doi.org/10.51891/rease.v11i7.19881

APPLICATION OF MATRIX METHODS IN CHEMICAL ENGINEERING: FROM THEORY TO PRACTICE WITH COMPUTATIONAL EXAMPLES

APLICAÇÃO DE MÉTODOS MATRICIAIS NA ENGENHARIA QUÍMICA: DA TEORIA À PRÁTICA COM EXEMPLOS COMPUTACIONAIS

APLICACIÓN DE MÉTODOS MATRICIALES EN INGENIERÍA QUÍMICA: DE LA TEORÍA A LA PRÁCTICA CON EJEMPLOS COMPUTACIONALES

Elilton Rodrigues Edwards[1]
Marley Rebouças Ferreira[2]
Henrique Silva Novais[3]

ABSTRACT: This article aimed to demonstrate the practical application of matrix methods in Chemical Engineering through the modeling and resolution of complex problems using computational tools. The study was developed based on the implementation of algebraic algorithms in Python, using the NumPy and SciPy libraries in the Jupyter Notebook environment. The methodology includes examples of solving linear systems of different orders, handling sparse and triangular matrices, in addition to the matrix representation of molecular interactions. The results showed that the computational application of these methods facilitates the simulation and analysis of real-world problems in chemical engineering, such as mass balances in separation processes and interconnected tank systems. It is concluded that matrix methods, when integrated with computational programming, offer a precise and efficient approach to scientific computing, expanding the possibilities for analysis and innovation in contemporary engineering.

Keywords: Matrix Methods. Scientific Computing. Chemical Engineering.

RESUMO: Este artigo teve como objetivo demonstrar a aplicação prática dos métodos matriciais em Engenharia Química por meio da modelagem e resolução de problemas complexos com ferramentas computacionais. O estudo foi desenvolvido a partir da implementação de algoritmos algébricos em Python, utilizando as bibliotecas NumPy e SciPy no ambiente Jupyter Notebook. A metodologia inclui exemplos de resolução de sistemas linear de diferentes ordens, tratamento de matrizes esparsas e triangulares, além da representação matricial de interações moleculares. Os resultados demonstraram que a aplicação computacional desses métodos facilita a simulação e análise de problemas reais da engenharia química, como balanços de massa em processos de separação e sistemas de tanques interconectados. Conclui-se que os métodos matriciais, quando integrados à programação computacional, oferecem uma abordagem precisa e eficiente à computação científica, ampliando as possibilidades de análise e inovação na engenharia contemporânea.

Palavras-chave: Métodos Matriciais. Computação Científica. Engenharia Química.

[1]Professor Universitário/Orientador Universidade Estadual de Santa Cruz – UESC.
[2]Discente Universidade Estadual de Santa Cruz – UESC.
[3]Discente Universidade Estadual de Santa Cruz – UESC.

RESUMEN: Este artículo tuvo como objetivo demostrar la aplicación práctica de los métodos matriciales en la Ingeniería Química mediante la modelación y resolución de problemas complejos con herramientas computacionales. El estudio fue desarrollado a partir de la implementación de algoritmos algebraicos en Python, utilizando las bibliotecas NumPy y SciPy en el entorno Jupyter Notebook. La metodología incluye ejemplos de resolución de sistemas lineales de diferentes órdenes, tratamiento de matrices dispersas y triangulares, además de la representación matricial de interacciones moleculares. Los resultados demostraron que la aplicación computacional de estos métodos facilita la simulación y el análisis de problemas reales de la ingeniería química, como balances de masa en procesos de separación y sistemas de tanques interconectados. Se concluye que los métodos matriciales, cuando se integran con la programación computacional, ofrecen un enfoque preciso y eficiente para la computación científica, ampliando las posibilidades de análisis e innovación en la ingeniería contemporánea.

Palabras clave: Métodos Matriciales. Computación Científica. Ingeniería Química.

INTRODUÇÃO

In recent years, engineering has undergone a profound transformation with the advent of new technologies, which make solving complex problems more accessible and efficient (Zawadzki P and Żywicki K, 2016; Xu LD, et al., 2018). Computers have become indispensable tools in this evolution process, providing resources for the application of robust mathematical and computational methods. These methods have allowed scientists, researchers, and engineers to translate natural phenomena into mathematical language, enabling the simulation of physical and chemical processes through advanced algorithms. Furthermore, techniques such as the use of Artificial Neural Networks (ANNs) and Artificial Intelligence (AI) have become increasingly common in engineering, enabling simulation and prediction models with a high degree of precision and applicability.

In this context, the use of algebraics, an area of mathematics, enables a greater understanding of the foundations of scientific computing. These methods are widely used to solve and model complex problems, both in engineering disciplines and in areas such as physics, chemistry, and also in AI including ANNs. Algebraic methods are fundamental tools for solving a multitude of engineering problems as they enable us to link mathematics with computing.

ANNs make extensive use of matrix methods and algebraic operations. The structure and functioning of ANNs depend directly on matrix calculations to manipulate weights and input data at each layer of the network. The operations of matrix multiplication, summation, and activation functions are fundamental for training and running networks and have roots in algebraic methods (Parr T and Howard J, 2018; Sun R, et al., 2024; Liu Z, et al., 2025).

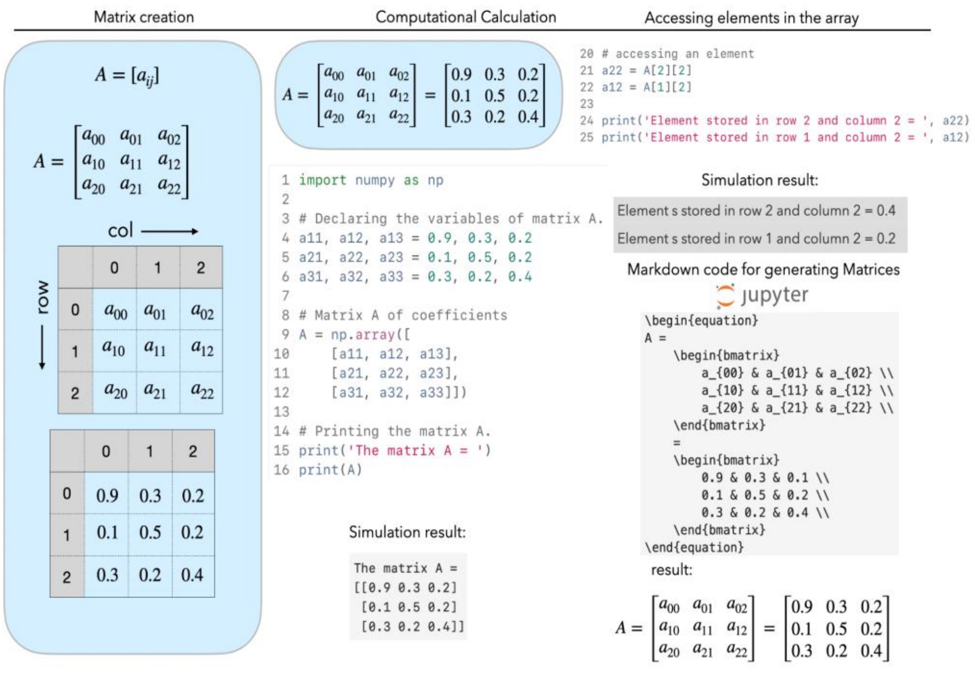
Algebraic Methods as the Basis of Scientific Computing

Algebraic methods, including matrix methods, are the basis of scientific computing and have essential applications in areas such as solving systems of linear equations, data analysis and development of predictive models. These methods provide tools to solve practical problems and enable the understanding and implementation of algorithms in various areas of engineering. In mathematical modeling, for example, they are used to accurately represent physical, chemical and biological systems, allowing numerical simulations and the development of realistic models. In Machine Learning, they are the basis for algorithms such as neural networks, Support Vector Machines (SVMs) and linear regression. Matrix notations, by organizing data logically, simplify the resolution of problems with multiple variables and facilitate the efficient execution of complex operations, thus being indispensable for scientific computing. These methods have evolved significantly with the advancement of high-performance computing techniques and the growing need to deal with large volumes of data in various application areas. For example, the use of sparse linear algebra has become indispensable to optimize resources in large-scale problems, where systems of equations are composed of matrices with few non-zero elements, significantly reducing the computational cost. Furthermore, new approaches, such as tensor methods, have been developed to model multidimensional systems in complex areas such as Big Data analysis and bioinformatics (Cichocki A, et al., 2016). These methods not only improve the efficiency of calculations but also increase the accuracy of scientific simulations and predictions. Combining these methods with adaptive algorithms and machine learning allows scientific computing to become increasingly versatile and accurate, paving the way for innovations in sectors such as health, environment and economics (Gomes C, et al., 2019).

Basic Matrix Notations and Element Identification

In a matrix, each element is indicated by a coordinate a_{ij} where i represents the row and j the column, allowing you to locate any value precisely, like a point on a graph. Figure 1 illustrates the use of coordinates to identify elements in a matrix. Numpy (Numeric Python) defaults to the first cell having zero coordinates, so rows and columns with zeros are taken into account when counting to identify the coordinates of a number in the cell. For example, the element in the second row and third column is accessed as $A[1][2]$ and has the value 0.2. Another element in the third row and third column is accessed as $A[2][2]$, and will have the value 0.4 (Golub GH and Van Loan CF, 2013). To create a 3×3 matrix in Jupyter Notebook, we can use the NumPy library, declaring numeric values in a list structure, where each line is separated by commas (Oliphant TE, 2006). Thus, each element of the matrix A is directly accessed by its articulations $a_{ij} = A[i][j]$, as shown in the figure. In Jupyter Notebook, Markdown cells allow text editing and the creation of mathematical formulas using LaTeX syntax, ideal for documentation and explanations. Code cells are used to execute *scripts*, allowing you to perform mathematical calculations, process data, and develop interactive programs.

Figure 1 – Notation for creating matrices and identifying each element in the cell.


Matrix creation
Computational Calculation
Accessing elements in the array
col
row
Simulation result:
Markdown code for generating Matrices
jupyter
result:

$$A = [a_{ij}]$$

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

	0	1	2
0	a_{00}	a_{01}	a_{02}
1	a_{10}	a_{11}	a_{12}
2	a_{20}	a_{21}	a_{22}

	0	1	2
0	0.9	0.3	0.2
1	0.1	0.5	0.2
2	0.3	0.2	0.4

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0.9 & 0.3 & 0.2 \\ 0.1 & 0.5 & 0.2 \\ 0.3 & 0.2 & 0.4 \end{bmatrix}$$

```python
import numpy as np

# Declaring the variables of matrix A.
a11, a12, a13 = 0.9, 0.3, 0.2
a21, a22, a23 = 0.1, 0.5, 0.2
a31, a32, a33 = 0.3, 0.2, 0.4

# Matrix A of coefficients
A = np.array([
    [a11, a12, a13],
    [a21, a22, a23],
    [a31, a32, a33]])

# Printing the matrix A.
print('The matrix A = ')
print(A)
```

Simulation result:

```
The matrix A =
[[0.9 0.3 0.2]
 [0.1 0.5 0.2]
 [0.3 0.2 0.4]]
```

```python
# accessing an element
a22 = A[2][2]
a12 = A[1][2]

print('Element stored in row 2 and column 2 = ', a22)
print('Element stored in row 1 and column 2 = ', a12)
```

Simulation result:

```
Element s stored in row 2 and column 2 = 0.4
Element s stored in row 1 and column 2 = 0.2
```

Markdown code for generating Matrices

```
\begin{equation}
A =
    \begin{bmatrix}
        a_{00} & a_{01} & a_{02} \\
        a_{10} & a_{11} & a_{12} \\
        a_{20} & a_{21} & a_{22} \\
    \end{bmatrix}
    =
    \begin{bmatrix}
        0.9 & 0.3 & 0.1 \\
        0.1 & 0.5 & 0.2 \\
        0.3 & 0.2 & 0.4 \\
    \end{bmatrix}
\end{equation}
```

result:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0.9 & 0.3 & 0.2 \\ 0.1 & 0.5 & 0.2 \\ 0.3 & 0.2 & 0.4 \end{bmatrix}$$

Source: EDWARDS ER, et al., 2025.

Matrix manipulation, such as addition, subtraction, multiplication, and division, plays a fundamental role in several technological areas. In artificial neural networks, for example, matrices are used to represent weights and biases, allowing efficient calculations during training and inference (Lei J et al., 2023; Kariri E et al., 2023).

