DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

2do cuatrimestre 2024 — Metodos Numericos

Integrante	LU
Francisco Fazzari	900/21
Felipe Miriuka	1693/21
Ignacio Fernández Oromendia	29/21

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

La motivación es aprender sobre algoritmos para resolver sistemas de ecuaciones, utilizando álgebra matricial y ver como estos algoritmos se traducen a sus implementaciones, y cuales son sus limites en computadoras. El objetivo es de probar distintos algoritmos, calcular sus tiempos y poder compararlos entre ellos para ver cuales son los más eficientes y veloces para cada problema a resolver.

El método principal utilizado es el de Eliminación Gaussiana con varios algoritmos, primero uno simple iterativo y otro con pivoteo. Luego Eliminación Gaussiana para el caso de matrices tridiagonales. Ambas eliminaciones tienen su implementación con factorización LU.

Palabras clave:
Sistemas de Ecuaciones, matrices, eliminacion Gaussiana, factorizacion LU, laplaciano, tridiagonal, pivoteo, complejidad numerica, difusion, error numerico.

1. Introducción Teórica

En este informe evaluaremos distintos algoritmos para resolver sistemas de ecuaciones lineales convirtiéndolos en sistemas matriciales para luego utilizar Eliminación Gaussiana, Factorización LU y Sustitución para resolverlos. El problema es que resolver sistemas matriciales es una tarea difícil teóricamente y computacionalmente más todavía, sin embargo existen algunos casos que son fáciles de resolver, como los de las matrices triangulares superiores, para estas matrices buscar la solución tiene un costo $O(n^2)$. La idea entonces seria de transformar las matrices comunes en matrices triangulares superiores, de ser posible, para facilitar la resolución, en este trabajo utilizamos distintos métodos para transformar matrices generales en matrices triangulares superiores, suponiendo que se pueden transformar, utilizando el algoritmo de Eliminación Gaussiana y luego una vez triangulada podemos utilizar sustitución para reconstruir las soluciones del sistema.

2. Desarrollo

2.1. Eliminación Gaussiana

2.1.1. Foward & Backward Substitution

El método de Eliminación Gaussiana es un método que permite resolver sistemas de ecuaciones lineales en donde hay la misma cantidad de incógnitas que de ecuaciones, para estos sistemas en particular podemos construir una matriz A y un vector b tal que la matriz representa el lado de las ecuaciones con solo variables y el vector b esta formado por los números que se despejan del otro lado de la igualdad. Así obtenemos un sistema que tiene la siguiente forma: $Ax = b$.

Ahora bien, supongamos que tenemos una **matriz triangular superior**, sabemos que su sistema asociado tiene en su última fila $a_n \times x_n = b_n$, con lo cual $x_n = \frac{b_n}{a_n}$. Luego la ante-última fila es $a_{n-1} \times x_{n-1} + a_n \times x_n = b_{n-1}$, pero ya sabemos el valor de x_n, por ende podríamos calcularlo. Iterando sobre este procedimiento podemos obtener una formula para calcular x_i.

$$x_i = \frac{b_i - \sum_{i=0}^{n} a_{ij} x_j}{a_{ii}}$$

Por lo tanto, podemos escribir un algoritmo con complejidad temporal de $O(n^2)$ que resuelva sistemas para matrices triangulares superiores.

Algorithm 1 Backward Substitution (A, b)

```
x = [0 ... 0]

for i ∈ [n − 1, 0] do
    suma ← 0
    for j ∈ [i, n − 1] do
        suma ← suma + A_ij × x_j
    end for
    x_i = (b_i − suma)/A_ii
end for

return x
```

Siguiendo la misma idea pero para **matrices triangulares inferiores**, implementamos el *Foward Substitution* para resolver este tipo de sistema, el cual sigue la misma idea que el algoritmo anterior pero recorriendo la matriz desde 0 hasta n.

