Miscelánea Matemática **80** (2025) 67-85 SMM

DOI: https://doi.org/10.47234/mm.8006

Complejidades matemáticas y computacionales

Andreas Wachtel

Departamento Académico de Matemáticas
ITAM
andreas.wachtel@itam.mx

1. Introducción

En este trabajo se da un ejemplo que justifica la importancia de continuar con la formación sólida de estudiantes en áreas como matemáticas aplicadas, ciencias de datos y computación o, más general, en las áreas STEAM *(Science, Technology, Engineering, Arts and Mathematics)*.

En estas áreas, frecuentemente nos encontramos en la necesidad de resolver problemas matemáticos. Usualmente, convertimos el proceso de resolver un problema en un algoritmo y lo implementamos en un *lenguaje* (de programación). Tal vez, al correr el código, el algoritmo nos parece lento, lo cual nos hace buscar consejos de una inteligencia artificial (I.A.) o entre nuestras amistades. Los consejos pueden sugerir que cambiemos el lenguaje o proponer un algoritmo alternativo.

Aquí se expone un problema matemático, que tiene un proceso de resolución (y complejidad matemática) que convertiremos en 4 algoritmos. Estos nos permitirán presentar experimentos y teoría que justificarán que distintos lenguajes (Python, Julia, MatLab, Octave) responden de manera distinta, en tiempo de cómputo y complejidad, a un mismo algoritmo.

Veremos que el algoritmo de menor costo computacional cambia con el lenguaje en que se programa, lo cual justificará que traducir algoritmos entre lenguajes, con o sin ayuda de una I.A., no siempre produce los algoritmos más eficientes. Más aún, pedirle a una I.A. un algoritmo, no siempre produce el algoritmo más eficiente, como se muestra al final de este trabajo.

Palabras clave: algoritmo eficiente, complejidad matemática, complejidad computacional.
Este trabajo fue apoyado por la Asociación Mexicana de Cultura A.C.

El principal objetivo de este artículo es encontrar algoritmos equivalentes, que resuelven un problema matemático, e identificar el que se comporta como la complejidad matemática predice, con el fin de reducir tiempo de cómputo y energía consumida en la solución del problema. Para poder profundizar el análisis, el resto del trabajo se enfoca en un problema sencillo, que es el proceso de solución de sistemas de ecuaciones lineales en forma triangular superior, con algoritmos que realizan la sustitución hacia atrás.

El estudio de sistemas de ecuaciones lineales en forma triangular superior es de suma importancia ya que aparecen en la solución de problemas más complicados. Por ejemplo, dado cualquier sistema lineal cuadrado $A\vec{x} = \vec{b}$, la eliminación gaussiana, la factorización QR y la factorización LU, primero convierten el sistema en uno triangular superior, véase [1, cap. 5]. Adicionalmente, un problema lineal de mínimos cuadrados, primero se convierte (usando una factorización QR) también en un sistema triangular superior, véase [1, cap. 6].

En la sección 2 plantearemos el problema y mostraremos que resolverlo tiene una complejidad teórica, que denominaremos *complejidad matemática*. Además, mostraremos los 4 algoritmos para resolverlo, que hacen matemáticamente los mismos cálculos, pero en orden diferente.

Los experimentos, en la sección 3, permiten concluir qué lenguajes aprovechan, de manera diferente, la comunicación interna de la computadora, ilustrada en la figura 1, ya que el costo computacional y complejidad de un algoritmo varía con el lenguaje. Los experimentos permiten identificar el algoritmo (entre los 4), que resuelve el problema de la sección 2, con el mayor ahorro de energía y con una «*complejidad computacional*» que se parece a la matemática.

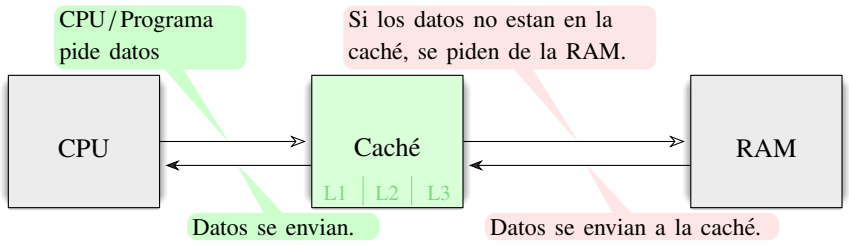


Figura 1. Comunicación simplificada entre CPU – Caché y RAM. Las abreviaciones *CPU* y *RAM* son las siglas de *central processing unit* y *random access memory*, respectivamente.

En la sección 4 se explica, más a fondo, la comunicación ilustrada en la figura 1, lo cual permitirá justificar teóricamente que dos algoritmos en Python tienen diferentes complejidades computacionales.

La conclusión del trabajo consiste de dos partes. Se presentará un resumen del conocimiento expuesto para poder mejorar la eficiencia de

futuras implementaciones de nuevos algoritmos, aún no conocidos o disponibles en el lenguaje. Además, se hace una comparación de nuestros mejores algoritmos con los proporcionados por ChatGPT, que motivará un mensaje final a estudiantes y profesores de las áreas STEAM.

2. Un problema matemático y cuatro algoritmos

A continuación, presentamos nuestro problema matemático sencillo con el fin de definir los 4 diferentes algoritmos. Para ello necesitamos la siguiente definición.

Definición 2.1. Una matriz $U \in \mathbb{R}^{n\times n}$ es *triangular superior* cuando $i > j \implies U_{ij} = 0$, *i.e.*, las entradas debajo de su diagonal son cero.

Problema 2.2. *Encuentre la solución $\vec{x}$ al sistema lineal $U\vec{x} = \vec{b}$, donde $U \in \mathbb{R}^{n\times n}$ es una matriz triangular superior y $\vec{x}, \vec{b} \in \mathbb{R}^{n\times 1}$ son vectores de n entradas en los reales.*

Antes de mostrar la solución general de este problema, ilustramos el problema y su solución para $n = 3$:

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Suponiendo que U tiene inversa las entradas de la solución $\vec{x}$ son:

$$x_3 = b_3/U_{33}, \;\; x_2 = (b_2 - x_3U_{23})/U_{22}, \;\; x_1 = (b_1 - x_2U_{12} - x_3U_{13})/U_{11}.$$

En general, cuando U tiene inversa, la solución $\vec{x}$ del problema 2.2 es única y sus entradas están dadas por

$$x_i = \left[b_i - \sum_{j=i+1}^{n} U_{ij}x_j \right] / U_{ii} \qquad \text{, para } i \text{ de } n, \ldots, 1\,. \tag{1}$$

Estas igualdades, con la i decreciente, también definen la *sustitución hacia atrás*, que se refiere al proceso de despejar las entradas de $\vec{x}$ en el orden $x_n, x_{n-1}, \ldots, x_1$. Históricamente, el autor de [6] especula que esa sustitución es conocida desde 1690 por Michel Rolle. Nuestros 4 algoritmos implementarán este proceso.

