DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1

Eliminación Gaussiana, matrices tridiagonales y difusión

Marzo-Abril 2024 Métodos Numéricos

Integrante	LU	Correo electrónico
Begalli, Juan Martin	139/22	juanbegalli@gmail.com
Carrillo, Mariano	358/22	carr.mariano@gmail.com
Serapio, Noelia	871/03	noeliaserapio@gmail.com

Resumen

En el presente trabajo buscamos resolver sistemas de ecuaciones con eliminación gaussiana sin y con pivoteo. Resolver sistemas triangulares aprovechando eficientemente que la matriz es triangular, calculando la eliminación gaussina y la factorización LU para este tipo de matrices. Veremos como utilizar la matriz de Laplaciano para calcular el u cuya derivada segunda es d y experimentaremos con difusión.

Desarrollamos los 4 algoritmos mencionados anteriormente y vimos el error numérico que se comete en el caso de la eliminación gaussiana con pivoteo. Comparamos los algoritmos de eliminación gaussiana con pivoteo vs tridiagonal y además, tridiagonal sin y con precómputo.

Como pensábamos pudimos comprobar en la experimentación que los tiempos de ejecución son mejores en el caso de tridiagonal con precómputo que sin él y a su vez este tiene mejores tiempos que realizar eliminación gaussiana con pivoteo (si se usan matrices tridiagonales). Pudimos deducir y experimentar con la ecuación implícita y explicita de la difusión donde vimos que la implícita siempre se comporta bien, mientras que la otra no.

Palabras Clave: *Eliminación gaussiana, Sistemas tridiagonales, Laplaciano discreto, Difusión*

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

En el presente trabajo práctico utilizaremos el modelo matemático de matrices para abordar los problemas relacionados a sistemas de ecuaciones lineales y su posterior aplicación para la simulación de procesos de difusión y para resolver ecuaciones diferenciales utilizando la matriz tridiagonal del operador laplaciano.

Primero implementaremos el algoritmo de eliminación Gaussiana sin pivote y luego con pivote, continuaremos con el caso de eliminación Gaussiana para matrices tridiagonales y este será nuestro punto de partida para el calculo de difusión. La eliminacion gaussiana es un algoritmo que se utiliza para resolver sistemas de ecuaciones lineales usando matrices. La idea es, representar al sistema como una matriz ampliada:

$$\begin{array}{r} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \vdots \qquad\qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{array} \quad \Rightarrow \quad A = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & b_n \end{array}\right]$$

Luego se busca obtener una matriz con forma escalonada al ir realizando operaciones (multiplicar una fila por un escalar no nulo, intercambiar de posición dos filas, sumar a una fila un múltiplo de otra). Para finalmente despejar las incógnitas sustituyendo hacia atrás. El elemento pivote de una columna especifica es aquel utilizado para colocar ceros por debajo de él [1]. En alguna implementación del algoritmo de eliminación gaussiana sin pivoteo, es decir sin intercambio de filas, puede ocurrir que nuestro algoritmo no encuentre solución aunque sí exista. Es por esto que se plantea otra versión realizando un pivoteo parcial entre filas, donde el pivote será el valor máximo de la columna en la cual estamos parados en ese momento. En ambas implementaciones la cantidad de operaciones elementales es del orden $O(n^3)$

Existe otro tipo de sistemas lineales que vale la pena analizar, son las llamadas **matrices tridiagonales**, que se ven así:

$$a_i x_i + b_i x_i + c_i x_{i+1} = d_i, \quad i = 1, 2, \ldots, n$$

$$\begin{bmatrix} b_1 & c_1 & 0 & \ldots & 0 \\ a_2 & b_2 & c_2 & \ldots & 0 \\ 0 & a_3 & b_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \ldots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

En este tipo de matrices buscamos resolver el sistema utilizando eliminación gaussiana, pero haciendo un uso inteligente de las operaciones para evitar hacer cuentas innecesarias, tomando solo 3 vectores que serian los de la tridiagonal. Para esto, emplearemos técnicas como la factorización LU y de esa forma resolveremos el sistema con complejidad $O(n)$ [3]. Es por esa razón que se plantea una versión modificada del algoritmo de eliminación gaussiana.

Una aplicación del algoritmo de eliminación gaussiana para sistemas tridiagonales es la de encontrar el vector solución de una ecuación diferencial. En este caso, nos basamos en que el llamado **Laplaciano**, cuando trabajamos en una dimensión, es equivalente a la derivada segunda. Buscaremos, entonces, resolver $\frac{d^2}{dx^2}u = d$ como:

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

Planteando la matriz, nos queda un sistema tridiagonal donde buscamos para cada d el vector u:

$$\begin{bmatrix} -2 & 1 & 0 & \ldots & 0 \\ 1 & -2 & 1 & 0 & \vdots \\ 0 & 1 & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \ldots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

A su vez, una motivación por la cual resulta interesante el desarrollo de estos algoritmos es que podemos utilizar estas herramientas para la simulación de un proceso estocástico de **difusión**. Pues este proceso modela como una entidad se difunde, típicamente, de un lugar de mayor concentración a uno de menos. Entonces, si modelamos la evolución promedio de unas partículas desde una posición inicial en función del tiempo, nos interesa encontrar dicha función. Podemos hacerlo si resolvemos un sistema tridiagonal usando eliminación gaussiana[1]. Los detalles sobre el desarrollo e implementación están explicados en la sección 2.5.

[1] https://hplgit.github.io/fdm-book/doc/pub/diffu/pdf/diffu-4print.pdf

2. Desarrollo

2.1. Eliminación gaussiana

Se implementaron algoritmos para, dada una matriz $A \in R^{n \times n}$ y un vector $b \in R^n$ resolver el sistema lineal $Ax = b$. Para ello, utilizamos *Python* haciendo uso de las características propias del lenguaje (sin librerías externas) para la implementación base. Las implementaciones se encuentran en los archivos *eliminacionGaussianaPivot.py* y *eliminacionGaussiana.py*

2.1.1. Sin pivoteo

Para implementar esta variante de la eliminación gaussiana primero tuvimos que pensar en los casos donde no hay solución al sistema (sea porque no existe o hay infinitas soluciones). Por eso, en primer lugar, chequeamos que el tamaño de A y de b coincidan, si no se levanta una excepción pues el sistema está mal definido. Luego de construir la matriz ampliada agregando el vector b a A, entramos al loop principal del algoritmo. En él tenemos que tener en cuenta si el elemento pivot de la iteración i (i.e el elemento de la diagonal i) es igual a 0, pues no podremos dividir por él. En ese caso, es necesario que el resto de la columna i sea 0 (para i+1 $\leq j \leq n$, $j = 0$), caso contrario el algoritmo concluye que el sistema no tiene solución. Es importante aclarar que esto sucede porque es un algoritmo de eliminación gaussiana **sin pivoteo**, pues como explicamos en la siguiente sección que haya un 0 en el elemento pivote no quiere decir que no haya solución. Es por eso que esta implementación es bastante limitada en cuanto a resultados.

