DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Eliminación Gaussiana, matrices tridiagonales y difusión

Métodos Numéricos

Joaquín Carrasco: joaquin.e.carrasco@gmail.com
Santiago Pensotti: santipensotti@gmail.com
Valentina Vul: valenvul@gmail.com

Resumen:

El objetivo de este trabajo es implementar y experimentar con algoritmos de eliminación gaussiana para resolver sistema de ecuaciones lineales. En particular, se analizó el caso de las matrices tridiagonales y su aplicación al modelado de un problema de difusión. Además se buscó comparar diferentes implementaciones de dicho algortimo con el fin de encontrar cuál es el más exacto y velóz.
Se utilizó python para implementar estas funciones y analizar sus complejidades temporales y error generado. A partir de esto, se llegó a la conclusión de que el algoritmo más favorable varía según el caso de uso. El algoritmo más veloz es el que resuelve sistemas de ecuaciones de matrices tridiagonales, realizando la triangulación en un etapa de precálculo y resolviendo el sistema a partir de esta matriz para todos los términos independientes que se presenten.
Este algoritmo solo puede ser utilizado en casos particulares. Cuando no se puede asegurar nada sobre la estructura de la matriz se llego a la conclusión de que es preferible el algoritmo de eliminación gaussiana con pivoteo. Esto se debe a que abarca un rango mayor de matrices que es capaz de resolver, y presenta un error menor sin tener una gran perdida en tiempo de cómputo.

Palabras clave:

Eliminación Gaussiana, Difusión, Error numérico

Índice

1. Introducción Teórica

El objetivo de este trabajo es la implementación y experimentación con algoritmos de eliminación gaussiana y el estudio particular del caso de matrices tridiagonales y su aplicación en el modelado de un problema de difusión.

Un sistema de ecuaciones lineales puede ser representado mediante una matriz A y un vector b, tal que cada fila de A corresponda a una ecuación distinta, sus elementos a los coeficientes de las incógnitas y los elementos de b a los términos independientes. De forma que:

$$Ax = b \tag{1}$$

con $A \in R^{n \times n}$, $x, b \in R^{n}$.

Representa un sistema de ecuaciones del tipo:

$$\begin{array}{cccccc}
a_{11} * x_1 + & a_{12} * x_2 + & \dots & + a_{1\,n-1} * x_{n-1} + & a_{1\,n} * x_n = & b_1 \\
a_{21} * x_1 + & a_{22} * x_2 + & \dots & + a_{2\,n-1} * x_{n-1} + & a_{2\,n} * x_n = & b_2 \\
\vdots \; + & \vdots \; + & \vdots & + \quad \vdots \; + & \vdots \; = & \vdots \\
a_{n\,1} * x_1 + & a_{n\,2} * x_2 + & \dots & + a_{n\,n-1} * x_{n-1} + & a_{n\,n} * x_n = & b_n
\end{array}$$

La resolución de estos sistemas se vuelve mucho más simple al tener una matriz triangular, ya que a partir de esta se puede aplicar una sustitución fila por fila [1]. Basándose en esto, la eliminación gaussiana propone un algoritmo que permite transformar una matríz dada en una triangular y así resolver el sistema de la forma antes presentada. Esta transformación se hace mediante operaciones elementales entre filas, lo que genera un sistema equivalente al original, es decir, con la misma solución[2]. Estos sistemas solo van a tener solución única si la matriz A es no singular, por lo que el algoritmo es trivial si A no es inversible.

La eliminación gaussiana presenta un problema con ciertos sistemas que, a pesar de tener solucion, presentan algún cero en la diagonal principal en cualquier paso de la eliminación, lo que deriva en una división por cero. Es por esto que se ideó una segunda versión que implementa otra operación elemental, el intercambio de filas. A este intercambio se lo llama pivoteo.

Este pivoteo no sólo sirve para evitar divisiones por cero, sino que también se puede realizar en todos los pasos. En cada instancia se busca el elemento mas grande de la columna y se pivotea en base a la fila en la que se encuentra. Esto evita amplificar errores de redondeo que se pueden haber causado al realizar operaciones previas [3].

A partir de estos dos algoritmos se pueden derivar nuevas versiones que aprovechan una estructura asegurada de la matriz dada. Dichas estructuras permiten disminuir la cantidad de pasos necesarios para resolver el sistema de ecuaciones.
Una de estas es la matriz tridiagonal. Estas matrices presentan una forma particular, de manera que todos los elementos que no están en la diagonal principal y en las diagonales secundarias inmediatas, son nulos. Al realizar la eliminación gaussiana de estas matrices se aprovecha el conocimiento previo de dónde hay ceros y, por cada paso, se realizan únicamente 3 modificaciones: el elemento de la diagonal de dicho paso, el elemento de la izquierda y el término independiente correspondiente a ese paso.

[1] Watkins, D. S. (2010). Fundamentals of matrix computations (2nd ed.). John Wiley & Sons, Inc. p.23
[2] Watkins, D. S. (2010). Fundamentals of matrix computations (2nd ed.). John Wiley & Sons, Inc. p.70
[3] Watkins, D. S. (2010). Fundamentals of matrix computations (2nd ed.). John Wiley & Sons, Inc. p.93

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & \ddots & \\ & & \ddots & \ddots & a_{n,n-1} \\ 0 & & & a_{n-1,n} & a_{n,n} \end{bmatrix}$$

Figura 1: Matriz tridiagonal

Una matríz de esta forma ampliamente utilizada en los campos de la matemática y la física es la matríz del operador laplaciano. En una dimensión, este operador respresenta la segunda derivada de una función continua *f(x)*. Al discretizar esta función, evaluándola en múltiples puntos de su dominio, se llega a la siguiente ecuación:

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

Siendo u el conjunto de valores discretos de la función f y d_i el iésimo valor de la aproximación discreta de la segunda derivada de *f*. De forma matricial esto se puede expresar como:

$$\begin{bmatrix} -2 & 1 & & & 0 \\ 1 & -2 & 1 & & \\ & 1 & -2 & \ddots & \\ & & \ddots & \ddots & 1 \\ 0 & & & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ \vdots \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ d_n \end{bmatrix} \tag{3}$$

En el caso particular del modelado de una difusión en el espacio y el tiempo, el operador laplaciano discreto nos permite aproximar la evolución temporal de la concentración de la cantidad que se esta difundiendo en diferentes puntos del espacio. Esto se puede hacer mediante la ecuación de difusión discreta, que define al incremenento de un paso k como:

$$u_i^{(k)} - u_i^{(k-1)} = \alpha(u_{i-1}^{(k)} - 2u_i^{(k)} + u_{i+1}^{(k)}) \tag{4}$$

Con α siendo el coeficiente de difusión adimensional. Mediante esta ecuación se puede calcular la variación de la concentración en distintos puntos, lo que permite modelar la difusión de una cantidad en un rango de tiempo.

2. Desarrollo

2.1. Sobre control de error numérico en los algoritmos

Los algoritmos fueron implementados utilizando la librería numpy, que sigue el estándar $IEEE754$[4]. Este provee una representación finita del conjunto de los números reales, por lo que no todos los números racionales logran ser representados. Se define así un techo y un piso de representación y un subconjunto de reales representados que no está distribuido uniformemente entre el máximo y el mínimo. Cuánto mayor sea la magnitud del número, menor es la probabilidad de tener una representación exacta.
Además hay una cantidad finita de bits destinados a esta representación por lo que muchos números se presentan redondeados siguiendo las reglas de este estándar.

Esta forma de representación puede causar distintos tipos de errores a la hora de hacer cálculos con la computadora. Al implementar los algoritmos descriptos más adelante se tuvieron en cuenta cuatro fuentes de error principales: el redondeo, la cancelación catastrófica [5], la división por números pequeños y la multiplicación por números muy grandes.

Estos pequeños errores se pueden magnificar a lo largo del arlgoritmo por lo que se eligieron las operaciones de tal forma tal que diminuya la probabilidad de que estos se generen, y se determinó un umbral de tolerancia que permite detectarlos. Se llegó al valor 10^{-8} ya que estos algoritmos fueron implementados en python, cuya precisión para números de punto flotante es de 10^{-16}, y no se espera perder más de la mitad de la presición con ninguno de los siguientes algoritmos [6].

