Tarea9_LeninAmangandi

January 7, 2026

1 Escuela Politécnica Nacional

1.1 Métodos Numéricos

Nombre: Lenin Amangandi

Tema: Ejercicios Unidad 04-A-B | Eliminación gaussiana vs Gauss-Jordan

Enlace GitHub Tarea 9

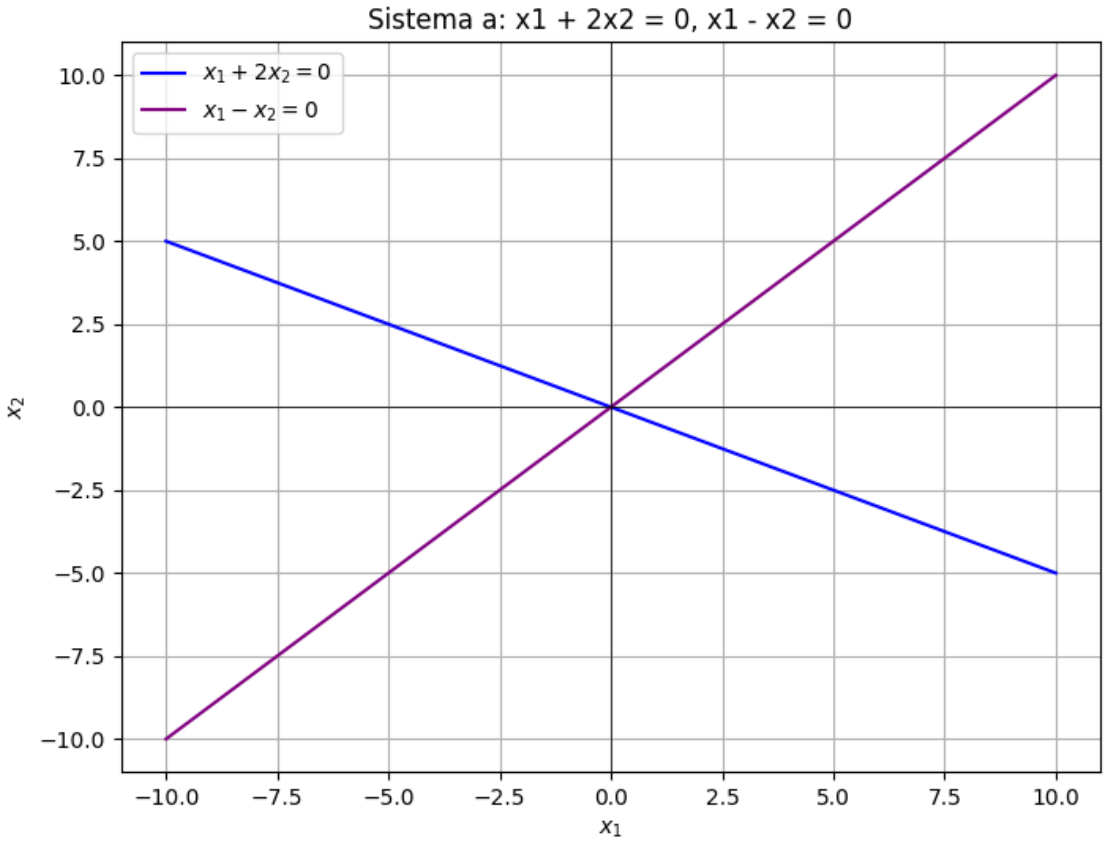
1.1.1 Ejercicio 1

Para cada uno de los siguientes sistemas lineales, obtenga, de ser posible, una solución con métodos gráficos. Explique los resultados desde un punto de vista geométrico.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

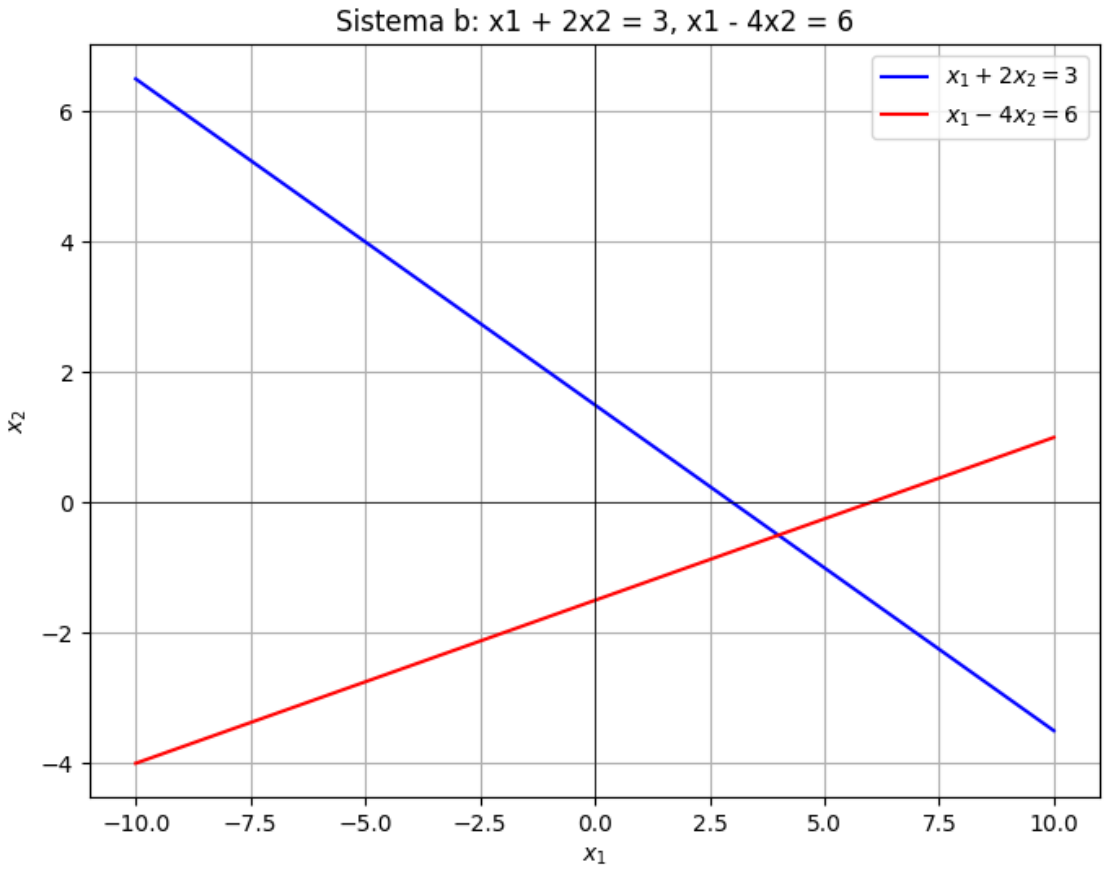
[3]:
```
# Sistema a: x1 + 2x2 = 0, x1 - x2 = 0
# Ecuaciones
# x1 + 2*x2 = 0
# x1 - x2 = 0

# Definir el rango de valores de x1
x1 = np.linspace(-10, 10, 400)

# Ecuación 1: x1 + 2*x2 = 0 -> x2 = -x1/2
x2_1 = -x1/2

# Ecuación 2: x1 - x2 = 0 -> x2 = x1
x2_2 = x1

# Graficar las rectas
plt.figure(figsize=(8, 6))
plt.plot(x1, x2_1, label=r'$x_1 + 2x_2 = 0$', color='blue')
plt.plot(x1, x2_2, label=r'$x_1 - x_2 = 0$', color='purple')

# Etiquetas y leyenda
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
```

```
plt.title('Sistema a: x1 + 2x2 = 0, x1 - x2 = 0')
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True)
plt.legend()

# Mostrar la gráfica
plt.show()
```