Otra parte importante a considerar sobre la implementación radica en el hecho de que el algoritmo hace uso de múltiples divisiones y multiplicaciones. Debemos notar que este hecho genera la posibilidad de que haya errores numéricos. Estos pueden darse, por ejemplo, si dividimos por valores cercanos al 0, sumamos números de diferente magnitud o si se da una *cancelación catastrófica*. No obstante, en esta parte del trabajo no nos preocupamos por dichos errores. Otra cosa importante una vez finalizada la triangulación de la matriz es que debemos chequear si se llego a un sistema compatible indeterminado o a un sistema incompatible, si se diera cualquiera de esos casos se levanta una excepción que indica que existen infinitas soluciones o ninguna, respectivamente.

La ultima parte del algoritmo consiste en despejar una de las incógnitas (la de la ultima fila) y realizar una sustitución hacia atrás de manera progresiva para ir despejando el resto de las incógnitas. Y cada incógnita despejada la guardamos en un vector que será el valor de retorno de nuestro algoritmo.

Pseudocodigo del algoritmo implementado:

in: $a \in R^{n \times n}$, $b \in R^n \Rightarrow$ **out:** $x \in R^n$

eliminacionGaussiana_sinPivoteo

1: $t_A \leftarrow tam(a)$, $t_b \leftarrow tam(b)$, $ampliarMatriz(a, b)$
2: **for** $i \leftarrow 1, ..., n$ **do**
3: **if** $\forall i \leq k \leq n\ a_{ki} = 0$ **then**
4: **no hay solución**
5: **else**
6: **seguir**
7: **end if**
8: **for** $j \leftarrow i + 1, ..., n$ **do**
9: $c \leftarrow a_{ji}/a_{ii}$
10: $Fila_j \leftarrow Fila_j - c \times Fila_i$
11: **end for**
12: **end for**
13: **if** $a_{nn} = 0$ **then**
14: **no hay solución única**
15: **end if**
16: $x_n \leftarrow a_{t_A, t_A+1}/a_{t_A, t_A}$
17: **for** $i \leftarrow n - 1, ..., 1$ **do**
18: sustitucion $= \sum_{j=i+1}^{t_A} a_{ij} x_j$
19: $x_i = (a_{i,n+1} -$sustitucion$)/a_{ii}$
20: **end for**
21: $resFinal \leftarrow (x_1, ..., x_n)$
22: **devolver** $resFinal$

El algoritmo no encontrara solución por ejemplo si tenemos el siguiente sistema (el caso de tests es el *test_error_4* del archivo *tests_1a.py*) mientras que si encontrara solución si se aplica el algoritmo de eliminación gaussina con pivoteo como veremos más adelante.

$$\begin{bmatrix} 0 & -2 & 1 \\ 2 & 3 & 1 \\ 3 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -14 \\ 1 \\ 1 \end{bmatrix}$$

2.1.2. Con pivoteo parcial

Basándonos fuertemente en nuestra implementación del algoritmo de eliminación gaussiana sin pivoteo pudimos implementar su variante con pivoteo. Para eso, antes de entrar a la parte principal del algoritmo donde se realiza la eliminación hacia adelante, se busca el elemento máximo de la columna. En principio creímos que bastaba con hacer pivoteo solamente en el caso donde hubiese un 0 en la diagonal en cuyo caso era obligatorio realizarlo (de lo contrario incurriríamos en el error del algoritmo anterior). No obstante, ya que la consigna lo requería, se buscó el máximo de la columna en cada iteración.

Otra parte importante a considerar sobre nuestra implementación es la de la detección de posibles errores. Debido a la aritmética finita con la que se trabaja, se introdujo un número límite de manera tal de que cada vez que fuéramos a dividir por un número se chequeara que estuviera por arriba de ese rango, de lo contrario se activaba un flag. En primera instancia establecimos un valor arbitrario para el ϵ considerado como tolerancia, pero luego decidimos introducirlo como parámetro de la función, para tener mayor libertad y comodidad a la hora de testear. A su vez, corroboramos que en los sistemas donde no existiesen soluciones o hubiesen infinitas [3] nuestro algoritmo lo detecte correctamente.

Pseudocodigo del algoritmo:

in: $a \in R^{n \times n}$, $b \in R^n \Rightarrow$ **out:** $x \in R^n$

eliminacionGaussiana_conPivoteo

1: $t_A \leftarrow$ longitud(a);$t_b \leftarrow$ longitud(b); $ampliarMatriz(a, b)$
2: **for** $i \leftarrow 1,...,t_A - 1$ **do**
3: **if** $a_{ii} = \text{máx}_{i \leq j \leq n}(|a_{ji}|)$ **then**
4: **seguir**
5: **else**
6: $filaPivot \leftarrow$a$_k$ con $a_k = \text{máx}_{i \leq j \leq n}(|a_{ji}|)$
7: $actual \leftarrow a_i$; $nuevo \leftarrow filaPivot$
8: $a_i \leftarrow nuevo$; $filaPivot \leftarrow actual$
9: **end if**
10: **for** $j \leftarrow i + 1$ **to** t_A **do**
11: $c \leftarrow a_{ji}/aii$
12: $Fila_j \leftarrow Fila_j - c \times Fila_i$
13: **end for**
14: **end for**
15: **if** $a_{nn} = 0$ **then**
16: **no hay solución única**
17: **end if**
18: $x_n \leftarrow a_{t_A, t_A+1}/a_{t_A, t_A}$
19: **for** $i \leftarrow n - 1, ..., 1$ **do**
20: sustitucion $= \sum_{j=i+1}^{t_A} a_{ij} x_j$
21: $x_i = (a_{i,n+1} -$sustitucion$)/a_{ii}$
22: **end for**
23: $resFinal \leftarrow (x_1, ..., x_n)$
24: **devolver** $resFinal$

El siguiente sistema no tiene solución, y se puede ver la excepción que devuelve al ejecutar *tests_2a.py* y corresponde al caso *test_error_5*.

