DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Métodos Numéricos

20 de abril de 2023 Metodos Numericos

Integrante	LU	Correo electrónico
Schwartzmann, Alejandro Ezequiel	390/20	a.schwartzmann@hotmail.com
Memoli Buffa, Pedro	376/20	demnitth@gmail.com
Collasius, Federico	164/20	fede.collasius@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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
https://exactas.uba.ar

PALABRAS CLAVE: Gaussian Elimination, Differential Equations

RESUMEN: En este trabajo implementamos y experimentamos con el algoritmo de eliminación gaussiana aplicado en particular a las matrices tridiagonales. En particular estudiamos su aplicación en la resolución aproximada de ecuaciones de Poisson y en el modelado de problemas de difusión. Validamos experimentalmente las complejidades teóricas e hicimos un análisis sobre el error numérico de la eliminación gaussiana, y lo relacionamos con el número de condición de las matrices. También estudiamos la convergencia de la aproximación de las soluciones en una ecuación de Poisson a su valor real.

Índice

1. Introducción

La resolución de sistemas lineales es un problema fundamental en el álgebra lineal que tiene aplicaciones en todas las ramas de la ciencia e ingeniería. Por lo tanto, es importante desarrollar algoritmos que puedan resolverlos en un tiempo y con un error numérico razonable.

El objetivo de este trabajo es implementar y experimentar sobre el algoritmo de Eliminación Gaussiana para resolver sistemas lineales. Se hace un estudio en particular del caso de matrices tridiagonales y su utilización para aproximar la solución de las ecuaciones de Poisson y de difusión en el caso unidimensional. Desarrollaremos sobre estos conceptos a continuación.

1.1. Resolución sistemas lineales

Un sistema lineal es un conjunto de ecuaciones de la forma

$$\begin{cases} a_{11}x_1 + \ldots & + a_{1n}x_n & = b_1 \\ \vdots + \vdots & + \vdots & = \vdots \\ a_{n1}x_1 + \ldots & + a_{nn}x_n & = b_n. \end{cases}$$

Se expresa en forma matricial como $Ax = b$, donde $A \in \mathbb{R}^{nxn}$ con $(A)_{ij} = a_{ij}$; $b \in \mathbb{R}^{nx1}$ con $(b)_{i1} = b_i$ y $x \in \mathbb{R}^{nx1}$ con $(x)_{i1} = x_i$. Resolver el sistema consiste en encontrar todos los x tales que el sistema $Ax = b$ sea verdadero.

Las soluciones de estos problemas permanecen invariantes frente a *operaciones elementales* sobre el sistema. Las operaciones consisten en sumarle a una fila un múltiplo no nulo de otra, intercambiar filas, o multiplicar una fila por una constante. En términos de la matriz asociada, las operaciones son equivalente a multiplicar a A a izquierda por una matriz elemental M. Esta es la propiedad fundamental en la que se basa el algoritmo de eliminación gaussiana. En este se multiplica A por izquierda con M, donde M es un producto de matrices elementales $M_1M_2...M_n$ y MA resulta triangular superior.

La eliminación gaussiana se puede dividir en dos algoritmos según se permita o no el intercambio de las filas de la matriz. De no permitirse, se dice que se tiene eliminación gaussiana *sin pivote*, y puede demostrarse que cuando no se permite pivote, puede no poder llevarse a cabo el algoritmo, mientras que si se permite siempre termina.

Para analizar el algoritmo es relevante estudiar su error numérico. Como las coordenadas de la matriz se representan con aritmética de punto flotante, muchos números tiene un decimal a partir del cual se pierde precisión. Esto causa que la constante b, tanto como la solución x que pueda proporcionar el algoritmo, va a tener un error respecto a su valor real. La relación de error relativo entre la solución y la constante está acotado según el *numero de condición* $k(A) = ||A||||A^{-1}||$ de la matriz A debido a la desigualdad:

$$\frac{||x - x_e||}{||x||} \leq k(A)\frac{||b - b_e||}{||b||}. \tag{1}$$

Esto determina que a mayor número de condición, mayor será la cota del error relativo posible que hay en la solución. Dicho error también está condicionado por la representación de números en computadoras 1 la cual intenta representar números racionales y reales, donde algunos poseen dígitos infinitos como $\frac{1}{3} \approx 0{,}3333...$, en una computadora con memoria finita. Veremos en la experimentación como surgen varios problemas a la hora de operar con números en la computadora.

En esta sección del trabajo desarrollamos como implementamos el algoritmo de eliminación gaussiana con y sin pivote. También analizamos su error numérico y tiempo de cómputo, comparando los casos en los que se añade el pivote y cuando no, y como varía según el número de condición de la matriz.

1.2. Sistemas tridiagonales

Un sistema tridiagonal tiene la forma $a_{i-1}x_{i-1}+a_ix_i+a_{i+1}x_{i+1} = b_i$. Se expresa en forma matricial como

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & \cdots & 0 \\ 0 & a_2 & b_3 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & c_{n-1} \\ 0 & 0 & \cdots & a_{n-1} & b_n \end{bmatrix} x = b. \tag{2}$$

Estos sistemas de ecuaciones pueden ser resueltos con eliminación gaussiana en $O(n)$, observando que en cada iteración del proceso de triangulación y despeje de soluciones solo es necesario hacer $O(1)$ operaciones aritméticas. Además, pueden guardarse las operaciones de las matrices elementales y de ahí reconstruir las soluciones para cualquier vector independiente b, evitando así el paso de triangular. A esto se le dice "pre computar".

En esta sección del trabajo llevamos a cabo la eliminación gaussiana modificada para tener una complejidad temporal y de memoria $O(n)$, guardando además las operaciones. Experimentamos sobre su tiempo de cómputo y lo comparamos con la eliminación gaussiana tridiagonal sin implementar el precomputo.

1.3. Ecuación de Poisson

Una aplicación de sistemas tridiagonales es un método numérico para aproximar la solución de una ecuación de Poisson de bordes nulos en el caso unidimensional. La ecuación de Poisson tiene la forma $\nabla^2 u = d$, donde u y d son funciones de $\mathbb{R}^n$ a $\mathbb{R}$, u es C^2 y d continua. En el caso unidimensional, esta ecuación se reduce a

$$\frac{d^2u(x)}{dx^2} = d(x). \tag{3}$$

Si se samplea la función $d(x)$ a partir del 0 en intervalos de Δx, se obtiene un vector $d = (d(\Delta x), ..., d(n\Delta x))$. Luego tomando diferencias discretas, la derivada segunda de $u(i\Delta x) = u_i$ puede aproximarse como

$$u_{i-1} - 2u_i + u_{i+1} = d_i\Delta^2 x \tag{4}$$

Asumiendo que $u_0 = u_{n+1} = 0$, u sampleado en los valores de d puede representarse como la solución del sistema tradicional formado por (2) para el caso en qué $u(0) = u((n+1)\Delta x) = 0$. Este puede resolverse utilizando el algoritmo de la sección anterior, cuya complejidad es $O(n)$.