Algorithm 2 Foward Substitution (A, b)

$x = [0 \dots 0]$

for $i \in [0, n-1]$ **do**
  $suma \leftarrow 0$
  for $j \in [0, i]$ **do**
    $suma \leftarrow suma + A_{ij} \times x_j$
  end for
  $x_i = (b_i - suma)/A_{ii}$
end for

return x

2.1.2. Eliminación Gaussiana sin pivoteo

Para la *Eliminación Gaussiana sin pivoteo* [3] planteamos un algoritmo el cual elige para cada fila i el número de la diagonal A_{ii} para luego dividir cada número de la columna i, A_{ji} donde $j = i + 1$, obteniendo un coeficiente m, el que se usará para restar la fila j desde la columna i en adelante. El pseudo-código del algoritmo se ve así:

Algorithm 3 Eliminación Gaussiana sin Pivoteo (A, b)

for $i \in [0, n-1]$ **do**
  for $j \in [i+1, n-1]$ **do**
    $m \leftarrow \frac{A_{ji}}{A_{ii}}$
    $b \leftarrow b_j - (m \times b_i)$
    for $k \in [i, n]$ **do**
      $A_{jk} \leftarrow A_{jk} - (m \times A_{ik})$
    end for
  end for
end for

return Backward Substitution(A, b)

Un problema de este más allá de su complejidad temporal $O(n^3)$ es que si el número elegido de la diagonal A_{ii} en la división para obtener el coeficiente m es 0 entonces estaríamos haciendo una división por 0. Con lo cual necesitamos evitar en cada elección de A_{ii} que este sea nulo.

Veamos el siguiente ejemplo donde empezamos restando $3F_1$ a F_2 y F_3:

$$\begin{bmatrix} 1 & 2 & 1 \\ 3 & 6 & 10 \\ 3 & 8 & 7 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 7 \\ 0 & 2 & 4 \end{bmatrix}$$

Una vez llegados a este paso $A_{11} = 0$ con lo cual si el algoritmo sigue estaríamos dividiendo por 0. ¿Que pasaría si pudiésemos permutar F_3 con F_2 en este paso?

2.1.3. Eliminación Gaussiana con pivoteo

Para resolver el caso de eliminación Gaussiana donde aparecen ceros en la diagonal utilizamos la técnica denominada **pivoteo** [2], en la cual se permutan las filas y/o columnas para no dividir por cero. Es posible que durante la eliminación Gaussiana hayan infinitas o no haya ninguna solución y en este caso el algoritmo no funciona y se detiene antes de dividir por cero. En nuestro caso implementamos el **pivoteo parcial**, el cual permuta filas únicamente.

Primero debemos encontrar el pivote, para ello buscamos el número más grande dentro de la columna i, una vez hallado el pivote, permutamos la 2 filas en cuestión y luego procedemos con la eliminación

gaussiana. Elegimos la heurística de pivotear con el número más grande para evitar errores por trabajar con aritmética finita, ya que si elegimos el más grande el número resultante de las divisiones será más pequeño, evitando así trabajar con números muy grandes.

Algorithm 4 Eliminación Gaussiana con Pivoteo (A, b)

```
for i ∈ [0, n − 1] do
    fila ← Encontrar Pivote(A, i)
    Permutar Filas(A, i, fila)
    Permutar Filas(b, i, fila)
    for j ∈ [i + 1, n − 1] do
        m ← A_ji / A_ii
        b ← b_j − (m × b_i)
        for k ∈ [i, n] do
            A_jk ← A_jk − (m × A_ik)
        end for
    end for
end for

return Backward Substitution(A^0, b)
```

Los siguientes pseudo-códigos representan la permutación de filas y la búsqueda del pivote.

Algorithm 5 Encontrar Pivote(A, i)

```
max ← |A_ii|
fila ← i

for k ∈ [i, n] do
    if |A_ki| > max then
        fila = k
        max = |A_ki|
    end if
end for

return fila
```

Algorithm 6 Permutar Filas(A, i, j)

```
copia = copiar(A_i)
for k ∈ [0, |A_i|] do
    A_ik = A_jk
    A_jk = copia_k
end for
A_i = A_j
A_j = fila
```

Aunque la Eliminación Gaussiana con pivoteo resuelve más casos que sin pivoteo seguimos teniendo dos casos donde el algoritmo no funciona. Para el primer caso, supongamos el siguiente sistema:

$$\left[\begin{array}{ccc|c} 3 & 6 & -3 & 3 \\ 2 & 4 & -2 & 2 \\ 1 & 2 & -1 & 1 \end{array}\right] \rightarrow \begin{bmatrix} F_2 = F_2 - \frac{2}{3}F_1 \\ F_3 = F_3 - \frac{1}{3}F_1 \end{bmatrix} \rightarrow \left[\begin{array}{ccc|c} 3 & 6 & -3 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