In the field of biometrics, both facial and fingerprint scanning rely on matrix operations to process and compare images at high speed (Gururaj HL et al., 2024; Bayram S et al., 2014). Furthermore, in the area of digital security, matrix manipulation is essential for encryption algorithms and data analysis, ensuring the protection of sensitive information (Kumar T and Chauhan S, 2018; Paul AJ et al., 2011; Kumar P et al., 2022). These applications demonstrate how matrix manipulation is at the heart of advanced technologies that drive innovation in diverse areas.

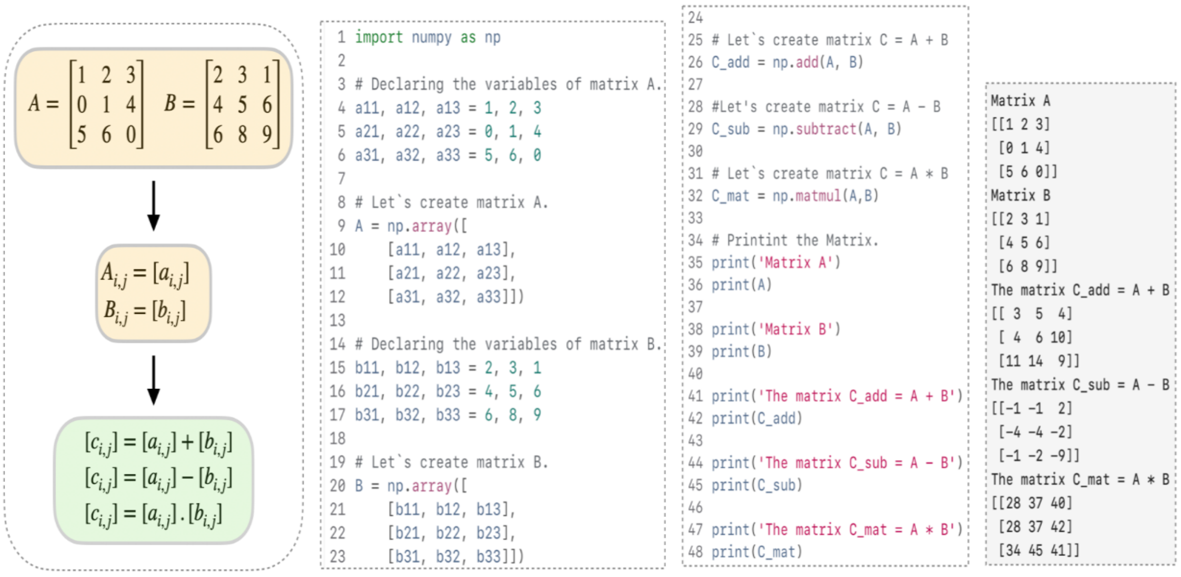
Algebraic Operations with Matrices

When adding or multiplying matrices, the computer performs sequential mathematical operations, row by row and column by column, according to the indexed positions. For example, when adding two matrices A and B of the same dimension, each element c_{ij} of the resulting matrix C is obtained by adding the corresponding elements a_{ij} and b_{ij}. This type of calculation is simplified by matrix notation, allowing the computer to access and operate on each individual value according to its specific position. Universal functions such as add, subtract and *matmul* are used to facilitate algebraic calculations.

Figure 2 shows the creation of matrices using the Numpy library and the algebraic manipulations of addition, multiplication and division. NumPy[4] (*Numerical Python*) is an essential package for scientific computing in Python, with fixed-size arrays and efficient data manipulation for advanced mathematical operations. The *add*, *subtract* and *matmul* functionalities are called universal functions (ufunc), which perform elementary operations efficiently on arrays. They are optimized to apply operations to all array elements, supporting both point-to-point operations and matrix calculations. The *add* and *subtract* functions allow you to add or subtract the corresponding values from cells a_{ij} and b_{ij}, saving algebraic calculation time, as long as the matrices have the same number of rows and columns. The *matmul* function performs the matrix product, multiplying the rows of a matrix A by the columns of another matrix B, as long as the number of columns of A is equal to the number of rows of B (Oliphant TE, 2006).

[4] https://numpy.org/

Figure 2 – Algebraic manipulation of matrices: addition, subtraction and multiplication.


$A = \begin{bmatrix}1&2&3\\0&1&4\\5&6&0\end{bmatrix}$ $B = \begin{bmatrix}2&3&1\\4&5&6\\6&8&9\end{bmatrix}$
$A_{i,j} = [a_{i,j}]$
$B_{i,j} = [b_{i,j}]$
$[c_{i,j}] = [a_{i,j}] + [b_{i,j}]$
$[c_{i,j}] = [a_{i,j}] - [b_{i,j}]$
$[c_{i,j}] = [a_{i,j}] . [b_{i,j}]$

```python
import numpy as np

# Declaring the variables of matrix A.
a11, a12, a13 = 1, 2, 3
a21, a22, a23 = 0, 1, 4
a31, a32, a33 = 5, 6, 0

# Let`s create matrix A.
A = np.array([
    [a11, a12, a13],
    [a21, a22, a23],
    [a31, a32, a33]])

# Declaring the variables of matrix B.
b11, b12, b13 = 2, 3, 1
b21, b22, b23 = 4, 5, 6
b31, b32, b33 = 6, 8, 9

# Let`s create matrix B.
B = np.array([
    [b11, b12, b13],
    [b21, b22, b23],
    [b31, b32, b33]])

# Let`s create matrix C = A + B
C_add = np.add(A, B)

#Let's create matrix C = A - B
C_sub = np.subtract(A, B)

# Let`s create matrix C = A * B
C_mat = np.matmul(A,B)

# Printint the Matrix.
print('Matrix A')
print(A)

print('Matrix B')
print(B)

print('The matrix C_add = A + B')
print(C_add)

print('The matrix C_sub = A - B')
print(C_sub)

print('The matrix C_mat = A * B')
print(C_mat)
```

```
Matrix A
[[1 2 3]
 [0 1 4]
 [5 6 0]]
Matrix B
[[2 3 1]
 [4 5 6]
 [6 8 9]]
The matrix C_add = A + B
[[ 3  5  4]
 [ 4  6 10]
 [11 14  9]]
The matrix C_sub = A - B
[[-1 -1  2]
 [-4 -4 -2]
 [-1 -2 -9]]
The matrix C_mat = A * B
[[28 37 40]
 [28 37 42]
 [34 45 41]]
```

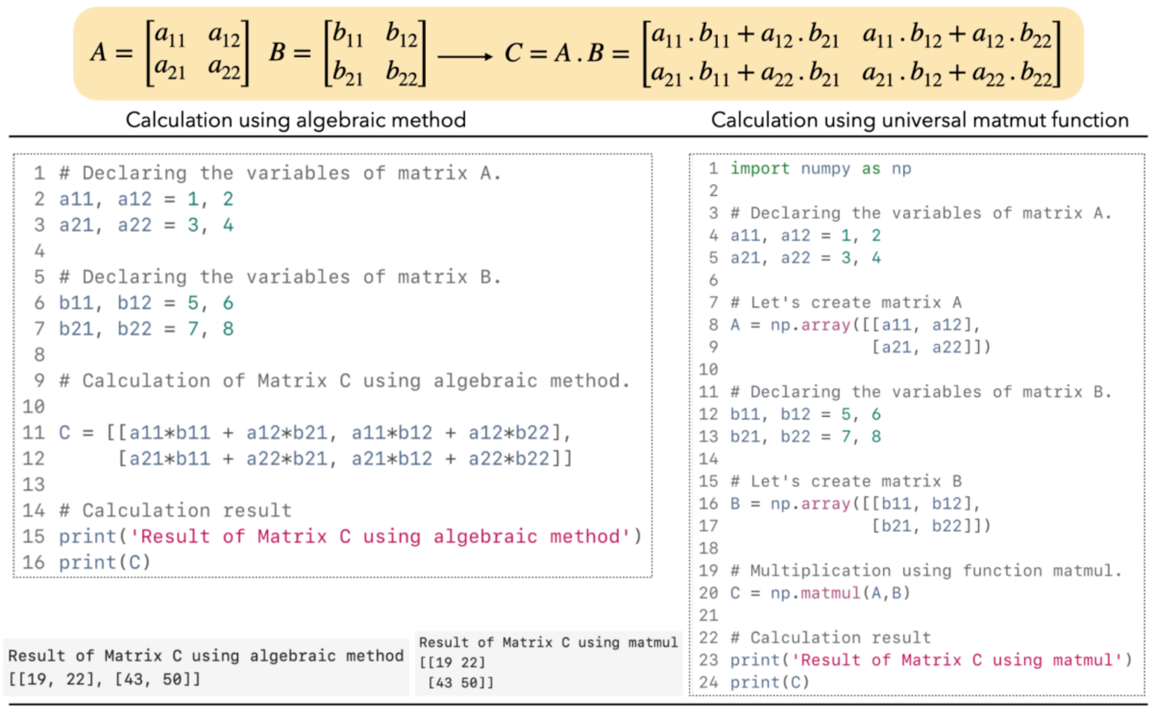
Source: EDWARDS ER, et al., 2025.

Data Storage and Access in Internal Tables

On the computer, matrices are stored in structures such as arrays or lists, which organize values by their i, j positions. In Python, for example, libraries like NumPy represent matrices as arrays, allowing direct access to elements through indexes. To access cell a_{12}, the program retrieves the second element from the first line (Oliphant TE, 2006).

When multiplying matrices, the program locates the necessary elements and performs the dot product between rows and columns, generating the new matrix. For example, consider the matrices A and B, both $2x2$ (Oliphant TE, 2006). In figure 3, the first script displays numerical calculations using algebraic methods, while the second uses Numpy's *matmul* function to perform multiplication in a single line of code.

Figure 3 – Matrix multiplication using algebraic method and Numpy's universal matmul function.



$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \longrightarrow C = A \,.\, B = \begin{bmatrix} a_{11} \,.\, b_{11} + a_{12} \,.\, b_{21} & a_{11} \,.\, b_{12} + a_{12} \,.\, b_{22} \\ a_{21} \,.\, b_{11} + a_{22} \,.\, b_{21} & a_{21} \,.\, b_{12} + a_{22} \,.\, b_{22} \end{bmatrix}$$

Calculation using algebraic method
Calculation using universal matmut function

```
# Declaring the variables of matrix A.
a11, a12 = 1, 2
a21, a22 = 3, 4

# Declaring the variables of matrix B.
b11, b12 = 5, 6
b21, b22 = 7, 8

# Calculation of Matrix C using algebraic method.

C = [[a11*b11 + a12*b21, a11*b12 + a12*b22],
     [a21*b11 + a22*b21, a21*b12 + a22*b22]]

# Calculation result
print('Result of Matrix C using algebraic method')
print(C)
```

```
Result of Matrix C using algebraic method
[[19, 22], [43, 50]]
```

```
Result of Matrix C using matmul
[[19 22]
 [43 50]]
```

```
import numpy as np

# Declaring the variables of matrix A.
a11, a12 = 1, 2
a21, a22 = 3, 4

# Let's create matrix A
A = np.array([[a11, a12],
              [a21, a22]])

# Declaring the variables of matrix B.
b11, b12 = 5, 6
b21, b22 = 7, 8

# Let's create matrix B
B = np.array([[b11, b12],
              [b21, b22]])

# Multiplication using function matmul.
C = np.matmul(A,B)

# Calculation result
print('Result of Matrix C using matmul')
print(C)
```

Source: EDWARDS ER, et al., 2025.

This example demonstrates how the data of A and B are accessed and processed by the rules of linear algebra and stored in the matrix C. The organization in internal tables guarantees the efficiency of the algorithms, widely applied in engineering, modeling and computational analysis.

METHODS

This study employed computational methods to solve matrix problems in engineering, utilizing the latest version of the Python programming language. The NumPy library, essential for matrix operations, and the Jupyter Notebook environment from the Anaconda distribution were used for calculations, enabling efficient documentation and parameter variation. Python was chosen for its extensive application in engineering, offering precision, efficiency, and reproducibility in solving complex problems.