Enseguida definimos la complejidad matemática (teórica) del problema 2.2, ya que el costo computacional de un buen algoritmo se debe comportar como esa complejidad predice.

2.1 La complejidad matemática

Para calcular las entradas de la solución $\vec{x}$ del problema 2.2, dados por la ecuación (1), se requieren operaciones (multiplicaciones y divisiones). Observamos que despejar la entrada x_i requiere calcular $\sum_{j=i+1}^{n} U_{ij}x_j$ con $n-i$ multiplicaciones y 1 división. Por lo tanto, despejar todas las entradas, requiere $\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$ multiplicaciones y n divisiones, obteniéndose un total de $n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ operaciones. Debido a $\frac{n^2}{2} \leqslant \frac{n(n+1)}{2} \leqslant n^2$ se dice que el número de operaciones crece como n^2. En este trabajo definimos a la *complejidad matemática* como el número de las multiplicaciones

$$\#\text{mults} = \frac{n(n-1)}{2}\,, \tag{2}$$

ya que esas realmente son las que causan el crecimiento n^2. Esta definición también simplificará la discusión de las complejidades computacionales presentada más adelante.

Comportamientos como el de la expresión (2) existen en varios problemas matemáticos. En la computadora, el crecimiento de operaciones involucradas en un problema se refleja en el tiempo $T(n)$, que un algoritmo tarda en resolver dicho problema y nos permite predecir cómo ese tiempo debe aumentar cuando se resuelve un problema más grande.

Por ejemplo, un algoritmo para el problema 2.2 debe tardar un tiempo $T(n)$ proporcional a n^2. Si suponemos que $T(n) = t_0 n^2$, donde t_0 representa un promedio de tiempo por operación independiente de n, entonces esta hipótesis implica que $T(3n) = t_0(3n)^2 = 3^2 T(n)$. Con ello en mente, si un algoritmo tarda 5 segundos para resolver un sistema con $n = 100$, es decir, $T(100) = 5s$, entonces resolver un sistema de dimensión $300 = 3n$ debe tardar aproximadamente $45s$, ya que $T(3n) \approx 3^2 T(n) = 9 \cdot 5s = 45s$. La aproximación se debe a que t_0 cambia con la dimensión y comunicación interna en la computadora.

2.2 Construcción de los cuatro algoritmos

La sustitución hacia atrás (1), que resuelve el problema 2.2, se puede implementar de 4 maneras distintas, que resultan en 4 algoritmos. Estos se distinguen en el orden que utilizan las entradas de la matriz U (por renglón o columna) y en el número de veces que se sobreescriben las entradas de $\vec{x}$. En teoría, matemáticamente los cuatro algoritmos hacen las mismas $\frac{1}{2}n(n+1)$ operaciones en orden diferente, es decir, la entrada i de la solución $\vec{x}$ está dada por la expresión (1), pero la suma es calculada en orden diferente, como veremos abajo. En la computadora, con sumas redondeadas, los resultados pueden diferir un poco, pero

esto no será objeto de estudio aquí. El lector interesado puede consultar [7].

Dentro de los cuatro algoritmos de abajo, los índices i, j son enteros y la segunda línea implementa el orden impuesto por la sustitución hacia atrás, mientras las líneas 3–6 distinguen a los algoritmos.

El primer algoritmo 2.1 calcula las entradas (1) con dos ciclos. El ciclo interior (líneas 3–5), para $i < n$, resta entrada por entrada la suma $\sum_{j=i+1}^{n} x_j U_{ij}$. Más aún, si se calcula la suma anterior como un producto punto entre $U_{i,[i+1:n]} = (U_{i,i+1}, \ldots, U_{i,n})$ y $\vec{x}_{i+1:n} = (x_{i+1}, \ldots, x_n)^\top$, entonces obtenemos el algoritmo 2.2 con solo un ciclo.

Algoritmo 2.1 (por renglón y entrada por entrada)**.**

1: *copiar* $\vec{x} \leftarrow \vec{b}$
2: ***for*** i *decreciente de* $n, \ldots, 1$ ***do***
3: ***for*** j *de* $n, \ldots, i+1$ ***do*** ▷ $(n, \ldots, n+1) = \emptyset$
4: *asignar* $x_i \leftarrow x_i - U_{ij} \cdot x_j$ ▷ *restar ent.* $\times$ *ent.*
5: ***end for***
6: *asignar* $x_i \leftarrow x_i / U_{ii}$
7: ***end for***

Algoritmo 2.2 (por renglón y vectorizado)**.**

1: *copiar* $\vec{x} \leftarrow \vec{b}$
2: ***for*** i *decreciente de* $n, \ldots, 1$ ***do***
3: ***if*** $i < n$ ***then***
4: *asignar* $x_i \leftarrow x_i - \left\langle U_{i,[i+1:n]}, \vec{x}_{[i+1:n]} \right\rangle$ ▷ *producto punto.*
5: ***end if***
6: *asignar* $x_i \leftarrow x_i / U_{ii}$
7: ***end for***

Los siguientes dos algoritmos resuelven el sistema leyendo las entradas de U por columnas y no por filas, como en los algoritmos anteriores. El algoritmo 2.3 tiene dos ciclos y en la línea 5 las restas se realizan individualmente.

Algoritmo 2.3 (por columna y entrada por entrada)**.**

1: *copiar* $\vec{x} \leftarrow \vec{b}$
2: ***for*** j *decreciente de* $n, \ldots, 1$ ***do***
3: *asignar* $x_j \leftarrow x_j / U_{jj}$
4: ***for*** i *decreciente de* $j-1, \ldots, 1$ ***do*** ▷ $(0, \ldots, 1) = \emptyset$
5: *asignar* $x_i \leftarrow x_i - U_{ij} \cdot x_j$ ▷ *restar ent.* $\times$ *ent.*
6: ***end for***
7: ***end for***

Si juntamos estas restas individuales de línea 5 del algoritmo 2.3 en una resta vectorial involucrando $U_{[1:j-1],j} = (U_{1,j}, \ldots, U_{j-1,j})^\top$ y

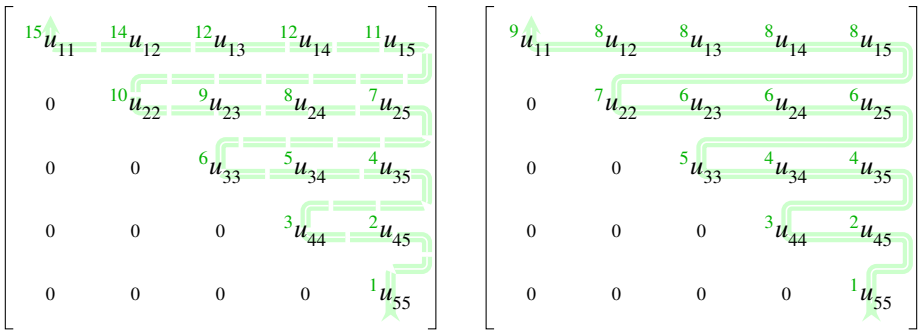
$\vec{x}_{[1:j-1]} = (x_1, \ldots, x_{j-1})^\top$, entonces obtenemos el algoritmo 2.4 con solo un ciclo.