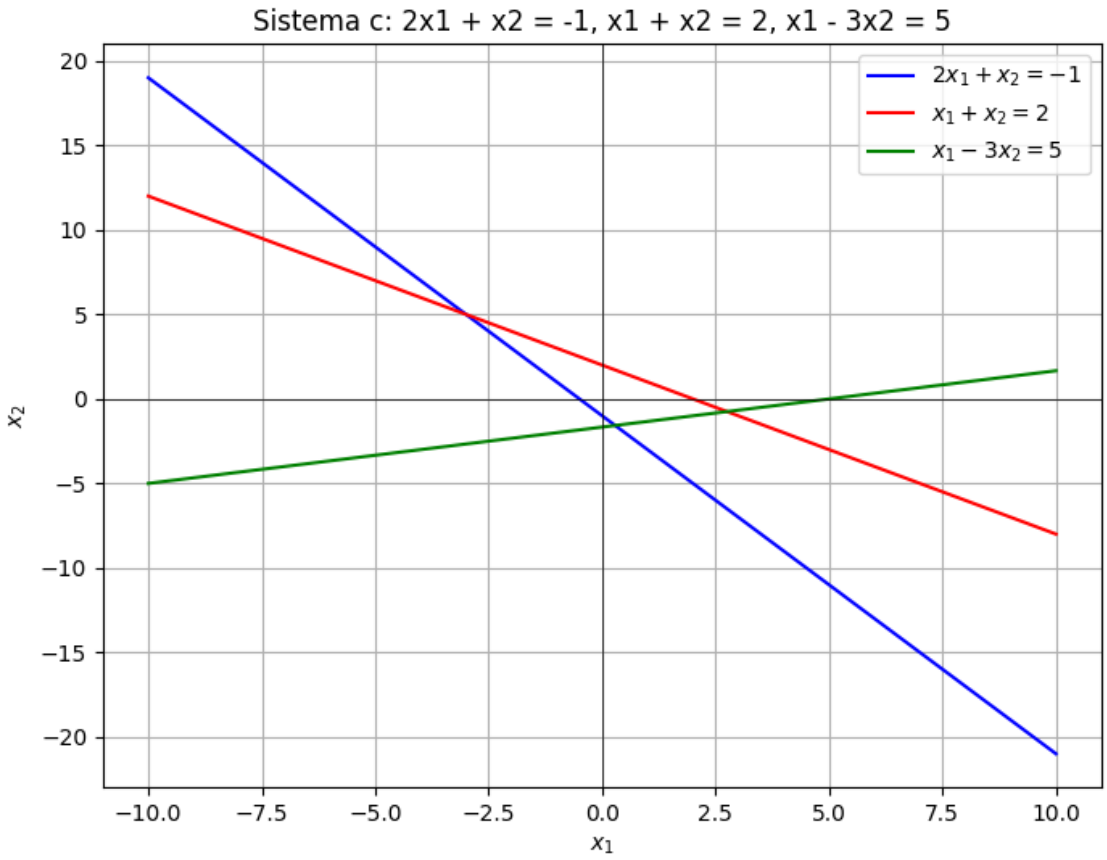
[4]:
```
# Sistema b: x1 + 2*x2 = 3, x1 - 4*x2 = 6
# Ecuaciones
# x1 + 2*x2 = 3
# x1 - 4*x2 = 6

# Ecuación 1: x1 + 2*x2 = 3 -> x2 = (3 - x1)/2
x2_1_b = (3 - x1) / 2

# Ecuación 2: x1 - 4*x2 = 6 -> x2 = (x1 - 6)/4
x2_2_b = (x1 - 6) / 4
```

```
# Graficar las rectas
plt.figure(figsize=(8, 6))
plt.plot(x1, x2_1_b, label=r'$x_1 + 2x_2 = 3$', color='blue')
plt.plot(x1, x2_2_b, label=r'$x_1 - 4x_2 = 6$', color='red')

# Etiquetas y leyenda
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title('Sistema b: x1 + 2x2 = 3, x1 - 4x2 = 6')
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True)
plt.legend()

# Mostrar la gráfica
plt.show()
```