2.2. Eliminacion Gaussiana sin pivoteo

A partir de la teoría explicada previamente se ideó la siguiente implementación del algoritmo de Eliminación Gaussiana sin pivoteo.

En este, primero se efectúa la triangulación de la matriz. Comenzando por la primera fila, y para cada fila i de la matriz, se sigue el siguiente procendimiento para todas las filas j por debajo de esta:
Se comienza por calcular el *pivot*[7]. Esto se logra dividiendo el elemento de la columna i en j que se busca anular, por el elemento de la diagonal en i. Esta división no se puede realizar si el divisor es igual a cero, por lo que antes de comenzar cada iteración, el algoritmo se asegura que el elemento en la diagonal sea no nulo.
Por lo explicado previamente esto se implementa mediante una tolerancia, ya que los errores de redondeo no nos aseguran una presición exacta en los coeficientes. Si se encuentra uno de estos elementos, se corta la ejecución y se devuleve una excepción informando la razón.
Luego, para cada elemento de la fila j, se le resta su elemento correspondiente de la fila i multiplicado por el pivot ya calculado. De la misma forma, se modifica la fila j del vector de términos independientes, utilizando el mismo pivot.

Una vez triangulada la matriz y modificado el vector, se comienza el proceso de sustitución para resolver el sistema. Se comienza por la última fila ya que esta contiene una única incógnita, y luego se recorre el sistema hacia arriba. Cada elemento i del vector solución, corresponde a la suma de todos los coeficientes de la matriz en la fila i, multiplicados por el elemento del vector solución en la posición correspondiente a la columna de estos. Como la matriz ya fue triangulada, todos los elementos en columnas previas a la diagonal son nulas, por lo que no es necesario realizar esas operaciones. El resultado de esa suma es restado del término independiente de la misma fila y el número resultante es dividido por el coeficiente que acompaña a la incógnita i.

$$0 * x_1 + ... + a_{i,i} * x_i + ... + a_{i,n} * x_n = b_i \tag{5}$$

[4] IEEE (2008). 754-2008 - IEEE Standard for Floating-Point Arithmetic. ieeexplore.ieee.org
[5] Término utilizado para describir lo que ocurre cuando se restan números muy similares.
[6] Python. (2015). PEP 485 – A Function for testing approximate equality. peps.python.org
[7] Asi se les llama a los cocientes por los que se multiplica una fila previo a restarselo a otra.

$$x_i = \frac{b_i - (a_{i,i+1} * x_{i+1} + ... + a_{i,n} * x_n)}{a_{i,i}} \tag{6}$$

Algorithm 1 Eliminación Gaussiana sin pivoteo $(in : A, b) \rightarrow x$

$tolerancia \leftarrow 1e^{-8}$
for $i = 0$ **to** $n - 1$ **do**
  if $|a_{ii}| < tolerancia$ **then** ▷ loop sobre las columnas
    Excepción: cero en (i, i)
    stop
  end if
  for $j = 0$ **to** $n - 1$ **do** fila i
    $p \leftarrow \frac{a_{ji}}{a_{ii}}$ ▷ p es el escalar que al restar la fila i escalada por p
    ▷ a la fila j vuelve 0 al elemento $a_j i$
    for $k = i$ **to** $n - 1$ **do** ▷ Antes de i ya son 0 porque fue triangulada
      $a_{jk} \leftarrow a_j k - p * a_i k$
    end for
    $b_j \leftarrow b_j - p * b_k$
  end for
end for
▷ Hasta acá A está triangulada
▷ Ahora queda resolver a A triangulada
$x \leftarrow VectorVacio(n)$
$x_{n-1} \leftarrow \frac{b_{n-1}}{a_{n-1\ n-1}}$
for $i = n - 1$ **to** 0 **do**
  $x_i \leftarrow b_i$
  for $j = i + 1$ **to** $n - 1$ **do**
    $x_i \leftarrow x_i - a_{i\,j} * x_j$ ▷ $x_i = b_i - \sum_{j=n-1}^{i+1} a_{i\,j} * x_j$
  end for
  $x_i \leftarrow \frac{x_i}{a_{i\,i}}$ ▷ $x_i = \frac{b_i - \sum_{j=n-1}^{i+1} a_{i\,j} * x_j}{a_{i\,i}}$
end for
return x

Para verificar la efectividad de este algoritmo primero se generaron matrices cuadradas con números aleatorios, mediante la función *random.rand*[8] de numpy, de tamaños 3, 5, 10 y 15. De manera similar se generó un vector del mismo tamaño que cada matriz, y se multiplicó a ambos, resultando en un nuevo vector. Se ejecutó la implementación del algoritmo con cada una de estas matrices y su multiplicación correspondiente. Observar que, al ejecutar el algoritmo con un vector de términos independientes que es el resultado de una multiplicación con la matriz, este sistema siempre tendrá solución única.
Luego, se comparó la respuesta dada por el algoritmo, con el vector original con el cual se realizó la multiplicación. Esto se realizó mediante la función *isclose*[9] que devuelve un vector con el valor *True* en cada posición del arreglo en la que se satisface la comparación. Al trabajar con números reales, es necesario tener en cuenta los errores de redondeo y esta función permite realizar la comparación con un umbral de tolerancia (por las razones desarrolladas previamente (2.1) se utilizó una tolerancia relativa de 10^{-8}).

Para comprobar qué ocurre en los casos en los que el sistema no está condicionado para ser resuelto con este algoritmo se idearon otros tests. Nuevamente se partió se matrices generadas de manera aleatoria. Como se desea observar el funcionamiento de la implementación cuando el sistema no tiene solución única, es necesario que la matriz evaluada sea singular. Para esto, se reemplazó una

[8] numpy.random.rand
[9] numpy.isclose

fila de la matriz por una combinacion lineal del resto de las filas, asegurandonos un sistema que no tiene solución. Esta metodología se basa en una propiedad que cumplen las matrices no singulares:

Una matriz $A \in \mathrm{R}^{nxn}$ es no signular $\iff$ todas sus columnas son linealmente independientes las unas de las otras.

Un caso ejemplo donde se levanta la excepción:

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \; b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{7}$$

$a_{0\,0} < tolerancia$ por lo que se detiene el algoritmo y eleva una excepción.

Este sistema tiene solución pero, al no implementar un intercambio de filas, este algoritmo no es capaz de resolverlo. Se puede observar que al intercambiar la fila A_1 por la fila A_2, la matriz ya queda triangulada y el vector solución del sistema es el

$$x = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} \tag{8}$$

Otro ejemplo, con una matriz que a primera vista parece condicionada:

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\downarrow$$

$$A_2 \leftarrow A_2 - (1/1) * A_1, \; b_2 \leftarrow b_2 - 1 * b_1$$
$$A_3 \leftarrow A_3 - (0/1) * A_1, \; b_3 \leftarrow b_3 - 0 * b_1$$

$$\downarrow$$

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \; b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\downarrow$$

$a_{22} < tolerancia$, por lo que se detiene el algoritmo y se eleva la excepción

2.3. Eliminacion Gaussiana con pivoteo

Como se ejemplificó anteriormente, el agregado de la permutación de filas supone una ampliación de los tipos de sistemas que el algoritmo es capaz de resolver, siempre que el sistema tenga solución única esta podrá ser encontrada. Además, esta operación agregada permite disminuir el error generado, en cada paso el cálculo del pivot se hace a partir del divisor de mayor magnitud encontrado, evitando (en la mayoría de los casos) el error generado por dividir por un número muy pequeño. Asimismo esta división reduce la probabilidad de que se propaguen nuevos errores a causa de la multiplicación por números de una magnitud muy grande, ya que el pivot siempre será lo más pequeño posible[10].

Esta implementación es muy similar a la anterior (19), con la excepción de que en cada paso se recorre la columna completa, con el tal de encontrar el elemento de mayor magnitud en esta. Una vez encontrado este, se intercambian tanto las filas de la matriz como las del vector independiente, con el fin de posicionar al número más grande posible en la diagonal.
Existe la posibilidad de que no se encuentre ningun número distinto a cero en dicha columna, lo que indica que la matriz es singular y el sistema no tiene solución. En por esto que, luego de realizar

[10] Heath, M. T. (2002). Scientific Computing: An Introductory Survey (2da ed.). p.71

el intercambio, se verifica que el elemento de la diagonal sea no nulo. De no ser así el algoritmo se detiene y devuelve una excepción. Nuevamente esto se verifica en base a una tolerancia (2.1), para tener en consideración los errores de redondeo que pudieron haber sido generados en pasos previos.