RESULT AND DISCUSSION

Application in Scientific Computing

The structured storage and calculation system allows mathematical operations to be performed in an optimized way, essential for scientific computing. This efficiency is especially important when dealing with large, complex matrices often found in engineering, physics, and chemistry (Heath R, 2018). For example, in many engineering problems, sparse matrices arise, which have a large number of zero elements. Although they seem simple, their storage and manipulation can consume a lot of memory and processing when specialized methods are not used, such as algorithms designed to deal exclusively with non-zero values. Another common challenge is the multiplication of very large matrices, such as those used in structural or fluid dynamics simulations. In these cases, scientific computing makes it possible to perform calculations that would be unfeasible manually, guaranteeing precision and efficiency even on massive scales (Golub GH and Loan CFV, 2013). Additionally, operations such as finding the transpose of a large matrix, essential in linear algebra and data analysis, require optimized approaches to minimize memory usage and execution time. These examples demonstrate how algebraic methods and advances in scientific computing are fundamental to solving real-world problems. They enable the application of structured and scalable processes, enabling advances in areas such as systems modeling, physical simulations, and data analysis in engineering and other exact sciences (Golub GH and Loan CFV, 2013).

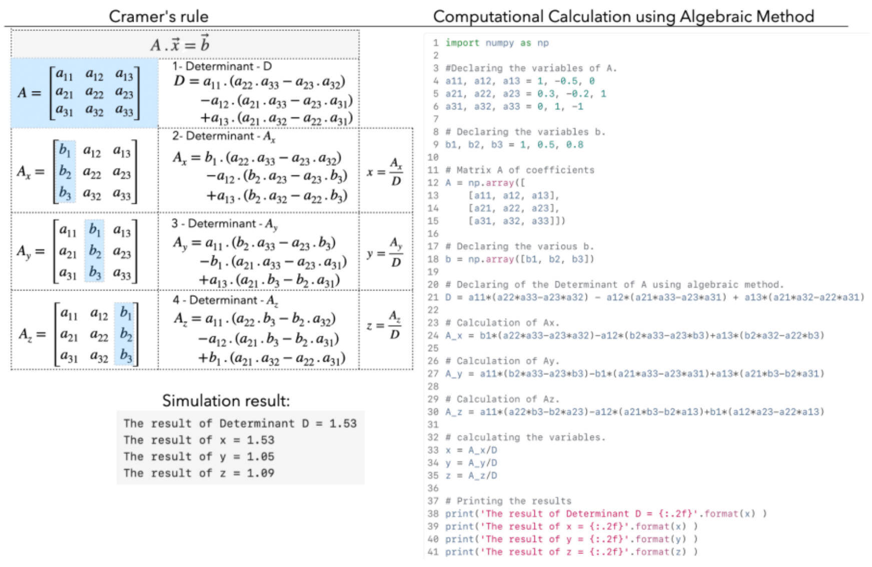
Computational Analysis in Solving 3x3 Linear Systems using Cramer's Rule.

Cramer's Method is a mathematical tool of great relevance in Engineering, especially in problems involving linear systems of equations. It allows you to solve systems with a single solution in an analytical way, being useful in situations where the number of unknowns is small, such as in the analysis of electrical circuits, in the sizing of structures and in solving problems related to mass and energy balances (Gilat A and Subramaniam V, 2000). Through the use of determinants, the method provides a straightforward approach to finding system variables, allowing accurate interpretations of results. Although it has limitations for larger systems due to computational cost, its application is fundamental in teaching analytical methods and understanding underlying mathematical principles (Anton H and Rorres C, 2013).

Furthermore, Cramer's Method serves as a basis for developing algorithms in engineering software, connecting theory to computational practice (Burden RL and Faires JD, 2011).

The technique solves n linear systems of equations with n unknowns, using determinants, as long as the determinant of the coefficient matrix is non-zero ($D \neq 0$). The figure 4 illustrates the application of Cramer's Method in obtaining the unknowns of a 3x3 system, highlighting the computational resolution based on algebraic methods. When applying the method, the values of the vector,$\vec{b}$, successively replace the values of the column corresponding to each unknown in the matrix of coefficients A, generating the matrices A_x, A_y and A_z. Then, the unknowns x, y and z are calculated by dividing the determinant of each matrix ($\Delta_x, \Delta_y, \Delta_z$) by the determinant of the original matrix (D). Furthermore, the figure presents the codes implemented in Python to calculate the unknowns of the linear system $A.\vec{x} = \vec{b}$, with the input of the values of the variables defined by the matrix and the corresponding vector (Meyer CD, 2000).

Figure 4 – Computational methods for solving Cramer's rule.



$$A.\vec{x} = \vec{b}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

1- Determinant - D

$$D = a_{11}.(a_{22}.a_{33} - a_{23}.a_{32}) - a_{12}.(a_{21}.a_{33} - a_{23}.a_{31}) + a_{13}.(a_{21}.a_{32} - a_{22}.a_{31})$$

$$A_x = \begin{bmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{bmatrix}$$

2- Determinant - A_x

$$A_x = b_1.(a_{22}.a_{33} - a_{23}.a_{32}) - a_{12}.(b_2.a_{23} - a_{23}.b_3) + a_{13}.(b_2.a_{32} - a_{22}.b_3)$$

$$x = \frac{A_x}{D}$$

$$A_y = \begin{bmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{bmatrix}$$

3 - Determinant - A_y

$$A_y = a_{11}.(b_2.a_{33} - a_{23}.b_3) - b_1.(a_{21}.a_{33} - a_{23}.a_{31}) + a_{13}.(a_{21}.b_3 - b_2.a_{31})$$

$$y = \frac{A_y}{D}$$

$$A_z = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{bmatrix}$$

4 - Determinant - A_z

$$A_z = a_{11}.(a_{22}.b_3 - b_2.a_{32}) - a_{12}.(a_{21}.b_3 - b_2.a_{31}) + b_1.(a_{21}.a_{32} - a_{22}.a_{31})$$

$$z = \frac{A_z}{D}$$

```
The result of Determinant D = 1.53
The result of x = 1.53
The result of y = 1.05
The result of z = 1.09
```

```python
import numpy as np

#Declaring the variables of A.
a11, a12, a13 = 1, -0.5, 0
a21, a22, a23 = 0.3, -0.2, 1
a31, a32, a33 = 0, 1, -1

# Declaring the variables b.
b1, b2, b3 = 1, 0.5, 0.8

# Matrix A of coefficients
A = np.array([
    [a11, a12, a13],
    [a21, a22, a23],
    [a31, a32, a33]])

# Declaring the various b.
b = np.array([b1, b2, b3])

# Declaring of the Determinant of A using algebraic method.
D = a11*(a22*a33-a23*a32) - a12*(a21*a33-a23*a31) + a13*(a21*a32-a22*a31)

# Calculation of Ax.
A_x = b1*(a22*a33-a23*a32)-a12*(b2*a33-a23*b3)+a13*(b2*a32-a22*b3)

# Calculation of Ay.
A_y = a11*(b2*a33-a23*b3)-b1*(a21*a33-a23*a31)+a13*(a21*b3-b2*a31)

# Calculation of Az.
A_z = a11*(a22*b3-b2*a23)-a12*(a21*b3-b2*a13)+b1*(a12*a23-a22*a13)

# calculating the variables.
x = A_x/D
y = A_y/D
z = A_z/D

# Printing the results
print('The result of Determinant D = {:.2f}'.format(x) )
print('The result of x = {:.2f}'.format(x) )
print('The result of y = {:.2f}'.format(y) )
print('The result of z = {:.2f}'.format(z) )
```

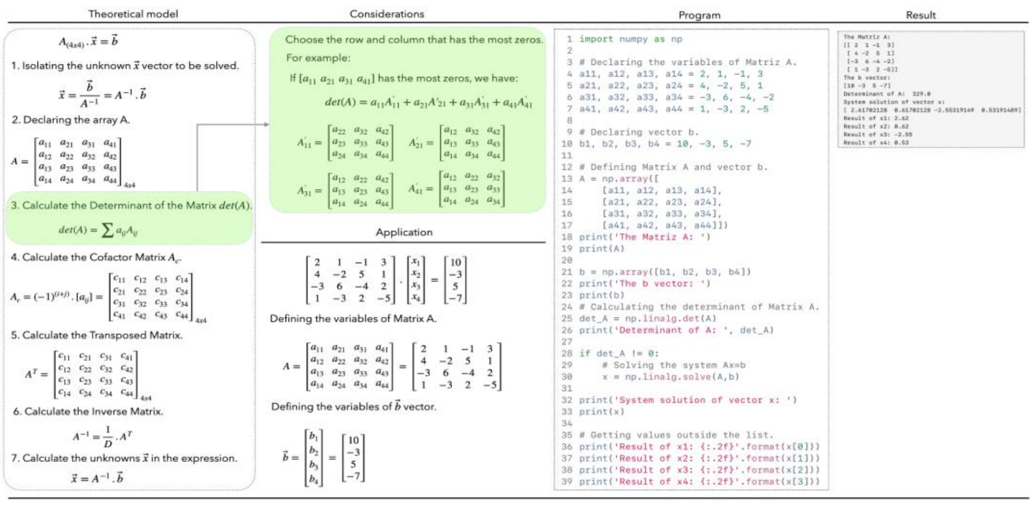
Source: EDWARDS ER, et al., 2025.

Computational Analysis in Solving 4x4 Linear Systems.

The use of computational methods to solve matrices, especially those with many rows and columns, is essential in several areas of engineering and other applied sciences. A 4x4 matrix, for example, is widely used in small-scale problems, such as in the analysis of electrical circuits and in the balance of forces in structures. However, as the number of rows and columns increases, such as in matrices used in the Finite Element Method (FEM) for modeling solids in CAD software, or in the analysis of complex chemical processes, such as separations in distillation towers, the use of computational methods becomes indispensable. These methods allow the manipulation and solution of highly complex systems, ensuring accuracy and efficiency, even in large-scale applications (Anton H and Rorres C, 2013). Furthermore, the use of matrices in areas such as banking cryptography reinforces their multidisciplinary importance. Thus, the use of numerical algorithms and computational tools provides a practical and powerful solution to problems that would be unfeasible to solve manually, connecting mathematical theory with real challenges (Burden RL and Faires JD, 2011).

To solve linear systems of the type $A.\vec{x} = \vec{b}$ the technique used consists of isolating the unknown $\vec{x}$ and declaring the numerical values of the matrix A. The calculation of the determinant is given by the expression $det(A) = \sum a_{ij}A_{ij}$. The cofactor matrix (A_C) is presented in item 4 of figure 5, while the transposed matrix (A^T) is obtained by swapping the matrix rows with columns, as illustrated in item 5. The inverse matrix is calculated by dividing the transposed cofactor matrix by the determinant, as shown in item 6 of the same figure. To calculate the determinant, it is recommended to choose the row or column with the most cells equal to zero, a consideration presented in the second column of the figure 5. An example application is shown in the applications section of this figure. Finally, the calculations were implemented with the help of the NumPy library, using the np.linalg.solve function, since the purely algebraic resolution would make this article excessively long.

Figure 5 – Computational methods in resolving 4x4 matrices.



Theoretical model

$$A_{(4x4)}.\vec{x} = \vec{b}$$

1. Isolating the unknown $\vec{x}$ vector to be solved.

$$\vec{x} = \frac{\vec{b}}{A^{-1}} = A^{-1}.\vec{b}$$

2. Declaring the array A.

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}_{4x4}$$

3. Calculate the Determinant of the Matrix $det(A)$.

$$det(A) = \sum a_{ij}A_{ij}$$

4. Calculate the Cofactor Matrix A_c.

$$A_c = (-1)^{(i+j)}.[a_{ij}] = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}_{4x4}$$

5. Calculate the Transposed Matrix.

$$A^T = \begin{bmatrix} c_{11} & c_{21} & c_{31} & c_{41} \\ c_{12} & c_{22} & c_{32} & c_{42} \\ c_{13} & c_{23} & c_{33} & c_{43} \\ c_{14} & c_{24} & c_{34} & c_{44} \end{bmatrix}_{4x4}$$

6. Calculate the Inverse Matrix.

$$A^{-1} = \frac{1}{D}.A^T$$

7. Calculate the unknowns $\vec{x}$ in the expression.

$$\vec{x} = A^{-1}.\vec{b}$$

Considerations

Choose the row and column that has the most zeros.