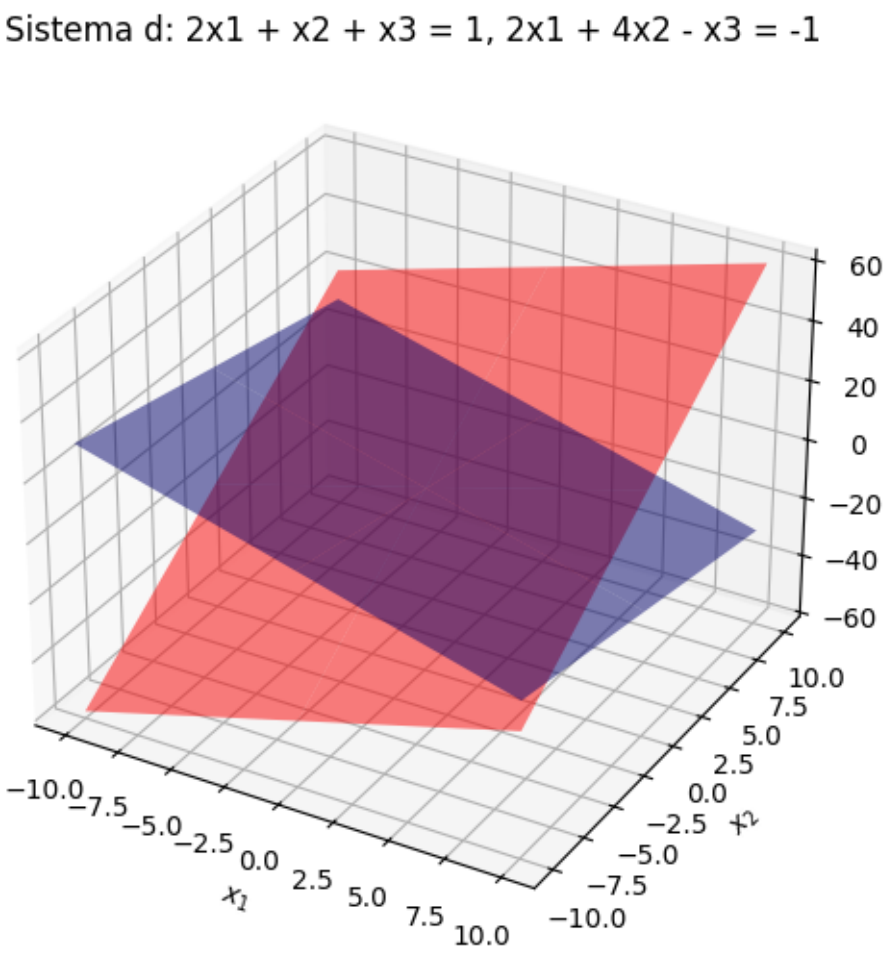
[5]:
```
# Sistema c: 2x1 + x2 = -1, x1 + x2 = 2, x1 - 3x2 = 5
# Ecuaciones
```

```
# 2*x1 + x2 = -1 -> x2 = -2*x1 - 1
x2_1_c = -2*x1 - 1

# x1 + x2 = 2 -> x2 = 2 - x1
x2_2_c = 2 - x1

# x1 - 3*x2 = 5 -> x2 = (x1 - 5)/3
x2_3_c = (x1 - 5) / 3

# Graficar las rectas
plt.figure(figsize=(8, 6))
plt.plot(x1, x2_1_c, label=r'$2x_1 + x_2 = -1$', color='blue')
plt.plot(x1, x2_2_c, label=r'$x_1 + x_2 = 2$', color='red')
plt.plot(x1, x2_3_c, label=r'$x_1 - 3x_2 = 5$', color='green')

# Etiquetas y leyenda
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title('Sistema c: 2x1 + x2 = -1, x1 + x2 = 2, x1 - 3x2 = 5')
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True)
plt.legend()

# Mostrar la gráfica
plt.show()
```



[6]:
```
# Sistema d: 2x1 + x2 + x3 = 1, 2x1 + 4x2 - x3 = -1
# Graficar en 3D

from mpl_toolkits.mplot3d import Axes3D

x1_4 = np.linspace(-10, 10, 200)
x2_4 = np.linspace(-10, 10, 200)
X1, X2 = np.meshgrid(x1_4, x2_4)

# Ecuaciones en 3D
x3_1_d = 1 - 2*X1 - X2
x3_2_d = 2*X1 + 4*X2 + 1

# Graficar
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X1, X2, x3_1_d, color='blue', alpha=0.5, rstride=100,␣
 ↪cstride=100)
```

```
ax.plot_surface(X1, X2, x3_2_d, color='red', alpha=0.5, rstride=100,␣
 ↪cstride=100)

ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
ax.set_zlabel(r'$x_3$')
ax.set_title('Sistema d: 2x1 + x2 + x3 = 1, 2x1 + 4x2 - x3 = -1')

plt.show()
```



1.1.2 Ejercicio 2

[24]:
```
import numpy as np

def gauss_elimination(A, b):
    n = len(b)
    # Elimination phase
```

```
    for i in range(n):
        # Pivoting: swap rows if necessary
        if A[i,i] == 0:
            for j in range(i+1, n):
                if A[j,i] != 0:
                    A[[i,j]] = A[[j,i]]
                    b[i], b[j] = b[j], b[i]
                    break
        for j in range(i+1, n):
            if A[j,i] != 0:
                factor = A[j,i] / A[i,i]
                A[j] = A[j] - factor * A[i]
                b[j] = b[j] - factor * b[i]

    # Back substitution phase
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - np.dot(A[i,i+1:], x[i+1:])) / A[i,i]

    return x

A_a = np.array([[-1, 4, 1], [5/3, 2/3, 2/3], [2, 1, 4]], dtype=float)
b_a = np.array([8, 1, 11], dtype=float)

A_b = np.array([[4, 2, -1], [1/9, 1/9, -1/3], [1, 4, 2]], dtype=float)
b_b = np.array([-5, -1, 9], dtype=float)

x_a = gauss_elimination(A_a, b_a)
x_b = gauss_elimination(A_b, b_b)

print("Solución del sistema a:", np.round(x_a, 2))
print("Solución del sistema b:", np.round(x_b, 2))
```

```
Solución del sistema a: [-1.  1.  3.]
Solución del sistema b: [-1.  1.  3.]
```