Hay matrices que, a pesar de ser no singulares, están mal condicionadas para este algoritmo. A pesar de que se logra llegar a una respuesta única, el error que se genera al ejecutar el programa lo rinde trivial. Esto ocurre ya que el número de condición [11] de la matriz es muy alto, indicando que el hecho de que las variaciones en la solución calculada con respecto a la real sean pequeñas, no asegura que lo mismo ocurrirá con la diferencia del vector independiente calculado que con el real. Es decir, si se consideran A, b y x tal que, $Ax = b$, y $\hat{x}$ y $\hat{b}$ siendo $\hat{x}$ la solución dada por el alrgoritmo de eliminación gaussiana con pivoteo tal que $A\hat{x} = \hat{b}$:

$$\text{el hecho que } |x - \hat{x}| \text{ sea pequeño, no implica que } |b - \hat{b}| \text{ tambien lo sea.}$$

Es por esto que, al finalizar la resolución del sistema, el algoritmo verifica que la distancia entre el vector resultante de multiplicar al vector solución por la matriz, y el vector del término independiente no sea menor a una tolerancia. Si lo fuese esto podría indicar errores de redondeo que, por lo explicado en el párrafo anterior, podrían volver a la respuesta irrelevante dada una matriz mal condicionada. Si esta condición se cumple, el usuario es advertido de la posibilidad del error.

[11] Heath, M. T. (2002). Scientific Computing: An Introductory Survey (2da ed.). p.59-63

Algorithm 2 Eliminación Gaussiana con pivoteo $(in : A, b) \rightarrow x$

$tolerancia \leftarrow 1e^{-8}$
$M \leftarrow A$
for $i = 0$ **to** $n - 1$ **do** ▷ loop triangulación
  $max \leftarrow i$
  for $fila = i + 1$ **to** $n - 1$ **do** ▷ loop elección del pivote
    if $|a_{fila\,i}| > a_{max\,i}$ **then**
      $max \leftarrow fila$
    end if
  end for
  $intercambioFilas(A, i, max)$
  $intercambioFilas(b, i, max)$
  if $|a_{ii}| < tolerancia$ **then**
    Excepción: cero en (i, i) ▷ Se detiene si pivot ¡$tolerancia$
    stop
  end if
  for $j = 0$ **to** $n - 1$ **do** fila i
    $p \leftarrow \frac{a_{ji}}{a_{ii}}$ ▷ p es el escalar que al restar la fila i escalada por p
    ▷ a la fila j vuelve 0 al elemento $a_j i$
    for $k = i$ **to** $n - 1$ **do** ▷ Antes de i ya son 0 porque fue triangulada
      $a_{jk} \leftarrow a_j k - p * a_i k$
    end for

    $b_j \leftarrow b_j - p * b_k$
  end for
end for
▷ Hasta acá A está triangulada
▷ Ahora queda resolver a A triangulada
$x \leftarrow VectorVacío(n)$
$x_{n-1} \leftarrow \frac{b_{n-1}}{a_{n-1\,n-1}}$
for $i = n - 1$ **to** 0 **do**
  $x_i \leftarrow b_i$
  for $j = i + 1$ **to** $n - 1$ **do**
    $x_i \leftarrow x_i - a_{i\,j} * x_j$ ▷ $x_i = b_i - \sum_{j=n-1}^{i+1} a_{i\,j} * x_j$
  end for

  $x_i \leftarrow \frac{x_i}{a_{i\,i}}$ ▷ $x_i = \frac{b_i - \sum_{j=n-1}^{i+1} a_{i\,j} * x_j}{a_{i\,i}}$
end for

$b' \leftarrow M * x$
$dist \leftarrow max(|b' - b|_2, |b - b'|_2)$
if $dist < tolerancia$ **then**
  print("se incurre en error numérico en base a una tolerancia")
end if

return x

De manera similar al algoritmo anterior (2.2), en un principio se verificó el correcto funcionamiento del algoritmo para sistemas con solución unica. La comparación de la respuesta dada por el algoritmo y la que generó el sistema fue realizada en base a una tolerancia de 10^{-8}.
Luego se buscó verificar que el algoritmo se comporte adecuadamente en los casos en los que la solución no exista o haya infinitas, es decir, la matriz pasada como parámetro sea singular. Se generaron matrices aleatorias, que luego fueron alteradas para asegurarse que sean singulares, y también vectores aleatorios. Luego se ejecutó la implementación en base a estos y se verificó que

la excepción levantada sea la correcta.

Este es un ejemplo con una matriz de coeficientes singular:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 0 & 0 & 0 \end{bmatrix} b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{9}$$

$$A_1 \leftarrow A_1 - 2 * A_0,\ b_1 \leftarrow b_1 - 2 * b_0$$
$$A_2 \leftarrow A_2 - 0 * A_0,\ b_2 \leftarrow b_2 - 0 * b_0$$
$$\downarrow$$
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} b = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

No se encuentra ningún elemento mayor a a_{22} por debajo de este, por lo que el pivoteo no efectúa nungún camibio de fila

$$\downarrow$$

$a_{i\,i} < tolerancia$ entonces

$$\downarrow$$

Excepción: "no se pudo resolver el sistema "

2.4. Matrices tridiagonales

Dadas matrices triadigonales, que solo contienen elementos no nulos en la diagonal principal y en las diagonales secundarias, es inconveniente triangularlas con los algoritmos desarrollados en las secciones previas. Teniendo en cuenta esto se desarrolló una versión simplificada de la eliminación gaussiana para ahorrar la ejecución pasos de innecesarios.

El funcionamiento básico del algoritmo es el mismo, se busca un factor tal que al multiplicar la fila i por este y restarselo a alguna de las filas inferiores anule el elemento de la columna i. La diferencia en este caso, es que la matriz ya presenta ceros en la mayoria de las posiciones donde se busca crearlos, por lo que sólo es necesario modificar una fila por paso de la ejecución. Además, se tiene la certeza de que cada fila contienen como máximo tres elementos no nulos, por lo que no es necesario realizar la resta en la fila completa(1).

Siguiendo esta idea, este nuevo algoritmo realiza sólo cuatro operaciones en cada paso de la ejecución, sin importar el tamaño de la matriz. Primero se calcula el pivot, con este se modifica el elemento por debajo de la diagonal en la fila *i+1*, el que se encuentra a su derecha y el elemento *i+1* del vector que contiene los términos independientes. El elemento que se encuentra a la derecha de la diagonal en la fila *i+1* no es necesario modificarlo ya que el elemento en esa columna de la fila i es nulo, y al por lo que la resta de este multiplicado por el pivot no genera alteración alguna.

Algorithm 3 Eliminación Gaussiana matriz tridiagonal $(in : A, b) \rightarrow x$

for $i = 0$ **to** $n - 1$ **do**
  if $a_{ii} = 0$ **then** ▷ loop sobre las columnas
    Error: Cero en la diagonal ▷ Se detiene si pivot 0
    stop
  end if
  $p \leftarrow \frac{a_{i+1,i}}{a_{ii}}$ ▷ p es el escalar que al restar la fila i escalada por p
  ▷ a la fila j vuelve 0 al elemento $a_j i$
  $A_k + 1_k + 1 \leftarrow A_k + 1_k + 1 - p * A_{kk} + 1$
  $b_{j+1} \leftarrow b_j + 1 - p * b_k$ ▷ b es el vector que tiene los terminos independientes
end for
return x
▷ Hasta acá A está triangulada
$x \leftarrow VectorVacío(n)$
if $A_n - 1_n - 1 = 0$ **then** ▷ El elemento de la matriz en la posición n-1,n-1 es el nulo no tiene solución
  Error: Cero en la solución
  stop
end if
$x_n - 1 \leftarrow \frac{b_{n1}}{A_n - 1, n_1}$
for $k = n - 2$ **to** 1 **do**
  $x_k \leftarrow \frac{k - A_{k\,k+}1 * x_{k+1}}{A_{k\,k}}$
end for

Otra forma de pensar este tipo de matrices es cómo si estuviesen dadas por tres vectores, uno por cada diagonal. Con el primer elemento de la diagonal inferior y el último de la superior igual a 0.