For example:

If $[a_{11}\ a_{21}\ a_{31}\ a_{41}]$ has the most zeros, we have:

$$det(A) = a_{11}A'_{11} + a_{21}A'_{21} + a_{31}A'_{31} + a_{41}A'_{41}$$

$$A'_{11} = \begin{bmatrix} a_{22} & a_{32} & a_{42} \\ a_{23} & a_{33} & a_{43} \\ a_{24} & a_{34} & a_{44} \end{bmatrix} \quad A'_{21} = \begin{bmatrix} a_{12} & a_{32} & a_{42} \\ a_{13} & a_{33} & a_{43} \\ a_{14} & a_{34} & a_{44} \end{bmatrix}$$

$$A'_{31} = \begin{bmatrix} a_{12} & a_{22} & a_{42} \\ a_{13} & a_{23} & a_{43} \\ a_{14} & a_{24} & a_{44} \end{bmatrix} \quad A'_{41} = \begin{bmatrix} a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \\ a_{14} & a_{24} & a_{34} \end{bmatrix}$$

Application

$$\begin{bmatrix} 2 & 1 & -1 & 3 \\ 4 & -2 & 5 & 1 \\ -3 & 6 & -4 & 2 \\ 1 & -3 & 2 & -5 \end{bmatrix} . \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 10 \\ -3 \\ 5 \\ -7 \end{bmatrix}$$

Defining the variables of Matrix A.

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 & 3 \\ 4 & -2 & 5 & 1 \\ -3 & 6 & -4 & 2 \\ 1 & -3 & 2 & -5 \end{bmatrix}$$

Defining the variables of $\vec{b}$ vector.

$$\vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} 10 \\ -3 \\ 5 \\ -7 \end{bmatrix}$$

Program

```python
import numpy as np

# Declaring the variables of Matriz A.
a11, a12, a13, a14 = 2, 1, -1, 3
a21, a22, a23, a24 = 4, -2, 5, 1
a31, a32, a33, a34 = -3, 6, -4, -2
a41, a42, a43, a44 = 1, -3, 2, -5

# Declaring vector b.
b1, b2, b3, b4 = 10, -3, 5, -7

# Defining Matrix A and vector b.
A = np.array([
    [a11, a12, a13, a14],
    [a21, a22, a23, a24],
    [a31, a32, a33, a34],
    [a41, a42, a43, a44]])
print('The Matriz A: ')
print(A)

b = np.array([b1, b2, b3, b4])
print('The b vector: ')
print(b)
# Calculating the determinant of Matrix A.
det_A = np.linalg.det(A)
print('Determinant of A: ', det_A)

if det_A != 0:
    # Solving the system Ax=b
    x = np.linalg.solve(A,b)

print('System solution of vector x: ')
print(x)

# Getting values outside the list.
print('Result of x1: {:.2f}'.format(x[0]))
print('Result of x2: {:.2f}'.format(x[1]))
print('Result of x3: {:.2f}'.format(x[2]))
print('Result of x4: {:.2f}'.format(x[3]))
```

Result

```
The Matriz A:
[[ 2  1 -1  3]
 [ 4 -2  5  1]
 [-3  6 -4 -2]
 [ 1 -3  2 -5]]
The b vector:
[10 -3  5 -7]
Determinant of A:  329.0
System solution of vector x:
[ 2.61702128  0.61702128 -2.55319149  0.53191489]
Result of x1: 2.62
Result of x2: 0.62
Result of x3: -2.55
Result of x4: 0.53
```

Source: EDWARDS ER, et al., 2025.

Computational Analysis in Solving 5x5 Linear Systems.

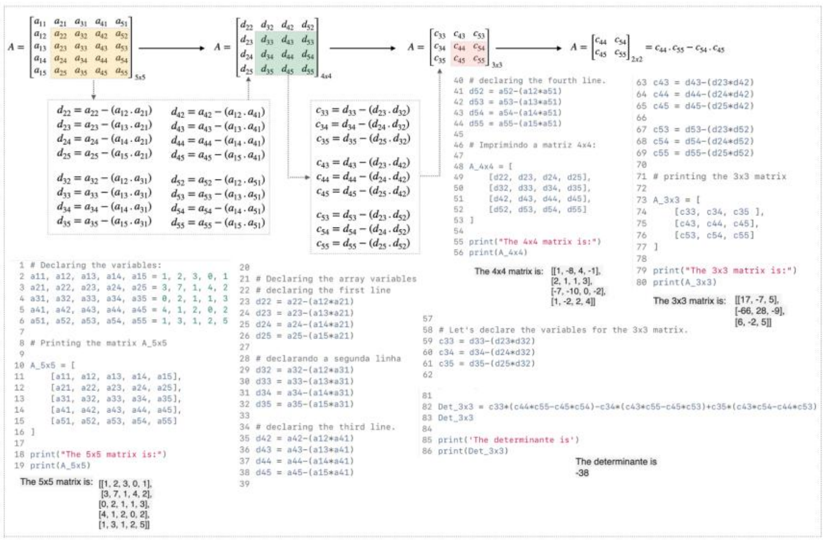
The figure 6 shows the resolution of a 5x5 matrix using the algebraic method, directly manipulating the rows and columns, without the need to resort to the cofactor definition, $(-1)^{i+j} det(A_{ij})$. The latter is generally used in determinant calculations using the row or column expansion approach (Golub GH and Loan CFV, 2013). If we used the expansion by cofactors to calculate the determinant of the 5x5 matrix, we could expand along the first row ($i = 1$), considering the sum over the columns ($j = 1$ *to* $j = 5$), according to the following expression:

$$det(A) = \sum_{j=1}^{5} (-1)^{i+j} det(A_{1j})$$

Where A_{ij} is the submatrix 4x4 obtained by removing the row $i = 1$ and the column j.

In the algebraic method used, the terms $(-1)^{i+j}$ are not necessary, as the matrix elements are manipulated directly, which simplifies the calculations and eliminates the need to resort to the classical definition of cofactors. This procedure offers an efficient alternative for calculating the determinant, avoiding complete expansion by rows or columns (Heath R, 2018). Furthermore, in the code presented, it is highlighted that exclusively algebraic deductions were used, without the aid of libraries or specific programming functions.

Figure 6 – Computational methods in resolving 5 x 5 matrices.



$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} & a_{51} \\ a_{12} & a_{22} & a_{32} & a_{42} & a_{52} \\ a_{13} & a_{23} & a_{33} & a_{43} & a_{53} \\ a_{14} & a_{24} & a_{34} & a_{44} & a_{54} \\ a_{15} & a_{25} & a_{35} & a_{45} & a_{55} \end{bmatrix}_{5x5} \longrightarrow A = \begin{bmatrix} d_{22} & d_{32} & d_{42} & d_{52} \\ d_{23} & d_{33} & d_{43} & d_{53} \\ d_{24} & d_{34} & d_{44} & d_{54} \\ d_{25} & d_{35} & d_{45} & d_{55} \end{bmatrix}_{4x4} \longrightarrow A = \begin{bmatrix} c_{33} & c_{43} & c_{53} \\ c_{34} & c_{44} & c_{54} \\ c_{35} & c_{45} & c_{55} \end{bmatrix}_{3x3} \longrightarrow A = \begin{bmatrix} c_{44} & c_{54} \\ c_{45} & c_{55} \end{bmatrix}_{2x2} = c_{44} . c_{55} - c_{54} . c_{45}$$

$$\begin{aligned} d_{22} &= a_{22} - (a_{12} . a_{21}) & d_{42} &= a_{42} - (a_{12} . a_{41}) \\ d_{23} &= a_{23} - (a_{13} . a_{21}) & d_{43} &= a_{43} - (a_{13} . a_{41}) \\ d_{24} &= a_{24} - (a_{14} . a_{21}) & d_{44} &= a_{44} - (a_{14} . a_{41}) \\ d_{25} &= a_{25} - (a_{15} . a_{21}) & d_{45} &= a_{45} - (a_{15} . a_{41}) \\ d_{32} &= a_{32} - (a_{12} . a_{31}) & d_{52} &= a_{52} - (a_{12} . a_{51}) \\ d_{33} &= a_{33} - (a_{13} . a_{31}) & d_{53} &= a_{53} - (a_{13} . a_{51}) \\ d_{34} &= a_{34} - (a_{14} . a_{31}) & d_{54} &= a_{54} - (a_{14} . a_{51}) \\ d_{35} &= a_{35} - (a_{15} . a_{31}) & d_{55} &= a_{55} - (a_{15} . a_{51}) \end{aligned}$$

$$\begin{aligned} c_{33} &= d_{33} - (d_{23} . d_{32}) \\ c_{34} &= d_{34} - (d_{24} . d_{32}) \\ c_{35} &= d_{35} - (d_{25} . d_{32}) \\ c_{43} &= d_{43} - (d_{23} . d_{42}) \\ c_{44} &= d_{44} - (d_{24} . d_{42}) \\ c_{45} &= d_{45} - (d_{25} . d_{42}) \\ c_{53} &= d_{53} - (d_{23} . d_{52}) \\ c_{54} &= d_{54} - (d_{24} . d_{52}) \\ c_{55} &= d_{55} - (d_{25} . d_{52}) \end{aligned}$$

```python
# Declaring the variables:
a11, a12, a13, a14, a15 = 1, 2, 3, 0, 1
a21, a22, a23, a24, a25 = 3, 7, 1, 4, 2
a31, a32, a33, a34, a35 = 0, 2, 1, 1, 3
a41, a42, a43, a44, a45 = 4, 1, 2, 0, 2
a51, a52, a53, a54, a55 = 1, 3, 1, 2, 5

# Printing the matrix A_5x5

A_5x5 = [
    [a11, a12, a13, a14, a15],
    [a21, a22, a23, a24, a25],
    [a31, a32, a33, a34, a35],
    [a41, a42, a43, a44, a45],
    [a51, a52, a53, a54, a55]
]

print("The 5x5 matrix is:")
print(A_5x5)
```

```
The 5x5 matrix is:  [[1, 2, 3, 0, 1],
                     [3, 7, 1, 4, 2],
                    [0, 2, 1, 1, 3],
                    [4, 1, 2, 0, 2],
                    [1, 3, 1, 2, 5]]
```

```python

# Declaring the array variables
# declaring the first line
d22 = a22-(a12*a21)
d23 = a23-(a13*a21)
d24 = a24-(a14*a21)
d25 = a25-(a15*a21)

# declarando a segunda linha
d32 = a32-(a12*a31)
d33 = a33-(a13*a31)
d34 = a34-(a14*a31)
d35 = a35-(a15*a31)

# declaring the third line.
d42 = a42-(a12*a41)
d43 = a43-(a13*a41)
d44 = a44-(a14*a41)
d45 = a45-(a15*a41)

# declaring the fourth line.
d52 = a52-(a12*a51)
d53 = a53-(a13*a51)
d54 = a54-(a14*a51)
d55 = a55-(a15*a51)

# Imprimindo a matriz 4x4:

A_4x4 = [
    [d22, d23, d24, d25],
    [d32, d33, d34, d35],
    [d42, d43, d44, d45],
    [d52, d53, d54, d55]
]

print("The 4x4 matrix is:")
print(A_4x4)
```

```
The 4x4 matrix is:  [[1, -8, 4, -1],
                    [2, 1, 1, 3],
                    [-7, -10, 0, -2],
                    [1, -2, 2, 4]]
```

```python

# Let's declare the variables for the 3x3 matrix.
c33 = d33-(d23*d32)
c34 = d34-(d24*d32)
c35 = d35-(d25*d32)

c43 = d43-(d23*d42)
c44 = d44-(d24*d42)
c45 = d45-(d25*d42)

c53 = d53-(d23*d52)
c54 = d54-(d24*d52)
c55 = d55-(d25*d52)

# printing the 3x3 matrix

A_3x3 = [
    [c33, c34, c35 ],
    [c43, c44, c45],
    [c53, c54, c55]
]

print("The 3x3 matrix is:")
print(A_3x3)
```

```
The 3x3 matrix is:  [[17, -7, 5],
                    [-66, 28, -9],
                    [6, -2, 5]]
```

```python

Det_3x3 = c33*(c44*c55-c45*c54)-c34*(c43*c55-c45*c53)+c35*(c43*c54-c44*c53)
Det_3x3

print('The determinante is')
print(Det_3x3)
```

```
The determinante is
-38
```

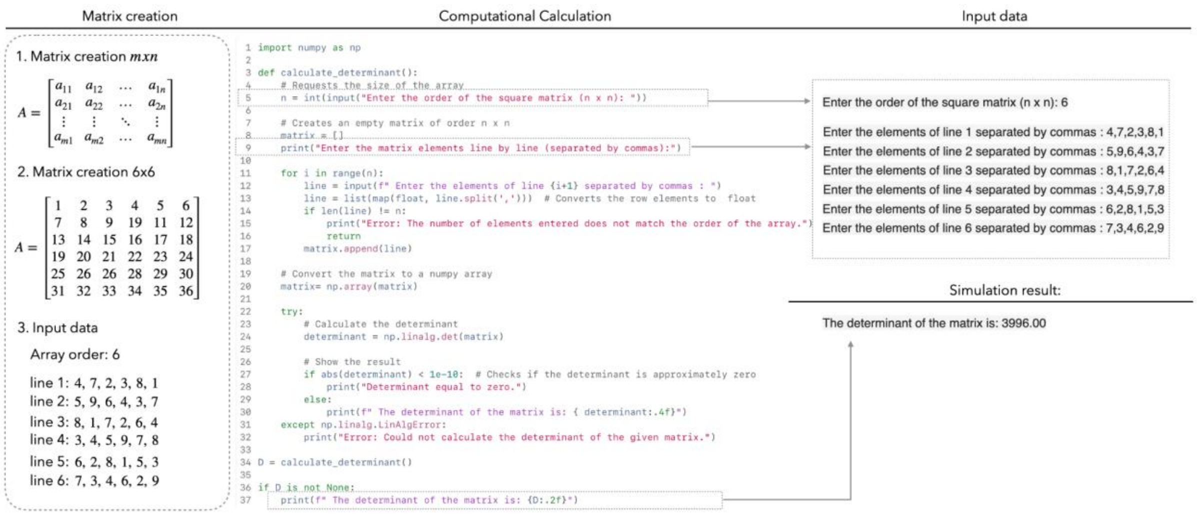
Source: EDWARDS ER, et al., 2025.