Algoritmo 2.4 (por columna y vectorizado)**.**

1: *copiar* $\vec{x} \leftarrow \vec{b}$
2: ***for*** j *decreciente de* $n, \ldots, 1$ ***do***
3: *asignar* $x_j \leftarrow x_j / U_{jj}$
4: ***if*** $1 < j$ ***then***
5: *asignar* $\vec{x}_{[1:j-1]} \leftarrow \vec{x}_{[1:j-1]} - x_j U_{[1:j-1,j]}$ ▷ *resta vectorial.*
6: ***end if***
7: ***end for***

Ya que los algoritmos 2.2 y 2.4 ocupan varias entradas de U en forma de vectores para hacer una sola asignación, decimos que estos dos algoritmos son *vectorizados*.

Finalmente, los caminos verdes en las figuras 2 y 3 visualizan el orden de acceso de cada algoritmo para matrices $U \in \mathbb{R}^{5\times 5}$. La notación ${}^{\kappa}u_{ij}$ indica que la entrada u_{ij} es la κ-ésima entrada leída por el algoritmo. Además, las figuras para los algoritmos vectorizados 2.2 y 2.4 muestran varias veces el mismo índice κ para indicar una operación de vectores.



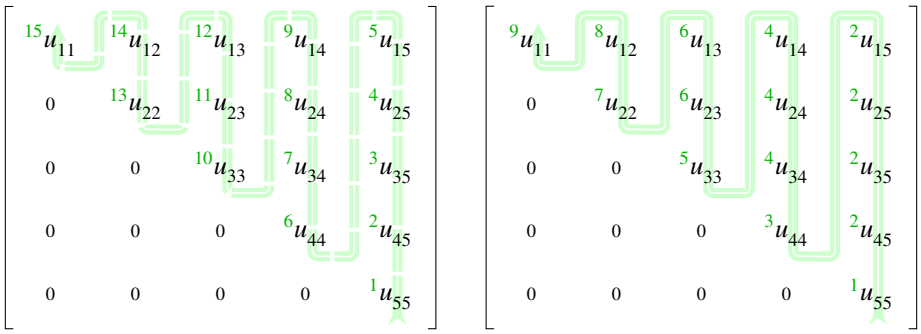
(a) orden del algoritmo 2.1 **(b)** orden del algoritmo 2.2

Figura 2. Como los algoritmos por renglones leen los datos de $U \in \mathbb{R}^{5\times 5}$.

Aunque resolver sistemas lineales es bien conocido por siglos [6], las técnicas para construir los algoritmos para la computadora, que resuelvan dichos sistemas, son más recientes y algunas de ellas son también bien conocidas. Por ejemplo, el algoritmo 2.1 se encuentra en [1, p. 95], mientras en [5, p. 81] se encuentran algoritmos para sistemas triangulares inferiores similares a nuestros algoritmos 2.1, 2.2 y 2.4.

3. Experimentos y observaciones

En esta sección descubrimos, con experimentos, que en los lenguajes Python y Octave uno de los algoritmos vectorizados es hasta 98 % más



(a) orden del algoritmo 2.3 **(b)** orden del algoritmo 2.4

Figura 3. Como los algoritmos por columnas leen los datos de $U \in \mathbb{R}^{5\times 5}$.

rápido que su versión de dos ciclos. Por otro lado, en Julia conviene usar un algoritmo de dos ciclos. Después, veremos que la complejidad de los algoritmos cambia con el lenguaje y que nuestro problema 2.2 se puede resolver en tiempos parecidos en los distintos lenguajes.

Los experimentos presentados miden el tiempo que los algoritmos tardan en resolver un sistema y se hicieron en una computadora con sistema operativo «Ubuntu 20.04» con 8 GiB[1] de *RAM* y una *CPU* de tipo Intel Core i5 (8^a generación) con caché de niveles $\{L1 \,|\, L2 \,|\, L3\}$ de tamaños $\{256\,\text{KiB} \,|\, 1\,\text{MiB} \,|\, 6\,\text{MiB}\}$[2].

Adicionalmente, existe una memoria virtual llamada *swap* que es muy lenta, por estar ubicada en el disco duro. Cabe señalar que antes de realizar los experimentos, para cada lenguaje, el sistema operativo fue reiniciado y se apagó el uso de la memoria virtual *swap*.

3.1 Algoritmos *vs* lenguajes

Aquí comparamos los 4 algoritmos presentados anteriormente, tomando como base los tiempos que tardan en resolver el problema 2.2 los lenguajes Python 3.9.18, Octave 9.2, MatLab R2024a y Julia 1.10.4. En cada experimento, los sistemas fueron de la forma $U\vec{x} = (1, \ldots, 1)^\top$ donde $U \in \mathbb{R}^{n\times n}$ fue triangular superior con la diagonal $\{1, 2, \ldots, n\}$ y entradas aleatorias en $[-1/2, 1/2]$ arriba de la diagonal.

Para cada algoritmo en cada lenguaje hemos medido $11 = m+1$ tiempos T_e para $e \in \{0, 1, \ldots, m\}$. Luego hemos ignorado el tiempo T_0 y calculado un promedio $\langle T\rangle = \frac{1}{m}\sum_{e=1}^{m} T_e$ el cual se encuentra en el cuadro 1. Consideramos que los tiempos obtenidos son confiables ya que para cada entrada medimos la desviación estándar $\left[\frac{1}{m-1}\sum_{e=1}^{m}\left[T_e - \langle T\rangle\right]^2\right]^{1/2}$ y esa fue menor que 3 % del promedio presentado $\langle T\rangle$.

[1] Unidades: $1B = 1\,byte = 8$ bits, $1\,\text{KiB} = 2^{10}\,bytes$, $1\,\text{MiB} = 2^{20}\,bytes$, $\text{GiB} = 2^{30}\,bytes$.

[2] Los tamaños de RAM y caché son resultados del comando "`sudo lshw -class memory`".