En este caso podemos ver que hay infinitas solución, es decir para cualquier x_2 y x_3 que se elija, se define x_1 en base a esos valores y se obtiene una solución.
Ahora veamos un ejemplo de un sistema inconsistente:

$$\left[\begin{array}{ccc|c} 4 & 2 & -2 & 6 \\ 2 & 1 & -1 & 3 \\ 2 & 1 & -1 & 4 \end{array}\right] \rightarrow \begin{bmatrix} F_2 = F_2 - \frac{1}{2}F_1 \\ F_3 = F_3 - \frac{1}{2}F_1 \end{bmatrix} \rightarrow \left[\begin{array}{ccc|c} 4 & 2 & -2 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$$

En este último ejemplo podemos ver que haciendo eliminación gaussiana llegamos a un sistema que no tiene solución.

2.1.4. Eliminación Gaussiana para un sistema tridiagonal

Hasta ahora habíamos tratado con matrices arbitrarias, en esta sección veremos como resolver sistemas de ecuaciones diferenciales con **matrices tridiagonales**. Una matriz tridiagonal es aquella que tiene su diagonal principal y sus dos diagonales adyacentes con números y el resto ceros. Con lo cual, si queremos realizar eliminación Gaussiana para estas matrices estamos desperdiciando mucho tiempo procesando ceros. Para evitar esto, desarrollamos un algoritmo para resolver sistemas tridiagonales que se aprovecha de la peculiar forma de estas matrices. Para esto dividimos la matriz en tres vectores a, b y c, donde a es la diagonal inferior adyacente a la principal, b es la diagonal principal y c la diagonal superior. Al aplicar eliminación Gaussiana solo sobre los elementos de estos vectores el resultado de los nuevos vectores a, b, c sera

$$a_i = a_i - (a_i/b_{i-1}) * b_{i-1} = 0$$
$$b_i = b_i - (a_i/b_{i-1})c_{i-1}$$
$$c_i = c_i$$
$$d_i = d_i - (a_i/b_{i-1})d_{i-1}$$

De esta forma, se consigue la matriz triangulada con complejidad O(n), ya que solamente estamos trabajando sobre los 3 vectores de tamaño n.

Algorithm 7 Eliminación Gaussiana Tridiagonal (a, b, c, d)

$a^0 = copia(a)$
$b^0 = copia(b)$
$c^0 = copia(c)$
$d^0 = copia(d)$

for $i \in [0, n]$ **do**
    $m = a_i^0/b_{i-1}$
    $a_i^0 = a_i^0 - m \times b_{i-1}^0$
    $b_i^0 = b_i^0 - m \times c_{i-1}^0$
    $d_i^0 = d_i^0 - m \times d_{i-1}^0$
end for

$x = [0 \dots 0]$
$x_{n-1} = d_{n-1}^0$
for $i \in [n-2, 0]$ **do**
    $x_i = \frac{d_i^0 - c_i^0 * x_{i+1}}{b_i^0}$
end for
return x

2.2. Factorización LU

Imaginémonos el caso en el que tengamos que resolver un sistema $Ax = b$ donde A es una matriz arbitraria conocida y **fija**. Decimos que es **fija** porque vamos a querer resolver este sistema para distintos b pero siempre con el mismo A. Estando parados acá no nos conviene utilizar Eliminación Gaussiana ya que tendremos un tiempo $O(kn^3)$ donde k es la cantidad de veces que multiplicamos por un b distinto. Sin embargo, podemos escribir a A como LU donde L es una matriz triangular inferior con 1 en la diagonal y U es un matriz triangular superior[4]. Luego tenemos que resolver el siguiente sistema:

$$Ax = b \iff LUx = b$$

Luego, si $Ux = y$ debemos resolver estos 2 sistemas:

$$Ly = b$$

$$Ux = y$$

Ahora bien, si ya tenemos L y U podemos hacer *backward* y *foward substitution* para despejar x y resolver el sistema con una complejidad $O(n^2)$. Para obtener estas matrices debemos aplicar este algoritmo:

Algorithm 8 Factorización LU (A)

```
L = I
for i ∈ [0, n − 1] do
    for j ∈ [i + 1, n − 1] do
        m ← A_ji / A_ii
        L_ji ← m
        for k ∈ [i, n − 1] do
            A_jk ← A_jk − m × A_ik
        end for
    end for
end for
return (L, A)
```

2.2.1. Factorización LU para matrices Tridiagonales

Para llevar la *factorización LU* a matrices tridiagonales hacemos un proceso similar que en la sección anterior, pero basándonos en el algoritmo de eliminación gaussiana para matrices tridiagonales. Vamos a querer calcular dos matrices LU tal que $A = LU$, donde L va a tener unos en la diagonal y va a tener el coeficiente m que se le resta a la fila $i+1$ en la eliminación gaussiana en la diagonal inferior adyacente a la principal. Luego U será la misma que se obtiene en la eliminación gaussiana.

Algorithm 9 Factorización LU Tridiagonal (a, b, c)

```
b^0 = copia(b)
l = [0, . . . , 0]

for i ∈ [1, n − 1] do
    m = a_i / b^0_{i−1}
    b^0_i = b^0_i − m × c_{i−1}
    l_i = m
end for

return l, b^0, c
```

2.3. Simulación de difusión

Para simular la difusión de un gas, liquido u otro tipo de elementos, podemos pensar el problema como si tuviese diferentes estados donde el *k-ésimo* estado depende del *k-1-ésimo* estado. Por otro lado, podemos modelar las dispersión de la difusión en 3 direcciones, viendo las posibles posiciones como si fuesen un vector columna u. Si la partícula se encuentra en la fila i en el estado $k-1$ estaría representada como $u_i^{(k-1)}$. Luego, si queremos simular el estado k sabemos que $u_i^{(k-1)}$ va a afectar con mayor y menor medida a $u_{i+1}^{(k)}$, $u_i^{(k)}$ y $u_{i-1}^{(k)}$.
Llegados a este punto podemos desarrollar una formula de difusión para el vector $u^{(k)}$ que depende de $u^{(k-1)}$ con una base en $u^{(0)}$.

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

Partiendo de está formula queremos despejar $u_i^{(k-1)}$ de la ecuación.

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

Como habíamos mencionado antes, $u^{(k)}$ es un vector columna, luego podemos pensar otro vector $(0, \ldots, 1, -2, 1, \ldots, 0)$ donde el primer 1 se encuentra en la posición $i-1$ y el segundo uno en $i+1$. Una pequeña aclaración, escribimos al vector columna u como si fuese vector fila pero traspuesto (u^t) para facilitar la lectura.

$$u_i^{(k-1)} = u_i^{(k)} - \alpha(0, \ldots, 1, -2, 1, \ldots, 0)(u_0^{(k)}, \ldots, u_{i-1}^{(k)}, u_i^{(k)}, u_{i+1}^{(k)}, \ldots, u_n^{(k)})^t$$

Luego, en lugar de pensar está formula como calculo de coordenadas podemos pensarla de forma matricial, donde el vector $u^{(k)}$ es multiplicado por el operador Laplaciano $(\nabla^2)[1]$, una matriz tridiagonal donde todos los elementos de la diagonal principal son -2 y sus adyacentes 1 (Similar al vector que habíamos usado anteriormente).

$$u^{(k-1)} = u^{(k)} - \alpha\nabla^2 u^{(k)}$$

Si sacamos factor común $u^{(k)}$ hacia la derecha obtenemos lo siguiente (I es la matriz identidad).

$$u^{(k-1)} = (I - \alpha\nabla^2)u^{(k)}$$

Finalmente tomando $A = (I - \alpha\nabla^2)$ tenemos una formula para cada estado k de la difusión

$$Au^{(k)} = u^{(k-1)}$$

Ahora que tenemos esta formula podemos simular la difusión de calor desde una punto calculando cada estado hasta el k-ésimo. Para realizar este calculo vamos generar el Laplaciano ∇^2 para luego generar A y así poder calcular las difusiones con m pasos. Para realizar el calculo de las difusiones también necesitamos un r para generar nuestro u_0 de la siguiente manera:

$$u_i^{(0)} = \begin{cases} 1 & \text{si } \lfloor \frac{n}{2} \rfloor - r < i < \lfloor \frac{n}{2} \rfloor + r, \\ 0 & \text{caso contrario.} \end{cases}$$