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \tag{10}$$

Basandose en esta representación sólo es necesario alterar los vectores a y b para triangular ya que los valores de c siempre se encuentran por encima de la diagonal principal. Aplicando las mismas transformaciones a otro vector, d, se consigue nuevamente un algoritmo que resuelve un sistema de ecuaciones dado por una matriz y un vector.

Algorithm 4 Eliminación Gaussiana sistema tridiagonal con vectores $(in : a, b, c, d) \rightarrow x$

$n = len(d)$
for $i = 0$ **to** $n - 1$ **do**
    if $a_{ii} = 0$ **then** ▷ loop sobre las columnas
        Error: Cero en la diagonal ▷ Se detiene si pivot 0
        stop
    end if
    $p \leftarrow \frac{a_{i+1}}{b_i}$ ▷ p es el escalar que al restar la fila i escalada por p
    $A_i + 1 \leftarrow A_i + 1 - p * b_i$
    $b_i + 1 \leftarrow b_i + 1 - p * c_i$
    $d_i + 1 \leftarrow c_i + 1 - p * d_i$
end for
▷ Hasta acá A está triangulada
$x \leftarrow VectorVacío(n)$
if $A_n - 1_n - 1 = 0$ **then** ▷ El elemento de la matriz en la posición n-1,n-1 es el nulo no tiene solución
    Error: Cero en la solución
    stop
end if
$x_n - 1 \leftarrow \frac{d_{n-1}}{b_{n-1}}$
for $i = n - 1$ **to** 0 **do**
    $x_i \leftarrow \frac{d_i - c_i * x_{i+1}}{b_1}$
end for
return x

Cada vez que cambia alguno de los vectores es necesario realizar toda la triangulación nuevamente. Para los casos particulares en los que se busca resolver múltiples sistemas con la misma matriz pero variando los términos independientes, es conveniente dividir el proceso en dos, con el fin de ahorrar tiempos de cómputo.
Primero se realiza un precálculo que triangula la matriz dada por los vectores *a, b* y *c*, y se guardan los *pivots*. Como el vector *a* siempre es nulo despés de la triangulación no es necesario devolverlo. De esta forma, al finalizar se puede ejecutar otro algoritmo que recibe los vectores *b* y *c* modificados, y un tercer vector *pivots*, que contiene los pasos que se realizaron para llegar a estos, y resuleva el sistema para cualquier *d* dado. Este segundo algoritmo realiza las operaciones necesarias en los terminos independientes y luego resuleve mediante substitución de la misma forma que todos los algoritmos previamente presentados.

Algorithm 5 Triangulacion tridiagonal $(in : a, b, c) \rightarrow b, c, pivots$

$n = len(d)$
$pivots \leftarrow VectorVacío(n)$
for $i = 0$ **to** $n - 1$ **do**
    if $b_i = 0$ **then** ▷ loop sobre las columnas
        Error: Çero en la diagonal" ▷ Se detiene si pivot 0
        stop
    end if
    $p \leftarrow \frac{a_{i+1}}{b_i}$ ▷ p es el escalar que al restar la fila i escalada por p
    $A_i + 1 \leftarrow A_i + 1 - p * b_i$
    $b_i + 1 \leftarrow b_i + 1 - p * c_i$
    $pivots_i \leftarrow p$
end for

return b,c, pivots

Algorithm 6 Resolver sistema tridiagonal $(in : b, c, pivots) \rightarrow x$

```
n = len(d)                                  ▷ Se usan los párametros dfevueltos por el algoritmo 4
for i = 1 to n do
    d_i ← d_i − pivots_{i+1} ∗ d_{i−1}
end for
x ← VectorVacio(n)
for i = n − 1 to0 do
    x ← (d_i − c_i ∗ x_{i+1}) / b_i
end for
return x
```

Al igual que en el caso de la eliminación gaussiana sin pivoteo, ninguno de estos algoritmos da una solución del sistema si se encuentran ceros en la diagonal principal en alguno de sus pasos. El intercambio de filas no se puede aplicar para este tipo de matrices ya que esta operación modificaría la estructura particular que se busca aprovechar.

Los tres algoritmos se testearon usando el mismo algoritmo que para la eliminación gaussiana (2.2), la diferencia es que en este caso se usaron matrices con estructura tridiagonal , además de ser no singulares.

2.5. Comparación de algoritmos

Una vez implementados todos loa algoritmos se buscó encontrar las ventajas y desventajas de cada uno. Para esto, se realizaron comparaciones en cuanto al timepo de cómputo y al error generado. Para realizar estas comparaciones se eligieron tamaños de matrices representativos y que se puedan correr en un tiempo acorde, teniendo en cuenta que para cada tamaño se realizó el cálculo múltiples veces.

En primer lugar, se evaluó la complejidad temporal de las implementaciones. Se evaluaron las diferencias entre la eliminación gaussiana con y sin pivoteo. Para el caso de matrices tridiagonales, se analizó el contraste entre la eliminación gaussiana con pivoteo y la eliminación gaussiana para matrices tridiagonales. Por último, se comparó el rendimiento temporal del algoritmo de eliminación tridiangular con y sin precomputo.

Con el mismo objetivo, se compararon las diferencias del error generado por el algoritmo de eliminación con y sin pivoteo, y se midió la efectividad de los algoritmos para matrices triangulares. Las dos versiones presentadas de este algoritmo realizan exactamente las mismas operaciones, por lo que la comparación del error es trivial ya que este es equivalente.

Las matrices utilizadas en la comparación fueron generadas aleatoriamente utilizando la función *random* de numpy. En el caso de matrices tridiagonales, se utilizó una función que devuelve matrices aleatorias con dicha estructura. Para medir los tiempos de los algoritmos, se utilizó la función de Python *timeit* [12], que corre una función una misma cantidad de veces y guarda los datos significativos. Luego se graficó el tiempo promedio, para esto se utilizó la librería matplotlib.

2.6. Aplicaciones

Una vez desarrollados todos los algoritmos, se pudieron encarar las aplicaciones de estos a casos y fenómenos particulares.
Dado un vector d, aproximación finita de una función continua, se busca encontrar al vector u tal que la función aproximada por d sea la derivada segunda de la función aproximada por u.

$$\frac{d^2}{dx^2}u = d \tag{11}$$

[12] docs.python.org

Para lograr esto se comenzó generando la matriz del operador laplaciano discreto (1) para el tamaño de d correspondiente. Como se explicó previamente, cuando el tamaño de la matriz y de los vectores tiende a infinito, multiplicar esta matriz por un vector que discretice la función u permite calcular su derivada segunda.
A partir de esta, se utilizó el algoritmo de eliminación gaussiana para matrices tridiagonales con el fin de que la solución del sistema equivalga al u buscado. Este algoritmo fue ejecutado con tres d diferentes de tamaño $n = 101$.

a) $d_i = \begin{matrix} 0 \\ 4/n \end{matrix} \quad i = [n/2] + 1$

b) $d_i = 4/n^2$

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

Las funciones resultantes fueron graficadas con respecto a una sección de su dominio, x. (2.6)