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 3 & 3 \\ 4 & 8 & 12 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

También se realizó un análisis sobre el error entre la solución esperada x y la entregada por nuestra implementación ($\hat{x}$). El análisis comparó la diferencia entre realizar operaciones con números de tipo flotante de 32 y 64 bits, usando como métrica la norma infinito ($\|\hat{x} - x\|_\infty$). La comparación se reflejó en un gráfico generado utilizando *Pyplot*. Para ello resolvimos el sistema $Ax = b$ conociendo los 3 vectores, pero variando elementos de la matriz A utilizando varios $\epsilon \in [10^{-6}, 10^0]$ espaciados logaritmicamente (se utilizó la función *logspace* de NumPy para generarlos). Este fue el esquema a seguir:

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

Generamos un total de 30 valores para ϵ pues intentamos con otras cantidades y esta nos pareció que era lo suficientemente representativa. No obstante, también experimentamos con 100 valores de ϵ para reforzar nuestras hipótesis y corroborar los resultados. Con esta experimentación esperábamos que el error en las soluciones entregadas por nuestra implementación, operando

con números de punto flotante de 64 bits, fuese bastante menor al error con flotantes de 32 bits. Pues consideramos que los principales errores iban a darse al realizar divisiones y multiplicaciones entre los números, y al tener un mayor margen de representación con 64 bits que con 32 bits el error sería menor.

Debido a que Python por defecto opera con números de punto flotante de 64 bits y a que nuestra implementación utiliza las listas nativas del lenguaje (no un *array* de *NumPy*) debimos utilizar lo que se conoce como *list comprehension*[2] para operar con flotantes de 32 bits. Pues fue la solución que encontramos para no tener que reescribir todo el algoritmo de eliminación gaussiana.

2.2. Sistemas tridiagonales

La implementación del algoritmo para matrices tridiagonales se encuentra en el archivo *eliminacionGaussianaTridiagonalVectores.py*. La variante con precomputo se encuentra en el archivo *LUTridiagonalVectores.py*

2.2.1. Eliminación Gaussiana para sistemas tridiagonales

Para realizar la implementación del algoritmo que resuelve sistemas tridiagonales, nos basamos fuertemente en el método de eliminación Gaussiana. Sabiendo que la matriz es tridiagonal, pudimos realizar solo las operaciones necesarias, dado que los elementos relevantes para resolver el sistema se encuentran en las 3 diagonales, las cuales pueden ser representadas cada una con un vector, o sea, que para resolver un sistema triadigonal podemos solo centrarnos en los 3 vectores de las diagonales principales más el vector d.
Si tenemos el sistema tridiagonal los vectores que utilizara nuestro algoritmo serán: $a = [0, a_2, a_3, ..., a_{n-1}, a_n]$,
$b = [b_1, b_2, b_3, ..., b_{n-1}, b_n]$, $c = [c_1, c_2, c_3, ..., c_{n-1}, 0]$. Para resolver un sistema triadiagonal se concatena el vector columna d a la derecha de la matriz generando una matriz aumentada. Decidimos implementar eliminación gaussiana con pivoteo parcial si algún elemento de la diagonal (i.e. el vector b) es cero, para tal motivo vamos a usar un vector extra que llamaremos c_c, el cual tendrá elementos diferentes de cero si viene de un switch entre filas y el valor de c_i por el que se switchea es diferente de cero, la matriz aumentada si consideramos este nuevo vector se vería como:

$$\begin{bmatrix} b_1 & c_1 & c\_c_1 & 0 & \dots & d_1 \\ a_2 & b_2 & c_2 & c\_c_2 & \dots & d_2 \\ 0 & a_3 & b_3 & \ddots & \vdots & d_3 \\ \vdots & \ddots & \ddots & \ddots & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c\_c_{n-2} & d_{n-2} \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} & d_{n-1} \\ 0 & \dots & 0 & a_n & b_n & d_n \end{bmatrix}$$

Luego el algoritmo viéndolo de un lado teórico se puede dividir en 2 etapas como la eliminación gaussiana tradicional. En la primer etapa, se busca obtener una matriz escalonada, para ello en este caso en particular:

- Si el elemento b_i es cero y el a_{i+1} es distinto de cero se hace un switch entre la fila i y la $i+1$ de la matriz aumentada, esto hará que el nuevo elemento que estará debajo de la diagonal (i.e. el a_{i+1}) sea cero, con lo cual ya no es necesario hacer nada mas en el paso i para este caso.
- Si b_i es distinto de cero, generamos un 0 en a_{i+1} realizando la siguiente operación en la fila $i+1$ $f(i+1) = f(i+1) - a_{i+1}/b_i * f(i)$. Lo que que se traduce en realizar las siguientes operaciones $a_{i+1} = a_{i+1} - a_{i+1}/b_i * b_i = 0$, $b_{i+1} = b_{i+1} - a_{i+1}/b_i * c_i$, $c_{i+1} = c_{i+1} - a_{i+1}/b_i * c\_c i = c_{i+1}$ pues $c\_c i$ sera cero si estamos en este caso y $d_{i+1} = d_{i+1} - a_{i+1}/b_i * d_i$.

El segundo paso, ya teniendo la matriz escalonada, se despejan las incógnitas, desde la posición x_n para atrás. Donde $x_n = d_n/b_n$, $x_{n-1} = [d_{n-1} - c_{n-1} * x_n]/b_{n-1}$ y los x_i con $n-1 > i \geq 1$ se calculan utilizando los valores anteriormente despejados, donde $x_i = [d_i - (c_i * x_{i+1} + c\_c_i * x_{i+2})]/b_i$.