Algorithm 10 Simular Difusión (A, n, r, m)

```
A = I − α∇²
u = Generar u0(n, r)
l, u1, u2 = Factorizar LU Tridiagonal(diagonales(A))
for k ∈ [1, m] do
    u_k = Resolver LU Tridiagonal(l, u1, u2, u_{k−1})
    u.append(u_k)
end for
return u
```

2.4. Simulación de difusión 2D

Para simular un efecto de difusión en dos dimensiones como seria el caso de la difusión del calor, podemos seguir la misma linea de razonamiento que para el caso de difusión en una dimensión, donde tenemos los diferentes estados que representan el tiempo donde el k-ésimo estado depende del k-1-ésimo estado. Una partícula en la posición i, j en el estado k esta representada como $u_{ij}^{(k)}$ y depende de $u_{i-1j}^{(k-1)}$, $u_{i+1j}^{(k-1)}$, $u_{ij-1}^{(k-1)}$, $u_{ij+1}^{(k-1)}$ y $u_{ij}^{(k-1)}$.
Podemos desarrollar una formula de difusión para la matriz $U^{(k)}$ que contiene todos los valores iniciales $u_{ij}^{(k)}$ depende de $U^{(k-1)}$ con una base en $U^{(0)}$.

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

La idea para el caso en dos dimensiones es transformar la matriz $U^{(k)} \in \mathbb{R}^{n \times n}$ en un vector donde las filas de $U^{(k)}$ pasan a estar una al lado de la otra obteniendo $u^{(k)} \in \mathbb{R}^{n^2}$ para luego resolver el sistema. El cual se vería de la siguiente manera:

$$Au^{(k)} = u^{(k-1)}$$

Donde la posición $(Au^{(k)})_{ij} = \sum_{q=1}^{n} a_{iq} u_q^{(k)} = u^{(k-1)}$

3. Resultados

3.1. Comparación de error numérico

Para ver como cambia el error numérico al aplicar eliminación gaussiana con pivoteo, vamos a definir $Ax = b$ de la siguiente manera, con un ϵ para variar el error numérico.

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

Se puede ver que para todo ϵ el resultado de Ax debería dar b, ya que al hacer la multiplicación de matrices el ϵ positivo y el negativo se cancelan, como en el siguiente ejemplo para la fila 1.

$$b_1 = 1 + 2 + \epsilon + 3 - \epsilon$$

$$b_1 = 1 + 2 + 3$$

Para el experimento elegimos valores de ϵ entre 10^{-6} y 10^{0} usando matrices de tipo *Float 32 bits* y *Float 64 bits*. Luego, graficamos el error máximo de las diferencias absolutas.

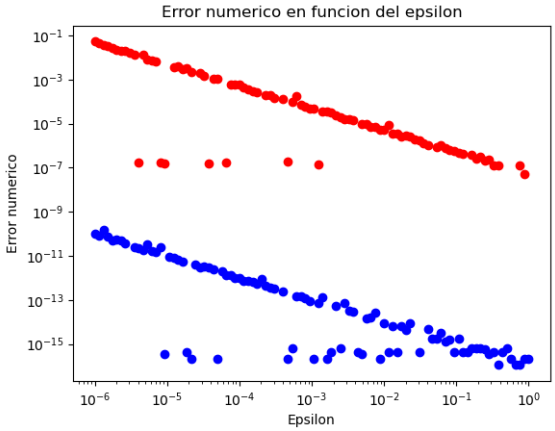


Figura 1: Error numérico

Se puede ver en el gráfico que al aumentar el ϵ tanto en 32 como 64 bits, cuanto más bajo es el ϵ mayor será el error numérico, ya que cuanto más bajo sea el ϵ mayor será la cantidad de decimales con la que estaremos trabajando, por lo tanto el error será mayor.

Por otro lado, se puede observar que el error es menor en 64 bits que en 32 bits. Esto se debe a que al tener menos bits para representar los decimales correctamente y hacer las operaciones es más impreciso, generando mayor error numérico al usar 32 bits.

Queremos destacar que con ciertos valores de ϵ tanto en 32 como 64 bits el error numérico baja más de lo esperado, e incluso hay momentos en donde este llega a 0. Esto se debe a que hay ϵ para los cuales las operaciones reducen el error numérico.