1.1.3 Ejercicio 3

[11]:
```
def solve_gauss_with_pivoting(A, b):
    """
    Resuelve un sistema de ecuaciones Ax=b usando eliminación gaussiana
    con pivoteo parcial para determinar si se requieren intercambios de fila.

    Args:
        A (list of lists): Matriz de coeficientes.
        b (list): Vector de términos constantes.
```

```
    Returns:
        tuple: Una tupla conteniendo la solución (np.array) y un booleano
               que es True si se realizaron intercambios de fila.
               Devuelve (None, bool) si no hay solución única.
    """
    # --- 1. Configuración Inicial ---
    # Convertimos a arrays de numpy para operaciones vectoriales
    A = np.array(A, dtype=float)
    b = np.array(b, dtype=float).reshape(-1, 1)
    n = len(b)

    # Creamos la matriz aumentada [A|b]
    M = np.hstack([A, b])
    print("--- Matriz Aumentada Original ---")
    print(M)
    print("-" * 50)

    swaps_performed = False

    # --- 2. Eliminación Gaussiana con Pivoteo Parcial ---
    print("--- Iniciando Eliminación Gaussiana ---")
    for i in range(n):
        # a) Búsqueda del pivote (pivoteo parcial)
        # Buscamos el máximo valor absoluto en la columna actual (desde la fila i hacia abajo)
        pivot_row_index = i
        for j in range(i + 1, n):
            if abs(M[j, i]) > abs(M[pivot_row_index, i]):
                pivot_row_index = j

        # b) Intercambio de filas si es necesario
        if pivot_row_index != i:
            print(f"-> Pivoteo requerido en columna {i+1}: Intercambiando fila {i+1} con fila {pivot_row_index+1}.")
            # Intercambiamos las filas completas
            M[[i, pivot_row_index]] = M[[pivot_row_index, i]]
            swaps_performed = True
            print("   Matriz después del intercambio:\n", M)

        # c) Verificación de singularidad
        # Si después del pivoteo el elemento en la diagonal sigue siendo cero, la matriz es singular.
        if M[i, i] == 0:
            print("\n! La matriz es singular (no tiene solución única).")
            return None, swaps_performed

        # d) Eliminación
```

```
        # Hacemos cero los elementos debajo del pivote
        for j in range(i + 1, n):
            multiplier = M[j, i] / M[i, i]
            M[j, :] -= multiplier * M[i, :]

        print(f"\nMatriz después de la eliminación en la columna {i+1}:\n", M)

    print("-" * 50)
    print("--- Matriz Triangular Superior Final ---")
    print(M)
    print("-" * 50)

    # --- 3. Sustitución hacia atrás ---
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        sum_ax = np.dot(M[i, i+1:n], x[i+1:n])
        x[i] = (M[i, n] - sum_ax) / M[i, i]

    # --- 4. Resultados ---
    print("--- Resultados ---")
    if swaps_performed:
        print("-> Se necesitaron intercambios de fila durante el proceso.")
    else:
        print("-> No se necesitaron intercambios de fila.")

    print("\nSolución Calculada:", x)
    return x, swaps_performed
```

[15]:
```
A_a = [[1, -1, 3],
       [3, -3, 1],
       [1,  1, 0]]

# Vector de términos constantes
b_a = [2, -1, 3]

print("****** RESOLVIENDO SISTEMA 'a' CON PIVOTEO ******")
solution_a, swaps_needed_a = solve_gauss_with_pivoting(A=A_a, b=b_a)
```

```
****** RESOLVIENDO SISTEMA 'a' CON PIVOTEO ******
--- Matriz Aumentada Original ---
[[ 1. -1.  3.  2.]
 [ 3. -3.  1. -1.]
 [ 1.  1.  0.  3.]]
--------------------------------------------------
--- Iniciando Eliminación Gaussiana ---
-> Pivoteo requerido en columna 1: Intercambiando fila 1 con fila 2.
   Matriz después del intercambio:
 [[ 3. -3.  1. -1.]
```

```
 [ 1. -1.  3.  2.]
 [ 1.  1.  0.  3.]]

Matriz después de la eliminación en la columna 1:
 [[ 3.         -3.          1.         -1.        ]
 [ 0.          0.          2.66666667  2.33333333]
 [ 0.          2.         -0.33333333  3.33333333]]
-> Pivoteo requerido en columna 2: Intercambiando fila 2 con fila 3.
   Matriz después del intercambio:
 [[ 3.         -3.          1.         -1.        ]
 [ 0.          2.         -0.33333333  3.33333333]
 [ 0.          0.          2.66666667  2.33333333]]

Matriz después de la eliminación en la columna 2:
 [[ 3.         -3.          1.         -1.        ]
 [ 0.          2.         -0.33333333  3.33333333]
 [ 0.          0.          2.66666667  2.33333333]]

Matriz después de la eliminación en la columna 3:
 [[ 3.         -3.          1.         -1.        ]
 [ 0.          2.         -0.33333333  3.33333333]
 [ 0.          0.          2.66666667  2.33333333]]
--------------------------------------------------
--- Matriz Triangular Superior Final ---
[[ 3.         -3.          1.         -1.        ]
 [ 0.          2.         -0.33333333  3.33333333]
 [ 0.          0.          2.66666667  2.33333333]]
--------------------------------------------------
--- Resultados ---
-> Se necesitaron intercambios de fila durante el proceso.

Solución Calculada: [1.1875 1.8125 0.875 ]
```

[16]:
```
A_b = [[2, -1.5, 3],
       [-1, 0, 2],
       [4, -4.5, 5]]
B_b = [1, 3, 1]

solve_gauss_with_pivoting(A_b, B_b)
```

```
--- Matriz Aumentada Original ---
[[ 2.  -1.5  3.   1. ]
 [-1.   0.   2.   3. ]
 [ 4.  -4.5  5.   1. ]]
--------------------------------------------------
--- Iniciando Eliminación Gaussiana ---
-> Pivoteo requerido en columna 1: Intercambiando fila 1 con fila 3.
   Matriz después del intercambio:
```

```
 [[ 4.  -4.5  5.   1. ]
 [-1.   0.   2.   3. ]
 [ 2.  -1.5  3.   1. ]]

Matriz después de la eliminación en la columna 1:
 [[ 4.    -4.5    5.     1.   ]
 [ 0.    -1.125  3.25   3.25 ]
 [ 0.     0.75   0.5    0.5  ]]

Matriz después de la eliminación en la columna 2:
 [[ 4.         -4.5         5.          1.        ]
 [ 0.         -1.125       3.25        3.25      ]
 [ 0.          0.          2.66666667  2.66666667]]

Matriz después de la eliminación en la columna 3:
 [[ 4.         -4.5         5.          1.        ]
 [ 0.         -1.125       3.25        3.25      ]
 [ 0.          0.          2.66666667  2.66666667]]
--------------------------------------------------
--- Matriz Triangular Superior Final ---
[[ 4.         -4.5         5.          1.        ]
 [ 0.         -1.125       3.25        3.25      ]
 [ 0.          0.          2.66666667  2.66666667]]
--------------------------------------------------
--- Resultados ---
-> Se necesitaron intercambios de fila durante el proceso.

Solución Calculada: [-1. -0.  1.]
```