Pseudocodigo del algoritmo:

[2] https://docs.python.org/3/tutorial/datastructures.html

in: $a, b, c, d \in R^n \Rightarrow$ **out:** $x \in R^n$

eliminacionGaussiana_tridiagonal

```
1:  t_d ← longitud(d) ; t_b ← longitud(b)
2:  for i ← 1, ..., t_d do
3:      if b_i = 0 then
4:          if a_{i+1} ≠ 0 then
5:              swap(b_i, a_{i+1}); swap(b_{i+1}, c_i); swap(d_i, d_{i+1})
6:              c_c_i ← c_{i+1}; c_{i+1} ← 0
7:          end if
8:      else
9:          cociente ← a_{i+1}/b_i
10:         a_{i+1} ← 0
11:         b_{i+1} ← b_{i+1} − cociente × c_i
12:         d_{i+1} ← d_{i+1} − cociente × d_i
13:     end if
14: end for
15: x_{t_d} ← d_{t_d}/b_{t_b}
16: x_{t_d−1} ← (d_{t_d−1} − c_{t_d−1} × x_{t_d})/b_{t_b−1}
17: for i ← t_d − 2, ..., 1 do
18:     x_i ← [(d_i − (c_i × x_{i+1} + c_c_i × x_{i+2})]/b_i
19: end for
20: res ← (x_1, ..., x_n)
21: devolver res
```

2.2.2. Factorización LU para sistemas tridiagonales

Una forma más eficiente de aplicar el algoritmo de eliminacion gaussiana para una matriz tridiagonal se basa en el precómputo de su factorización LU. Decidimos realizar la factorización LU de la matriz teniendo en cuenta que es tridiagonal y solo realizando las operaciones que tienen relevancia (i.e solo operaciones sobre los vectores a, b y c). Cabe destacar que utilizando esta técnica, al tener precomputados los vectores a,b,c creemos que el sistema se puede resolver más rápido a que si no tuviéramos este precómputo. Vamos a querer escribir $A \in R^{n\times n}$ como el producto de $L \in R^{n\times n}$ triangular inferior con unos en la diagonal por $U \in R^{n\times n}$ triangular superior. Es decir, $A = LU$. Resolver el sistema $Ax = b$ es equivalente a $LUx = b$ que se puede resolver en dos pasos:

$$Ly = b$$
$$Ux = y$$

Deseamos matrices tales que:

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & 0 \\ 0 & a_3 & b_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \dots & 0 & a_n & b_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_2 & 1 & 0 & \dots & 0 \\ 0 & l_3 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & l_n & 1 \end{bmatrix} \begin{bmatrix} u_1 & c_1 & 0 & \dots & 0 \\ 0 & u_2 & c_2 & \dots & 0 \\ 0 & 0 & u_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \dots & 0 & 0 & u_n \end{bmatrix} =$$

$$\begin{bmatrix} u_1 & c_1 & 0 & \dots & 0 \\ l_2u_1 & l_2c_1 + u_2 & c_2 & \dots & 0 \\ 0 & l_3u_2 & l_3c_2 + u_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \dots & 0 & l_nu_{n-1} & l_nc_{n-1} + u_n \end{bmatrix}$$

Donde despejando se obtiene:

$$u_1 = b_1 \qquad u_2 = b_2 - l_2c_1 \qquad u_3 = b_3 - l_3c_2 \qquad u_{n-1} = b_{n-1} - l_{n-1}c_{n-2} \qquad u_n = b_n - l_nc_{n-1}$$
$$l_2 = a_2/u_1 \qquad l_3 = a_3/u_2 \qquad l_4 = a_4/u_3 \qquad l_n = a_n/u_{n-1}$$

lu_tridiagonal_vectores realizara el precómputo de los vectores a, b y c (i.e. se realiza la factorización LU en este método) en los mismos vectores, lo que significa que no se utilizara memoria extra. Donde en el vector a quedara guardado los valores correspondientes a l, b corresponderá a los valores de u, mientras que se c no cambia.
Luego teniendo la factorización LU resolvemos el sistema en dos pasos haciendo:

$$Ly = b$$
$$Ux = y$$

En el código esto se realiza con el método **resolverSistemaLU** el cual se encarga de resolver el sistema en cuestión, llamando a los métodos **obtenerYDeMatrizL** que resuelve la primer ecuación y luego se invoca a **obtenerXDeMatrizU** que resuelve la segunda ecuación. Dichos métodos están optimizados y solo realizan operaciones con los vectores a, b, c y d, no con la matriz completa (ya que fuera de los vectores de la triadiagonal la matriz vale cero).

Pseudocodigo del algoritmo:

inout: $a, b, c \in R^n$

lu_tridiagonal_vectores

1: $t_a \leftarrow longitud(a)$; $t_b \leftarrow longitud(b)$;
2: $t_c \leftarrow longitud(c)$
3: **if** $t_a = t_b$ and $t_b = t_c$ **then**
4: **for** $k \leftarrow 1, ..., t_a$ **do**
5: **if** $b_k - 1 \neq 0$ **then**
6: $a_k \leftarrow a_k / b_{k-1}$
7: $b_k \leftarrow b_k - a_k * c_{k-1}$
8: **end if**
9: **end for**
10: **end if**

in: $b, c, y \in R^n \Rightarrow$ **out:** $res \in R^n$

obtenerXDeMatrizU

1: $tamB \leftarrow longitud(b)$; $tamC \leftarrow longitud(c)$; $tamY \leftarrow longitud(y)$
2: **if** $b_{tamB-1} \neq 0$ **then**
3: $y_{tamY-1} \leftarrow y_{tamY-1} / b_{tamB-1}$
4: **end if**
5: **for** $\leftarrow tamB - 2, ..., 0$ **do**
6: **if** $b_i \neq 0$ **then**
7: $y_i \leftarrow (y_i - c_i * y_{i+1}) / b_i$
8: **end if**
9: **end for**
10: $res \leftarrow (y_1, ..., y_n)$
11: **devolver** res

in: $a, b, c, d \in R^n \Rightarrow$ **out:** $x \in R^n$

resolverSistemaLU

1: y ← obtenerYDeMatrizL(a, d)
2: x ← obtenerXDeMatrizU(b, c, y)
3: **devolver** x

in: $a, d \in R^n \Rightarrow$ **out:** $res \in R^n$

obtenerYDeMatrizL

1: **for** $k \leftarrow 1, ..., longitud(A)$ **do**
2: $y_i \leftarrow d_i - a_i * d_{i-1}$
3: **end for**
4: $res \leftarrow (y_1, ..., y_n)$
5: **devolver** res