Cuadro 1. Tiempo esperado para resolver el sistema $U\vec{x} = \vec{b}$ con $n = 2000$

	algoritmo 2.1	algoritmo 2.2	algoritmo 2.3	algoritmo 2.4
Python	$684.940\,ms$	$3.767\,ms$	$716.216\,ms$	$12.912\,ms$
Octave	$9\,738.600\,ms$	$35.562\,ms$	$9\,895.400\,ms$	$30.817\,ms$
MatLab	$9.607\,ms$	$10.375\,ms$	$4.499\,ms$	$6.367\,ms$
Julia	$6.669\,ms$	$8.907\,ms$	$2.177\,ms$	$8.239\,ms$

Primero, los experimentos en el cuadro 1 muestran que fijar un algoritmo y comparar sus tiempos en 4 lenguajes no es una forma justa de comparar velocidades de lenguajes. Puesto que un lenguaje pierde con un algoritmo, pero gana con otro. Por ejemplo, Python gana con el algoritmo 2.2, Julia con algoritmo 2.3 y MatLab con algoritmo 2.4.

Segundo, para cada lenguaje existen dos algoritmos (mucho) más rápidos que los otros. Abajo, en la sección 3.2, presentamos experimentos que justifican que solo uno de estos dos tiene un comportamiento del tiempo que refleja la complejidad matemática (2), es decir, esperamos que el tiempo se comporte como $T(n) \approx t_0\, n^2$ donde $t_0 > 0$ es constante.

3.2 Tiempo *vs* dimensión

Para ahorrar energía solo se comparan los dos mejores algoritmos por lenguaje. Los experimentos en esta sección se hicieron igual como descrito en las secciones 3 y 3.1, pero duplicando la dimensión n. Es decir, por teoría esperamos que el tiempo crezca por el factor $2^2 = 4$, ya que $T(2n) \approx 4T(n)$. Abajo mostraremos para cada algoritmo seleccionado su tiempo esperado y su desviación estándar.

3.2.1. Python

En Python los algoritmos vectorizados 2.2 y 2.4 fueron los mejores del cuadro 1. El cuadro 2 muestra sus tiempos y se ve que el algoritmo 2.2 gana. Más aún, se puede ver que el tiempo del algoritmo 2.2, digamos $T_{2.2}(n)$, crece más lento que el esperado por el factor 4, excepto para el cambio de 8000 a 16000. Sin embargo, también notamos que $T_{2.2}(16000) = 136\,ms < 4^2 \cdot 9.29\,ms = 4^2 \cdot T_{2.2}(4000)$, lo cual respeta la complejidad teórica (2) y muestra que $T_{2.2}(8000)$ es atípicamente corto.

Por otro lado, el tiempo que tarda el algoritmo 2.4 crece (de fila a fila) por un factor mayor que 4. Más adelante, en la sección 4, explicamos las razones. Por el momento podemos concluir que el promedio del tiempo por operación t_0 no es constante y depende de algo más.

Cuadro 2. Crecimientos de tiempos esperados en Python 3.9.18

n	algoritmo 2.2	algoritmo 2.4
2 000	$3.77\,ms \pm 11\,\mu s$	$12.91\,ms \pm 70\,\mu s$
4 000	$9.29\,ms \pm 66\,\mu s$	$58.82\,ms \pm 550\,\mu s$
8 000	$25.79\,ms \pm 569\,\mu s$	$275.27\,ms \pm 1.93\,ms$
16 000	$136.01\,ms \pm 3.83\,ms$	$1\,525.00\,ms \pm 4.58\,ms$

3.2.2. Octave

En Octave los algoritmos vectorizados 2.2 y 2.4 fueron los mejores del cuadro 1. En los tiempos del cuadro 3 se ve que el algoritmo 2.4 gana. Si duplicamos la dimensión n, entonces observamos que $T(2n) \leq 4T(n)$ para cada fila, lo cual respeta la complejidad matemática (2).

El algoritmo 2.2 cumple lo mismo en las primeras 3 filas, sin embargo siempre es más lento y su último salto del tiempo tiene un cociente de $3424/382 = 8.9634 \gg 4$ que es mucho más grande que 4. Ese factor muestra un crecimiento incompatible con la complejidad teórica (2).

Cuadro 3. Crecimientos de tiempos esperados en Octave 9.2.0.

n	algoritmo 2.2	algoritmo 2.4
2 000	$35.56\,ms \pm 0.51\,ms$	$30.82\,ms \pm 0.25\,ms$
4 000	$106.14\,ms \pm 0.84\,ms$	$67.52\,ms \pm 0.51\,ms$
8 000	$382.00\,ms \pm 5.70\,ms$	$177.68\,ms \pm 5.30\,ms$
16 000	$3\,424.40\,ms \pm 133.85\,ms$	$584.76\,ms \pm 8.08\,ms$

3.2.3. MatLab

En MatLab los algoritmos 2.3 y 2.4 fueron los mejores del cuadro 1. Ambos ocupan las entradas de U por columna y su crecimiento del tiempo computacional refleja la complejidad matemática (2).

Cuadro 4. Crecimientos de tiempos esperados en MatLab R2024a

n	algoritmo 2.3	algoritmo 2.4
2 000	$4.48\,ms \pm 48\,\mu s$	$6.25\,ms \pm 78\,\mu s$
4 000	$17.78\,ms \pm 151\,\mu s$	$19.79\,ms \pm 253\,\mu s$
8 000	$70.41\,ms \pm 934\,\mu s$	$68.45\,ms \pm 483\,\mu s$
16 000	$282.38\,ms \pm 2.20\,ms$	$264.74\,ms \pm 3.01\,ms$

3.2.4. Julia

En Julia los algoritmos 2.1 y 2.3 fueron los mejores del cuadro 1. Ambos usan dos ciclos. Los tiempos en el cuadro 5 fueron obtenidos con la funcionalidad `btime` de Julia. Se ve que el algoritmo 2.3 gana y si duplicamos la dimensión n, entonces su tiempo crece aproximadamente por el factor $4 = 2^2$ lo cual coincide con la complejidad matemática (2). Por otro lado, los tiempos del algoritmo 2.1 crecen de fila en fila por factores $40/7 > 5 > 4, 215/40 > 5 > 4$ y $1298/215 > 6 > 4$, es decir, su complejidad computacional es más grande que la matemática.