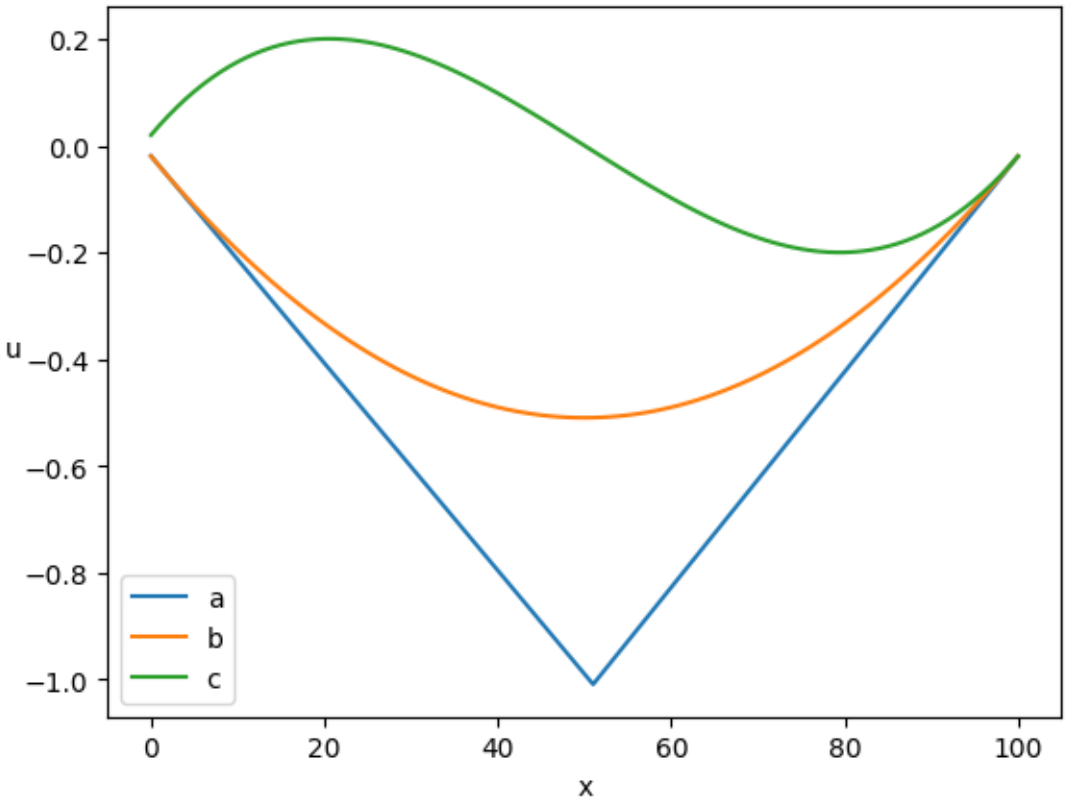

Figura 2: Funciones asociadas a los distintos d

Como fue nombrado antes, el operador laplaciano se puede utilizar para el modelado de una difusión en el espacio en donde la diferencia de concentración en cada paso se representa como una fracción del operador laplaciano.
Para facilitar los cálculos, esto se puede reordenar para llegar a una operacion matricial del tipo

$$Au^{(k)} = u^{(k-1)} \tag{12}$$

Tomando la ecuación original (1):

$$u_i^{(k)} - u_i^{(k-1)} = \alpha(u_{i-1}^{(k)} - 2u_i^{(k)} + u_{i+1}^{(k)}) \tag{13}$$

$$u_i^{(k)} = \alpha u_{i-1}^{(k)} - 2\alpha u_i^{(k)} + \alpha u_{i+1}^{(k)} + u_i^{(k-1)} \tag{14}$$

$$u_i^{(k)} - \alpha u_{i-1}^{(k)} + 2\alpha u_i^{(k)} - \alpha u_{i+1}^{(k)} = u_i^{(k-1)} \tag{15}$$

$$-\alpha u_{i-1}^{(k)} + (1+2\alpha) u_i^{(k)} - \alpha u_{i+1}^{(k)} = u_i^{(k-1)} \tag{16}$$

Cada elemento i del vector $u^{(k-1)}$, es igual al producto entre la fila i de una matriz A y el vector columna $u^{(k)}$. A partir del desarrollo previo se llegó a que dicha matriz tiene esta forma:

$$A = \begin{bmatrix} 1+2\alpha & -\alpha & & & 0 \\ -\alpha & 1+2\alpha & -\alpha & & \\ & -\alpha & 1+2\alpha & \ddots & \\ & & \ddots & \ddots & -\alpha \\ 0 & & & -\alpha & 1+2\alpha \end{bmatrix} \tag{17}$$

Como se puede observar el resultado es una matriz tridiagonal, por lo que se utilizó el algoritmo de eliminación gaussiana para matrices tridiagonales para calcular el valor que toma u en cada paso. Para lograr modelar esta difusión en un periodo de tiempo, se resolvió el sistema de manera iterativa, con una iteración por cada Δt evaluado. La matriz derivada de la ecuación de difusión discreta no varia en ningún paso, es por esto que primero se realizó un precómputo para triangularla, y luego se fue alterando el término independiente a partir del último u calculado. Cada uno de estos vectores resultantes fue almacenado en una matriz, en la que cada paso de la difusión está representando por una columna.
Con esta metodología se graficó (2.6) la evolución de un vector u a lo largo de 1000 iteraciones, tomando la condición inicial:

$$u_i^{(0)} = \begin{matrix} 0 \\ 1 \end{matrix} \quad \begin{matrix} \\ [n/2] - r < i < [n/2] + r \end{matrix} \tag{18}$$

Con $n = 101$, $r = 10$ y $\alpha = 1$

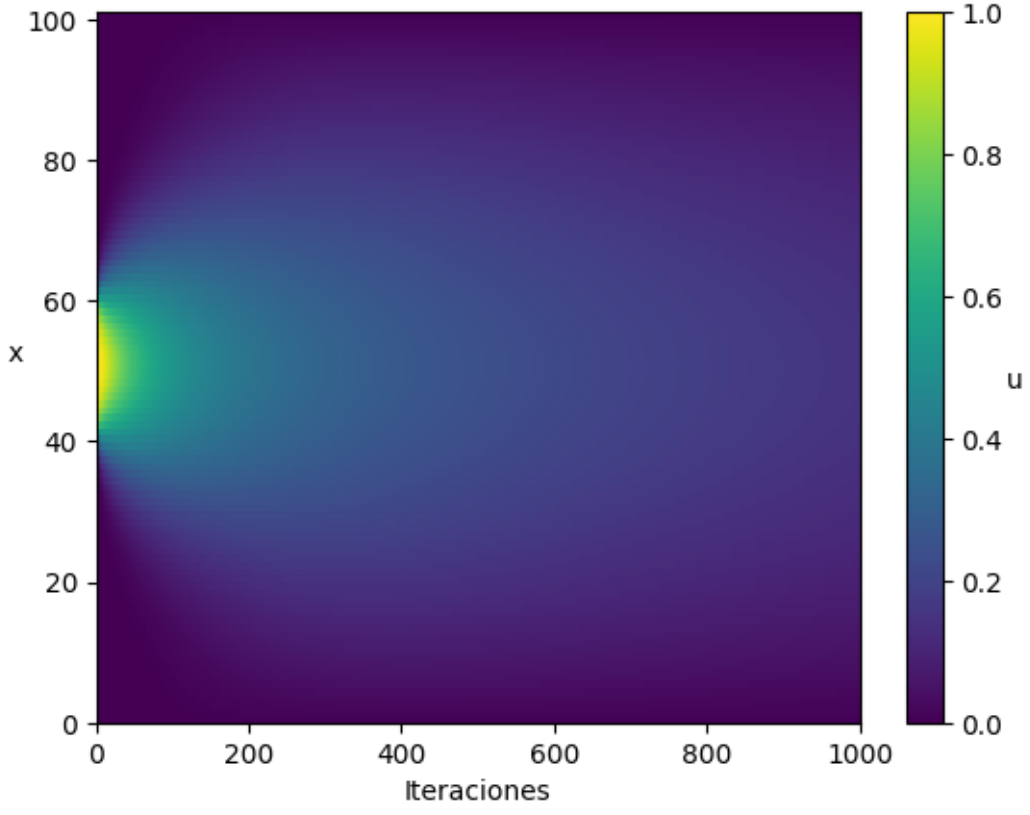

Figura 3: Difusion en el tiempo

3. Resultados y discusión

Estos algoritmos presentan beneficios y desventajas. La favorabilidad del uso de uno sobre el otro es dependiente del uso puntual que se le busca dar. Esta desición dependerá principalmente de tres factores, las certezas que se tienen sobre la estructura de la matriz, la velocidad de cómputo y la exactitud de la solución.

Para calcular el error se siguieron los siguientes parametros:
Se definió al $error_x$ como la distancia entre $\hat{x}$, resultado del la ejecución del algoritmo, y el x real que genera el sistema. De forma tal que:

$$error_x = ||x - \hat{x}||_2 \tag{19}$$

Para generar los parámetros en base a los cuales se van a ejecutar los algoritmos, primero se crearon matrices aleatorias A con numpy. Para los casos en los que se testeó el algoritmo de eliminación gaussiana sin pivoteo, se verificó que estas puedan ser resueltas sin intercambio de filas.
Luego se generó de igual forma un vector x de la misma dimensión. Al A con este, se consiguió un vector con términos independientes b. Al crear los parámetros de esta forma se aseguró que el sistema siempre tenga solución, y se pudo almacenar al x real con el cual se va a realizar la comparación. Finalmente, se claculó $\hat{x}$ ejecutando el algoritmo a testear con la matriz y elvector antes definidos.
Una vez hechos todos estos cálculos es posible medir el error. Se evaluaron 8 dimensiones distintas uniformemente distribuidas dentro del rango. En cada dimensión se evaluaron 200 sistemas aleatorios distintos. Se calcularon el promedio y la mediana y fueron expresados en el gráfico 3.