En esta parte del trabajo implementamos el algoritmo y comparamos la solución aproximada con la real. También estudiamos su resultado para algunas ecuaciones particulares.

1.4. Ecuación de difusión

La ecuación de difusión es $\frac{df(x,t)}{dt} = \alpha\frac{d^2f(x,t)}{dx^2}$ y es utilizada para modelar diversos procesos que involucren difusión, por ejemplo como se expande y comporta un gas, hasta aplicaciones en otras disciplinas como economía. En el caso discreto t toma valores naturales y puede representar un k-esimo paso de algún proceso que modele difusión. Con el método de aproximación presentado en 1.3, la ecuación puede aproximarse como

$$u_i^k - u_i^{k-1} = \alpha(u_{i-1}^k - 2u_i^k + u_{i+1}^k). \tag{5}$$

Este es un sistema que puede representarse en forma matricial como $Au^k = u^{k-1}$, donde A es:

$$\begin{bmatrix} 1+\alpha & -\alpha & 0 & \cdots & 0 \\ -\alpha & 1+\alpha & -\alpha & \cdots & 0 \\ 0 & -\alpha & 1+\alpha & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & -\alpha \\ 0 & 0 & \cdots & -\alpha & 1+\alpha \end{bmatrix}. \tag{6}$$

Por el método presentado en la sección 1.2, si n es la cantidad de iteraciones y se quiere obtener todo u^k, basta triangular A en $O(n)$ y luego despejar u^k recursivamente utilizando los multiplicadores guardados. La complejidad resultante es $O(n^2)$.

En esta sección del trabajo implementamos el algoritmo para $\alpha = 1$ y experimentamos con una condición inicial que concentre 1s en el centro y 0s en los bordes.

2. Desarrollo

2.1. Resolución de sistemas lineales

Desarrollamos en las siguientes secciones la implementación del algoritmo de Eliminación Gaussiana para resolver sistemas de ecuaciones lineales. En la sección 2.1.1 mostramos la implementación de los casos con pivoteo y sin pivoteo. Luego en la sección 2.1.2 describimos los experimentos sobre su error numérico y tiempo de cómputo en cada caso.

2.1.1. Implementación: Eliminación Gaussiana

La idea de la eliminación gaussiana sin pivote es usar operaciones elementales —excluyendo intercambio de filas— para generar una matriz triangular. Esto se hace iterando sobre cada columna i y restando múltiplos de la fila i al resto en el rango $i+1$ a n. Cabe destacar que esta versión del algoritmo puede no terminar como mostramos a continuación. La implementación es la siguiente:

Algorithm 1 Eliminación Gaussiana sin Pivote

Require: A: matriz cuadrada de tamaño $n \times n$, b: vector de tamaño n.
Ensure: x: vector solución del sistema $Ax = b$.

$n \leftarrow \text{TAM}(A[0])$
for $i \leftarrow 0$ to $n-1$ **do**
    if $\text{isCLOSE}(A[i,i], 0)$ and $i \neq n-1$ **then**
        return -1
    end if
    for $j \leftarrow i+1$ to n **do**
        $m \leftarrow A[j,i]/A[i,i]$
        for $k \leftarrow i$ to n **do**
            $A[j,k] \leftarrow A[j,k] - m \times A[i,k]$
        end for
        $b[j] \leftarrow b[j] - m \times b[i]$
        return -1
    end for
end for
$x \leftarrow \text{ZEROS}(n)$
for $i \leftarrow n-1$ **downto** 0 **do**
    if $isClose(\text{A}(i,i), 0)$ and $i = n-1$ **then**
        continue
    end if
    for $j \leftarrow i+1$ to n **do**
        $b[i] \leftarrow b[i] - A[i,j] \times x[j]$
    end for
    $x[i] \leftarrow b[i]/A[i,i]$
end for
return x

Algorithm 2 isClose

function $\text{isCLOSE}(a, b, rel\_tol = 1e-06, a\_tol = 1e-08)$
    return $|a-b| \leq a\_tol + rel\_tol * absolute(max(a,b))$
end function

1. Input: Toma $A \in R^{nxn}$, y un vector de términos independientes $b \in R^n$ que representan el sistema de ecuaciones lineales $Ax = b$ a resolver.

2. Iteración: Itera sobre las columnas de A de 1 a n

3. Ceros en columna: Para ubicar 0 en la columna a partir de la fila i, la operación que realiza es multiplicar la i-ésima fila por el escalar apropiado $(A_{ji}/A_{ii}) = \lambda$, y usarla para restar $F_j - \lambda * F_i$. Esta operación se repite para todas las filas debajo de la fila del pivote, logrando los 0 desde la j-ésima fila en la i-ésima columna. [1]

4. Repetición: Se aumenta el índice i hasta llegar a la n-ésima fila.

5. Resolver el sistema: Como se termina con una matriz triangular superior, se tiene que para $a_n x_n = b_n$, con lo cual simplemente despeja x_n, esto implica pasar del otro lado de la igualdad los elementos: $x_n = b_n/a_n$. Esto se generaliza, partiendo del x_n despejado de la última fila, se sube por cada fila y utiliza los coeficientes conocidos para despejar la variable x desconocida.

El algoritmo corre $O(n)$ iteraciones para el ciclo según el ítem 2. Restar una fila por otra tiene una complejidad $O(n)$, ya que se hacen $O(n)$ restas entre elementos (ítem 3). La complejidad temporal por iteración es entonces $O(n^2)$, y entonces el algoritmo tiene una complejidad $O(n^3)$.

Como trabajamos con representación de punto flotante, sabemos que los números representados tienen un grado de imprecisión pequeño pero no nulo. No todos los números pueden representarse de forma exacta, y este problema se acentúa para números muy grandes, ya que la representación contiene menos bits para representar la parte decimal, porque necesita de ellos para representar la parte entera.