2.3. Laplaciano Discreto

Con el objetivo de probar la implementación del algoritmo de eliminación gaussiana para sistemas tridiagonales resolvermos la ecuación diferencial para 3 vectores d. Para ello, utilizamos la matriz tridiagonal del operador laplaciano unidimensional(∇^2) pues como explicamos en la sección 1, esta matriz (L) equivale a la $\frac{d^2}{dx^2}$. Decidimos utilizar la eliminación Gaussiana con precómputo para hallar soluciones al sistema $Lu = d$. Aquí utilizaremos tres vectores d, los tres de tamaño $n = 101$.

$$d_i^a = \begin{cases} \frac{4}{n} & \text{si } i = (n/2) + 1 \\ 0 & \text{si no} \end{cases}$$

$$d_i^b = \frac{4}{n^2}$$

$$d_i^c = (-1 + 2i/(n-1))12/n^2$$

$$\mathrm{L} = \begin{bmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \vdots \\ 0 & 1 & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 1 & -2 \end{bmatrix}$$

Los u hallados equivalen a encontrar $\frac{d^2}{dx^2}u = d$. Es decir, al resolver estos sistemas de ecuaciones obtenemos una aproximación de la función cuya derivada segunda es d. En la figura 3.2 se graficaron las funciones cuya derivada segunda era d.

2.4. Tiempos de cómputo

En este caso realizamos dos comparaciones de tiempos:

En la primera comparamos los tiempos mínimos de usar Eliminación Gaussiana con pivoteo y Eliminación Gaussiana en un sistema tridiagonal, usando en ambos casos la matriz laplaciana y un vector, d, generado aleatoriamente utilizando *NumPy*. Para los test usamos diferentes tamaños de matrices que van desde 1x1 hasta 50x50, luego usamos timeit para medir los tiempos, generando 20 repeticiones con cada tamaño. A partir de ahí tomamos los valores mínimos dados por el timeit para luego graficarlos en función del tamaño de la matriz. En este caso suponemos que usando Eliminación Gaussiana con pivoteo debería tener una complejidad mayor al otro y mientras más aumenta el tamaño de la matriz más diferencia habría en los tiempos de ejecución. La figura 3.3.1 posee los resultados de dicha comparación en escala logarítmica.

En la segunda, decidimos comparar los tiempos entre la implementación de Eliminación Gaussiana para un sistema tridiagonal y su variante con precómputo. Una cosa a tener en cuenta es que medimos el tiempo que nos tomó precomputar la matriz que fuimos reutilizando a lo largo de las iteraciones (solo en la variante que lo utiliza), el cual fue sumado únicamente en la primera iteración. Por esa razón, podemos suponer que en la primera iteración los tiempos serán similares pero luego la variante con precómputo será más rápida, pues la implementación normal debe volver a realizar muchas operaciones que en realidad no son necesarias.

En esta última comparación nos encontramos con el problema de que era necesario trabajar con matrices bastante grandes para lograr resultados que nos permitiesen llegar a conclusiones concretas. A su vez, también debimos ajustar los parámetros *repeat* y *number* de las funciones *timeit*, ya que si en cada repetición corríamos nuestro algoritmo pocas veces, no estábamos seguros de que los tiempos recibidos fuesen lo suficientemente representativos. Por esta razón, es que debimos realizar muchos experimentos para medir los tiempos de computo, hasta encontrar los parámetros adecuados. Observamos que al tener que utilizar funciones auxiliares como *np.allclose* para realizar comparaciones entre números de punto flotante, nuestra implementación se vio altamente enlentecida con los parámetros que generaron los resultados presentados.

2.5. Difusión

A partir de la ecuación de difusión discreta a la cual expresamos de manera que el incremento en el paso k, $u^{(k)} - u^{(k-1)}$ es una fracción α del operador laplaciano aplicada a $u^{(k-1)}$

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

Podemos obtener, distribuyendo α y despejando $u_i^{(k)}$, de forma explícita el estado en el paso k en función del estado k - 1.

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

Así deducimos la matriz tridiagonal A $\in R^{n\times n}$ tal que $u^{(k)} = Au^{(k-1)}$:

$$\text{A}= \begin{bmatrix} -2\alpha+1 & -\alpha & 0 & \dots & 0 \\ -\alpha & -2\alpha+1 & -\alpha & 0 & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -\alpha & -2\alpha+1 & -\alpha \\ 0 & \dots & 0 & -\alpha & -2\alpha+1 \end{bmatrix}$$

Gracias a esto podemos observar como α es un factor importante en cuanto al control de la tasa de difusión. Supusimos que a mayor valor de α más rápida es la difusión (difunde más partículas en una unidad de tiempo), mientras que un valor menor significa una difusión más lenta.

Para simular la difusión y corroborar nuestras hipótesis, necesitamos una condición inicial $u^{(0)}$ (vector de tamaño 101 en este caso) para comenzar a iterar. Los elementos de $u^{(0)}$ los inicializamos de forma tal:

$$u_i^{(0)} = \begin{cases} 1 & \text{si } 40 < i < 60 \\ 0 & \text{si no} \end{cases}$$

Así obtenemos un vector donde se tiene una alta concentración en el centro y nula en los bordes.

Luego se computó la difusión para 1000 iteraciones de $u^{(k)} = Au^{(k-1)}$ partiendo de la condición inicial de $u^{(0)}$, utilizando distintos valores de α. Así obtuvimos los valores que aparecen en la figura 4. Durante la experimentación usamos diferentes valores para α en el caso explicito, ya que veíamos que obteníamos diferentes resultados a los esperados y con los valores elegidos nos pareció que se expresaban mejor esos cambios inesperados.

Otra manera es, a partir de la ecuación discreta de la difusión, despejar $u^{(k-1)}$ y así obtener la ecuación implícita:

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

Deduciendo así la matriz tridiagonal A $\in R^{n\times n}$ tal que A$u^{(k)} = u^{(k-1)}$:

$$A = \begin{bmatrix} 2\alpha+1 & -\alpha & 0 & 0 & 0 \\ -\alpha & 2\alpha+1 & -\alpha & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & -\alpha & 2\alpha+1 & -\alpha \\ 0 & 0 & 0 & -\alpha & 2\alpha+1 \end{bmatrix}$$

Luego computando la difusión, de la misma manera que con la ecuación implícita, ahora lo hacemos con la implícita obteniendo los resultados que aparecen en la figura 3.4.

