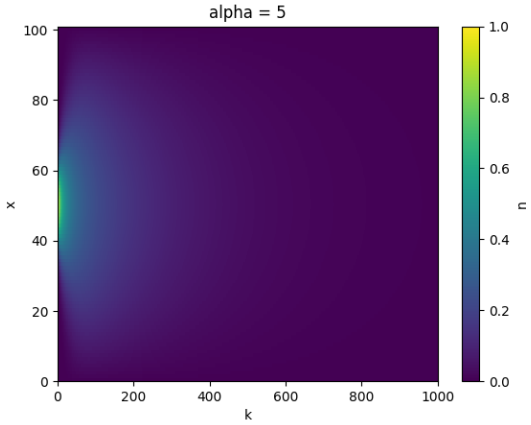

Figura 9: Difusión 1D ($\alpha = 5$)

Analizando cada imagen, podemos concluir que la difusión es proporcional al α, mientras más grande el α mayor es la difusión de calor a lo largo del tiempo, y cuando es más pequeño el α la difusión es menor. Esto explica cómo en cada imagen mientras mas crece α, el calor va quedando menos concentrado en el medio.

4.6. Simulación de sistemas de difusión de calor en 2D

Para este experimento se simuló una difusión 2D con los siguientes parámetros: ($n = 15$, $m = 100$, $\alpha = 0{,}1$, $con\_pivoteo = True$), lo que dió los siguientes resultados para cada instante de tiempo (Figura 10, 11, 12, y 13):

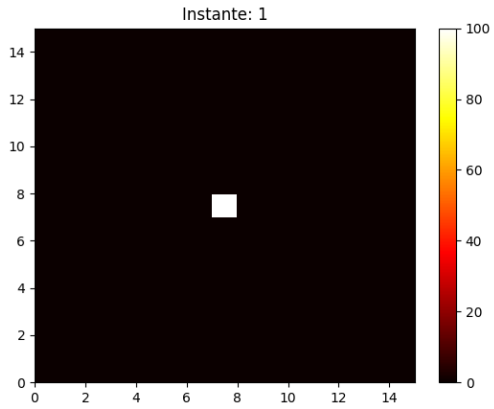

Figura 10: Difusión 2D (Instante 1)

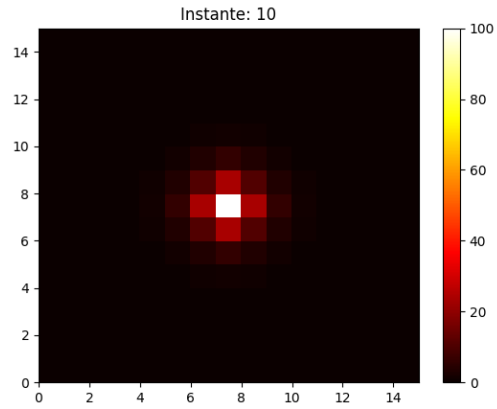

Figura 11: Difusión 2D (Instante 10)

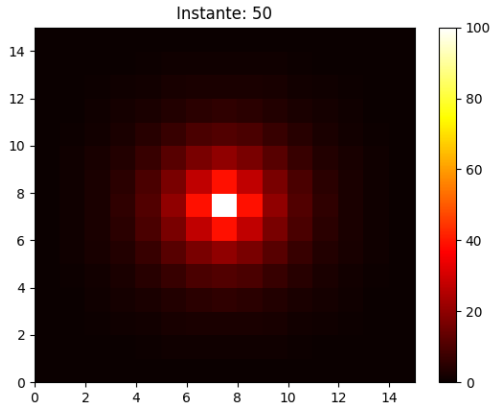

Figura 12: Difusión 2D (Instante 50)

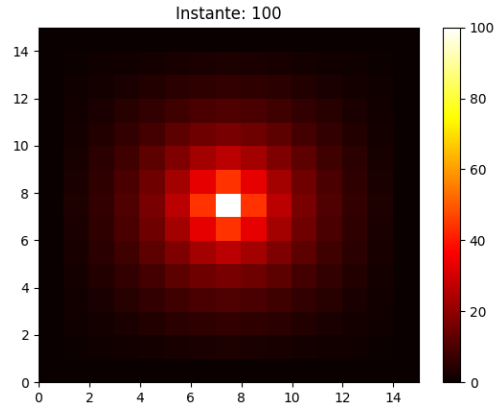

Figura 13: Difusión 2D (Instante 100)

Se puede observar como el calor se propaga hacia los bordes hasta cierto punto, debido a que los bordes se mantienen con temperatura 0 siempre. En experimentos previos se vió que si no se mantenía la fuente de calor en el centro de forma constante el calor no se disipaba, y que si no se mantenían los bordes con temperatura 0 el calor llegaba a ocupar la totalidad de la placa.