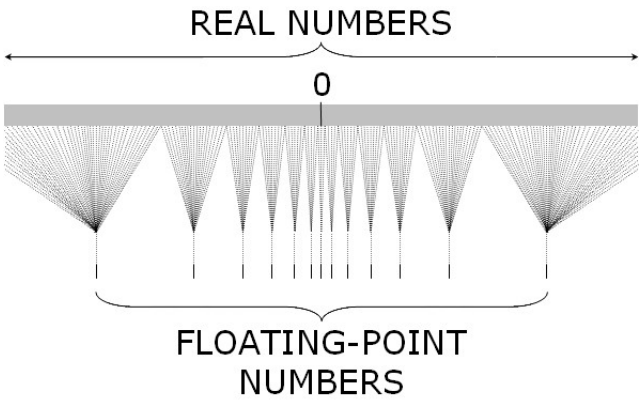

Figura 1: Esquematización del intervalo de números reales que representa cada numero flotante. Se observa que mayores números pierden precisión en la representación.

Esto genera un grave error cuando el algoritmo intenta reconstruir la solución dividiendo $b[i]/A[i,i]$, donde se puede tener un número muy cercano a 0, y además si $b[i]$ es un número muy grande, esta operación va a generar un resultado con errores en la representación. Para salvar esta situación se implementa la función isClose(a, b) que evalúa si dos números son suficientemente cercanos, usando como criterio un valor absoluto y relativo más según un criterio motivado por la implementación de numpy3]. Cuando isClose(A[i, i], 0) es verdadero durante la triangulación, el programa retorna -1. Esto evita que un multiplicador sea un número tan grande que su error respecto al valor real sea inaceptable. Cabe destacar que hay otras fuentes de error, como multiplicar por números de orden muy alto, restar números similares o sumar números de distinta magnitud[2]. Consideramos de cualquier forma que para la absoluta mayoría de matrices, la principal fuente de error es la posible división de

[1] $Aji - Aii * Aji/Aii = 0$

un número muy cercano a 0.

El problema de esta implementación es que el algoritmo puede no terminar. Cuando intenta triangular la matriz, si encuentra un 0 en la diagonal, y todavía no termino de poner 0 debajo, no puede continuar porque λ queda expresado como el cociente entre un $a \in \mathbb{R}$ y 0. Un ejemplo sería la matriz:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

En el segundo paso de la eliminación Gaussiana:

$$A^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

Si se continúa con el algoritmo se llega a un multiplicador $\lambda = A_{31}/A_{22} = 1/0$ que se indefine.

Para salvar esta situación implementamos un intercambio de filas en el algoritmo seleccionando un *pivote*. Si encuentra un 0 en la diagonal, intenta intercambiar la j-ésima fila por otra, que este debajo de ella, y que su elemento A_{ji} no nulo y el máximo de la columna, esto no solo salva al algoritmo, logrando que encuentre soluciones si encuentra un 0 en la columna, sino que además agrega estabilidad numérica porque, como veremos más adelante en la experimentación, debido a la representación de punto flotante cuando se intenta calcular el cociente entre un número y en el denominador se encuentra un número cercano a 0, empiezan a aparecer errores en el cálculo. Esta idea la sistematizamos de la siguiente manera en el algoritmo siguiente:

1. Selección de Pivote: En cada iteración i de las columnas de la matriz, se encuentra él j tal que $|A_{ji}|$ sea máximo para j en el rango de i a n. Si el máximo $|A_{ji}|$ es 0 pasa a la siguiente iteración; si no es nulo, la fija F_j se la denomina *pivote* y se intercambia la fila F_i por F_j. Si es nulo o él por error de aritmética de punto flotante interseca el 0, el algoritmo se saltea la fila.

La introducción del pivote introduce $O(n^2)$ operaciones aritméticas, por lo que la complejidad total del algoritmo sigue siendo $O(n^3)$.

Llevar a cabo siempre el pivoteo asegura que en la diagonal de la iteración i-esima, el elemento $a_i i$ es lo mayor posible. Esto reduce considerablemente el error numérico, ya que los multiplicadores son números menores con mayor precisión en su representación de punto flotante.

Con el algoritmo modificado, luego de la segunda iteración de la implementación con pivote, la matriz A resulta

$$A^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

Gracias al pivote el algoritmo puede terminar de triangular la matriz.

Aun así existen limitaciones a la aplicación de la Eliminación Gaussiana. Cuando el sistema tiene infinitas soluciones o ninguna, si bien este algoritmo termina y triangula la matriz, no logra despejar una única solución. Como se ve en el siguiente sistema de ecuaciones representado por la matriz extendida A:

$$A = \begin{bmatrix} 1 & 2|0 \\ 2 & 4|0 \end{bmatrix}$$

Cuando aplicamos eliminación Gaussiana la matriz queda expresada como:

$$A^{(1)} = \begin{bmatrix} 1 & 2|0 \\ 0 & 0|0 \end{bmatrix}$$

Algorithm 3 Eliminacion_Gaussiana(A, b)

```
1: n ← shape of A[0]
2: for i ← 0 to n − 1 do
3:     if i ≠ n − 1 then
4:         max ← 0
5:         row ← 0
6:         for w ← i + 1 to n do
7:             if A_{w,i} > max then
8:                 max ← A_{w,i}
9:                 row ← w
10:            end if
11:        end for
12:        if isClose(max, 0) then
13:            return -1
14:        else
15:            A_{i,row} ← A_{row,i}
16:        end if
17:    end if
18:    for j ← i + 1 to n do
19:        m ← A_{j,i}/A_{i,i}
20:        for k ← i to n do
21:            A_{j,k} ← A_{j,k} − m · A_{i,k}
22:        end for
23:        b_j ← b_j − m · b_i
24:    end for
25: end for
26: x ← zeros(n)
27: for i ← n − 1 downto 0 do
28:    if isClose(A_{i,i}, 0) then
29:        return -1
30:    end if
31:    for j ← i + 1 to n do
32:        b_i ← b_i − A_{i,j} · x_j
33:    end for
34:    x_i ← b_i/A_{i,i}
35: end for
36: return x
```

El conjunto solución de este sistema es infinito, ya que la última ecuación es descartada, $0*x_1+0*x_2 = 0$ vale para todo x_1, x_2 e introduce un grado de libertad en las soluciones. Esto ocurre cuando aparece un 0 en la diagonal luego de que la matriz esté triangulada, en cuyo caso retorna -1.