Computational Resolution of nxn Matrix

As the order of a matrix increases, calculations using algebraic methods become progressively more complex, making it necessary to use computational libraries, such as NumPy. This library allows the construction and manipulation of matrices of order n x n efficiently (Golub GH e Van Loan CF, 2013). Matrix calculations follow well-defined patterns and the identification of these patterns makes it possible to develop algorithms capable of calculating fundamental properties, such as the determinant, as long as it is non-zero (Heath R, 2018).

Figure 7 illustrates an example of Python code that allows the user to construct a matrix of order n x n. The user inserts the elements of each row of the matrix separated by commas, facilitating data entry. The program calculates the determinant of the given matrix, as long as $D \neq 0$. This type of approach is especially useful in application areas involving highly complex calculations, such as fluid flow analysis using the Finite Element Method (FEM), where high-order matrices are often required to solve multidimensional problems (Saad Y, 2003).

Figure 7 – Computational methods for solving the n x n Matrix.



1. Matrix creation mxn

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

2. Matrix creation 6x6

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 19 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 26 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{bmatrix}$$

3. Input data

Array order: 6

line 1: 4, 7, 2, 3, 8, 1
line 2: 5, 9, 6, 4, 3, 7
line 3: 8, 1, 7, 2, 6, 4
line 4: 3, 4, 5, 9, 7, 8
line 5: 6, 2, 8, 1, 5, 3
line 6: 7, 3, 4, 6, 2, 9

```python
import numpy as np

def calculate_determinant():
    # Requests the size of the array
    n = int(input("Enter the order of the square matrix (n x n): "))

    # Creates an empty matrix of order n x n
    matrix = []
    print("Enter the matrix elements line by line (separated by commas):")

    for i in range(n):
        line = input(f" Enter the elements of line {i+1} separated by commas : ")
        line = list(map(float, line.split(',')))  # Converts the row elements to  float
        if len(line) != n:
            print("Error: The number of elements entered does not match the order of the array.")
            return
        matrix.append(line)

    # Convert the matrix to a numpy array
    matrix= np.array(matrix)

    try:
        # Calculate the determinant
        determinant = np.linalg.det(matrix)

        # Show the result
        if abs(determinant) < 1e-10:  # Checks if the determinant is approximately zero
            print("Determinant equal to zero.")
        else:
            print(f" The determinant of the matrix is: { determinant:.4f}")
    except np.linalg.LinAlgError:
        print("Error: Could not calculate the determinant of the given matrix.")

D = calculate_determinant()

if D is not None:
    print(f" The determinant of the matrix is: {D:.2f}")
```

```
Enter the order of the square matrix (n x n): 6

Enter the elements of line 1 separated by commas : 4,7,2,3,8,1
Enter the elements of line 2 separated by commas : 5,9,6,4,3,7
Enter the elements of line 3 separated by commas : 8,1,7,2,6,4
Enter the elements of line 4 separated by commas : 3,4,5,9,7,8
Enter the elements of line 5 separated by commas : 6,2,8,1,5,3
Enter the elements of line 6 separated by commas : 7,3,4,6,2,9
```

```
The determinant of the matrix is: 3996.00
```

Source: EDWARDS ER, et al., 2025.

Sparse Matrix Computational Resolution

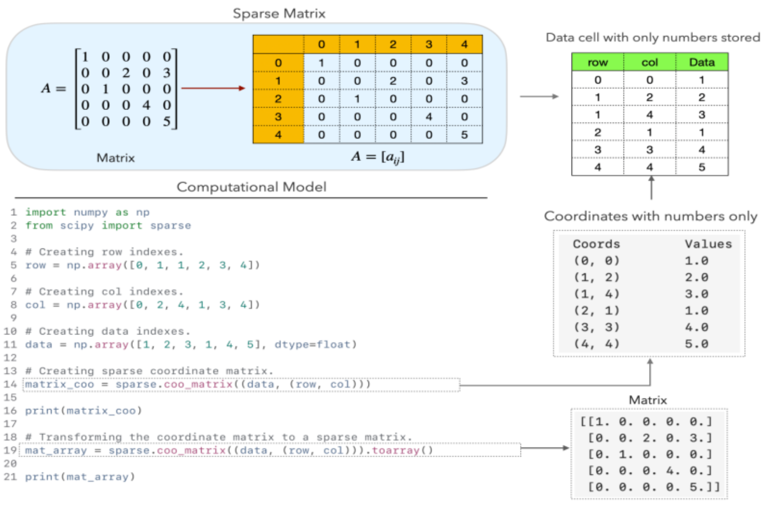
A sparse matrix is characterized by the presence of a large number of zero elements in its structure. Direct storage of these values can lead to unnecessary computational consumption, especially in large-scale problems. Sparse matrices are widely used in engineering calculations, such as in structural analysis using the finite element method, as well as in electrical, mechanical and chemical engineering applications (Stroher GR et al., 2021; Chapra SC and Canale RP, 2016).

Figure 8 illustrates a matrix A composed predominantly of zero elements. Each element of this matrix is represented as a_{ij}, where i denotes the row and j the column. To simplify computational operations and optimize memory usage, it is common to convert sparse matrices into structures that store only the non-zero elements and their respective positions (Saad Y, 2003; Duff IS et al., 2017).

One of the most widely used computational representations for sparse matrices relies on three primary vectors: the row vector, which denotes the row indices of nonzero elements; the col vector, which stores the corresponding column indices; and the data vector, which contains the numeric values of these elements (Golub GH and Loan CFV, 2013; Jones E et al., 2001). The data cell table with only numeric stored in Figure 8 exemplifies such storage structures, considering only nonzero elements. This approach significantly reduces memory consumption and processing time, thus facilitating the solving of large-scale problems.

Moreover, several computational libraries implement these optimized structures. For instance, Python's scipy.sparse library provides efficient formats such as CSR (*Compressed Sparse Row*) and COO (*Coordinate List*), which are widely employed in numerical methods and computational simulations (Jones E et al., 2001). By leveraging these optimized representations, complex calculations can be performed efficiently while conserving computational resources.

Figure 8 – Method for creating sparse matrix.


Sparse Matrix
Matrix
$A = [a_{ij}]$
Data cell with only numbers stored
Computational Model
Coordinates with numbers only
Matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 3 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

	0	1	2	3	4
0	1	0	0	0	0
1	0	0	2	0	3
2	0	1	0	0	0
3	0	0	0	4	0
4	0	0	0	0	5

row	col	Data
0	0	1
1	2	2
1	4	3
2	1	1
3	3	4
4	4	5

```python
import numpy as np
from scipy import sparse

# Creating row indexes.
row = np.array([0, 1, 1, 2, 3, 4])

# Creating col indexes.
col = np.array([0, 2, 4, 1, 3, 4])

# Creating data indexes.
data = np.array([1, 2, 3, 1, 4, 5], dtype=float)

# Creating sparse coordinate matrix.
matrix_coo = sparse.coo_matrix((data, (row, col)))

print(matrix_coo)

# Transforming the coordinate matrix to a sparse matrix.
mat_array = sparse.coo_matrix((data, (row, col))).toarray()

print(mat_array)
```

```
Coords      Values
(0, 0)      1.0
(1, 2)      2.0
(1, 4)      3.0
(2, 1)      1.0
(3, 3)      4.0
(4, 4)      5.0
```

```
[[1. 0. 0. 0. 0.]
 [0. 0. 2. 0. 3.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 4. 0.]
 [0. 0. 0. 0. 5.]]
```

Source: EDWARDS ER, et al., 2025.

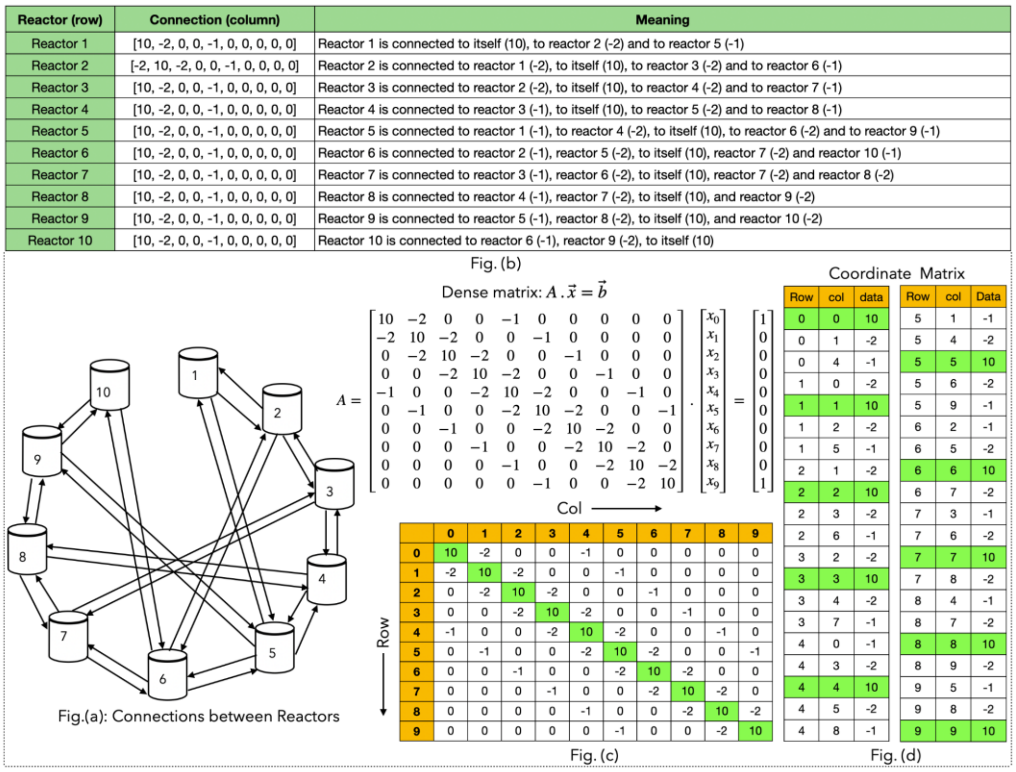
Figure 9 presents an example of the application of the mentioned techniques in solving a problem involving the concentration of a certain product that circulates between ten interconnected tanks, Fig. 9(a). The corresponding Fig. 9(b) describes the connections between the tanks, as well as the respective concentrations. To model the problem, the matrix equation $A.\vec{x} = \vec{b}$ was used, where A is a dense matrix representing the coefficients of the linear system, $\vec{x}$ is the vector of unknowns (concentrations to be determined) and $\vec{b}$ is the vector of independent terms.

The complete data in Fig. 9(c) presents the values of the matrix A arranged so that the first element is stored in the row $i = 0$ and in the column $j = 0$, following the indexing pattern. However, to optimize computational calculations, it is possible to represent this matrix in coordinate form, storing only non-zero elements.