[16]:
```
(array([-1., -0.,  1.]), True)
```

[17]:
```
A_c = [[2, 0, 0, 0],
       [1, 1.5, 0, 0],
        [0, -3, 0.5, 0],
         [2, -2, 1, 1]]

B_c = [3, 4.5, -6.6, 0.8]

solve_gauss_with_pivoting(A_c, B_c)
```

```
--- Matriz Aumentada Original ---
[[ 2.   0.   0.   0.   3. ]
 [ 1.   1.5  0.   0.   4.5]
 [ 0.  -3.   0.5  0.  -6.6]
 [ 2.  -2.   1.   1.   0.8]]
--------------------------------------------------
--- Iniciando Eliminación Gaussiana ---
```

```
Matriz después de la eliminación en la columna 1:
 [[ 2.   0.   0.   0.   3. ]
 [ 0.   1.5  0.   0.   3. ]
 [ 0.  -3.   0.5  0.  -6.6]
 [ 0.  -2.   1.   1.  -2.2]]
-> Pivoteo requerido en columna 2: Intercambiando fila 2 con fila 3.
   Matriz después del intercambio:
 [[ 2.   0.   0.   0.   3. ]
 [ 0.  -3.   0.5  0.  -6.6]
 [ 0.   1.5  0.   0.   3. ]
 [ 0.  -2.   1.   1.  -2.2]]

Matriz después de la eliminación en la columna 2:
 [[ 2.          0.          0.          0.          3.        ]
 [ 0.         -3.          0.5         0.         -6.6       ]
 [ 0.          0.          0.25        0.         -0.3       ]
 [ 0.          0.          0.66666667  1.          2.2       ]]
-> Pivoteo requerido en columna 3: Intercambiando fila 3 con fila 4.
   Matriz después del intercambio:
 [[ 2.          0.          0.          0.          3.        ]
 [ 0.         -3.          0.5         0.         -6.6       ]
 [ 0.          0.          0.66666667  1.          2.2       ]
 [ 0.          0.          0.25        0.         -0.3       ]]

Matriz después de la eliminación en la columna 3:
 [[ 2.          0.          0.          0.          3.        ]
 [ 0.         -3.          0.5         0.         -6.6       ]
 [ 0.          0.          0.66666667  1.          2.2       ]
 [ 0.          0.          0.         -0.375      -1.125     ]]

Matriz después de la eliminación en la columna 4:
 [[ 2.          0.          0.          0.          3.        ]
 [ 0.         -3.          0.5         0.         -6.6       ]
 [ 0.          0.          0.66666667  1.          2.2       ]
 [ 0.          0.          0.         -0.375      -1.125     ]]
------------------------------------------------
--- Matriz Triangular Superior Final ---
[[ 2.          0.          0.          0.          3.        ]
 [ 0.         -3.          0.5         0.         -6.6       ]
 [ 0.          0.          0.66666667  1.          2.2       ]
 [ 0.          0.          0.         -0.375      -1.125     ]]
------------------------------------------------
--- Resultados ---
-> Se necesitaron intercambios de fila durante el proceso.

Solución Calculada: [ 1.5  2.  -1.2  3. ]
```

[17]:
```
(array([ 1.5,  2. , -1.2,  3. ]), True)
```

[18]:
```
A_d = [[1, 1, 0, 1],
       [2, 1, -1, 1],
       [4, -1, -2, 2],
       [3, -1, -1, 2]]

B_d = [2, 1, 0, -3]

solve_gauss_with_pivoting(A_d, B_d)
```

```
--- Matriz Aumentada Original ---
[[ 1.  1.  0.  1.  2.]
 [ 2.  1. -1.  1.  1.]
 [ 4. -1. -2.  2.  0.]
 [ 3. -1. -1.  2. -3.]]
--------------------------------------------------
--- Iniciando Eliminación Gaussiana ---
-> Pivoteo requerido en columna 1: Intercambiando fila 1 con fila 3.
   Matriz después del intercambio:
 [[ 4. -1. -2.  2.  0.]
 [ 2.  1. -1.  1.  1.]
 [ 1.  1.  0.  1.  2.]
 [ 3. -1. -1.  2. -3.]]

Matriz después de la eliminación en la columna 1:
 [[ 4.   -1.   -2.    2.    0.  ]
 [ 0.    1.5   0.    0.    1.  ]
 [ 0.    1.25  0.5   0.5   2.  ]
 [ 0.   -0.25  0.5   0.5  -3.  ]]

Matriz después de la eliminación en la columna 2:
 [[ 4.         -1.         -2.          2.          0.        ]
 [ 0.          1.5         0.          0.          1.        ]
 [ 0.          0.          0.5         0.5         1.16666667]
 [ 0.          0.          0.5         0.5        -2.83333333]]

Matriz después de la eliminación en la columna 3:
 [[ 4.         -1.         -2.          2.          0.        ]
 [ 0.          1.5         0.          0.          1.        ]
 [ 0.          0.          0.5         0.5         1.16666667]
 [ 0.          0.          0.          0.         -4.        ]]

! La matriz es singular (no tiene solución única).
```