2.1.2. Experimentación: Eliminación Gaussiana

Para este algoritmo propusimos dos tipos de experimentos, analizar su tiempo de ejecución y su error numérico, causados por la representación con punto flotante y también por el número de condición, $\kappa(A)$ de la matriz dada. [1]

Para evaluar el tiempo de ejecución del algoritmo, generamos matrices cuadradas identidad extendidas con un vector $b = (1)_n$, es decir un vector de 1s. Y para verificar la variación del comportamiento en el tiempo lo que hicimos fue cambiar el tamaño de las matrices, con un tamaño inicial de 100×100, hasta 700×700, con saltos discretos de 150 en el tamaño, para disminuir el tiempo que conlleva testear. Luego ejecutamos el algoritmo 11 veces para cada matriz y graficamos el mínimo y la mediana del tiempo de cómputo en función del tamaño. Esperamos que el gráfico en escala logarítmica sea paralelo a una recta de pendiente 3, o en caso contrario que tenga una pendiente mayor a 2 y menor a 3. Esto sería consistente con que la complejidad temporal sea $O(n^3)$.

Ahora como también nos interesa estudiar el error inherente a la representación de IEEE 754 en aritmética de punto flotante. Testeamos el comportamiento del algoritmo cuando recibe números con propiedades [2] que como consecuencia del funcionamiento de la representación de punto flotante conjeturamos que pueden generar errores muy pequeños, pero si la matriz dada tiene un $\kappa(M)$ suficientemente grande, puede potencialmente perderse mucha precisión como se observa en la desigualdad (1).

El primer caso que nos interesa es la cancelación catastrófica, en la cual se restan dos números muy cercanos entre sí, generando una pérdida de precisión. Para testear esta hipótesis generamos una matriz donde forzamos durante la eliminación gaussiana la resta entre dos números de magnitud muy grande muy similares.

El segundo caso es él de suma de números de distinta magnitud, con lo cual el número más grande "absorbe" al número más pequeño. Teniendo en cuenta esto, generamos una matriz donde el primer elemento de la diagonal es muy grande, y en la segunda fila existe un elemento muy pequeño.

Por último, el caso de división por cero, donde el resultado puede ser infinito, NaN (Not a Number) o un valor arbitrario, dependiendo de la implementación de la máquina. Para este caso generamos una matriz donde aparece un número muy cercano a 0 en la diagonal y por el algoritmo de la eliminación Gaussiana divide a otros elementos dentro de la matriz.

Para todos los casos testeados, la matriz generada fue de 2x2 de forma que sea simple resolverla analíticamente. Se utilizan números racionales que cumplen las propiedades para experimentar con el error y tienen un número de condición alto para que los errores se propaguen causando un error numérico considerable. Se espera que en estos casos se llegue a observar las limitaciones del algoritmo en términos de inestabilidad numérica. Cabe destacar que para testear esto deshabilitamos el return -1 cuando el máximo elemento de interés de la columna es cercano a 0.

2.1.3. Implementación: Matrices Tridiagonales

Dada una matriz tridiagonal $A \in \mathbb{R}^{nxn}$, hay a lo sumo 3n - 2 coordenadas no nulas cuya posición en la matriz es conocida. Esto permite una implementación del algoritmo de eliminación gaussiana que resuelve un sistema $Ax = d_i$ en $O(n)$ operaciones aritméticas. Observamos también que pueden almacenarse los multiplicadores en el proceso de triangular -precomputo- para así resolver un conjunto d de constantes independientes. Presentamos una implementación para matrices que nunca requieren pivote:

Algorithm 4 Eliminación Gaussiana para Matrices Tridiagonales con precomputo

Require: a, b, c: vectores de tamaño n que representan las diagonales de la matriz (2). d un vector de constantes independientes.

Ensure: *sols*: array cuyo elemento i-esimo es la solución del sistema $Ax = d_i$.

```
1: n ← LEN(b)
2: mult_upper ← []
3: mult_lower ← []
4: for i ← 0 to n − 1 do
5:     if b[i] = 0 then
6:         return -1
7:     end if
8:     m ← a[i]/b[i]
9:     b[i + 1] ← b[i + 1] − m × c[i]
10:    mult_upper.append(m)
11: end for
12: for i ← n − 1 to 1 do
13:    if b[i] = 0 then
14:        return -1
15:    end if
16:    m ← c[i − 1]/b[i]
17:    mult_lower.append(m)
18: end for
19: sols ← []
20: for cte ← d do
21:    x ← ARRAY(dtype = float, size = n)
22:    for i ← 1 to n − 1 do
23:        x[i] ← x[i] − x[i − 1] × mult_upper[i − 1]
24:    end for
25:    for i ← n − 2 downto 0 do
26:        x[i] ← x[i] − x[i + 1] × mult_lower[n − 2 − i]
27:    end for
28:    for i ← 0 to n − 1 do
29:        x[i] ← x[i] × 1/b[i]
30:    end for
31:    sols.append(x)
32: end for
33: return sols
```

1. En las líneas 4 a 10 guarda los multiplicadores de una triangulación superior en el array mult_upper, observando que solo hay dos elementos que se modifican por iteración (a[i] pasa a ser 0 y b[i + 1] se modifica).

2. En las líneas 12 a 18 se repite el procedimiento para una triangulación inferior, guardando los multiplicadores correspondientes en el array mult_lower.

3. En las líneas 19 a 33 Reconstruye las soluciones a partir de los multiplicadores almacenados iterando sobre cada constante independiente d_i en d.

La matriz se representa con 3 vectores y así en un espacio de memoria dé $O(n)$. En el ítem 1. se lleva a cabo la eliminación gaussiana que produce una matriz triangular superior. Cada iteración tiene una complejidad $O(1)$ ya que hay solo dos elementos por fila que tienen que ser modificados, y solo es relevante la fila $i + 1$. Por lo tanto, la complejidad de 3. es $O(n)$.

En el ítem 2. se lleva a cabo la eliminación gaussiana en sentido inverso y modificando los elementos superiores a la diagonal, produciendo así una matriz diagonal. La complejidad es $O(n)$ al igual que en el ítem anterior.

Finalmente en el ítem 3. se construyen las soluciones a partir de las operaciones almacenadas observando, que en la triangulación *superior*, en cada iteración a la constante independiente d_i se la modifica como $d[i] = d[i] - d[i-1]v_u[i-1]$. Cuando se triangula superiormente se modifica $d[i] = d[i] - d[i+1]v_u[n-2-i]$. Para obtener la solución final queda dividir a $d[i]$ por $b[i]$, ya que el sistema equivalente resulta así $Ix = x$. Por eso el factor $1/b[i]$. La complejidad de ese paso es $O(kn)$ con k la cantidad de sistemas a resolver (la cantidad de elementos en el array d). Concluyendo así que el algoritmo tiene una complejidad de $O(kn)$.