3. Resultados y discusión

3.1. Eliminacion Gaussiana

Tal como se mencionó en la sección 2.1.2 luego de haber implementado el algoritmo de eliminación gaussiana con pivoteo, se realizó un análisis del error numérico entre la solución esperada y la entregada por la implementación. Para ello, utilizamos números de punto flotante de 32 bits y 64 bits. Tal como habíamos supuesto, los resultados muestran que el error en números de 32 bits es mayor al error en 64 bits. Y a su vez, tal como se observa en la figura 2 hay una linealidad entre el valor de ϵ y $\|\hat{x} - x\|_\infty$, pues cuanto más chico es el ϵ más grande fue el error encontrado. Esto es debido a que al sumar o restar valores tan chicos todas las operaciones acarrean más error (por trabajar con aritmética finita), y por lo tanto, en los sucesivos pasos de la eliminación gaussiana el error se irá acumulando.

En relación a esto último, el origen del error está en que al querer sumar o restar dos numeros de diferente magnitud (en este caso, un ϵ muy chico con un entero) tienen que coincidir los exponentes por como se estructuran los números de punto flotante (su mantisa). Para hacerlos coincidir se requiere hacer *shifts* del exponente, por lo que si realizó muchos, es probable que pierda digitos y en consecuencia precisión. Entonces, podríamos decir que cuantos más bits tenga nuestro número, mayor margen para realizar *shifts* tengo, y pierdo menos precisión.

Sin embargo, al experimentar, nos dimos cuenta que existían ciertos valores de ϵ que terminaban generando un error mucho menor al esperado (basandonos en la visible linealidad que sigue el gráfico). La razón por la que estos *outliers* existen se debe a que la resta o suma de ciertos valores de ϵ se *comportan bien* con los valores enteros iniciales. Es decir, ocurre que los coeficientes iniciales de la matriz (una vez aplicados los ϵ) no acarrean tanto error al operar entre ellos, como sí lo pueden hacer valores cercanos a ellos en otra iteración con un ϵ diferente de magnitud similar.

A su vez, nos parece pertinente observar que estos *outliers* se observan en mayor cantidad para las operaciones realizadas en 32 bits que en 64. Si comparamos el grafico de la izquierda que utiliza una muestra de 30 epsilons con el de la derecha que utiliza 100, se observa que aumenta la cantidad de estos en ambas implementaciones. Es decir, debimos generar una muestra más grande de ϵ para que en la implementación de 64 bits hubiese más casos borde, lo que nos habla de una mayor *estabilidad* por parte de esta representación.

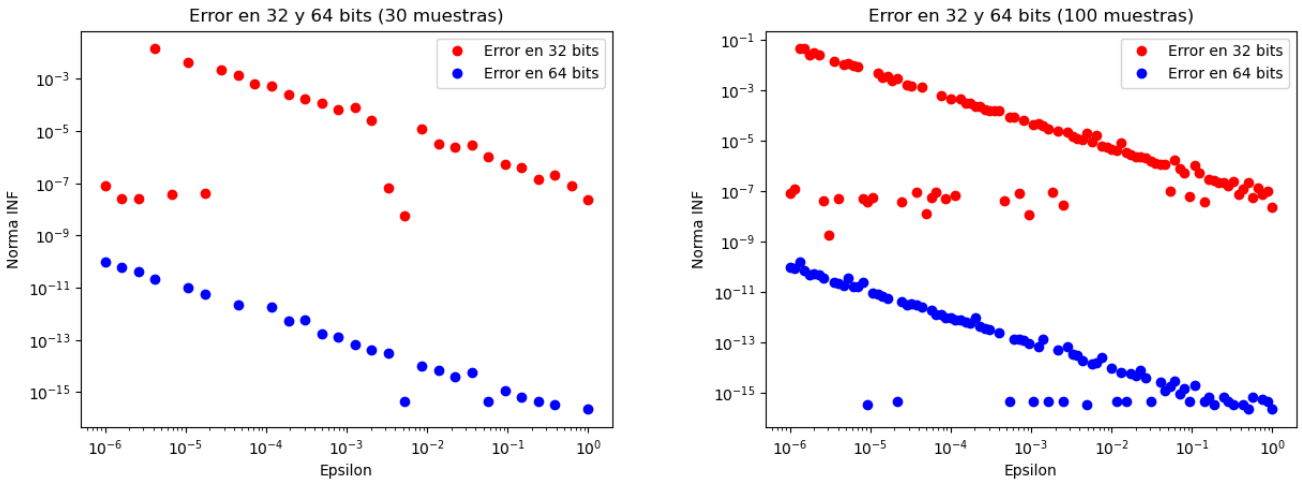

Figura 1: Error numérico en eliminación gaussiana

3.2. Laplaciano Discreto

El codigo de este grafico se encuentra en el archivo *graficoEcuacionDiferencial.py*

Al resolver, hallamos funciones aproximadas cuyas derivadas segundas se correspondían con 3 vectores d_a, d_b, d_c. En el gráfico aparecen las tres funciones halladas (u^a, u^b, u^c), en el cuál elegimos las X en el rango desde 0 a 100.

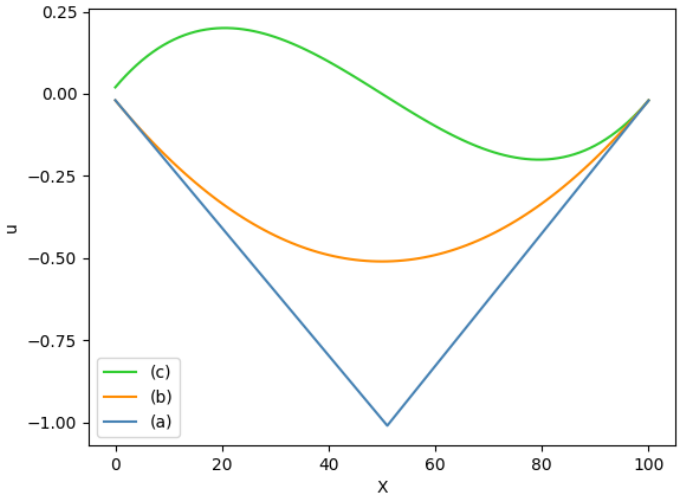

Figura 2: Funciones halladas de cada una de las $\frac{d^2}{dx^2}u = d$, usando $\mathrm{L}u^x = d^x$.