Cuadro 5. Crecimientos de tiempos esperados en Julia 1.10.4.

n	algoritmo 2.1	algoritmo 2.3
2 000	$6.669\,ms$	$2.177\,ms$
4 000	$39.671\,ms$	$8.604\,ms$
8 000	$215.876\,ms$	$33.659\,ms$
16 000	$1\,268.000\,ms$	$134.464\,ms$

3.3 Ahorros de tiempo, energía y el ambiente

Para Julia, el cuadro 5 muestra que el algoritmo 2.3 ahorra, por sistema, entre 59 % y 87 % de tiempo comparado con el algoritmo 2.1. Los otros cuadros permiten hacer cálculos similares. Tales ahorros (pequeños o no) se acumulan. Por ejemplo, existen métodos iterativos que encuentran la solución de un problema resolviendo muchos sistemas cuadrados de ecuaciones lineales. Por ejemplo, el método Simplex que resuelve problemas económicos, véase [4, cap. 3], o el método de potencia inversa que aproxima eigenvectores, véase [9, cap. 12].

Entonces, ser capaz de diseñar varios algoritmos e identificar el algoritmo más eficiente (y posteriormente usarlo), sirve para reducir la cantidad de energía consumida (se mide en kWh), ya que, un algoritmo eficiente da el resultado en menos tiempo y calienta menos la computadora, lo cual reduce la ventilación requerida y permite mantener la computadora más tiempo en el «modo de ahorro de energía». Alternativamente, se pueden resolver más instancias del mismo tipo de problema (en un lapso de tiempo) con menos computadoras. Cada kWh menos importa, ya que tiene un impacto ambiental. Por ejemplo, en México (2022) cada kWh consumido causó 421g de emisiones de carbono[3].

[3] www.cencepower.com/calculators/kwh-to-co2-calculator

3.4 Comparación justa entre los lenguajes

El cuadro 6 combina las mejores columnas de los cuadros 2, 3, 4 y 5, es decir, se muestran los tiempos del mejor algoritmo (aquí presentado) en cada lenguaje. Se puede ver que los tiempos de resolver el problema matemático 2.2 en los distintos lenguajes no son muy distintos si usamos el algoritmo «adecuado» en cada lenguaje.

Cuadro 6. El mejor algoritmo por lenguaje:

n	Python algoritmo 2.2	Octave algoritmo 2.4	MatLab algoritmo 2.4	Julia algoritmo 2.3
2 000	$3.77\,ms$	$30.82\,ms$	$6.25\,ms$	$2.177\,ms$
4 000	$9.29\,ms$	$67.52\,ms$	$19.79\,ms$	$8.604\,ms$
8 000	$25.79\,ms$	$177.68\,ms$	$68.45\,ms$	$33.659\,ms$
16 000	$136.01\,ms$	$584.76\,ms$	$264.74\,ms$	$134.464\,ms$

Más aún, los mejores algoritmos nos dejan adivinar (e indican) cómo el lenguaje almacena la matriz U en la memoria principal (RAM) de la computadora. Por ejemplo, en Octave, MatLab y Julia los algoritmos rápidos son orientados por columna y sugieren que U es almacenada por columna.

4. Sobre la complejidad computacional

Los cuadros 2 (Python), 3 (Octave) y 5 (Julia) muestran que las complejidades computacionales (cuando n crece) de dos algoritmos, que resuelven el mismo problema matemático, difieren en cada lenguaje. Más aún, el cuadro 1 muestra tiempos significativamente diferentes entre los algoritmos por lenguaje.

En esta sección justificamos el hecho de que el algoritmo 2.4 no respeta la complejidad matemática (2) y que el algoritmo 2.2 tiende a respetarla cuando programamos los algoritmos en Python. Las diferencias de los tiempos y complejidades de computación se pueden justificar después de entender una versión simplificada del transporte de datos entre CPU, Caché y RAM, y cómo se almacenan los datos en la RAM. Abajo simplificamos dicho transporte, separamos el trabajo de los 4 algoritmos en trabajo común e individual y terminamos descubriendo que el trabajo individual es el que causa las complejidades diferentes entre los algoritmos 2.4 y 2.2.

4.1 El transporte de datos y principios de localidad

Las ideas en esta sección se basan en [2]. Técnicamente, cada algoritmo consiste de instrucciones que piden usar los datos U y $\vec{b}$ que se encuentran en la RAM. La comunicación entre CPU – Caché – RAM está diseñada para respetar dos principios de localidad. Estos se basan en la hipótesis de que en cada instante un programa solo usa una fracción de los datos y las variables.

Por ejemplo, resolviendo nuestro sistema $U\vec{x} = \vec{b}$ podemos considerar que al despejar cada entrada $x_i = (b_i - \sum_{j=i+1}^{n} x_j U_{ij})/U_{ii}$ solo se requiere un renglón de U, la entrada de b_i, la variable x_i y las variables anteriores $x_{i+1}, \ldots, x_n$. Digamos que el despeje tarda un instante. En ese no se requiere otro renglón de U, ni otra entrada de $\vec{b}$, es decir, si U tiene 100 renglones, entonces en cada instante cada despeje requiere aproximadamente 1 % de los datos. En este ejemplo se reconocen los siguientes dos principios, bajo cuales los diseñadores de *hardware* construyeron la comunicación entre CPU – caché – RAM:

- *El principio de localidad en el tiempo.*
  Se espera que un programa que usa una variable la vuelva a usar de nuevo muy pronto, por ejemplo, las variables $x_{i+1}, \ldots, x_n$ se usan para calcular x_i.
- *El principio de localidad en el espacio.*
  Se espera que un programa que usa un dato, tiende a usar datos cercanos muy pronto, por ejemplo, entradas del renglón i de U.

Hoy en día, comúnmente, un CPU recibe/escribe datos e instrucciones en paquetes de 64 bits. La localidad en el tiempo y el acceso individual a variables motivan el acceso individual y rápido entre la CPU y la caché, es decir, se transporta en cada dato 64 bits individualmente, mientras la localidad en el espacio y la distancia a la RAM motivan el transporte agrupado (en bloques) entre la caché y la RAM. La distancia entre la RAM y la caché supera la de la CPU y la caché, por lo cual, el transporte entre la caché y la RAM es más lento, por un factor entre 10 y 100, véase [8]. Por lo anterior, la comunicación entre la caché y la RAM fue diseñada para siempre transportar (al menos) un vector de datos de, digamos, $C \cdot (64\,\text{bits})$, aún cuando la caché solo pide un dato. Para simplificar pensamos que $C = 2^p > 1$ es constante, para alguna $p \in \mathbb{N}$. La figura 4 fue inspirada por [3] y visualiza lo escrito anteriormente.