2.1.4. Experimentación: Matrices tridiagonales

Experimentamos sobre el tiempo de cómputo del algoritmo para el caso *con* precomputo y *sin* precomputo. El caso sin precomputo corresponde a correr el algoritmo propuesto una vez para cada constante independiente (d contiene solo un elemento que varía cuando se quiere resolver otro sistema, mientras que con precomuto d puede tener todas las constantes independientes que se quiera).

Para esto generamos matrices tridiagonales con la diagonal principal compuesto a de 2s y las diagonales adyacentes de 1s. El algoritmo resuelve en total 15 sistemas iguales con Ax = b siendo b un vector de 1s. Graficamos la mediana del tiempo para 20 mediciones vs el tamaño de la matriz en un rango de tamaños de 500 a 50000 en escala logarítmica. Dada que la complejidad es $O(n)$, esperamos ver una recta en escala logarítmica paralela a otra de pendiente 1. Es esperable también que el tiempo con precomputo sea considerablemente menor al algoritmo sin precomputo.

Para ver como afecta la cantidad de sistemas a resolver a la mejora de la implementación con precomputo, hacemos un gráfico de tiempo medio vs cantidad sistemas a resolver con las matrices y sistemas del experimento anterior. La dimensión de la matriz es de 5000 x 5000.

No se experimenta sobre su error numérico, ya que previamente lo estudiamos en el caso de matrices generales.

2.2. Ecuación de Poisson

Presentamos en la sección 2.2.1 la implementación de un algoritmo que aproxime la solución de la ecuación de Poisson para un caso de bordes fijos en 0. En 2.2.2 se presenta la experimentación sobre el tiempo de cómputo y como varía la aproximación de la solución con la solución analítica.

2.2.1. Implementación

El siguiente algoritmo encuentra una aproximación de $\frac{d^2}{dx^2}u = d$ dado un sampleo de d entre bordes nulos de u en intervalos regulares Δx.

Algorithm 5 Solución Aproximada

Require: n: tamaño de la matriz, d: vector $\in \mathbb{R}^n$ que representa $(d(\Delta x), ...d(n\Delta x))$..
Ensure: Retorna la solución el sistema $Au = d$ que aproxima $u = (u(\Delta x), ...u(n\Delta x))$ con $u(0) = u((n+1)\Delta x) = 0$.
1: $a \leftarrow$ vector de tamaño $n-1$ lleno de 1's
2: $b \leftarrow$ vector de tamaño n lleno de -2's
3: $c \leftarrow$ vector de tamaño $n-1$ lleno de 1's
4: $u \leftarrow$ EliminacionGaussianaTridiagonal(a, b, c, d)
5: **return** $u[0]$

1. Genera un operador *laplaciano* A como una matriz tridiagonal de dimensión n con a y c compuesto por 1s, y b compuesto por -2s dada la notación (2). El operador se representa con los arrays a, b y c.

2. Resuelve el sistema $Au = d$ para obtener $u = (u(\Delta x), ...u(n\Delta x))$ utilizando el algoritmo de matrices tridiagonales presentado previamente.

3. Retorna a u como un array.

Generar el operador laplaciano (2) tiene una complejidad $O(n)$ donde n es la cantidad de sampleos sobre la función d. Resolver el sistema con el algoritmo en la sección 2.1.3 es $O(n)$, por lo que la complejidad temporal total del algoritmo es $O(n)$. Dado un Δx suficientemente pequeño y un n suficientemente grande, se puede aproximar u por una distancia arbitraria a su valor real.

2.2.2. Experimentación

Estudiamos sobre su tiempo de cómputo en función del tamaño del sampleo, aproximando la solución de la ecuación $\frac{d^2}{dx^2}u = 2$. Para esto tomamos $\Delta x = 1/n$ donde n es el tamaño del sampleo de forma que el vector $d\Delta x$ esté formado por coordenadas que son todas c. Para ver cualitativamente como aproxima la solución, se grafica el u analítico del caso anterior $x^2 + c$, y superponemos la aproximación para Δx para $1/5, 1/50$ y $1/500$. Se estima que cada vez sea más similar.

También ejecutamos el algoritmo para tres problemas

$$d_1 = \begin{cases} 0 \\ 4/n \quad i = \lfloor n/2 \rfloor + 1 \end{cases}$$

$$d_2 = 4/n^2$$

$$d_3 = (-1 + 2i/(n-1))12/n^2$$

con $\Delta x = 1$ y $n = 101$ y los comparamos.

2.3. Ecuación de difusión

Presentamos en la sección 2.3.1 la implementación de un algoritmo que simula un proceso de difusión. Experimentamos en la sección 2.3.2 con un gráfico de calor pcolor para observar el proceso de difusión dada una condición inicial que concentre 1s en el centro y 0s fuera.

2.3.1. Implementación

Algorithm 6 Función de Difusión

```
1: function DIFUSION(c_i, k)
2:     n ← len(c_i)
3:     pasos_difusion ← [c_i]
4:     a ← vector de tamaño n − 1 lleno de 1's
5:     b ← vector de tamaño n lleno de -3's
6:     c ← vector de tamaño n − 1 lleno de 1's
7:     precomputo ← PrecomputoEliminacionGaussiana(a,b,c)
8:     for i ← 0 to k − 1 do
9:         resultado ← CalculoSolucion(precomputo, pasos_difusion[length(pasos_difusion) - 1])
10:        pasos_difusion.append(resultado)
11:    end for
12:    return pasos_difusion
13: end function
```

1. Input: Un array c_i que representa la condición inicial, y un entero positivo que representa la cantidad de iteraciones del proceso a ejecutar.

2. En las líneas 2-6 genera un operador *difusión* A como una matriz tridiagonal de dimensión n con la diagonal principal compuesta de -3s y las diagonales adyacentes de 1s. Esto representa a la matriz (6) con $\alpha = 1$.

3. En 7 calcula el precomputo de la eliminación gaussiana triangonal, que corresponde a los multiplicadores obtenidos en la primer mitad del algoritmo de matrices tridiagonales (Ver apéndice para la implementación explicita de PrecomputoEliminacion).

4. De 8 a 11 resuelve a partir del precomputo iterativamente los sistemas $Au^i = u^{i-1}$ hasta un k final fijo y guarda las soluciones. (Ver apéndice para la implementación explicita de CalculoSolucion)

Generar el operador y calcular el precomputo tiene una complejidad $O(n)$ como se menciono en la sección de Poisson y matrices tridiagonales. Resolver un sistema $Au^i = u^{i-1}$ toma un tiempo $O(n)$, y al ejecutarse k veces (la cantidad de ejecuciones del proceso), el tiempo final resulta $O(kn)$.