3.2. Verificación de la implementación para sistemas Tridiagonales

Para verificar la implementación de nuestros algoritmos para sistemas tridiagonales resolvemos 3 simples ecuaciones diferenciales. En estos sistemas queremos encontrar u para el problema $\frac{d^2}{dx^2}u = d$ utilizando la matriz tridiagonal del operador laplaciano para los siguientes sistemas:

(a) $d_i = \begin{cases} \frac{4}{n} & \text{si } i = \lfloor \frac{n}{2} \rfloor + 1, \\ 0 & \text{caso contrario} \end{cases}$

(b) $d_i = 4/n^2$

(c) $d_i = (-1 + \frac{2i}{n-1})\frac{12}{n^2}$

Para resolver estas ecuaciones usamos la matriz del operador Laplaciano (∇^2), calculamos una factorización LU para ya tener el precómputo en los 3 casos y resolver el sistema. Los resultados obtenidos se pueden ver en la siguiente figura.

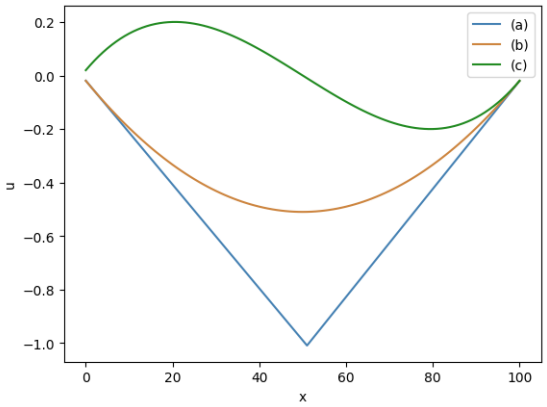
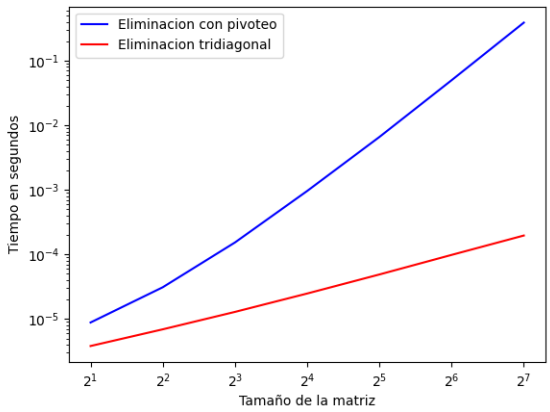


Figura 2: Resultados de u para distintas d

3.3. Comparación de tiempos entre eliminación gaussiana con pivoteo y tridiagonal

En este experimento, comparamos el tiempo de los algoritmos de eliminación gaussiana con pivoteo y eliminación gaussiana tridiagonal con matrices laplacianas (∇^2) de tamaño 2^i, donde i se encuentra entre 1 y 7. Esperamos que el tiempo de la eliminación gaussiana tridiagonal sea mucho menor al de la eliminación gaussiana con pivoteo ya que se aprovecha de la forma peculiar de la matriz tridiagonal.



Al comparar el tiempo que tarda el algoritmo de eliminación Gaussiana con pivoteo y el tridiagonal, se puede ver como la eliminación Gaussiana tridiagonal tiene una menor pendiente, esto se debe a que el gráfico incrementa de forma logarítmica en ambos ejes y la complejidad del algoritmo con pivoteo es de $O(n^3)$, mientras que el tridiagonal es de $O(n)$, por ende la recta con complejidad cubica tiene una pendiente mas alta que la recta con complejidad lineal.

3.4. Comparación de tiempos entre eliminación Gaussiana tridiagonal con y sin precómputo

En este experimento comparamos el tiempo que tarda en resolver cada algoritmo n veces el sistema $Ax = b$ utilizando matrices laplacianas cuadradas de 50×50. Donde el algoritmo sin precómputo utilizará eliminación gaussiana mientras que el algoritmo con precómputo aprovecha que la matriz A es la misma y utiliza la factorización LU para el primer caso y luego simplemente resuelve el sistema con algoritmos de substitution. Con lo cual esperamos que el tiempo de eliminación gaussiana tridiagonal sea más lento que la factorización LU.