En base a la función u^a se podría decir que no alcanza máximos, ni mínimos y que no es diferenciable en el punto mínimo del gráfico (cuando se juntan las 2 restas), lo que significa que no existe la derivada segunda en este caso, o sea, no se puede calcular d para x = 50 aproximadamente.
En base a la función u^b se podría decir que d alcanza su valor máximo en X = 50 aproximadamente, en el gráfico se ve que para este valor de X hay un mínimo relativo.
Mientras que para la función u^c se pude decir que alcanza un máximo en X = 20 y un mínimo en X = 80 aproximadamente, lo que significa que d en X = 20 alcanza su valor mínimo y d para X = 80 alcanza su valor máximo.

3.3. Tiempos de cómputo

Los archivos que contienen el codigo de esta implementación son *5a.py* y *5b.py*

3.3.1. Pivoteo vs. Tridiagonal

Analizando los resultados presentados por la figura 3 podemos ver la gran diferencia entre ambos algoritmos.Si bien empiezan a la par para la matriz 1x1, ya en la matriz 5x5 empieza a haber diferencia en los tiempos de ejecución y así sigue aumentando la diferencia entre ambos a medida que crece el tamaño (n) de la matriz. El de Eliminación Gaussiana tridiagonal parecería tener una complejidad O(n) aproximadamente, en cambio la de Eliminación Gaussiana Pivoteo parecería ser $\mathrm{O}(n^3)$. Por lo tanto como supusimos previamente, el algoritmo que usa pivoteo es significativamente más lento que el de tridiagonal (siempre teniendo en cuenta que estamos hablando de casos donde la matriz es tridiagonal).

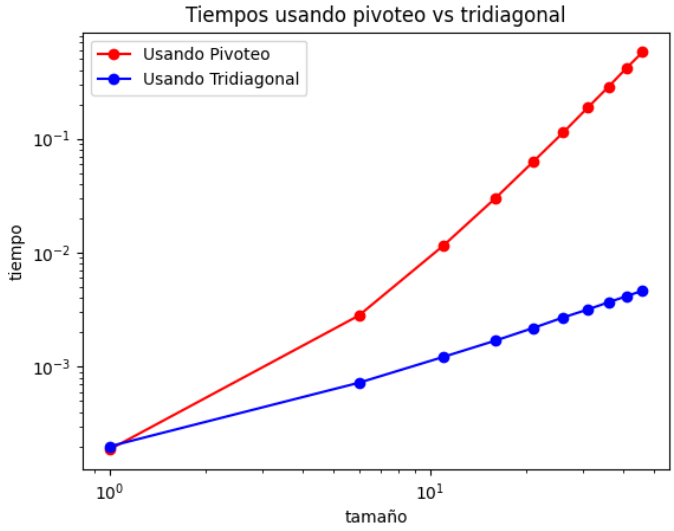

Figura 3: Tiempos mínimos vs tamaños

3.3.2. Tridiagonal sin precómputo vs. Tridiagonal con precómputo

En el caso de precómputo contra la implementación sin ella, vemos como en la primer iteración ambas tardan aproximadamente el mismo tiempo pues en esta iteración es cuando calculamos la factorización LU, es decir, hacemos el precómputo. Sin embargo, ya a partir de la segunda repetición nos damos cuenta como con el precómputo la complejidad se reduce de manera importante y luego se mantiene constante para el resto de las repeticiones, mientras tanto el caso sin factorización LU se mantiene constante con respecto a su primer iteración, por lo tanto no tiene una mejora de tiempo. Esto se debe a que al tener las matrices L y U ya computadas no hay que volver a hacer la Eliminación Gaussiana para la matriz entonces se ahorra mucho tiempo en cada iteración.

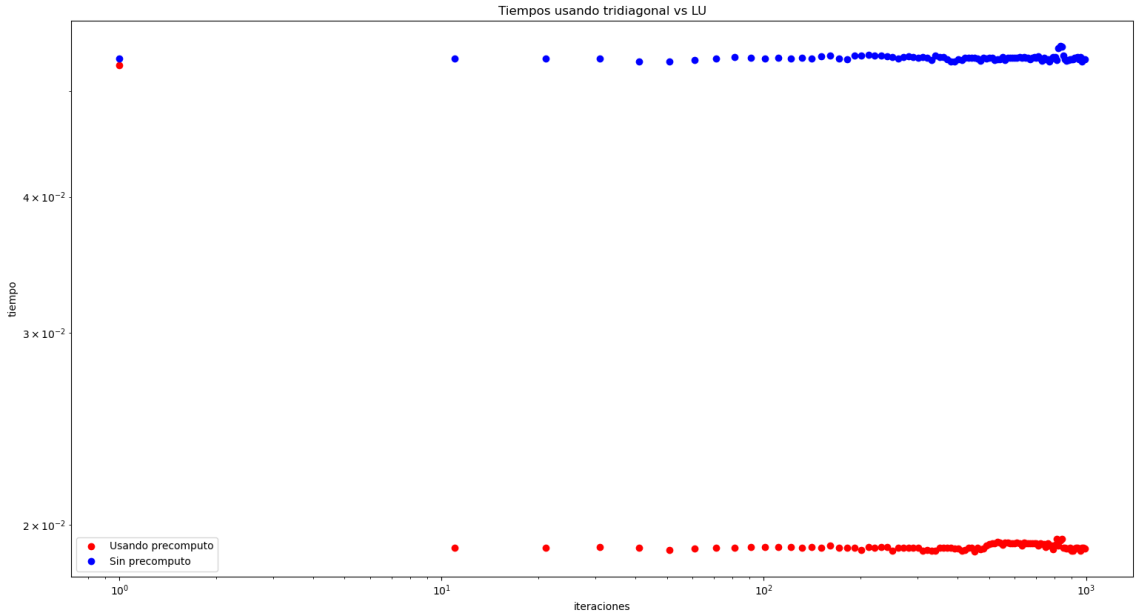

Figura 4: Comparación entre EG tridiagonales con precómputo y sin

3.4. Difusión

Los códigos que contienen este experimento estan en los archivos difusionExplicita.py y difusionImplicita.py

Como pueden observar en las figuras 5 y 6, los resultados no son iguales. En el caso implícito(figura 6) se comportó como supusimos, ya que a medida que aumentamos el α la dispersión aumentaba de manera consistente. Por otro lado, el resultado que no esperábamos en el caso explicito, fue que hasta $\alpha = 0{,}5$ se comportaba normal pero luego uno probaba con $\alpha = 0{,}501$ y, si bien en el gráfico aún pudimos observar la difusión ya vemos que no es consistente con los gráficos anteriores o con los gráficos de la figura 5. Finalmente cuando usamos $\alpha = 0{,}502$ observamos como los resultados dejaban de ser los esperados totalmente y se volvía un gráfico en el que no se puede visualizar ningún tipo de difusión como las anteriores; de ese valor en adelante los resultados empiezan a fallar.