2.3.2. Experimentación

Se corre el proceso de difusión con la condición inicial

$$u_i^{(0)} = \begin{cases} 0 & \\ 1 & \lfloor n/2 \rfloor - r < i < \lfloor n/2 \rfloor + r \end{cases}$$

Dado que el sistema modela difusión, suponemos que la concentración inicial de 1s se esparcirá a los bordes. Para observar el proceso se grafica con un gráfico pcolor que grafique los resultados en función del paso i-esimo, y coloree el eje Y basándose en el número de cada coordenada.

3. Resultados y Discusión

3.1. Eliminación Gaussiana: Tiempo de Ejecución

Se grafican los resultados de mediana y mínimo del tiempo de ejecución para 11 mediciones en función del tamaño de la matriz en escala logarítmica. Se adiciona una función cúbica para comparar.

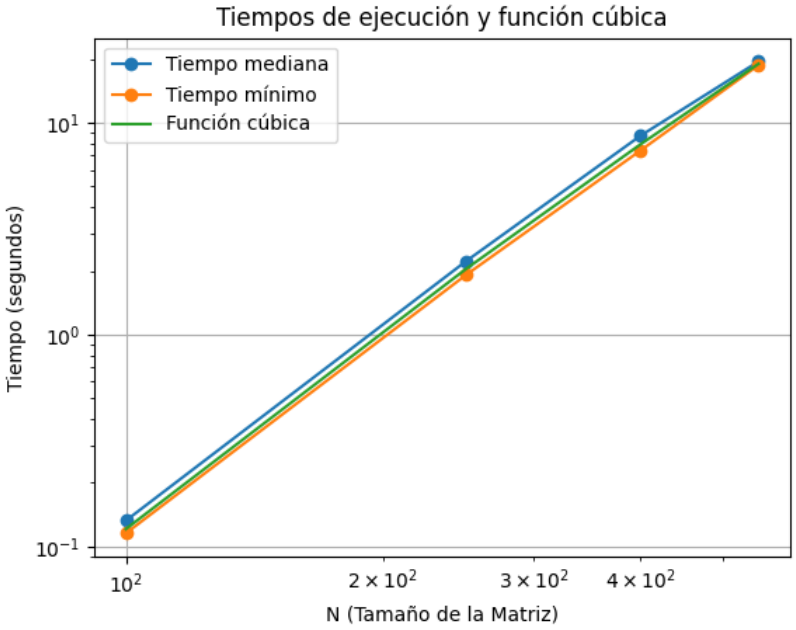

Figura 2: Tiempos mínimos y medianos vs una función cúbica en escala logarítmica. Tiempos en función del tamaño de la matriz

Como esperábamos, el comportamiento del algoritmo es $O(n^3)$ al ser las rectas aproximadamente paralelas al de la función cubica. La complejidad temporal del algoritmo sugiere que puede no ser escalable para entradas muy grandes y puede requerir mejoras o alternativas para manejarlas de manera eficiente.

3.2. Error Numérico

Presentamos en el cuadro 1 cada sistema de ecuaciones lineales representado con su respectiva matriz extendida, su número de condición $\kappa(A)$ y además con su conjunta solución exacta expresada en números racionales, junto con el error del resultado provisto por el algoritmo respecto del resultado exacto. Definimos b constante para los 3 sistemas: $\begin{bmatrix} 5 \\ -\frac{1}{2} \end{bmatrix}$

Matriz A: $\begin{bmatrix} 4 \times 10^{-6} & 7 \\ 9 & 3 \times 10^{6} \end{bmatrix}$

Matriz B: $\begin{bmatrix} 4 \times 10^{-12} & 7 \\ 9 & 3 \end{bmatrix}$

Matriz C: $\begin{bmatrix} 4 \times 10^{-4} & 7 \\ 9 & 3 \times 10^{-4} \end{bmatrix}$

Solución	x_1	x_2	$\kappa(M)$	**Error**
A	$-2{,}94117 \times 10^{5}$	$8{,}82352 \times 10^{-1}$	$3{,}18 \times 10^{12}$	$\big(5{,}8 \times 10^{-11}, 0{,}0\big)$
B	$-1{,}25000 \times 10^{6}$	$-1{,}68066 \times 10^{-1}$	$3{,}9 \times 10^{24}$	$\big(-1{,}2 \times 10^{6}, -8{,}8 \times 10^{-1}\big)$
C	$2{,}94116 \times 10^{11}$	$-1{,}68066s \times 10^{7}$	$3{,}93 \times 10^{8}$	$\big(2{,}9 \times 10^{11}, -1{,}6 \times 10^{7}\big)$

Cuadro 1: Tabla de soluciones. Error expresa la diferencia entre el vector solución esperado y el provisto por el algoritmo.

Como esperábamos, al forzar el algoritmo a operar sobre matrices que tienen en sus celdas elementos que complican la precisión de las operaciones, el resultado entregado es erróneo. Se concluye que el algoritmo puede potencialmente retornar resultados con un error órdenes de magnitud distinta al valor real como muestra el cuadro 2. Por esto es muy importante tener una rutina de *warning* cuando el algoritmo trabaja con tanto error, que en nuestro caso se corresponde a retornar -1.

Matriz	x_1	x_2
A	$\frac{30000007}{102}$	$\frac{22500001}{25500000}$
B	$-\frac{46525000000000}{15749999999997}$	$\frac{22500000000001}{31499999999994}$
C	$\frac{-87537500}{1574999997}$	$\frac{1125005000}{1574999997}$

Cuadro 2: Tabla de soluciones exactas.

3.3. Matrices tridiagonales

Se grafica en la figura 4 el tiempo de cómputo de la resolución de 15 sistemas tridiagonales en función del tamaño de la matriz. Todos los sistemas se componen de 2s en la diagonal principal y 1s en las diagonales adyacentes. La solución es un vector de 1s. Esto se hace para un rango de matrices de 500 a 50000. Se superponen los gráficos para cuando se implementa el precomputo y cuando no se hace.