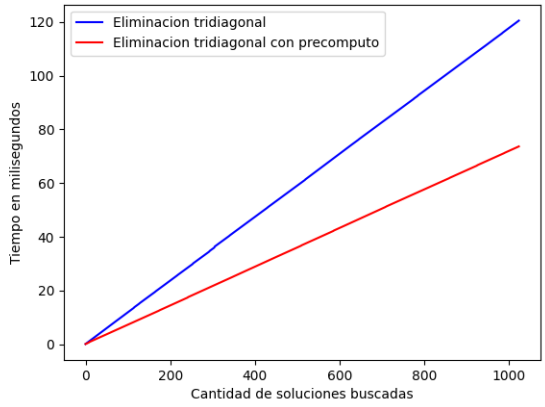


Figura 3: Eliminación Gaussiana tridiagonal vs Factorización LU y Substitution

Se puede observar que, como era esperado, el algoritmo de factorización LU es mejor en tiempo que la eliminación gaussiana, ya que no realiza pasos innecesarios. Además la eliminación gaussiana tiene complejidad $O(n^3)$ mientras que la factorización LU es $O(n^2)$, lo cual se ve claramente en el gráfico, cuantas más soluciones buscadas más diferencia hay entre una y otra.

3.5. Simulación de difusión

En este experimento decidimos observar el impacto en la simulación del valor α, quien representa cual es la potencia de la difusión. Esperamos que cuanto más bajo este coeficiente sea, menos difusión habrá, ya que es lo que representa. En la figura podemos observar el experimento para 4 valores distintos de α.

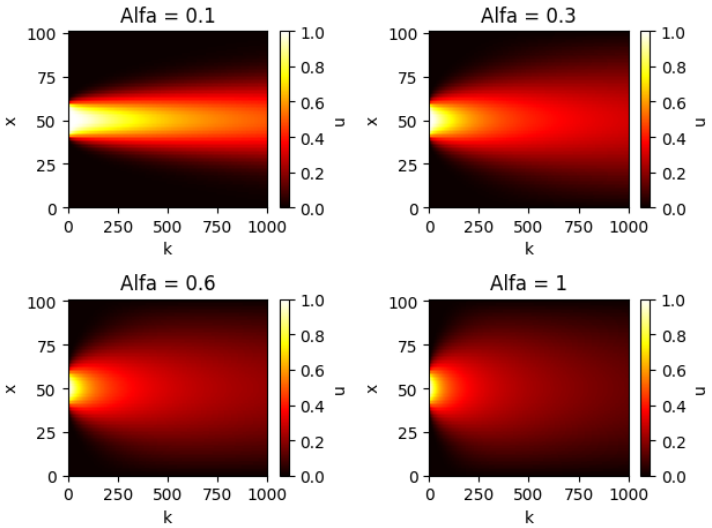


Figura 4: Simulación de difusión para distintos valores de α

3.6. Simulación de difusión 2D

En este experimentos observamos la difusión en dos dimensiones a través del tiempo de una placa de calor. Esperamos que mientras más pase el tiempo más se difunde el calor de la placa hacia afuera.

