Escuela Politécnica Nacional

[Tarea 09] Ejercicios Unidad 04-A-B | Eliminación gaussiana vs Gauss-Jordan

Nombre: Sebastián Morales

Fecha: 15/07/2025

Curso: GR1CC

Repositorio:

https://github.com/SebastianMoralesEpn/Github1.0/tree/5b30199197715b349a37c4ad463c03324e4c788f/Tareas/%5BTarea%2009%5D%20Ejercicios%20Unidad%2004-A-B%20%20Eliminaci%C3%B3n%20gaussiana%20vs%20Gauss-Jordan

Conjunto de Ejercicios

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.

a. $\begin{aligned} x_1 + 2x_2 &= 0, \\ x_1 - x_2 &= 0. \end{aligned}$

b. $\begin{aligned} x_1 + 2x_2 &= 3, \\ -2x_1 - 4x_2 &= 6. \end{aligned}$

c. $\begin{aligned} 2x_1 + x_2 &= -1, \\ x_1 + x_2 &= 2, \\ x_1 - 3x_2 &= 5. \end{aligned}$

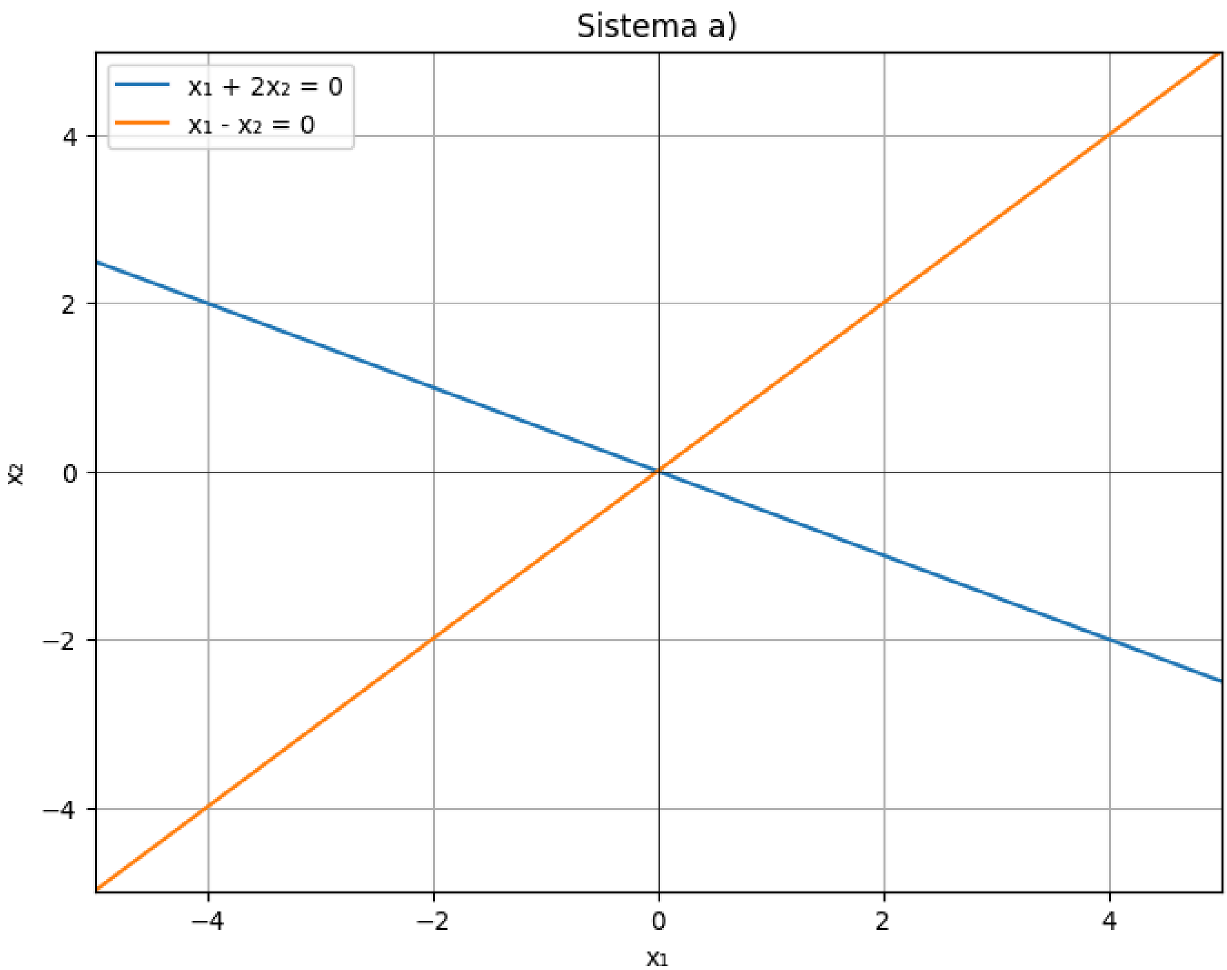
d. $\begin{aligned} 2x_1 + x_2 + x_3 &= 1, \\ 2x_1 + 4x_2 - x_3 &= -1. \end{aligned}$