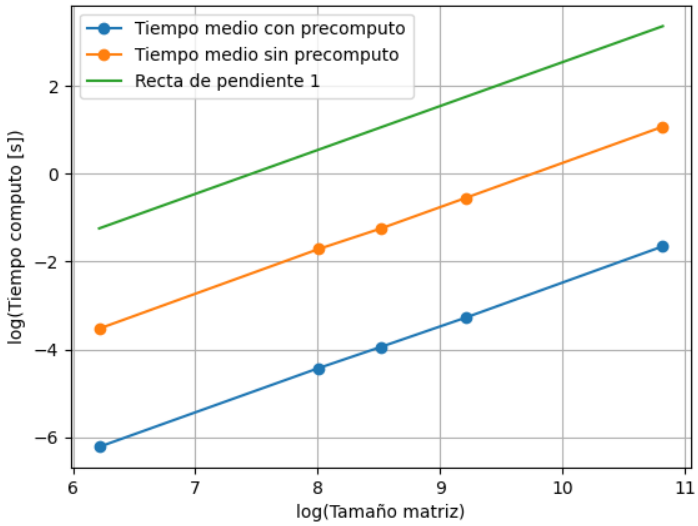

Figura 3: Mediana del tiempo de cómputo para la resolución de 15 sistemas tridiagonales con y sin precomputo en escala logarítmica. Se agrega una recta de pendiente 1 para comparar la complejidad teórica con el tiempo de cómputo real.

Las rectas son claramente paralelas a una de pendiente 1, lo que indica que el tiempo de cómputo es lineal al tamaño de la matriz. Esto es consistente con la complejidad temporal teórica del algoritmo que es $O(n)$. Se observa además que resolver los 15 sistemas con precomputo tiene un tiempo dos órdenes de magnitud menor que resolverlos sin precomputo.

Ahora mostramos los resultados de como varia esta mejora de tiempo con la cantidad de sistemas a resolver. Graficamos el tiempo con y sin precomputo vs la cantidad de sistemas a resolver:

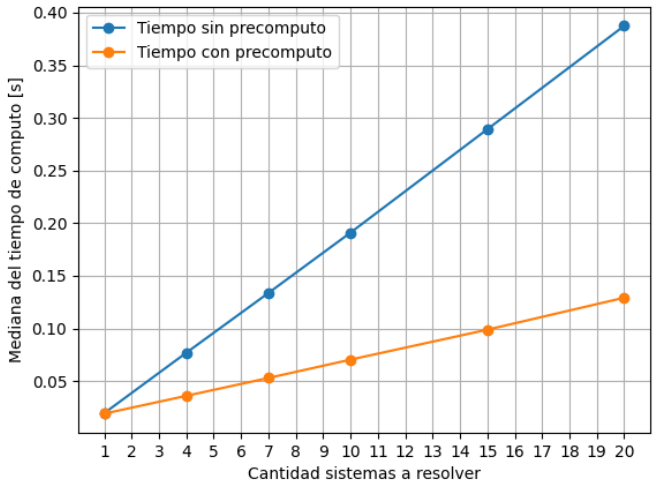

Figura 4: Mediana del tiempo con y sin precomputo vs la cantidad de sistemas a resolver. Se observa una clara mejora en el caso con precomputo a medida aumenta la cantidad de sistemas a resolver.

Se concluye que guardar las operaciones de la triangulación y reutilizarlas produce un algoritmo órdenes de magnitud más eficiente. Esta mejora crece a medida se resuelven más sistemas. Al resolver matrices tridiagonales siempre es mejor implementar el precomputo, ya que tampoco aumenta el tiempo para resolver sistemas individuales..

3.4. Ecuación de Poisson

3.4.1. Resultados diversos

Se grafican las soluciones de las aproximaciones para las ecuaciones de Poisson con términos d_i especificados en la sección 2.2.2:

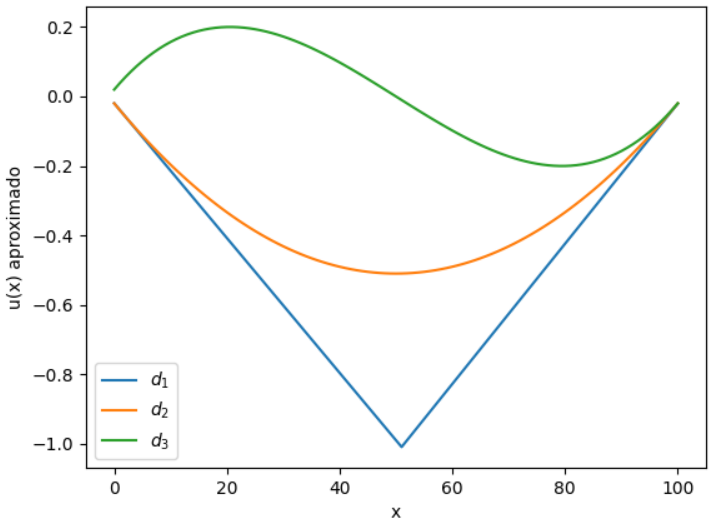

Figura 5: Resultado de la resolución numérica para las constantes d_i dadas.

3.4.2. Aproximación de la solución

Se grafica el u analítico $u(x) = x^2 + c$ correspondiente a la solución de la ecuación diferencial $u''(x) = 2$; junto con los resultados del algoritmo para un sampleo de la función d con un Δx de $1/5, 1/50, 1/500$ y $1/5000$ en el vector $(d(\Delta x), ...d(n\Delta x))$. Se agrega también la curva de la solución analítica real correspondiente a $u(0) = u(1) = 0$.

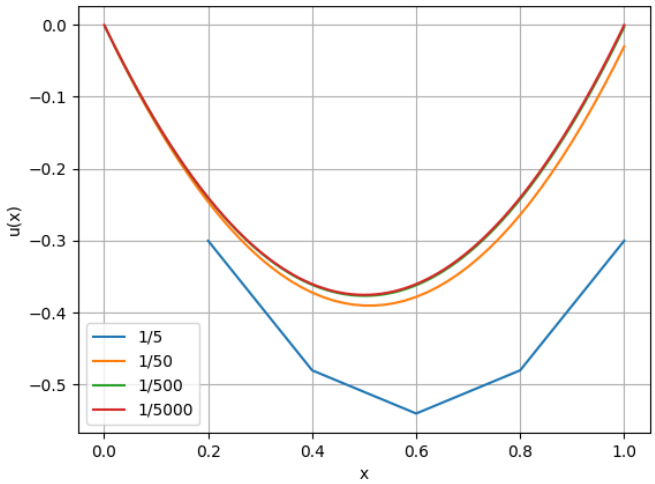

Figura 6: Gráfico de soluciones aproximadas para un sampleo de d en Δx de $1/5, 1/50, 1/500$ y $1/5000$ con una cantidad de sampleos de $5, 50, 500$ y 5000 respectivamente. Se observa como a menor Δx y mayor n la solución aproximada llega a ser indistinguible de la real.