Se desestimó al error calculado en base a b, $error_b = ||b - A * \hat{x}||_2$, ya que varía en base al número de condición de la matriz, y no necesariamente refleja la exactitud de los algoritmos.

En los casos del algoritmo de eliminación gaussiana con y sin pivoteo, se puede observar que el más simple de los dos tiene un tiempo de cómputo menor. Esto ocurre ya que, al no tener que realizar comparaciones, por cada iteración debe realizar menos operaciones. De todas formas, ambos tienen una compléjidad teórica de $O(n^3)$ por lo que la diferencia en velocidades no es significativa.

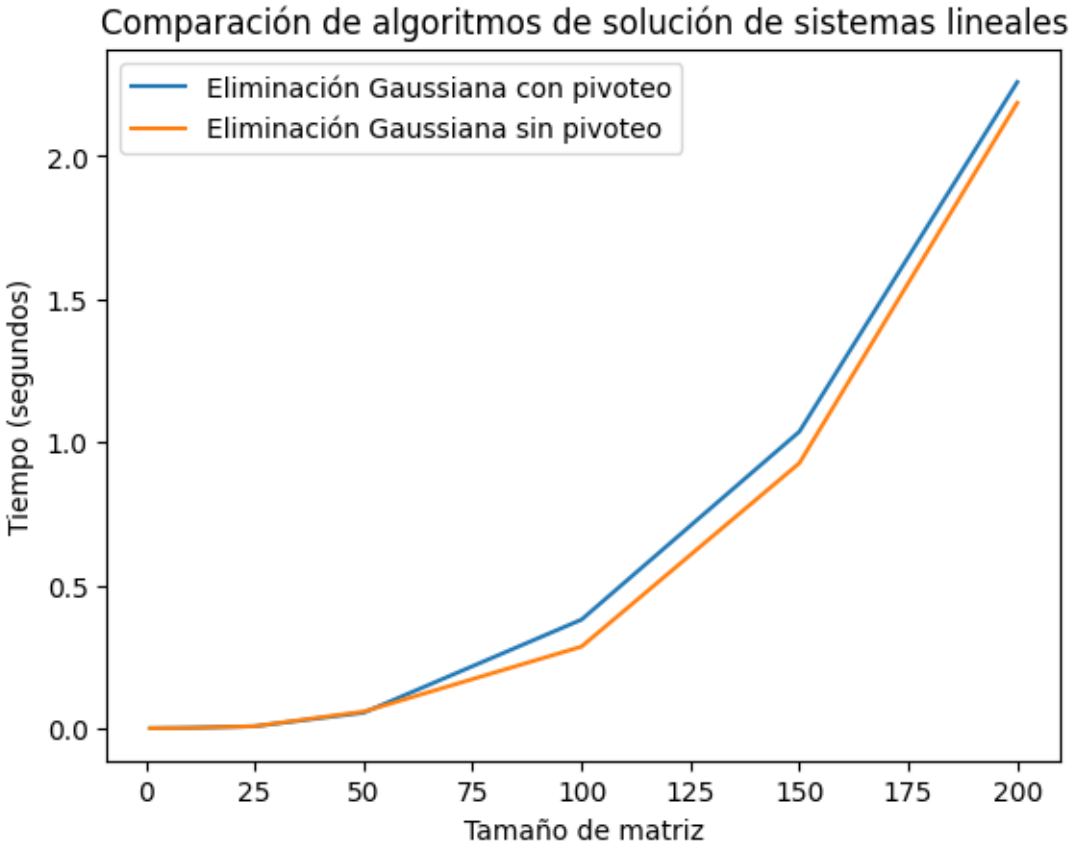

Figura 4: Comparación temporal de algoritmos de eliminación gaussiana

Es por esto que, en la mayoria de los casos resulta preferible utilizar el algoritmo con pivoteo. Este no solo permite resolver una mayor cantidad de sistemas, dado que resuelve todos los que tienen solución única, sino que también disminuye el error numérico en gran medida.

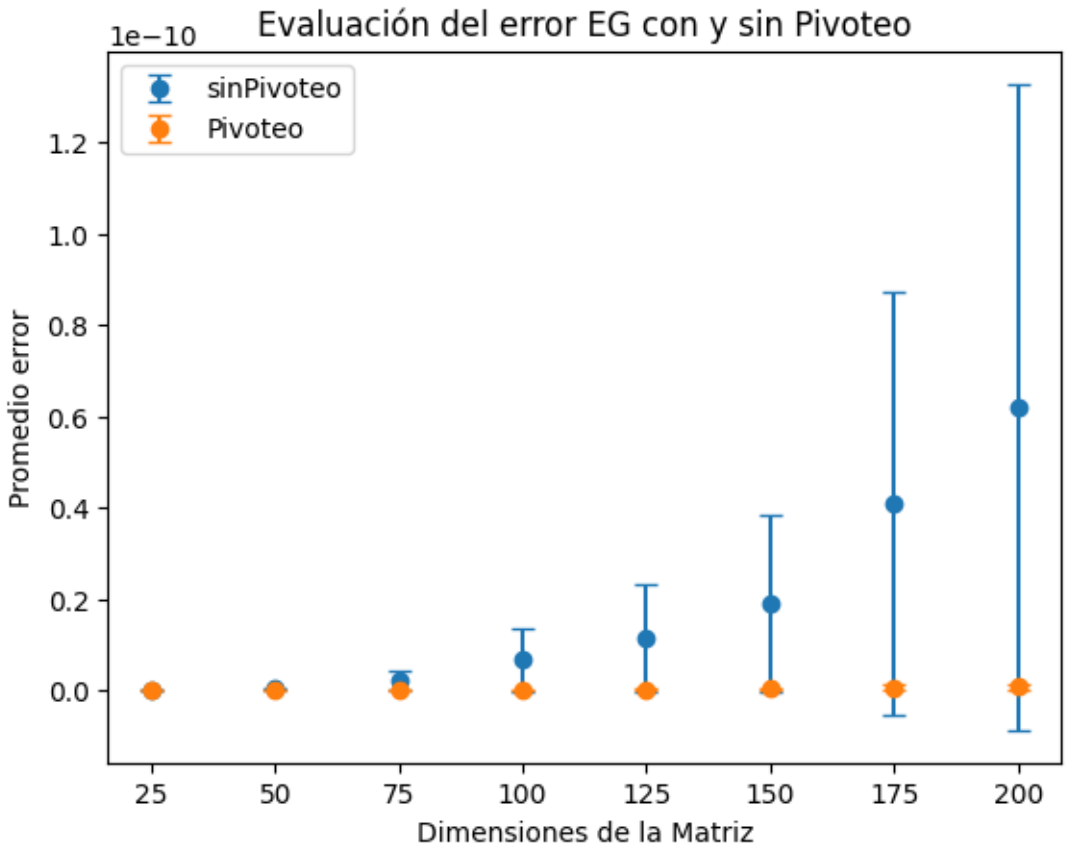

Figura 5: Comparación algoritmo eliminación gaussiana. En cada dimensión se midió el error de 200 sistemas de ecuaciones de cada método y se calculó el promedio y el desvío estandar

En este gráfico se observa que, las precauciones tomadas para disminuir los errores fueron altamente efectivas. Aún así es imposible erradicarlo por completo. Esto se debe a que los errores de redondeo así como el overflow y el underflow son imprevenibles.

Por otro lado, si se conoce la estructura de la matriz con la que se va a operar de antemano, es altamente ventajoso idear un algoritmo que que se aproveche de esta y no realice cómputos de más. El siguiente gráfico refleja cómo el algoritmo de eliminación gaussiana para matrices tridiagonales tiene una complejidad lineal. Esto se adapta a las conjeturas hechas en las secciones previas (2.4), en las que se analiza que se realizan una cantidad de operaciones fija en cada paso de la eliminación sin importar el tamaño de la matriz. Por el lado contrario, el algoritmo que implementa el pivoteo debe recorrer todas las filas de la matriz en cada paso, por lo que el costo se vuelve exponencial.