Literal a.

In [2]:

```python
# Sistema a)
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 100)
y1 = (-x)/2  # x₁ + 2x₂ = 0 → x₂ = -x₁/2
y2 = x       # x₁ - x₂ = 0 → x₂ = x₁

plt.figure(figsize=(8, 6))
plt.plot(x, y1, label='x₁ + 2x₂ = 0')
plt.plot(x, y2, label='x₁ - x₂ = 0')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
```

```
plt.legend()
plt.title('Sistema a)')
plt.xlabel('x₁')
plt.ylabel('x₂')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()

# Solución exacta
A = np.array([[1, 2], [1, -1]])
b = np.array([0, 0])
sol = np.linalg.solve(A, b)
print(f"Solución exacta: x₁ = {sol[0]}, x₂ = {sol[1]}")
```

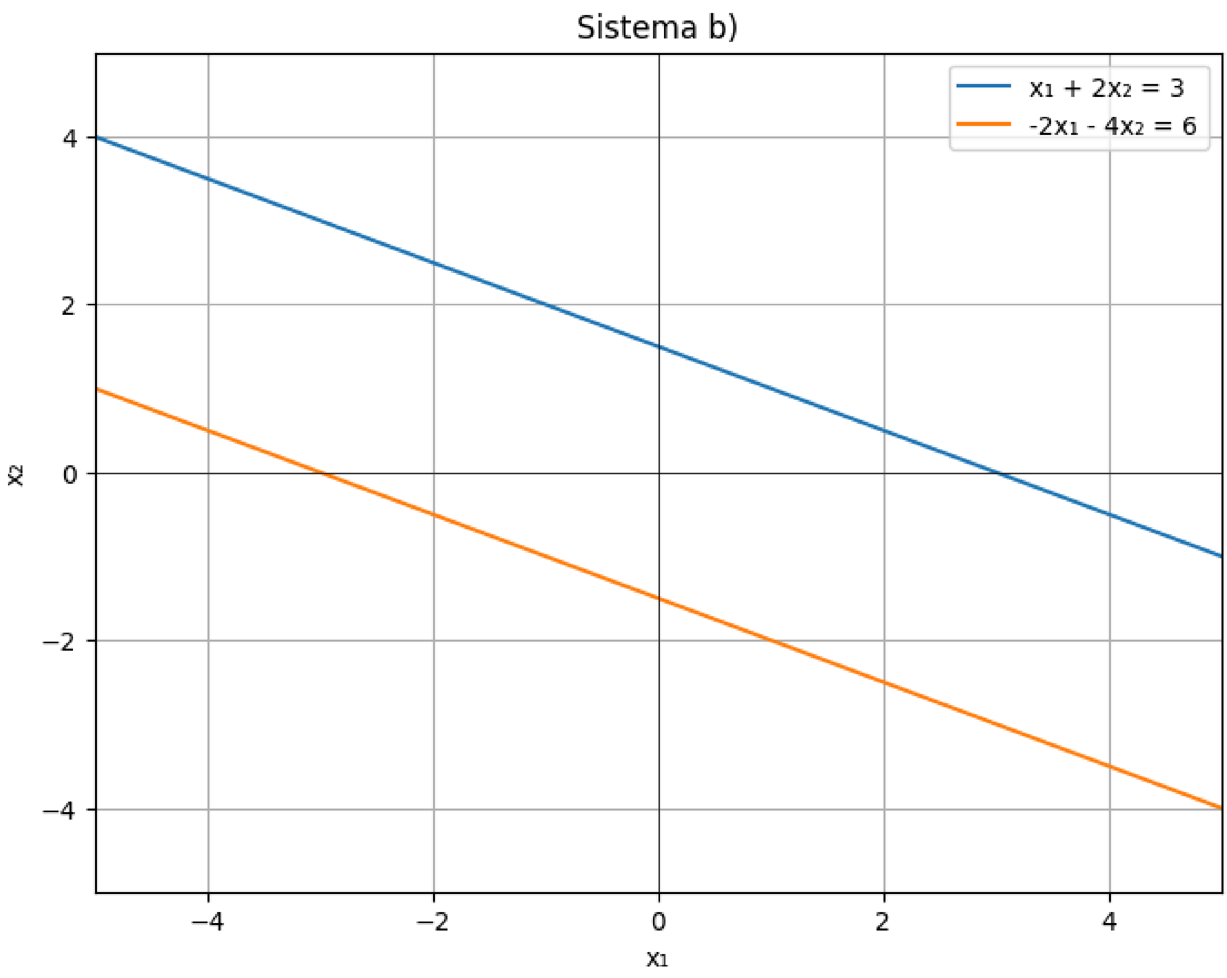



```
Solución exacta: x₁ = 0.0, x₂ = -0.0
```