Figure 9(d) shows this representation, indicating the positions of the rows (i) and column (j), in addition to the corresponding numerical values. This approach significantly reduces the memory used and speeds up the resolution of the linear system, since only the relevant cells of the matrix are considered. The application of these computational techniques enables the efficient analysis of interconnected tank systems, being widely used in simulations and modeling of industrial processes.

Figure 10 presents the script containing the complete computational calculations, including the declaration of input data: row, col and data, which represent, respectively, the rows, columns and numerical values of the matrix in coordinated form (*COO*). In the code, the data was initially printed in sparse format (*COO format*) to demonstrate the optimized structure, and later converted to dense matrix format using the toarray() method. To solve the linear system $A.\vec{x} = \vec{b}$, the iterative method *LSQR* was used, suitable for sparse matrices and large problems (Saad Y, 2003; Duff IS et al., 2017). After resolution, the concentrations calculated for each tank were printed, along with their respective units, allowing a detailed analysis of the results. This approach demonstrates how the application of sparse representation techniques, combined with efficient numerical methods, facilitates the resolution of complex problems involving interconnected tank systems.

Figure 9 – Coordinate notation for creating sparse matrices.



Reactor (row)	Connection (column)	Meaning
Reactor 1	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 1 is connected to itself (10), to reactor 2 (-2) and to reactor 5 (-1)
Reactor 2	[-2, 10, -2, 0, 0, -1, 0, 0, 0, 0]	Reactor 2 is connected to reactor 1 (-2), to itself (10), to reactor 3 (-2) and to reactor 6 (-1)
Reactor 3	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 3 is connected to reactor 2 (-2), to itself (10), to reactor 4 (-2) and to reactor 7 (-1)
Reactor 4	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 4 is connected to reactor 3 (-1), to itself (10), to reactor 5 (-2) and to reactor 8 (-1)
Reactor 5	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 5 is connected to reactor 1 (-1), to reactor 4 (-2), to itself (10), to reactor 6 (-2) and to reactor 9 (-1)
Reactor 6	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 6 is connected to reactor 2 (-1), reactor 5 (-2), to itself (10), reactor 7 (-2) and reactor 10 (-1)
Reactor 7	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 7 is connected to reactor 3 (-1), reactor 6 (-2), to itself (10), reactor 7 (-2) and reactor 8 (-2)
Reactor 8	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 8 is connected to reactor 4 (-1), reactor 7 (-2), to itself (10), and reactor 9 (-2)
Reactor 9	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 9 is connected to reactor 5 (-1), reactor 8 (-2), to itself (10), and reactor 10 (-2)
Reactor 10	[10, -2, 0, 0, -1, 0, 0, 0, 0, 0]	Reactor 10 is connected to reactor 6 (-1), reactor 9 (-2), to itself (10)

Fig. (b)

Dense matrix: $A\,.\vec{x} = \vec{b}$

$$A = \begin{bmatrix} 10 & -2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & 10 & -2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 10 & -2 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -2 & 10 & -2 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & -2 & 10 & -2 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & -2 & 10 & -2 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & -2 & 10 & -2 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -2 & 10 & -2 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & -2 & 10 & -2 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -2 & 10 \end{bmatrix} . \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Col ⟶

Row ↓

	0	1	2	3	4	5	6	7	8	9
0	10	-2	0	0	-1	0	0	0	0	0
1	-2	10	-2	0	0	-1	0	0	0	0
2	0	-2	10	-2	0	0	-1	0	0	0
3	0	0	-2	10	-2	0	0	-1	0	0
4	-1	0	0	-2	10	-2	0	0	-1	0
5	0	-1	0	0	-2	10	-2	0	0	-1
6	0	0	-1	0	0	-2	10	-2	0	0
7	0	0	0	-1	0	0	-2	10	-2	0
8	0	0	0	0	-1	0	0	-2	10	-2
9	0	0	0	0	0	-1	0	0	-2	10

Fig. (c)

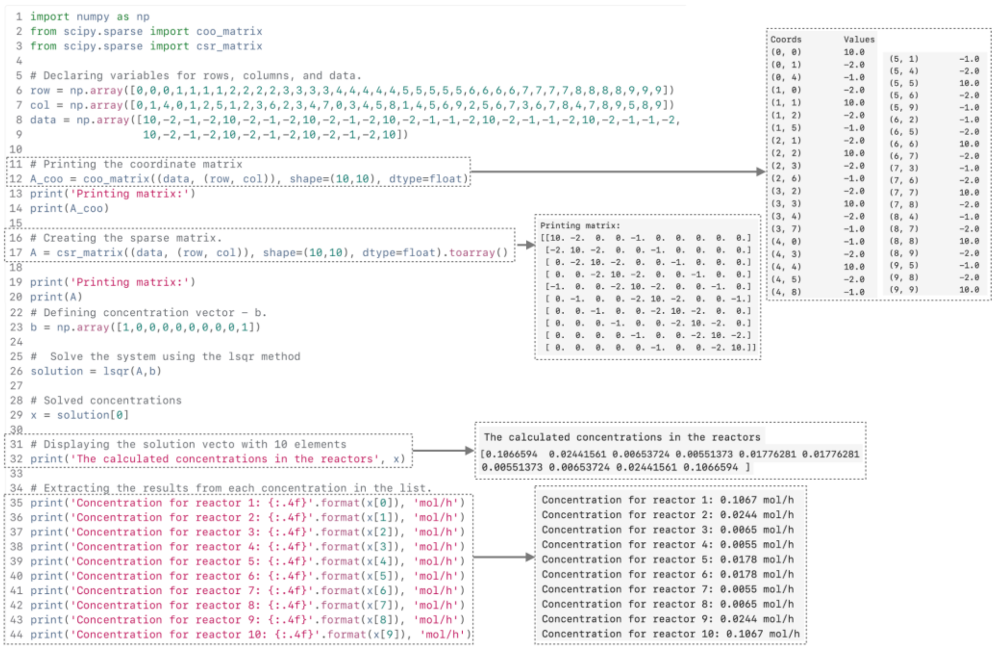
Coordinate Matrix

Row	col	data	Row	col	Data
0	0	10	5	1	-1
0	1	-2	5	4	-2
0	4	-1	5	5	10
1	0	-2	5	6	-2
1	1	10	5	9	-1
1	2	-2	6	2	-1
1	5	-1	6	5	-2
2	1	-2	6	6	10
2	2	10	6	7	-2
2	3	-2	7	3	-1
2	6	-1	7	6	-2
3	2	-2	7	7	10
3	3	10	7	8	-2
3	4	-2	8	4	-1
3	7	-1	8	7	-2
4	0	-1	8	8	10
4	3	-2	8	9	-2
4	4	10	9	5	-1
4	5	-2	9	8	-2
4	8	-1	9	9	10

Fig. (d)

Source: EDWARDS ER, et al., 2025.

Figure 10 – Computational calculation for an interconnected tank system.



```python
import numpy as np
from scipy.sparse import coo_matrix
from scipy.sparse import csr_matrix

# Declaring variables for rows, columns, and data.
row = np.array([0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,4,5,5,5,5,5,6,6,6,6,7,7,7,7,8,8,8,8,9,9,9])
col = np.array([0,1,4,0,1,2,5,1,2,3,6,2,3,4,7,0,3,4,5,8,1,4,5,6,9,2,5,6,7,3,6,7,8,4,7,8,9,5,8,9])
data = np.array([10,-2,-1,-2,10,-2,-1,-2,10,-2,-1,-2,10,-2,-1,-1,-2,10,-2,-1,-1,-2,10,-2,-1,-1,-2,
                 10,-2,-1,-2,10,-2,-1,-2,10,-2,-1,-2,10])

# Printing the coordinate matrix
A_coo = coo_matrix((data, (row, col)), shape=(10,10), dtype=float)
print('Printing matrix:')
print(A_coo)

# Creating the sparse matrix.
A = csr_matrix((data, (row, col)), shape=(10,10), dtype=float).toarray()

print('Printing matrix:')
print(A)
# Defining concentration vector - b.
b = np.array([1,0,0,0,0,0,0,0,0,1])

#  Solve the system using the lsqr method
solution = lsqr(A,b)

# Solved concentrations
x = solution[0]

# Displaying the solution vecto with 10 elements
print('The calculated concentrations in the reactors', x)

# Extracting the results from each concentration in the list.
print('Concentration for reactor 1: {:.4f}'.format(x[0]), 'mol/h')
print('Concentration for reactor 2: {:.4f}'.format(x[1]), 'mol/h')
print('Concentration for reactor 3: {:.4f}'.format(x[2]), 'mol/h')
print('Concentration for reactor 4: {:.4f}'.format(x[3]), 'mol/h')
print('Concentration for reactor 5: {:.4f}'.format(x[4]), 'mol/h')
print('Concentration for reactor 6: {:.4f}'.format(x[5]), 'mol/h')
print('Concentration for reactor 7: {:.4f}'.format(x[6]), 'mol/h')
print('Concentration for reactor 8: {:.4f}'.format(x[7]), 'mol/h')
print('Concentration for reactor 9: {:.4f}'.format(x[8]), 'mol/h')
print('Concentration for reactor 10: {:.4f}'.format(x[9]), 'mol/h')
```

```
Coords        Values
(0, 0)        10.0          (5, 1)        -1.0
(0, 1)        -2.0          (5, 4)        -2.0
(0, 4)        -1.0          (5, 5)        10.0
(1, 0)        -2.0          (5, 6)        -2.0
(1, 1)        10.0          (5, 9)        -1.0
(1, 2)        -2.0          (6, 2)        -1.0
(1, 5)        -1.0          (6, 5)        -2.0
(2, 1)        -2.0          (6, 6)        10.0
(2, 2)        10.0          (6, 7)        -2.0
(2, 3)        -2.0          (7, 3)        -1.0
(2, 6)        -1.0          (7, 6)        -2.0
(3, 2)        -2.0          (7, 7)        10.0
(3, 3)        10.0          (7, 8)        -2.0
(3, 4)        -2.0          (8, 4)        -1.0
(3, 7)        -1.0          (8, 7)        -2.0
(4, 0)        -1.0          (8, 8)        10.0
(4, 3)        -2.0          (8, 9)        -2.0
(4, 4)        10.0          (9, 5)        -1.0
(4, 5)        -2.0          (9, 8)        -2.0
(4, 8)        -1.0          (9, 9)        10.0
```

```
Printing matrix:
[[10. -2.  0.  0. -1.  0.  0.  0.  0.  0.]
 [-2. 10. -2.  0.  0. -1.  0.  0.  0.  0.]
 [ 0. -2. 10. -2.  0.  0. -1.  0.  0.  0.]
 [ 0.  0. -2. 10. -2.  0.  0. -1.  0.  0.]
 [-1.  0.  0. -2. 10. -2.  0.  0. -1.  0.]
 [ 0. -1.  0.  0. -2. 10. -2.  0.  0. -1.]
 [ 0.  0. -1.  0.  0. -2. 10. -2.  0.  0.]
 [ 0.  0.  0. -1.  0.  0. -2. 10. -2.  0.]
 [ 0.  0.  0.  0. -1.  0.  0. -2. 10. -2.]
 [ 0.  0.  0.  0.  0. -1.  0.  0. -2. 10.]]
```

```
The calculated concentrations in the reactors
[0.1066594  0.02441561 0.00653724 0.00551373 0.01776281 0.01776281
 0.00551373 0.00653724 0.02441561 0.1066594 ]
```

```
Concentration for reactor 1: 0.1067 mol/h
Concentration for reactor 2: 0.0244 mol/h
Concentration for reactor 3: 0.0065 mol/h
Concentration for reactor 4: 0.0055 mol/h
Concentration for reactor 5: 0.0178 mol/h
Concentration for reactor 6: 0.0178 mol/h
Concentration for reactor 7: 0.0055 mol/h
Concentration for reactor 8: 0.0065 mol/h
Concentration for reactor 9: 0.0244 mol/h
Concentration for reactor 10: 0.1067 mol/h
```