[18]:
```
(None, True)
```

1.1.4 Ejercicio 4

[19]:
```
def solve_gauss_32bit(A, b):
    """
    Resuelve un sistema Ax=b usando eliminación gaussiana con pivoteo parcial
    y forzando el uso de aritmética de precisión de 32 bits (float32).

    Args:
        A (list of lists): Matriz de coeficientes.
        b (list): Vector de términos constantes.
    """
    # --- 1. Configuración Inicial con Precisión de 32 bits ---
    # La clave es dtype=np.float32. NumPy forzará a que todos los
    # números y cálculos se mantengan en esta precisión.
    try:
        A_32 = np.array(A, dtype=np.float32)
        b_32 = np.array(b, dtype=np.float32).reshape(-1, 1)
        n = len(b_32)
    except ValueError:
        print("Error: Asegúrate de que todas las filas de la matriz A tengan el␣
↪mismo número de columnas.")
        return None

    # La matriz aumentada se crea y se mantiene en 32 bits.
    M = np.hstack([A_32, b_32])
    print("--- Matriz Aumentada Original (convertida a precisión 32-bit) ---")
    print(M)
    print("-" * 60)

    # --- 2. Eliminación Gaussiana con Pivoteo Parcial ---
    print("--- Iniciando Eliminación Gaussiana (aritmética 32-bit) ---")
    for i in range(n):
        # Pivoteo parcial para estabilidad numérica
        pivot_row_index = i
        # Buscamos la fila con el mayor pivote en la columna actual
        for j in range(i + 1, n):
            if abs(M[j, i]) > abs(M[pivot_row_index, i]):
                pivot_row_index = j

        # Si encontramos un mejor pivote, intercambiamos filas
        if pivot_row_index != i:
            M[[i, pivot_row_index]] = M[[pivot_row_index, i]]
            print(f"-> Pivoteo: Fila {i+1} <-> Fila {pivot_row_index+1}")

        # Verificación de singularidad (si el mejor pivote es 0, la matriz es␣
↪singular)
        if M[i, i] == 0:
```

```
            print("\n! La matriz es singular (no tiene solución única).")
            return None

        # Eliminación: hacemos cero los elementos debajo del pivote
        for j in range(i + 1, n):
            # Todas las operaciones (división, multiplicación, resta)
            # se realizan manteniendo la precisión de 32 bits.
            multiplier = M[j, i] / M[i, i]
            M[j, :] -= multiplier * M[i, :]

    print("\n--- Matriz Triangular Superior Final ---")
    print(M)
    print("-" * 60)

    # --- 3. Sustitución hacia atrás ---
    # El vector solución también se inicializa y calcula en 32 bits
    x = np.zeros(n, dtype=np.float32)
    for i in range(n - 1, -1, -1):
        # np.dot y el resto de operaciones respetan el dtype de los arrays
        sum_ax = np.dot(M[i, i+1:n], x[i+1:n])
        x[i] = (M[i, n] - sum_ax) / M[i, i]

    # --- 4. Resultados ---
    print("--- Resultados ---")
    print("Solución Calculada (precisión 32-bit):")
    # Imprimimos con alta precisión para ver el valor decimal exacto que␣
 ↪representa el float32
    for i in range(n):
        print(f"  x{i+1} = {x[i]:.8f}")

    return x
```

[20]:
```
A_a = [[1/4, 1/5, 1/6],
       [1/3, 1/4, 1/5],
       [1/2, 1,   2]]
b_a = [9, 8, 8]
solve_gauss_32bit(A_a, b_a)
print("\n" + "="*80 + "\n")
```

```
--- Matriz Aumentada Original (convertida a precisión 32-bit) ---
[[0.25       0.2        0.16666667 9.        ]
 [0.33333334 0.25       0.2        8.        ]
 [0.5        1.         2.         8.        ]]
------------------------------------------------------------
--- Iniciando Eliminación Gaussiana (aritmética 32-bit) ---
-> Pivoteo: Fila 1 <-> Fila 3

--- Matriz Triangular Superior Final ---
```

```
[[ 0.5         1.          2.          8.        ]
 [ 0.         -0.4166667  -1.1333333   2.6666665 ]
 [ 0.          0.         -0.01733333  3.0800002 ]]
------------------------------------------------------------
--- Resultados ---
Solución Calculada (precisión 32-bit):
  x1 = -227.07696533
  x2 = 476.92321777
  x3 = -177.69236755

================================================================================
```

[21]:
```
A_b = [[3.333, 15920, -10.333],
       [2.222, 16.71, 9.612],
       [1.5611, 5.1791, 1.6852]]
b_b = [15913, 28.544, 8.4254]
solve_gauss_32bit(A_b, b_b)
print("\n" + "="*80 + "\n")
```

```
--- Matriz Aumentada Original (convertida a precisión 32-bit) ---
[[ 3.3330e+00  1.5920e+04 -1.0333e+01  1.5913e+04]
 [ 2.2220e+00  1.6710e+01  9.6120e+00  2.8544e+01]
 [ 1.5611e+00  5.1791e+00  1.6852e+00  8.4254e+00]]
------------------------------------------------------------
--- Iniciando Eliminación Gaussiana (aritmética 32-bit) ---

--- Matriz Triangular Superior Final ---
[[ 3.3329999e+00  1.5920000e+04 -1.0333000e+01  1.5913000e+04]
 [ 0.0000000e+00 -1.0596623e+04  1.6500668e+01 -1.0580122e+04]
 [ 0.0000000e+00  0.0000000e+00 -5.0780745e+00 -5.0786133e+00]]
------------------------------------------------------------
--- Resultados ---
Solución Calculada (precisión 32-bit):
  x1 = 0.99970937
  x2 = 1.00000012
  x3 = 1.00010610

================================================================================
```