Literal b

In [6]:
```
# Sistema b)
x = np.linspace(-5, 5, 100)
y1 = (3 - x)/2      # x₁ + 2x₂ = 3 → x₂ = (3 - x₁)/2
y2 = (-6 - 2*x)/4   # -2x₁ - 4x₂ = 6 → x₂ = (-6 - 2x₁)/4

plt.figure(figsize=(8, 6))
```

```
plt.plot(x, y1, label='x₁ + 2x₂ = 3')
plt.plot(x, y2, label='-2x₁ - 4x₂ = 6')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.legend()
plt.title('Sistema b)')
plt.xlabel('x₁')
plt.ylabel('x₂')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()

# Verificación de soluciones
print("Las dos ecuaciones representan la misma recta (la segunda es -2 veces l
print("Por lo tanto, hay infinitas soluciones")
```

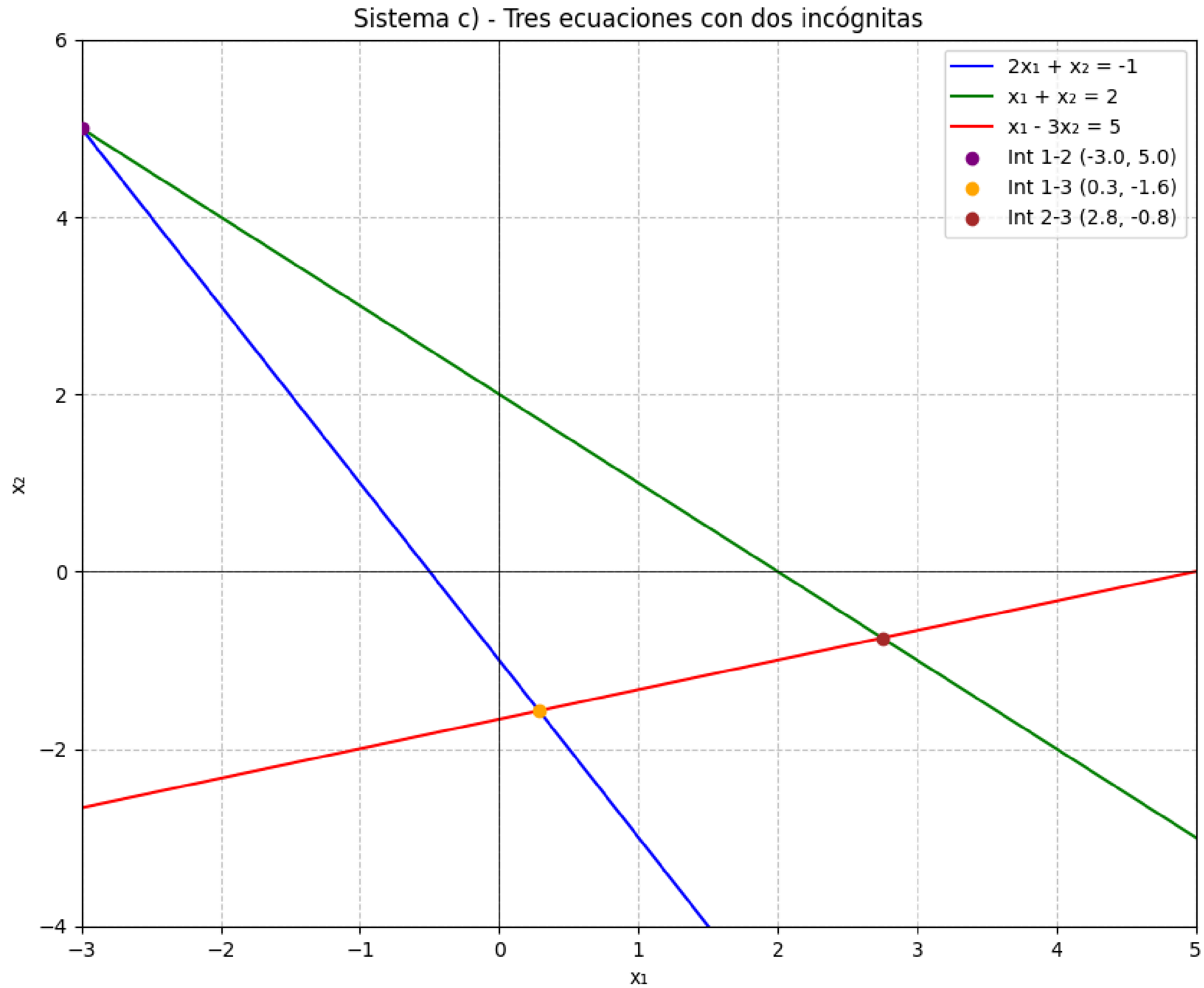



```
Las dos ecuaciones representan la misma recta (la segunda es -2 veces la primer
a)
Por lo tanto, hay infinitas soluciones
```

Literal C

In [7]:
```
# Sistema c)
```

```
x = np.linspace(-5, 5, 100)
y1 = -1 - 2*x  # 2x₁ + x₂ = -1 → x₂ = -1 - 2x₁
y2 = 2 - x     # x₁ + x₂ = 2 → x₂ = 2 - x₁
y3 = (x - 5)/3 # x₁ - 3x₂ = 5 → x₂ = (x₁ - 5)/3

plt.figure(figsize=(10, 8))
plt.plot(x, y1, label='2x₁ + x₂ = -1', color='blue')