Suponemos que las entradas de $U, \vec{b}$ y $\vec{x}$ son *números de precisión doble*, es decir, cada entrada ocupa 64 bits. Una introducción a esos números se encuentra, por ejemplo en [5, p. 23]. Dado que cada entrada

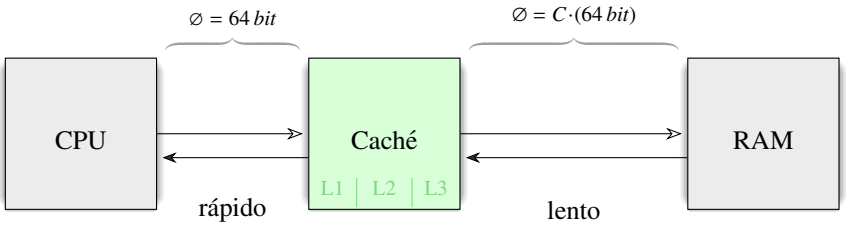


Figura 4. Velocidades simplificadas entre CPU – Caché y RAM.

de U ocupa 64 bits, el pedido de una entrada de U en la RAM, causa el transporte de un *bloque*, *i.e.*, de C entradas de U a la caché.

4.2 Trabajo común e individual

Los 4 algoritmos (en la sección 2.2) tienen en común la instrucción de copiar el lado derecho $\vec{x} \leftarrow \vec{b}$ y las n divisiones $x_i \leftarrow x_i/U_{ii}$. Lo anterior lo consideramos *trabajo común*. Concluimos que la complejidad computacional se ve afectada por el *trabajo individual*, es decir, por cómo cada algoritmo lee las entradas de U o por las veces que asigna $x_i \leftarrow (\dots)$.

Para simplificar nuestras conclusiones sobre un lenguaje, los siguientes puntos determinan el trabajo individual por algoritmo y el cuadro 7 resume los resultados.

- En el algoritmo 2.1 la asignación en la línea 4 es efectuada para $i \in \{1, \dots, n-1\}$ y $j \in \{i+1, \dots, n\}$, es decir, una entrada x_i se cambia $\sum_{j=i+1}^{n} 1 = (n-i)$ veces y en total las entradas de $\vec{x}$ se cambian y asignan $\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$ veces. Eso queda resumido en el 1^{er} renglón del cuadro 7.
- En el algoritmo 2.2 la asignación en la línea 4 es efectuada para $i \in \{1, \dots, n-1\}$, es decir, cada entrada x_i se cambia solo una vez y esto causa $n-1$ cambios y asignaciones, véase el cuadro 7.
- En el algoritmo 2.3 la asignación en la línea 5 es efectuada para $j \in \{2, \dots, n\}$ e $i \in \{1, \dots, j-1\}$, es decir, para cada j se cambian $j-1$ entradas de $\vec{x}$. En suma las entradas de $\vec{x}$ se cambian y asignan $\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{1}{2}n(n-1)$ veces, véase el cuadro 7.
- En el algoritmo 2.4 la asignación en la línea 5 es efectuada para $j \in \{2, \dots, n\}$, es decir, tenemos $n-1$ asignaciones, pero para cada j se cambian y escriben $j-1$ entradas de $\vec{x}$. En suma las entradas de $\vec{x}$ se sobre-escriben $\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{1}{2}n(n-1)$ veces, véase el cuadro 7.

Cuadro 7. Trabajo individual: asignaciones "$\leftarrow$" y variables cambiados.

	$\lvert\{\leftarrow\}\rvert$	$\lvert\{x_i \text{ cambiado}\}\rvert$	U leída por
alg. 2.1	$\frac{1}{2}n(n-1)$	$\frac{1}{2}n(n-1)$	renglón
alg. 2.2	$n-1$	$n-1$	renglón (vector)
alg. 2.3	$\frac{1}{2}n(n-1)$	$\frac{1}{2}n(n-1)$	columna
alg. 2.4	$n-1$	$\frac{1}{2}n(n-1)$	columna (vector)

En el cuadro 7 se ve que los algoritmos 2.2 y 2.4 agrupan asignaciones "$\leftarrow$" y leen entradas de U como vectores. Se ve que el algoritmo 2.4 cambia más veces las entradas de $\vec{x}$, sin embargo, ambos algoritmos se aceleran bastante en Python, véase el cuadro 1. Por otro lado, en Python los algoritmos 2.1 y 2.3 son lentos, ya que Python no agrupa automáticamente ni el renglón leído ni las asignaciones. A continuación veremos que es más sencillo comprobar que el algoritmo 2.4 tiene complejidad computacional más grande que la matemática (2), que mostrar la mejor complejidad del algoritmo 2.2.

4.3 Sobre el algoritmo 2.4

Comparando el cuadro 2 con el cuadro 7 observamos que leer entradas de U por renglones da el algoritmo más eficiente. Sabiendo que el transporte entre RAM y caché es por bloques $C \cdot (64\,\text{bits})$, es decir, se transportan C entradas juntas de U, podemos concluir que Python almacena U en la RAM renglón por renglón. Para visualizar el orden de acceso del algoritmo 2.4 en la RAM, retomamos una matriz $U \in \mathbb{R}^{5\times 5}$ y el orden de acceso en la figura 3(b). Dado que U se acomoda por renglones en la memoria, obtenemos el siguiente orden de acceso:

$$\left|\overset{9}{u_{11}}\ \overset{8}{u_{12}}\ \overset{6}{u_{13}}\ \overset{4}{u_{14}}\ \overset{2}{u_{15}}\right|0\ \overset{7}{u_{22}}\ \overset{6}{u_{23}}\ \overset{4}{u_{24}}\ \overset{2}{u_{25}}\left|0\ 0\ \overset{5}{u_{33}}\ \overset{4}{u_{34}}\ \overset{2}{u_{35}}\right|0\ 0\ 0\ \overset{3}{u_{44}}\ \overset{2}{u_{45}}\left|0\ 0\ 0\ 0\ \overset{1}{u_{55}}\right|$$

Este ejemplo muestra que si $C \leq 5$, entonces el algoritmo 2.4 (línea 5) tiene que leer una parte de 4 renglones para hacer el trabajo individual que requiere la 5^a columna $(u_{15}, u_{25}, u_{35}, u_{45})$ y para eso la caché pide 4 bloques de la RAM, es decir, $4C$ entradas. Similarmente, para obtener (u_{14}, u_{24}, u_{34}) la caché necesita pedir 3 bloques, que resultan en $3C$ entradas, etc.

Generalizamos esta intuición. El algoritmo (línea 5) necesita $(j-1)$ entradas de cada columna $j \in \{2, \ldots, n\}$, y bajo la hipótesis $C \leq n$ esas entradas en la RAM tienen una distancia $n+1$. Por lo tanto, si $C \leq n$, entonces se transportan $j-1$ bloques, es decir, $C(j-1)$ entradas para que la CPU obtenga las entradas de la columna j y pueda realizar

los cálculos instruidos por el algoritmo. Sumando sobre las columnas $j \in \{2, \dots, n\}$, la línea 5 del algoritmo 2.4 causa un transporte de

$$\sum_{j=2}^{n} C(j-1) = C \sum_{j=1}^{n-1} j = C\frac{n(n-1)}{2} \tag{3}$$

entradas de la RAM a la caché, lo cual aumenta el trabajo individual. Concluimos que el trabajo individual (3) del algoritmo 2.4 es por lo menos C veces el costo matemático (2). Eso es un factor constante, pero cuando n cruza el intervalo $[C, 2C]$, tenemos $C \approx n-1$ y la igualdad estima un costo $C \approx n-1$ veces mayor.