Se observa como para $\Delta x = 1/500$ y $1/5000$ la solución es indistinguible en el rango visto de la solución real. Esto es consistente con lo esperado para lo cual si Δx decrece y n crece la solución

puede aproximarse arbitrariamente a la real. Es interesante el efecto de corrimiento a la derecha, producto de que el algoritmo retorna una solución asumiendo que $u(0) = u(\Delta x(n+1)) = 0$. Cuando Δx es suficientemente chico el efecto del corrimiento es despreciable.

No se puede hacer esto para las constantes d_i ya que no sabemos de qué función realmente son sampleadas. Esto se pudo hacer para $u''(x) = 2$ porque ya conocemos la función $d(x) = 2$ a partir de la cual se samplea d_i en el sistema (4).

3.5. Ecuación de difusión

3.5.1. Resultado

Se grafica la evolución temporal del proceso de difusión para 101 iteraciones sobre la condición inicial con un r de 20 y un n de 101.

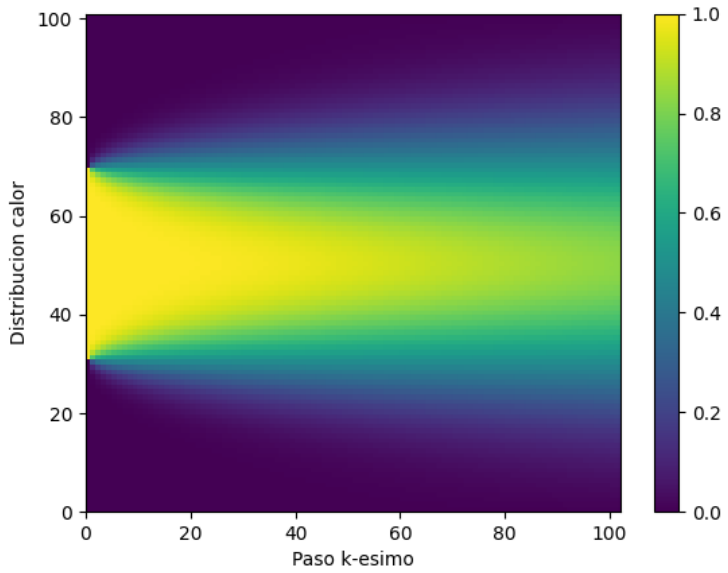

Figura 7: Gráfico de la distribución del "calor" (colores asociados a valores numéricos) en función del índice del paso.

Se observa el fenómeno de difusión esperado, en el cual los valores cercanos a 1 (en amarillo claro), pasan a mezclarse con los cercanos a 0 (en azul oscuro) y viceversa. Este fenómeno se acentúa a medida transcurre el proceso.

4. Conclusiones

Implementamos y experimentamos exitosamente con el algoritmo de eliminación Gaussiana. Analizando su tiempo de cómputo consideramos que se validó que su complejidad temporal es $O(n^3)$ al ser está aproximadamente paralela a una recta de pendiente 3 en escala logarítmica. Estudiando su error numérico debido a aritmética de punto flotante, concluimos que para matrices de número de condición relativamente alto, el resultado del algoritmo puede ser incorrecto por muchos órdenes de magnitud. Por esto recomendamos implementar un sistema de *warning* que corte el algoritmo pasando una cota máxima de error permitido.

También implementamos y experimentamos con un algoritmo de eliminación gaussiana modificado para matrices tridiagonales. Por la estructura de la matriz se pudo obviar muchas operaciones y así reducir la complejidad tanto temporal como de memoria a $O(n)$. Además, implementamos un sistema de "precomputo"que almacena las operaciones para así reutilizarlas y resolver muchos sistemas con la misma matriz de forma más eficiente. Comparamos el tiempo de cómputo entre ambas y concluimos que se valido que el tiempo de cómputo sigue una complejidad lineal al ser paralela a una recta de pendiente 1 en escala logarítmica. Además, el tiempo fue dos órdenes de magnitud menor implementando el precomputo que sin hacerlo.

Aplicamos esto a ecuaciones diferenciales. Implementamos un algoritmo eficiente para matrices tridiagonales con el fin de aproximar las soluciones de una ecuación de Poisson en el caso unidimensional. Se samplea la segunda derivada en intervalos equiespaciados Δx dentro de un rango donde la solución tiene bordes nulos, y se resuelve un sistema $Au = d$ donde A es el operador laplaciano. Observando su convergencia cuando Δx tiende a 0, consideramos que el algoritmo puede aproximar soluciones de forma exitosa y en un tiempo eficiente $O(n)$.

Finalmente, simulamos de forma exitosa procesos de difusión recursivos aplicando el precomputo del algoritmo de eliminación gaussiana para matrices triangulares.

5. Apéndice

5.1. Implementación de Eliminación Gaussiana Tridiagonal para difusión

Algorithm 7 Precomputo de Eliminación Gaussiana

```
function PrecomputoEliminacionGaussiana(a, b, c)
    n ← len(b)
    mult_upper ← []
    mult_lower ← []
    for i ← 0 to n − 2 do
        if b[i] == 0 then
            return -1
        end if
        mji ← a[i]/b[i]
        a[i] ← a[i] − mji ∗ b[i]
        b[i + 1] ← b[i + 1] − mji ∗ c[i]
        mult_upper.append(mji)
    end for
    for i ← n − 1 to 1 do
        if b[i] == 0 then
            return -1
        end if
        m_ji ← c[i − 1]/b[i]
        c[i − 1] ← c[i − 1] − m_ji ∗ b[i]
        mult_lower.append(m_ji)
    end for
    return [mult_upper, mult_lower, b]
end function
```

Algorithm 8 Cálculo de Solución

```
function CalculoSolucion(precomputo, cte)
    mult_upper ← precomputo[0]
    mult_lower ← precomputo[1]
    b ← precomputo[2]
    n ← len(b)
    x ← [cte[i] for i in range(n)]
    for i ← 1 to n − 1 do
        x[i] ← x[i] − x[i − 1] ∗ mult_upper[i − 1]
    end for
    for i ← n − 2 to 0 do
        x[i] ← x[i] − x[i + 1] ∗ mult_lower[n − 2 − i]
    end for
    for i ← 0 to n − 1 do
        x[i] ← x[i] ∗ (1/b[i])
    end for
    return x
end function
```

6. Referencias

Referencias

[1] IEE. *Aritmético Punto Flotante*. https://ieeexplore.ieee.org/document/30711

[2] What Every Computer Scientist Should Know About Floating-Point Arithmetic, by David Goldberg. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#674

[3] NumPy. *NumPy Documentation*. https://numpy.org/doc/stable/reference/generated/numpy.isclose.html