Esta diferencia entre los resultados nos indica que la formulación implícita es estable en cuanto a valores de α refiere, mientras que la formulación explícita no lo es (ver [2]).

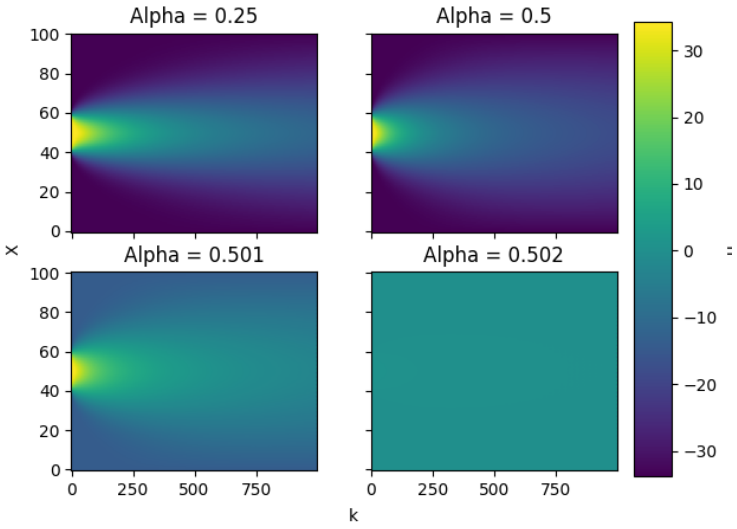

Figura 5: Simulación de difusión a través del tiempo a partir de la ecuación explicita

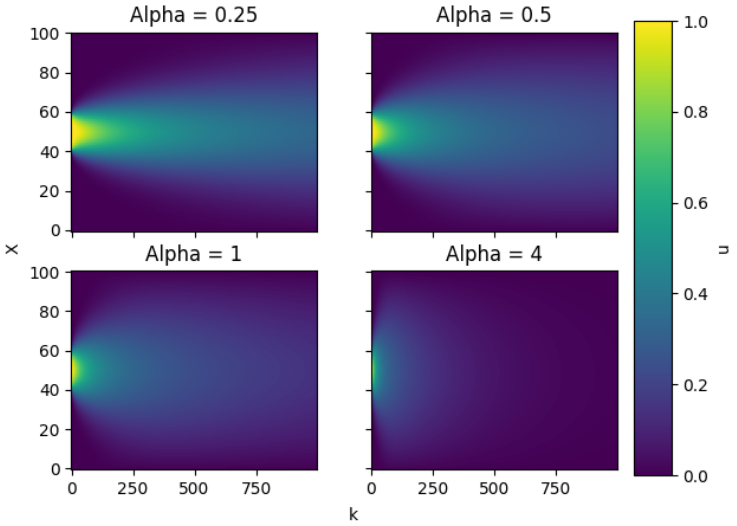

Figura 6: Simulación de difusión a través del tiempo a partir de la ecuación implícita

4. Conclusiones

A lo largo de este trabajo pudimos implementar y estudiar diferentes métodos de realizar Eliminación Gaussiana (EG), para resolver sistemas de ecuaciones. Programamos la EG con y sin pivoteo, observando los casos en los que el pivoteo es indispensable para poder resolver el sistema, y cómo este método no implica un aumento significativo en el tiempo de ejecución. Ambos son de complejidad $O(n^3)$. A su vez, el hecho de que estos algoritmos realicen múltiples operaciones aritméticas nos sirvió para corroborar la precisión de usar numeros de punto flotante de 32 bits contra 64 bits.

Así mismo pudimos profundizar sobre los sistemas tridiagonales y su resolución, la cual pudimos hacer con un algoritmo que posee un tiempo de ejecución de O(n). Además implementamos otro algoritmo para resolver estos sistemas, en el cual se busca primero la factorización LU. En este caso concluimos que si vamos a usar la misma matriz tridiagonal, de un considerable tamaño, para resolver diferentes sistemas, hay una mejora importante en el caso de usar el precómputo de las matrices L y U.

Finalmente investigamos un par de aplicaciones para este tipo de matrices. Una de ellas fue usar la matriz tridiagonal dada por la aproximación discreta unidimensional del operador Laplaciano, para encontrar la aproximación de una función con base en sus derivadas segundas. La otra aplicación que hallamos fue la de aprovechar la aproximación discreta del proceso estocástico de difusión (usando una fracción del operador Laplaciano). Lo abordamos tanto de forma implícita como explicita, en ambos casos partiendo desde la misma condición inicial y, luego, resolviendo iterativamente simulando así la difusión a través del tiempo. En el caso implícito se comportó como habíamos conjeturado produciendo una difusión coherente y constante. No obstante en el caso explicito empezó comportándose como esperábamos pero luego de cierto valor la difusión deja de ser coherente y empieza a ser irregular.

En resumen, en este informe ofrecemos una percepción detallada sobre la Eliminación Gaussiana y su variante para matrices tridiagonales, así como su aplicación en la resolución de sistemas lineales, cálculo de funciones y en la simulación de procesos físicos.

Referencias

[1] J. Douglas Faires Richard L. Burden. *Análisis numérico*. International Thomson Editores, 2002.

[2] H. P Langtangen. *Finite difference methods for diffusion processes*. University of Oslo, 2013.

[3] Timohty Sauer. *Numerical Analysis*. Pearson, 3rd Edition, 2017.