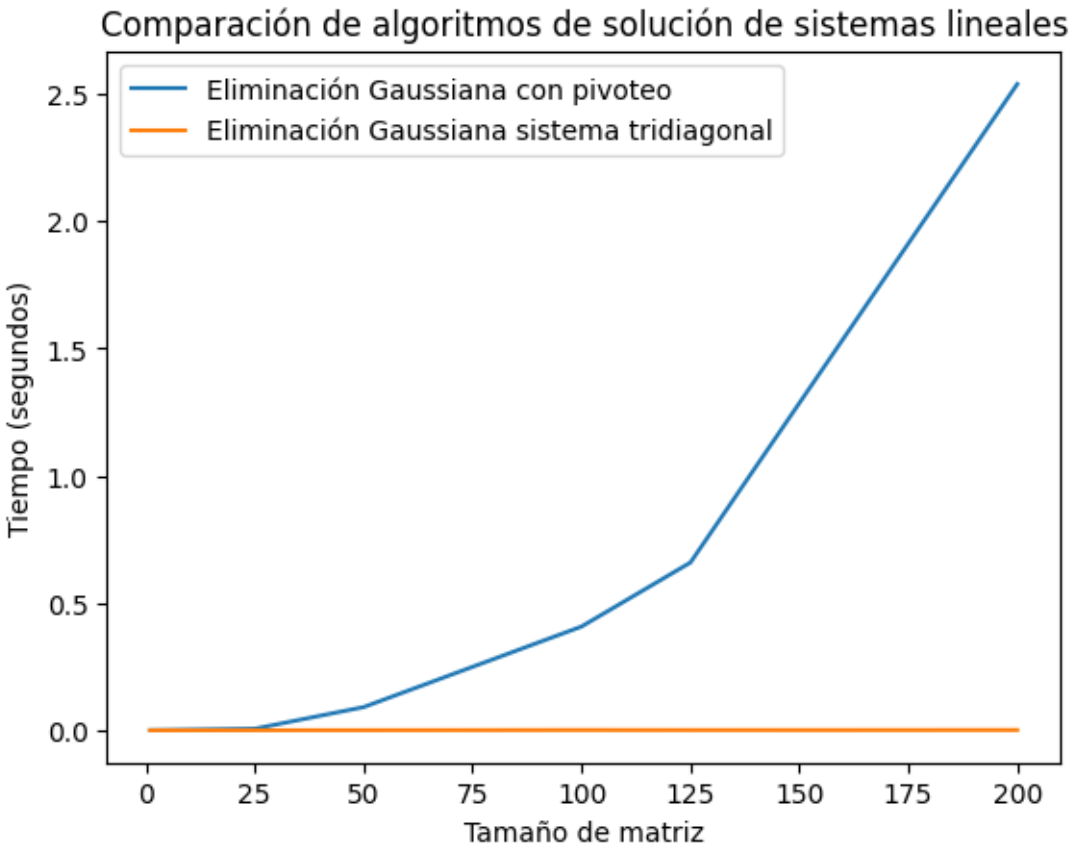

Figura 6: Resolución sistema tridiagonal

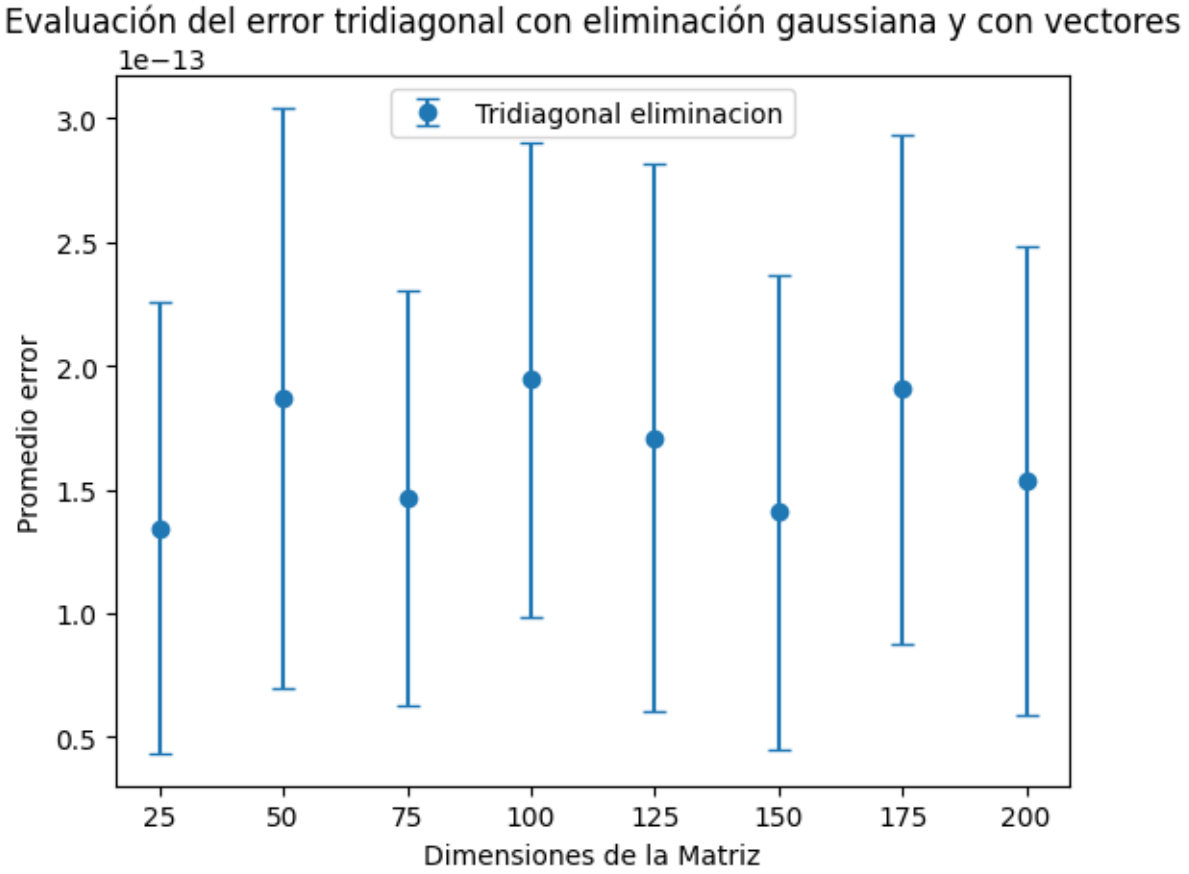

Figura 7: Error algoritmo sistema tridiagonal

Se puede observar que el algoritmo para matrices tridiagonales no genera errores significativos. Todos los valores calculados se encuentran por debajo del umbral de tolerancia de 10^{-8} establecido al comienzo del informe. Esto indica que no hay ninguna ventaja aparente a utilizar el algoritmo con pivoteo por encima de este.

El mismo beneficio se presenta a la hora de resolver el sistema tridiagonal en dos partes. Con los ejemplos de aplicaciones dados antes (2.5) se pudo observar que hay múltiples usos de estos algoritmos en los que se debe resolver un sistema repetidas veces variando sólo el término independiente. En estos casos es un desperdicio de tiempo de cómputo realizar la misma triangulación en todas las repeticiones. Si se la triangula en un principio y se alamacenan los pivots utilizados para hacerlo, la primera ejecución del código tendrá la misma complejidad pero, en el resto de las repeticiones esta se verá altamente reducida.