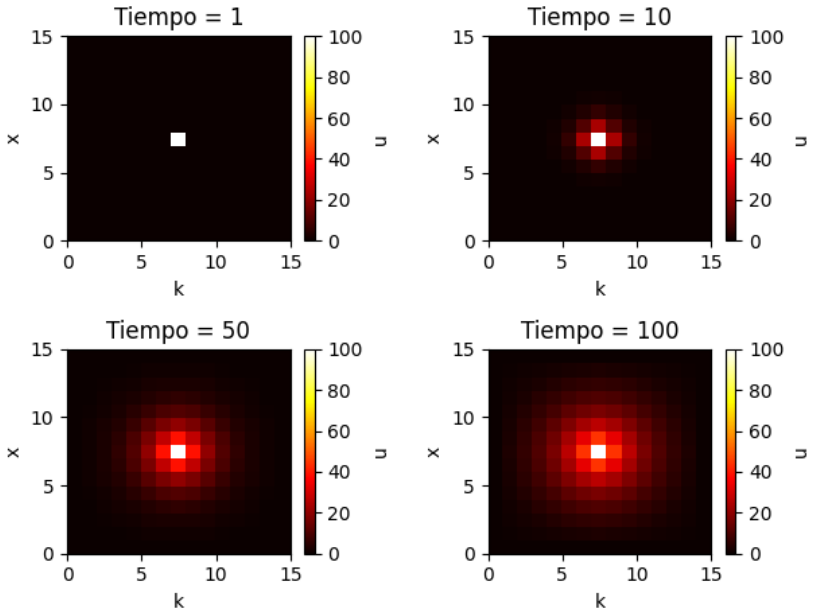


Figura 5: Simulación de difusión de una placa de calor en distintos tiempos t

3.6.1. Método de Eliminación Gaussiana utilizado

Este experimento mide los tiempos de las distintas implementaciones de Eliminación Gaussiana que podemos utilizar. Vamos a comparar con y sin pivoteo, lo esperado es que por cómo funciona el pivoteo, el algoritmo que lo utiliza sea más lento a medida que el tamaño de la matriz crece.

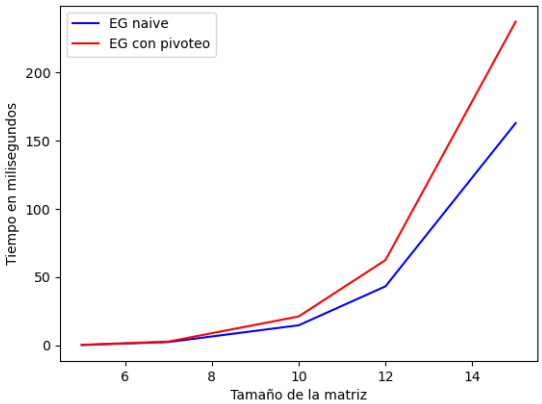


Figura 6: Simulación de difusión de una placa de calor en distintos tiempos t

Se puede ver en el gráfico que al usar eliminación gaussiana con pivoteo, el algoritmo es más lento que al usar la implementación sin pivoteo. Esto es lo esperado, ya que al tener que buscar el mayor número en cada columna en cada iteración, el algoritmo va a tardar más. En este caso, no sirve de mucho pivotear, ya que en este caso se está trabajando sobre una matriz con banda pentadiagonal, entonces al hacer pivoteo probablemente no se encuentre en la columna valor mayor al de la diagonal.

4. Conclusiones

En conclusión, pudimos evaluar los diferentes algoritmos para realizar *Eliminación Gaussiana* dependiendo el contexto dado. Que queremos decir con esto, si uno quiere resolver un sistema lineal siempre tiene la posibilidad de realizar una eliminación gaussiana común, es decir sin pivoteo, pero si uno sabe en que contexto está trabajando puede aprovecharse de las características de este contexto para poder hacer uso más eficiente del sistema que debe resolver. Por ejemplo, en el caso de la *simulación de difusión* Fig. 4 en una dimensión, nos aprovechamos de que la matriz usada para simular cada estado era tridiagonal, con lo cual, en lugar de usar un algoritmo de orden cúbico pudimos utilizar un algoritmo cuadrático para resolver el mismo problema.

Otro ejemplo se puede ver en el case de Fig. 3 donde en lugar de calcular siempre el mismo algoritmo una y otra vez nos aprovechamos de un precómputo con la *Factorización* LU para no tener que calcular la Eliminación Gaussiana una y otra vez.

Por otro lado, pudimos experimentar como afecta la representación de números reales en una computadora, ya que, como pudimos ver en la Fig. 1, en ciertas ocasiones llevar un cálculo teórico a la práctica tiene otros desafíos como por ejemplo la representación de números reales.

Además, vimos como resolviendo sistema abstractos como $Ax = b$ pueden representar una simulación de difusión de calor con distintos parámetros, algo que uno no se lo imagina estudiando el ámbito teórico.

Referencias

[1] Hans Petter Langtangen. Finite difference methods for diffusio processes. *Technical report, University of Oslo*, 2013.

[2] J. Douglas Faires Richard L. Burden. *Análisis numérico.* International Thomson Editores, 7th edition, 2002. Chapter 6.2.

[3] Timothy Sauer. *Numerical Analysis.* Pearson, 2rd edition, 2017. Chapter 2.1.1.

[4] Timothy Sauer. *Numerical Analysis.* Pearson, 2rd edition, 2017. Chapter 2.2.1.