4.4 Sobre el algoritmo 2.2

Para visualizar el orden de acceso del algoritmo 2.2 en la RAM, retomamos una matriz $U \in \mathbb{R}^{5\times 5}$ y el orden de acceso en la figura 2(b). Dado que Python acomoda U por renglones en la memoria, obtenemos el siguiente orden de acceso:

$$\left| \overset{9}{u_{11}} \ \overset{8}{u_{12}} \ \overset{8}{u_{13}} \ \overset{8}{u_{14}} \ \overset{8}{u_{15}} \right| 0 \ \overset{7}{u_{22}} \ \overset{6}{u_{23}} \ \overset{6}{u_{24}} \ \overset{6}{u_{25}} \left| 0 \ 0 \ \overset{5}{u_{33}} \ \overset{4}{u_{34}} \ \overset{4}{u_{35}} \right| 0 \ 0 \ 0 \ \overset{3}{u_{44}} \ \overset{2}{u_{45}} \left| 0 \ 0 \ 0 \ 0 \ \overset{1}{u_{55}} \right|$$

Podemos concluir que para calcular x_i, las entradas necesarias de U están juntas (principio de localidad en el espacio) en la memoria. A continuación calculamos cuántas entradas se transportan de la RAM a la caché.

La línea 4 del algoritmo lee para cada x_i por lo menos $(n-i)$ entradas del renglón i, es decir, las entradas de $(U_{i,i+1}, \dots, U_{i,n})$. De hecho se transportan tantos bloques de $C\cdot(64\,\text{bits})$ de RAM a caché como sean necesarios para entregar las $n-i$ entradas a la CPU. La descomposición módulo $C \geqslant 1$ de Álgebra permite encontrar el número de bloques para enviar $n-i$ entradas. Por ejemplo, para $n-1$ sabemos $n-1 = \ell C + r$ para dos naturales $\ell, r \in \mathbb{N}$ con $0 \leqslant r < C$. Entonces, si $r = 0$, se envían ℓ bloques y si $1 \leqslant r < C$ se envían $\ell + 1$ bloques. Para evitar esta distinción de casos existe la función *techo* $\lceil\cdot\rceil : \mathbb{R} \to \mathbb{N}$ que para $x \in \mathbb{R}$ da el primer natural no menor que x, *i.e.*, $\lceil x \rceil = \min\{p \in \mathbb{N} \colon x \leqslant p\}$. Entonces, el valor $\left\lceil \frac{n-i}{C} \right\rceil$ nos dice cuantos bloques se requieren para transportar $n-i$ entradas. Por ejemplo, si $2C < n-i \leqslant 3C$, entonces se transportan $\left\lceil \frac{n-i}{C} \right\rceil = 3$ bloques, que resultan ser $3C$ entradas.

Lo anterior es válido para $i \in \{1, \dots, n-1\}$. Entonces, en total el número de bloques transportados por el trabajo individual es:

$$\sum_{i=1}^{n-1} \left\lceil \frac{n-i}{C} \right\rceil$$

y, dado que cada bloque transporta C entradas (de U), el número de entradas transportadas por el trabajo individual es:

$$C \sum_{i=1}^{n-1} \left\lceil \frac{n-i}{C} \right\rceil = \sum_{i=1}^{n-1} C \left\lceil \frac{n-i}{C} \right\rceil . \tag{4}$$

Para evitar la evaluación exacta de esta suma y simplificar nuestros argumentos, usamos la cota inferior $x \leqslant \lceil x \rceil$ y que la función techo respeta a la cota superior $\lceil x \rceil - 1 < x \iff \lceil x \rceil < x + 1$ que se debe a la propiedad del mínimo y la distancia entre los naturales. Estas propiedades nos ayudan a encerrar cada sumando como sigue: para cada $i \in \{1, \ldots, n-1\}$ se cumple

$$n - i = C \, \frac{n-i}{C} \leqslant C \left\lceil \frac{n-i}{C} \right\rceil < C \left(\frac{n-i}{C} + 1 \right) = n - i + C \, .$$

Usando estas cotas y la identidad $\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, el número de entradas transportadas por el trabajo individual (4) queda estimado como sigue:

$$\begin{aligned} \frac{n(n-1)}{2} = \sum_{i=1}^{n-1}(n-i) &\leqslant \sum_{i=1}^{n-1} C \left\lceil \frac{n-i}{C} \right\rceil \\ &< \sum_{i=1}^{n-1}(n-i+C) = \frac{n(n-1)}{2} + nC \, , \end{aligned} \tag{5}$$

es decir, para realizar el trabajo individual del algoritmo 2.2 se transportan entre $\frac{1}{2}n(n-1)$ y $\frac{1}{2}n(n-1) + nC$ entradas de RAM a caché.

Para $C \geqslant 2$ y $n > 4$ se puede ver que la cota superior es menor que el número de entradas transportadas por el algoritmo 2.4, véase (3). Con el fin de cuantificar este beneficio comparamos con el costo matemático.

Para esto, dividimos las cotas (5) entre el costo matemático dado en ecuación (2), el cual representa el número de los productos $\{x_j U_{ij}\}$ y, por lo tanto, también el mínimo de entradas transportadas para hacer el trabajo individual. El resultado es:

$$1 \leqslant \frac{2}{n(n-1)} \sum_{i=1}^{n-1} C \left\lceil \frac{n-i}{C} \right\rceil < 1 + \frac{2C}{n} \, . \tag{6}$$

La cota inferior dice algo evidente: siempre se transportan al menos tantas entradas como requeridas por el costo matemático. Suponiendo que **la RAM alcanza** para almacenar $U, \vec{x}$ y $\vec{b}$, la cota superior tiene las siguientes dos interpretaciones:

- Para $n \geqslant C$ se tiene $1 + 2C/n \leq 3$, es decir, el costo del transporte individual del algoritmo 2.2 es menor que 3 veces el costo

matemático. Recuerde que el transporte individual causado por el algoritmo 2.4 era C-veces mayor, véase (3).