Se puede observar que en el caso de matrices tridiagonales, si se realiza una eliminación gaussiana con pivoteo se tiene un error más pequeño que si se realiza sin esta. Esto era de esperarse porque se evita un error númerico por divisió muy pequeña. Igual en ambos casos el error es muy chico y por debajo de la tolerancia esperada. Otra observación es que si se compara la figura (5)

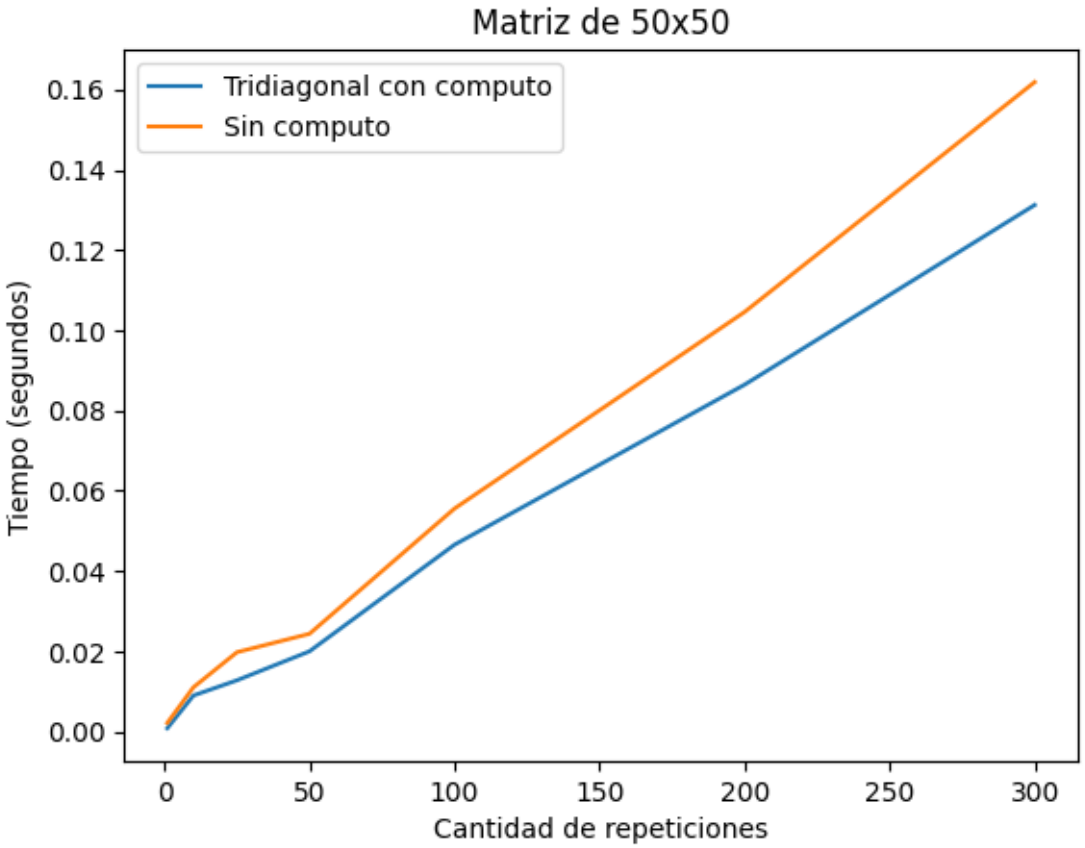

Figura 8: Tiempo distintas repeticiones precómputo y sin cómputo

Este gráfico muestra las diferencias que se producen al usar precómputo en una matriz tridiagonal de dimensión 50 en un diferente rango de repeticiones. Como se había teorizado, el computo previo acelera la ejecución del algortimo. Esta diferencia se remarca cuantas más veces es necesario generar una solución.

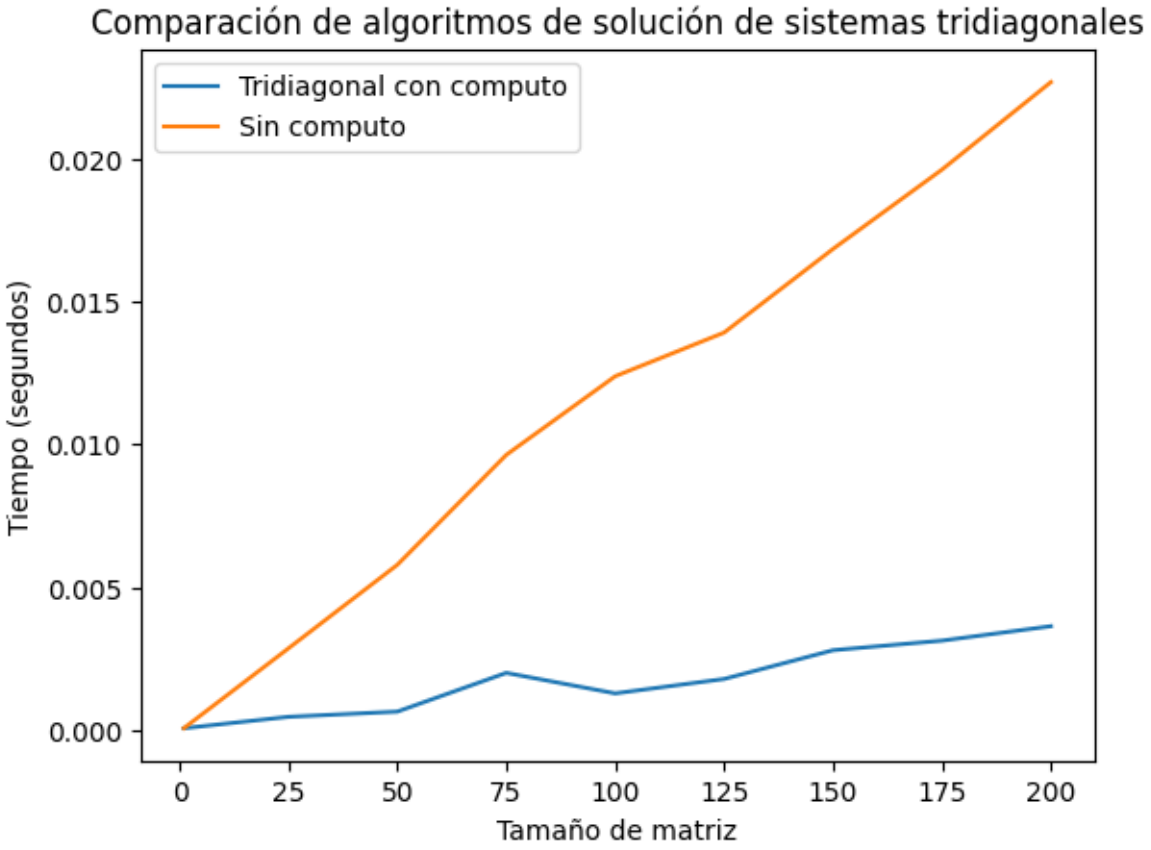

Figura 9: Resolución sistema tridiagonal

Por otro lado, este segundo gráfico ilustra cómo esta diferencia se hace mas notoria al aumentar la dimension de la matriz sobre la que se debe operar. Para cada tamaño se resolvió el sistema diez veces cambiando únicamente el término independiente en cada una. Como se había concluido previamente, ambos algoritmos tienen una compléjidad temporal lineal, pero la pendiente de la que no realiza un precómputo es significativamente mayor.

El error generado por los tres algoritmos para matrices tridiagonales implementados en este trabajo es el mismo. Los tres algoritmos realizan las mismas operaciones en cada paso, por lo que todo error de redondeo o de cálculo generado en uno, es replicado por los otros dos.

4. Conclusiones

La resolución de sistemas de ecuaciones es un problema que se presenta ampliamente en diversos campos de la ciencia, por lo que es importante idear y optimizar algoritmos, tanto temporalmente como con respecto al error que producen.

Si no se pueden tener certezas sobre la estructura de la matriz que será estudiada, el algoritmo de eliminación gaussiana con pivoteo nos asegura que, de tener solución el sistema, esta será encontrada. Asimismo asegura una mayor efectividad a la hora de producir estas soluciones. De todas formas, si lo que se busca es velocidad este algoritmo realizas una cantidad de operaciones mucho mas elevada que la de los otros presentados en este trabajo.

Cuando se presentan aplicaciones puntuales que permiten tener seguridades sobre la forma de los parámetros, tomar ventaja de estos a la hora de idear los algoritmos es de gran beneficio.

Por último, el error numérico es inevitable a la hora de ralizar cálculos con la computadora. Es de gran importancia conocer las fuentes de errores y tenerlas en cuenta ya que estos pueden hacer que el algoritmo termine no cumpliendo su finción. Támbien es necesario, dentro de lo posible, operar con matrices que se encuentren bien condicionadas para el algoritmo elegido. Esto permite reducir el error en gran magnitud.