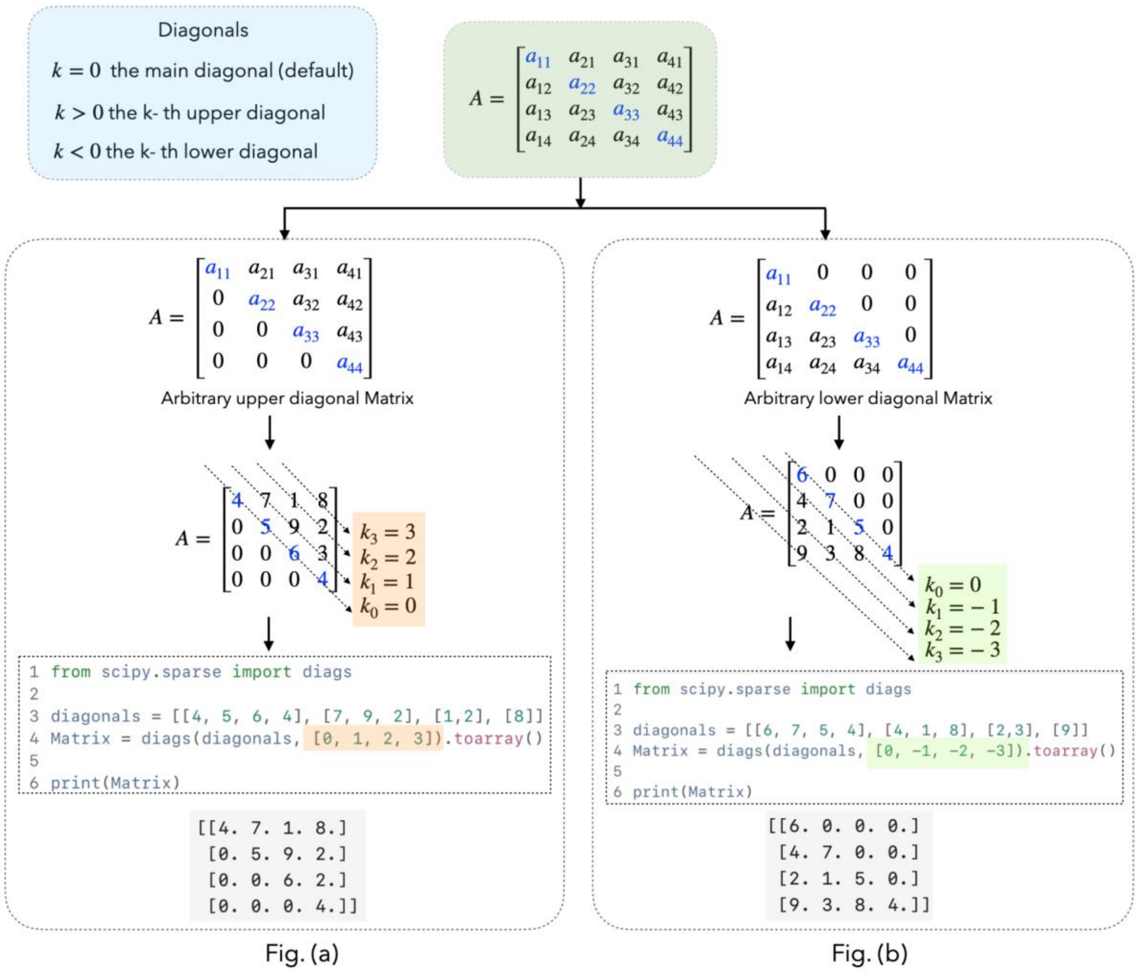
Source: EDWARDS ER, et al., 2025.

Applications of Triangular Matrices in Engineering Problems.

Triangular matrices, whether upper or lower, have a simplified structure in which all elements below the main diagonal (in the case of upper triangular matrices) or above the main diagonal (in the case of lower triangular matrices) are equal to zero. This characteristic allows a significant reduction in computational effort, especially in operations such as solving linear systems and matrix decompositions (Golub GH e Van Loan CF, 2013; Heath R, 2018). In engineering problems, these matrices arise naturally in several applications, such as in the analysis of civil and mechanical engineering structures, in the Finite Element Method (FEM), in electrical engineering and circuit analysis, in control and system dynamics, in signal processing and data analysis, in chemical engineering and process modeling (Mathews JH e Fink KD, 2012; da Silva Lessa L et al., 2020), as well as in iterative solutions and numerical methods (Saad Y, 2003; Duff IS et al., 2017). One of the main advantages of triangular matrices is in the solution of linear systems, since they allow solving equations such as $L.y = b$ or $U.x = u$ through direct substitution, reducing the number of necessary operations. Furthermore, these matrices present high computational efficiency compared to full matrices and guarantee greater numerical stability. Methods such as LU and Cholesky decomposition, which use triangular matrices, ensure greater accuracy in calculations, especially in large-scale problems (Golub GH e Van Loan CF, 2013).

Figure 11 presents the matrix $A = a_{ij}$ and the representations of its upper and lower diagonals using the *scipy.sparse* library functionality. In the Fig. 11(a), on the left, you can see the impression of an upper diagonal, where $k = 0$ corresponds to the main diagonal, and the positive values of $k(k > 0)$ indicate the upper diagonals, located above the main diagonal (Jones E et al., 2001). In the Fig.11(b), on the right, the lower diagonals are displayed, characterized by negative values of $k(k < 0)$, which correspond to the diagonals located below the main diagonal. Using the *diags* function, provided by the *scipy.sparse* library, it is possible to generate and manipulate sparse matrices from their diagonals. The k parameter allows you to directly specify which diagonal will be accessed or constructed, with $k = 0$ being the main reference diagonal. This approach offers an efficient way of working with diagonal and sparse matrices, being widely used in numerical problems and computational simulations, where the direct manipulation of diagonals is essential to reduce computational cost (Silva LL, et al., 2020; Saad Y, 2003).

Figure 11 – Printing of upper and lower triangular matrices.


Diagonals
$k = 0$ the main diagonal (default)
$k > 0$ the k- th upper diagonal
$k < 0$ the k- th lower diagonal
Arbitrary upper diagonal Matrix
$k_3 = 3$, $k_2 = 2$, $k_1 = 1$, $k_0 = 0$
Arbitrary lower diagonal Matrix
$k_0 = 0$, $k_1 = -1$, $k_2 = -2$, $k_3 = -3$
Fig. (a)
Fig. (b)

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ 0 & a_{22} & a_{32} & a_{42} \\ 0 & 0 & a_{33} & a_{43} \\ 0 & 0 & 0 & a_{44} \end{bmatrix} \qquad A = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{12} & a_{22} & 0 & 0 \\ a_{13} & a_{23} & a_{33} & 0 \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

$$A = \begin{bmatrix} 4 & 7 & 1 & 8 \\ 0 & 5 & 9 & 2 \\ 0 & 0 & 6 & 3 \\ 0 & 0 & 0 & 4 \end{bmatrix} \qquad A = \begin{bmatrix} 6 & 0 & 0 & 0 \\ 4 & 7 & 0 & 0 \\ 2 & 1 & 5 & 0 \\ 9 & 3 & 8 & 4 \end{bmatrix}$$

```python
from scipy.sparse import diags

diagonals = [[4, 5, 6, 4], [7, 9, 2], [1,2], [8]]
Matrix = diags(diagonals, [0, 1, 2, 3]).toarray()

print(Matrix)
```

```
[[4. 7. 1. 8.]
 [0. 5. 9. 2.]
 [0. 0. 6. 2.]
 [0. 0. 0. 4.]]
```

```python
from scipy.sparse import diags

diagonals = [[6, 7, 5, 4], [4, 1, 8], [2,3], [9]]
Matrix = diags(diagonals, [0, -1, -2, -3]).toarray()

print(Matrix)
```

```
[[6. 0. 0. 0.]
 [4. 7. 0. 0.]
 [2. 1. 5. 0.]
 [9. 3. 8. 4.]]
```

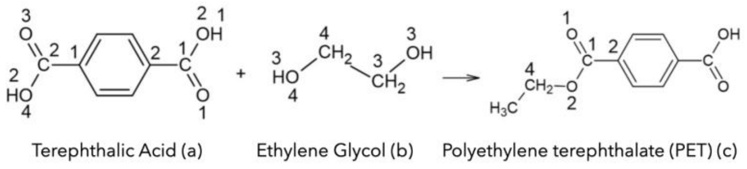
Source: EDWARDS ER, et al., 2025.

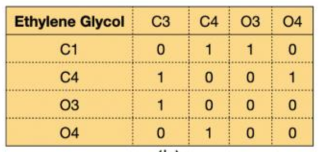
Matrix Representation of Molecular Interactions.

The use of matrix methods in problem solving offers the advantage of being able to identify rows and columns of each stored element. This knowledge is particularly useful in identifying the chemical reactions that occur between compounds, allowing the determination of the elements that react with each other to form new compounds (Leach AR, 2001). For example, by analyzing the chemical reaction between Terephthalic Acid and Ethylene Glycol, we can follow the bonds formed to produce Polyethylene Terephthalate (PET) (De Paoli, et al., 2009).

The figure 12 shows the representative matrix of bonds in Terephthalic Acid (a) and we can simplify the structure by considering the essential bonds, such as those with the COOH group, the benzene ring, and the specific carbon and oxygen atoms that participate in the reaction. In the bond matrix of Ethylene Glycol (b), we focus on the two carbon atoms (C3 and C4) and the oxygen atoms (O3 and O4) that react with Terephthalic Acid. And in the Polyethylene Terephthalate (PET) matrix (c), after the reaction, the bonds between Terephthalic Acid and Ethylene Glycol form Polyester. The final matrix represents the new bonds formed between the groups of the two compounds. Matrices A and B show the internal bonds in the starting compounds. Matrix C represents the polymer formed with new cross-links between the carbons and oxygens of the reactants (De Paoli, et al., 2009; Rosa AC, et al., 2023).

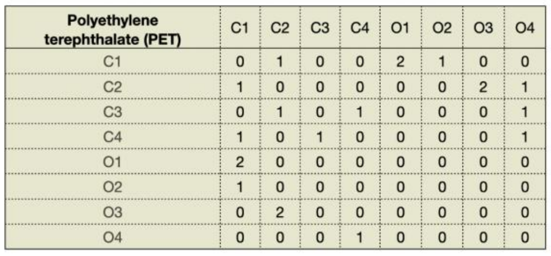
Figure 12 – Identification of chemical interactions through matrices.





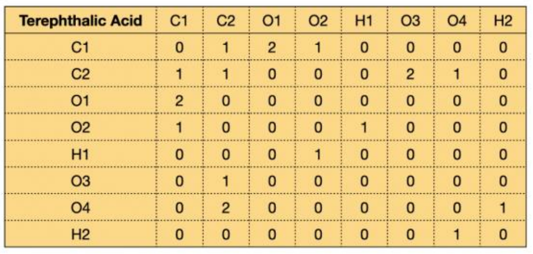
Ethylene Glycol	C3	C4	O3	O4
C1	0	1	1	0
C4	1	0	0	1
O3	1	0	0	0
O4	0	1	0	0

(b)



Polyethylene terephthalate (PET)	C1	C2	C3	C4	O1	O2	O3	O4
C1	0	1	0	0	2	1	0	0
C2	1	0	0	0	0	0	2	1
C3	0	1	0	1	0	0	0	1
C4	1	0	1	0	0	0	0	1
O1	2	0	0	0	0	0	0	0
O2	1	0	0	0	0	0	0	0
O3	0	2	0	0	0	0	0	0
O4	0	0	0	1	0	0	0	0

(c)



Terephthalic Acid	C1	C2	O1	O2	H1	O3	O4	H2
C1	0	1	2	1	0	0	0	0
C2	1	1	0	0	0	2	1	0
O1	2	0	0	0	0	0	0	0
O2	1	0	0	0	1	0	0	0
H1	0	0	0	1	0	0	0	0
O3	0	1	0	0	0	0	0	0
O4	0	2	0	0	0	0	0	1
H2	0	0	0	0	0	0	1	0

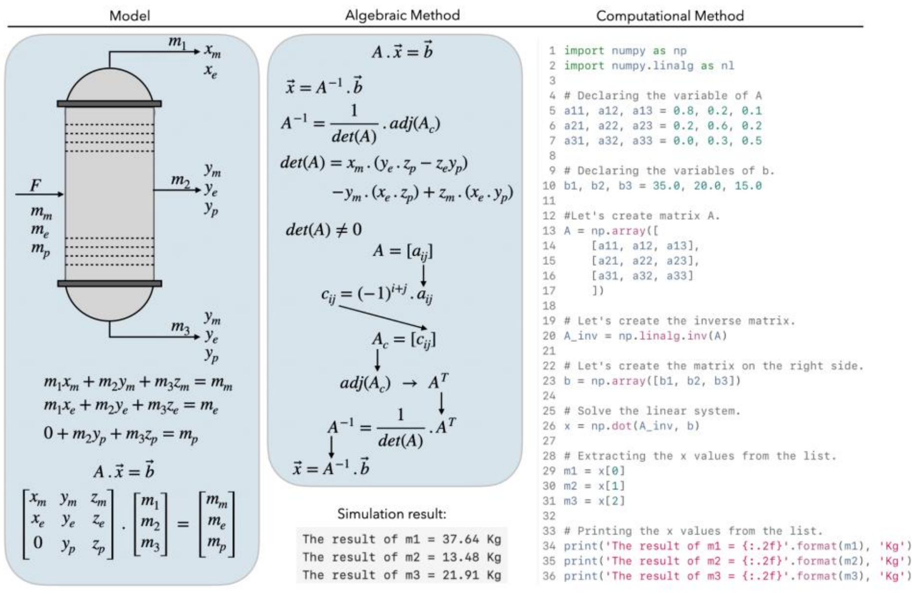
(a)

Source: EDWARDS ER, et al., 2025.