Podesmos concluir entonces que la temperatura de los bordes afecta directamente a la distribución final del calor.

Para finalizar, se hicieron dos simulaciones de la difusión con los mismos parámetros, pero una usando Pivoteo y en otra no. Se midieron los tiempos que tomó cada una de las 100 simulaciones de difusión en cada lapso de tiempo y se representaron los resultados en el siguiente boxplot (Figura 14):

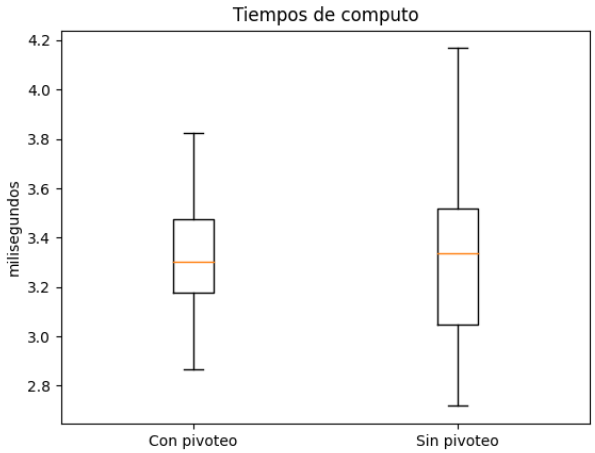

Figura 14: Comparación tiempo de cómputo de difusión entre EG Con Pivoteo y EG Sin Pivoteo

Analizando la distribución del tiempo para EG Con Pivoteo y EG Sin Pivoteo para calcular la difusión, se puede ver como la versión Sin Pivoteo es mas rápida para el cálculo de la difusión. Esto ocurre gracias a que EG Sin Pivoteo no realiza pivoteo en ninguna iteración, por lo que hace un paso menos que EG Con Pivoteo para calcular la difusión.

Podemos concluir que si nos importa menos la precisión de las soluciones y más la velocidad de cómputo, es más beneficioso en cuanto al tiempo utilizar EG Sin Pivoteo, aunque desde el enfoque de nuestra investigación recomendamos utilizar la versión EG Con Pivoteo para no obtener errores inesperados.

5. Conclusiones

Como conclusión, tras los experimentos y las implementaciones realizadas, se puede apreciar el gran poder de cómputo de los sistemas matriciales de ecuaciones para la resolución de problemas de análisis matemático como lo fueron las ecuaciones diferenciales y para la simulación de fenómenos de difusión de calor como se vieron en los últimos experimentos.

Vimos que trabajar con un algoritmo de Eliminación Gaussiana con pivoteo nos permite obtener soluciones mas precisas. Por otro lado, la versión sin pivoteo, aunque nos ofrece un pequeño grado de mayor velocidad y simpleza, puede no hallar solución en casos que esta sea única. Adicionalmente, podría realizarse otra variación del algoritmo en la cual únicamente se realice el pivoteo cuando el valor de la diagonal sea exactamente cero, o menor a un valor predefinido por el usuario, para que él mismo pueda elegir un balance entre precisión de la solución y tiempo de cómputo.

También analizamos cómo varía el grado de error con los tipos de datos float32 y float64, y concluimos que es preferible trabajar con float64 para cálculos que requieran una extrema precisión, pues este tipo produce menor grado de error numérico.

De manera complementaria, durante el desarrollo de este trabajo se realizó una variante del algoritmo de Eliminación Gaussiana tridiagonal que permite un pivoteo entre las filas k y $k+1$, cuyo código quedará adjunto. Además, planteamos la posibilidad de implementar en un futuro una variación de este algoritmo con precómputo, para así maximizar su performance.