[22]:
```
A_d = [[2,  1, -1,  1, -3],
       [1,  0,  2, -1,  1],
       [0, -2, -1,  1, -1],
       [3,  1, -4,  0,  5],
       [1, -1, -1, -1,  1]]
b_d = [7, 2, -5, 6, -3]
solve_gauss_32bit(A_d, b_d)
```

```
print("\n" + "="*80 + "\n")
```

```
--- Matriz Aumentada Original (convertida a precisión 32-bit) ---
[[ 2.  1. -1.  1. -3.  7.]
 [ 1.  0.  2. -1.  1.  2.]
 [ 0. -2. -1.  1. -1. -5.]
 [ 3.  1. -4.  0.  5.  6.]
 [ 1. -1. -1. -1.  1. -3.]]
------------------------------------------------------------
--- Iniciando Eliminación Gaussiana (aritmética 32-bit) ---
-> Pivoteo: Fila 1 <-> Fila 4
-> Pivoteo: Fila 2 <-> Fila 3

--- Matriz Triangular Superior Final ---
[[ 3.          1.         -4.          0.          5.          6.        ]
 [ 0.         -2.         -1.          1.         -1.         -5.        ]
 [ 0.          0.          3.5000002  -1.1666666  -0.50000006  0.8333334 ]
 [ 0.          0.          0.          1.6666666  -6.285714    1.8095238 ]
 [ 0.          0.          0.          0.         -4.885715   -0.45714247]]
------------------------------------------------------------
--- Resultados ---
Solución Calculada (precisión 32-bit):
  x1 = 1.88304090
  x2 = 2.80701756
  x3 = 0.73099399
  x4 = 1.43859613
  x5 = 0.09356716

================================================================================
```

1.1.5 Ejercicio 5

[23]:
```
import sympy

def analyze_system_with_parameter(A_symbolic, b_symbolic, symbol):
    """
    Analiza un sistema de ecuaciones lineales Ax=b con un parámetro simbólico.

    Args:
        A_symbolic (sympy.Matrix): Matriz de coeficientes con el símbolo.
        b_symbolic (sympy.Matrix): Vector de constantes con el símbolo.
        symbol (sympy.Symbol): El símbolo utilizado en las matrices.
    """
    if not A_symbolic.is_square:
        print("Error: La matriz de coeficientes debe ser cuadrada.")
        return
```

```
    # --- ANÁLISIS DE LA MATRIZ Y SU DETERMINANTE ---
    print("--- Análisis del Determinante ---")

    # 1. Calcular el determinante en términos del símbolo
    det_A = A_symbolic.det()
    print(f"La matriz de coeficientes A es:\n{A_symbolic}")
    print(f"\nEl determinante de A es: det(A) = {sympy.simplify(det_A)}")

    # 2. Encontrar los valores críticos del símbolo resolviendo det(A) = 0
    print("\nSe busca para qué valores del símbolo el determinante es cero.")
    critical_values = sympy.solve(det_A, symbol)
    print(f"El determinante es cero cuando {symbol} está en {critical_values}.")
    print("-" * 60)

    # --- ANÁLISIS DE LOS CASOS CRÍTICOS (det(A) = 0) ---
    print("--- Análisis de los Casos Críticos (No hay solución única) ---")
    for val in critical_values:
        print(f"\nAnálisis para {symbol} = {val}:")

        # Sustituir el valor crítico en la matriz aumentada
        A_sub = A_symbolic.subs(symbol, val)
        b_sub = b_symbolic.subs(symbol, val)
        augmented_matrix = A_sub.row_join(b_sub)
        print("Matriz aumentada para este valor:\n", augmented_matrix)

        # Llevar a la forma escalonada reducida por filas (RREF) para analizar␣
↪el rango
        rref_matrix, _ = augmented_matrix.rref()
        print("\nForma escalonada reducida por filas (RREF):\n", rref_matrix)

        # Verificar si hay una contradicción (fila [0, 0, ..., n] con n != 0)
        inconsistent = any(all(elem == 0 for elem in row[:-1]) and row[-1] != 0␣
↪for row in rref_matrix.tolist())

        if inconsistent:
            print(f"\nConclusión (a): Para {symbol} = {val}, el sistema es␣
↪inconsistente y NO TIENE SOLUCIONES.")
        else:
            print(f"\nConclusión (b): Para {symbol} = {val}, el sistema es␣
↪consistente y tiene INFINITAS SOLUCIONES.")
    print("-" * 60)

    # --- ANÁLISIS DEL CASO GENERAL (det(A) != 0) ---
    print("--- Análisis del Caso General (Solución única) ---")
    print(f"Existe una solución única siempre que {symbol} no sea uno de los␣
↪valores críticos {critical_values}.")
```

```
    # 3. Usar la Regla de Cramer para encontrar la solución general
    print("\nUsando la Regla de Cramer para encontrar la solución en términos␣
 ↪del símbolo:")

    solutions = []
    for i in range(A_symbolic.cols):
        # Crear la matriz Ai reemplazando la columna i por el vector b
        Ai = A_symbolic.copy()
        Ai[:, i] = b_symbolic

        det_Ai = Ai.det()
        # Simplificar la solución xi = det(Ai) / det(A)
        xi = sympy.simplify(det_Ai / det_A)
        solutions.append(xi)
        print(f"  x{i+1} = det(A{i+1}) / det(A) = ({sympy.simplify(det_Ai)}) /␣
 ↪({sympy.simplify(det_A)}) = {xi}")

    print("\nConclusión (c): La solución única para un  dado (distinto de los␣
 ↪valores críticos) es:")
    for i, sol in enumerate(solutions):
        print(f"  x{i+1} = {sol}")
# --- DATOS DE ENTRADA DEL PROBLEMA ---

# 1. Definir 'alpha' como un símbolo matemático
alpha = sympy.Symbol(' ')

# 2. Definir la matriz de coeficientes A con el símbolo
A_system = sympy.Matrix([
    [1,     -1,   alpha],
    [-1,     2,  -alpha],
    [alpha,  1,       1]
])

# 3. Definir el vector de constantes b
b_system = sympy.Matrix([-2, 3, 2])

# --- LLAMADA A LA FUNCIÓN DE ANÁLISIS ---
analyze_system_with_parameter(A_symbolic=A_system, b_symbolic=b_system,␣
 ↪symbol=alpha)
```

```
--- Análisis del Determinante ---
La matriz de coeficientes A es:
Matrix([[1, -1,  ], [-1, 2, - ], [ , 1, 1]])

El determinante de A es: det(A) = 1 -  **2
```

```
Se busca para qué valores del símbolo el determinante es cero.
El determinante es cero cuando   está en [-1, 1].
------------------------------------------------------------
--- Análisis de los Casos Críticos (No hay solución única) ---

Análisis para   = -1:
Matriz aumentada para este valor:
 Matrix([[1, -1, -1, -2], [-1, 2, 1, 3], [-1, 1, 1, 2]])

Forma escalonada reducida por filas (RREF):
 Matrix([[1, 0, -1, -1], [0, 1, 0, 1], [0, 0, 0, 0]])

Conclusión (b): Para   = -1, el sistema es consistente y tiene INFINITAS
SOLUCIONES.

Análisis para   = 1:
Matriz aumentada para este valor:
 Matrix([[1, -1, 1, -2], [-1, 2, -1, 3], [1, 1, 1, 2]])

Forma escalonada reducida por filas (RREF):
 Matrix([[1, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 1]])

Conclusión (a): Para   = 1, el sistema es inconsistente y NO TIENE SOLUCIONES.
------------------------------------------------------------
--- Análisis del Caso General (Solución única) ---
Existe una solución única siempre que   no sea uno de los valores críticos [-1,
1].

Usando la Regla de Cramer para encontrar la solución en términos del símbolo:
  x1 = det(A1) / det(A) = (-  - 1) / (1 -  **2) = 1/(  - 1)
  x2 = det(A2) / det(A) = (1 -  **2) / (1 -  **2) = 1
  x3 = det(A3) / det(A) = (  + 1) / (1 -  **2) = -1/(  - 1)

Conclusión (c): La solución única para un   dado (distinto de los valores
críticos) es:
  x1 = 1/(  - 1)
  x2 = 1
  x3 = -1/(  - 1)
```