Application of Matrix Methods in Chemical Engineering: A Mass Balance Case.

Matrix algebraic methods are fundamental tools for solving mass balance problems in separation equipment, being especially useful for rearranging systems of equations in matrix form $A.\vec{x} = \vec{b}$ (Hinsen K, 2000). The figure 13 illustrates a separator model with an inlet stream, F, containing the flows of methylene, ethylene and propylene with three corresponding outlet streams, represented by the unknown masses m_1, m_2 and m_3 of each component. The algebraic resolution method adopted is based on the condition that the determinant of the matrix A is non-zero ($det(A) \neq 0$), guaranteeing the existence of the inverse matrix (Chapra SC and Canale RP, 2016). The cofactor matrix A_c, previously detailed in this work, is used to determine the adjoint matrix $adj(A)$, which, in turn, allows calculating the transposed matrix A^T, which is then obtained through the relation $A^{-1} = adj\,(A)/det(A)$, and the solution of the system is found by multiplying $x = A^{-1}.b$ (Chapra SC and Canale RP, 2016). Although it is possible to solve the system algebraically, manual implementation of these calculations results in an extensive and impractical process for full inclusion in a scientific article, due to the high number of steps and lines of code generated. To overcome this limitation, the NumPy library was used, which allows the efficient determination of the unknowns m_1, m_2 and m_3 associated with the masses of the components at the separator output. This approach combines numerical precision with computational efficiency, aligning with best practices for chemical engineering applications.

Figure 13 – Application of matrix methods in the separation process.


Model
Algebraic Method
Computational Method
F
m_m
m_e
m_p
m_1
x_m
x_e
m_2
y_m
y_e
y_p
m_3
y_m
y_e
y_p
$m_1 x_m + m_2 y_m + m_3 z_m = m_m$
$m_1 x_e + m_2 y_e + m_3 z_e = m_e$
$0 + m_2 y_p + m_3 z_p = m_p$
$A.\vec{x} = \vec{b}$
$\begin{bmatrix} x_m & y_m & z_m \\ x_e & y_e & z_e \\ 0 & y_p & z_p \end{bmatrix} . \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} m_m \\ m_e \\ m_p \end{bmatrix}$
$A.\vec{x} = \vec{b}$
$\vec{x} = A^{-1}.\vec{b}$
$A^{-1} = \frac{1}{det(A)}.adj(A_c)$
$det(A) = x_m.(y_e.z_p - z_e y_p) - y_m.(x_e.z_p) + z_m.(x_e.y_p)$
$det(A) \neq 0$
$A = [a_{ij}]$
$c_{ij} = (-1)^{i+j}.a_{ij}$
$A_c = [c_{ij}]$
$adj(A_c) \rightarrow A^T$
$A^{-1} = \frac{1}{det(A)}.A^T$
$\vec{x} = A^{-1}.\vec{b}$
Simulation result:

```
The result of m1 = 37.64 Kg
The result of m2 = 13.48 Kg
The result of m3 = 21.91 Kg
```

```python
import numpy as np
import numpy.linalg as nl

# Declaring the variable of A
a11, a12, a13 = 0.8, 0.2, 0.1
a21, a22, a23 = 0.2, 0.6, 0.2
a31, a32, a33 = 0.0, 0.3, 0.5

# Declaring the variables of b.
b1, b2, b3 = 35.0, 20.0, 15.0

#Let's create matrix A.
A = np.array([
    [a11, a12, a13],
    [a21, a22, a23],
    [a31, a32, a33]
    ])

# Let's create the inverse matrix.
A_inv = np.linalg.inv(A)

# Let's create the matrix on the right side.
b = np.array([b1, b2, b3])

# Solve the linear system.
x = np.dot(A_inv, b)

# Extracting the x values from the list.
m1 = x[0]
m2 = x[1]
m3 = x[2]

# Printing the x values from the list.
print('The result of m1 = {:.2f}'.format(m1), 'Kg')
print('The result of m2 = {:.2f}'.format(m2), 'Kg')
print('The result of m3 = {:.2f}'.format(m3), 'Kg')
```

Source: EDWARDS ER, et al., 2025.

The results presented in this paper highlight the importance of linear algebra in solving Chemical Engineering problems. Matrix and algebraic methods are indispensable tools in scientific computing, as they allow their application in computational methods that are used in several scientific areas. The examples presented in this paper range from the resolution of simple linear systems to more complex cases such as the resolution of sparse and triangular matrices, demonstrating that the correct application of this technique provides a safer and more efficient approach in solving complex problems. Matrix methods applied to solving Chemical Engineering problems have proven particularly useful for mass balance analysis, chemical process simulations, and modeling of interconnected reactor systems, such as tank and reactor networks. In addition, the application of techniques for solving sparse and triangular matrices in large-scale problems revealed a significant reduction in computational cost, reinforcing the importance of optimized representations in systems with large dimensions. Matrix methods play a fundamental role in emerging areas such as AI and ANN, mainly in the manipulation of weights and data in computational models. The algebraic basis of these techniques allows the development of algorithms capable of handling large volumes of data and simultaneous calculations, expanding the scope of scientific computing to solve problems previously considered intractable. The use of multiplication, addition, and matrix activation functions in ANNs highlights how a deep understanding of algebraic methods is essential for advances in this area. Therefore, the adequate understanding and correct application of matrix and algebraic methods not only allow the resolution of specific problems in several areas, but also provide researchers, engineers, and other professionals with the ability to understand the intrinsic relationship between algebra and scientific computing. This understanding is essential to translate natural and social phenomena into accurate mathematical and computational models, promoting innovative and sustainable solutions to current challenges. Therefore, this work highlights that scientific computing, based on matrix and algebraic methods, is an indispensable tool for scientific and technological advancement. Its application transcends disciplinary boundaries, consolidating itself as a pillar for innovation in areas ranging from traditional engineering to exact, biological, and social sciences.

CONCLUSION

Computational advances have enabled researchers and engineers to address complex problems through the application of mathematical methods, particularly algebraic techniques designed to solve these challenges. The integration of computational tools with mathematical techniques has facilitated effective solutions to problems involving matrices, streamlining analysis and calculation processes. This study presents the main techniques for the computational resolution of matrices using algebraic methods, emphasizing efficient and optimized approaches. Additionally, a computational script was developed and detailed to solve matrix systems of any dimension, based on the definition of matrix order and the elements stored in each cell. Practical applications of these techniques were subsequently explored in Chemical Engineering problems, demonstrating their effectiveness in modeling and solving complex systems.

The results obtained indicate that the techniques described in this study can be employed by professionals from various fields of Engineering to address problems involving matrix methods, supported by algebraic and computational tools. This work, therefore, provides readers with a comprehensive perspective, from theoretical foundations to practical implementation, fostering the adoption of computational approaches in solving complex Engineering challenges.

REFERÊNCIAS

ANTON H, RORRES C. Elementary Linear Algebra: Applications Version. Wiley, Hoboken. 2013; 120–135.

BAYARAM S, et al. Sensor fingerprint identification through composite fingerprints and group testing. IEEE Transactions on information forensics and security, 2014; 10(3): 597–612.

BURDEN RL, FAIRES JD. Numerical Analysis. Brooks/Cole, Boston, 2011; 80–85200205.

CICHOCKI A. et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. Foundations and Trends® in Machine Learning, 2016; 9(4-5): 249–429.

CHAPRA SC, CANALE R.P. Métodos Numéricos Para Engenharia, 7a ed. McGraw Hill Brasil, São Paulo, Brasil, 2016.

DE PAOLI, et al. Poli(tereftalato de etileno), pet: uma revisão sobre os processos de síntese, mecanismos de degradação e sua reciclagem. Polímeros, 2009; 19(2): 73–84.

DUFF IS, at al. Direct Methods for Sparse Matrices, 1a ed. Oxford University Press, Oxford, Reino Unido, 2017; 103–145.

GILAT A, SUBRAMANIAM V. Métodos Numéricos Para Engenheiros e Cientistas: Uma Introdução Com Aplicações em MATLAB. Bookman Companhia Ed, São Paulo, 2000; 480.

GOLUB GH, LOAN CFV. Matrix Computations, 4th edn. Johns Hopkins University Press, Baltimore, 2013; 78–1128185200205.

GOMES C. et al. Farnsworth, A., Fern, A., Fern, X., et al.: Computational sustainability: Computing for a better world and a sustainable future. Communications of the ACM62(9), 2019; 56–65.

GURURAJ HL, et al. A comprehensive review of face recognition techniques, trends, and challenges. IEEE Access12. 2024; 107903–107926.

HINSEN K. The molecular modeling toolkit: A new approach to molecular simulations. Journal of Computational Chemistry, 2000; 21(2): 79–85.

JONES E, et al. Scipy: Open source scientific tools for python. SciPy Library Documentation, 2001.

KARIRI E, et al. Exploring the advancements and future research directions of artificial neural networks: a text mining approach. Applied Sciences. 2023; 13(5): 3186.

KUMAR T, CHAUHAN S. Image cryptography with matrix array symmetric key using chaos based approach. International Journal of Computer Network and Information Security, 2018; 4(3): 60.

KUMAR P, et al. Encryption algorithm using matrix manipulation. In: 2022 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES), Greater Noida, India, 2022: 125–129.

LEI J, et al. Toward matrix multiplication for deep learning inference on the xilinx versal. In: 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, Naples, Italy, 2023; 42: 227–234.

LEACH AR. Molecular Modelling: Principles and Applications. Pearson Education, Harlow, UK, 2001.

LIU, Z, et al. Neural networks with trainable matrix activation functions. Journal of Machine Learning for Modeling and Computing, 2025; 6(2): 1–11.

MEYER CD. Matrix Analysis and Applied Linear Algebra. SIAM, Philadelphia, 2000; 150–155.

MATHEWS JH, FINK KD. Numerical Methods for Mathematics, Science, and Engineering, 4th edn. Prentice-Hall, Upper Saddle River, New Jersey, EUA, 2012; 201–239.

OLIPHANT TE. A Guide to NumPy. Kluwer, USA 2006; 35–3845485055.

PAUL A, et al. Matrix based cryptographic procedure for efficient image encryption. Recent Advances in Intelligent Computational Systems. IEEE, Trivandrum, Índia, 2011; 227–234.

PARR T, HOWARD J. The matrix calculus you need for deep learning. Journal arXiv preprint arXiv:1802.01528, 2018; 56 (v3)

ROSA AC, et al. Uma aplicação de matrizes em grafos químicos. Revista de Ciências Exatas e da Terra, 2023.

SAAD Y. Interactive Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia 2003; 34–60.

SILVA LL, et al. Sparse matrices for transient simulations with computing memory reduction. Electric Power Systems Research, 2020; 183: 106266.

STROHER GR, et al. Avaliação do formato de armazenamento compressed sparse row para resolução de sistemas de equações linear esparsos. Revista Brasileira de Computação Aplicada, 2021; 13(2): 73–84.

SUN R, et al. A.: Neuralmatrix: Compute the entire neural networks with linear matrix operations for efficient inference. arXiv preprint, Journal arXiv:2305, 2024: 14405 (v4)

SURVEY SCAI. A Guide to NumPy, McGraw-Hill, New Yorkpp. 2018; 324–326328330330335.

XU DX et al. Industry 4.0: state of the art and future trends. International journal of production research, (2018); 56(8): 2941–2962.

ZAWADZKI P, ŻYWICKI, K. Smart product design and production control for effective mass customization in the industry 4.0 concept. Management and production engineering review, 2016; 78:105–112.