- Además, la cota superior (6) da el siguiente límite:

$$\lim_{n\to\infty} \frac{2}{n(n-1)} \sum_{i=1}^{n-1} C \left\lceil \frac{n-i}{C} \right\rceil \leqslant \lim_{n\to\infty} \left[1 + \frac{2C}{n} \right] = 1 \,, \tag{7}$$

es decir, para $n \geqslant C$ y creciendo, el trabajo individual cada vez se parece más al trabajo matemático (2).

Lo anterior confirma el crecimiento de los tiempos del algoritmo 2.2 en el cuadro 2 ya que en los experimentos la RAM alcanzó.

5. Resumen y conclusión

Terminamos este artículo resumiendo lo aprendido y comparándolo con respuestas de una inteligencia artificial.

Las secciones 2, 3 y 4 de este artículo justifican los siguientes puntos:

- El proceso de resolver un problema matemático se puede convertir en varios algoritmos.
- Fijando el lenguaje, la complejidad de los algoritmos varía.
- Fijando el algoritmo, su complejidad varía con el lenguaje.

En conclusión, hemos visto y observado que un lenguaje impone cómo almacena datos en la RAM y cómo corre las instrucciones del algoritmo. Básicamente, se puede decir que un algoritmo es eficiente si es compatible con lo que impone el lenguaje. Además, usar un algoritmo eficiente reduce el consumo de energía, como argumentado en la sección 3.3.

Si una persona diseña e implementa algoritmos, entonces, saber de matemáticas sirve para reconocer y definir algoritmos equivalentes. Por otro lado, conocer el lenguaje elegido y cómo este responde a la comunicación interna de la computadora, permite reconocer un algoritmo eficiente, o para aprovechar el conocimiento en otros problemas y nuevos algoritmos.

Comparamos con ChatGPT.

Hasta este punto, este artículo fue escrito y todos los algoritmos fueron implementados sin ayuda de la inteligencia artificial *(I.A.)*.

Debido a la época en la que vivimos, el 26 de mayo de 2024, el autor le dio la siguiente instrucción a la I.A. llamada *ChatGPT*:

> *I study mathematics. I have a linear system with an invertible upper triangular matriz U and a right-hand side b. Please write a fast algorithm in Python that solves this system.*

La misma instrucción también se hizo para cada uno de los lenguajes Julia, Octave y MatLab. Las respuestas para cada lenguaje fueron algoritmos matemáticamente correctos, pero todos son más lentos que los presentados en el cuadro 6. Esto lo confirma el cuadro 8 que presenta la comparación entre los tiempos de nuestros algoritmos $\langle T \rangle_{nuestro}$, del cuadro 6, y promedios de tiempos que se tardaron los algoritmos propuestos por la I.A. $\langle T \rangle_{I.A.}$ para resolver sistemas como los descritos en la sección 3, es decir, en la misma computadora y bajo las mismas circunstancias. Además, los algoritmos propuestos por la ChatGPT eran similares al algoritmo 2.1 (para Python) y al algoritmo 2.2 (para Octave, MatLab y Julia). Pero estos algoritmos los hemos descartados como óptimos.

Cuadro 8. Tiempos nuestros y tiempos de los algoritmos propuestos por la I.A.:

$n = 4000$	Python	Octave	MatLab	Julia
$\langle T \rangle_{nuestro}$	$9.29\,ms$	$67.52\,ms$	$19.79\,ms$	$8.604\,ms$
$\langle T \rangle_{ChatGPT}$	$1964.22\,ms$	$97.84\,ms$	$59.52\,ms$	$24.759\,ms$

Estos resultados sugieren que la respuesta de una ChatGPT no es del todo satisfactoria. Claramente, en la instrucción arriba se podría mencionar que intentamos resolver sistemas con muchas incógnitas, lo cual hace que ChatGPT da un algoritmo en Python tan eficiente como el nuestro, pero **solo** en Python. En los otros lenguajes la respuesta sigue siendo una versión del algoritmo 2.2, que fue descartada como eficiente.

Por lo anterior concluimos que se debe continuar con la formación sólida, especializada y crítica de matemáticas aplicadas y áreas relacionadas. En este documento, las matemáticas fueron particularmente útiles para encontrar las equivalencias algebraicas que nos permitieron encontrar diferentes algoritmos equivalentes para resolver sistemas lineales. Y los conocimientos del área de computación sobre la comunicación interna de la computadora, ver la sección 4, combinados con el Álgebra de números enteros, desigualdades y el Cálculo (límites) nos permitieron demostrar los resultados (6) y (7) que justifican un algoritmo eficiente y la identidad (3) que descarta un algoritmo no eficiente. Por lo tanto, este artículo representa el siguiente mensaje para los profesores y estudiantes de las áreas STEAM:

> *Se pueden beneficiar de herramientas nuevas como* ChatGPT*, pero se requiere una formación sólida, especializada, y crítica en las áreas STEAM para poder analizar las respuestas de dichas herramientas y obtener el mayor beneficio.*

Agradecimientos

El autor agradece enormemente la ayuda de la profesora Ana Lidia Franzoni del Departamento de Computación, ITAM, quien ayudó a formular la instrucción dada a la I. A. Más aún, ella mostró interés en la pregunta y confirmó las respuestas con la I.A. llamada *ChatGPT 4*.

Finalmente, el autor agradece los consejos de los revisores los cuales mejoraron y simplificaron la presentación.

Bibliografía

[1] U. M. Ascher y C. Greif, *A first course on numerical methods*, SIAM, 2011.
[2] D. August, «Memory Caching: Computer Architecture and Organization», Lecture notes as pdf www.cs.princeton.edu/courses/archive/fall15/cos375/lectures/16-Cache-2x2.pdf, 2015, visitado Abril 2024, 1–49.
[3] Cybercomputing, «A Cybercomputing Page», www.cybercomputing.co.uk/Languages/Hardware/cacheCPU.html, 2024, visitado Agosto 2023.
[4] M. C. Ferris, O. L. Mangasarian, y S. J. Wright, *Linear Programming with MatLab*, SIAM, 2007.
[5] I. Gladwell, J. G. Nagy, y W. E. Ferguson Jr., *Introduction to Scientific Computing*, 2011.
[6] J. F. Grcar, «Mathematicians of Gaussian Elimination», *Notices of the AMS* vol. 58 (2011) 782–792.
[7] N. J. Higham, «How Accurate is Gaussian Elimination?», *Numerical Analysis 1989, Proceedings of the 13th Dundee Conference* vol. 228 (1990) 137–154.
[8] G. Kranz, J. Toigo, y S. Peterson, «Cache vs. RAM: Differences between the two memory types», www.techtarget.com/searchstorage/answer/What-is-the-difference-between-cache-memory-and-RAM-cache, 08 2021, visitado Abril 2024.
[9] T. Sauer, *Numerical Analysis*, Pearson, 2012, Second Edition.