1.1.6 Ejercicio 7

[25]:
```python
import numpy as np

A = np.array([
    [1, 2, 0, 3],
    [1, 0, 2, 2],
    [0, 0, 1, 1]
```

```
])
x = np.array([1000, 500, 350, 400])
b = np.array([3500, 2700, 900])

# Calculamos el consumo actual
consumo_actual = A @ x

print("Consumo actual de alimentos:", consumo_actual)
print("Suministro disponible:", b)

# Verificamos si hay suficiente alimento
diferencia = b - consumo_actual
suficiente = np.all(diferencia >= 0)

if suficiente:
    print("\nRespuesta a: Sí hay suficiente alimento para el consumo actual.")
    print("Excedentes por tipo de alimento:", diferencia)
else:
    print("\nRespuesta a: No hay suficiente alimento para el consumo actual.")
    print("Déficits por tipo de alimento:", -diferencia[diferencia < 0])
```

```
Consumo actual de alimentos: [3200 2500  750]
Suministro disponible: [3500 2700  900]

Respuesta a: Sí hay suficiente alimento para el consumo actual.
Excedentes por tipo de alimento: [300 200 150]
```

[27]:
```
def max_incremento(A, b, x, especie):
    # Calculamos el consumo actual
    consumo_actual = A @ x
    excedente = b - consumo_actual

    # Obtenemos la columna de coeficientes para la especie
    a_j = A[:, especie]

    # Calculamos el máximo incremento posible
    incrementos = excedente / a_j
    max_incremento = np.min(incrementos[a_j > 0])

    return max_incremento

print("\nParte b: Máximo incremento individual por especie")
for j in range(len(x)):
    inc = max_incremento(A, b, x, j)
    print(f"Especie {j+1}: puede aumentar en {inc:.2f} unidades")
```

```
Parte b: Máximo incremento individual por especie
```

```
Especie 1: puede aumentar en 200.00 unidades
Especie 2: puede aumentar en 150.00 unidades
Especie 3: puede aumentar en 100.00 unidades
Especie 4: puede aumentar en 100.00 unidades

C:\Users\User\AppData\Local\Temp\ipykernel_2428\1786683539.py:10:
RuntimeWarning: divide by zero encountered in divide
  incrementos = excedente / a_j
```

[28]:
```python
A_sin_1 = np.delete(A, 0, axis=1)
x_sin_1 = np.delete(x, 0)

# Nuevo consumo sin especie 1
consumo_sin_1 = A_sin_1 @ x_sin_1
excedente_sin_1 = b - consumo_sin_1

print("\nParte c: Extinción de especie 1")
print("Excedente disponible:", excedente_sin_1)

# Calculamos incremento posible para cada especie restante
for j in range(len(x_sin_1)):
    # Sumamos 1 porque eliminamos la columna 0 (especie 1)
    inc = max_incremento(A_sin_1, b, x_sin_1, j)
    print(f"Especie {j+2}: puede aumentar en {inc:.2f} unidades")
```

```
Parte c: Extinción de especie 1
Excedente disponible: [1300 1200  150]
Especie 2: puede aumentar en 650.00 unidades
Especie 3: puede aumentar en 150.00 unidades
Especie 4: puede aumentar en 150.00 unidades

C:\Users\User\AppData\Local\Temp\ipykernel_2428\1786683539.py:10:
RuntimeWarning: divide by zero encountered in divide
  incrementos = excedente / a_j
```

[29]:
```python
# Eliminamos la especie 2 (columna 1)
A_sin_2 = np.delete(A, 1, axis=1)
x_sin_2 = np.delete(x, 1)

# Nuevo consumo sin especie 2
consumo_sin_2 = A_sin_2 @ x_sin_2
excedente_sin_2 = b - consumo_sin_2

print("\nParte d: Extinción de especie 2")
print("Excedente disponible:", excedente_sin_2)

# Calculamos incremento posible para cada especie restante
for j in range(len(x_sin_2)):
```

```
    # Ajustamos el índice porque eliminamos la columna 1 (especie 2)
    especie_original = j if j < 1 else j + 1
    inc = max_incremento(A_sin_2, b, x_sin_2, j)
    print(f"Especie {especie_original+1}: puede aumentar en {inc:.2f} unidades")
```

```
Parte d: Extinción de especie 2
Excedente disponible: [1300  200  150]
Especie 1: puede aumentar en 200.00 unidades
Especie 3: puede aumentar en 100.00 unidades
Especie 4: puede aumentar en 100.00 unidades
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_2428\1786683539.py:10:
RuntimeWarning: divide by zero encountered in divide
  incrementos = excedente / a_j
```