plt.plot(x, y2, label='x₁ + x₂ = 2', color='green')
plt.plot(x, y3, label='x₁ - 3x₂ = 5', color='red')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True, linestyle='--', alpha=0.7)

# Intersecciones dos a dos
A = np.array([[2, 1], [1, 1]])
b1 = np.array([-1, 2])
sol1 = np.linalg.solve(A, b1)

A = np.array([[2, 1], [1, -3]])
b2 = np.array([-1, 5])
sol2 = np.linalg.solve(A, b2)

A = np.array([[1, 1], [1, -3]])
b3 = np.array([2, 5])
sol3 = np.linalg.solve(A, b3)

plt.scatter(sol1[0], sol1[1], color='purple', zorder=5, label=f'Int 1-2 ({sol1
plt.scatter(sol2[0], sol2[1], color='orange', zorder=5, label=f'Int 1-3 ({sol2
plt.scatter(sol3[0], sol3[1], color='brown', zorder=5, label=f'Int 2-3 ({sol3[

plt.legend()
plt.title('Sistema c) - Tres ecuaciones con dos incógnitas')
plt.xlabel('x₁')
plt.ylabel('x₂')
plt.xlim(-3, 5)
plt.ylim(-4, 6)
plt.show()

# Verificación de solución común
try:
    A = np.array([[2, 1], [1, 1], [1, -3]])
    b = np.array([-1, 2, 5])
    sol, residuals, _, _ = np.linalg.lstsq(A, b, rcond=None)
    print(f"Solución aproximada: x₁ = {sol[0]:.2f}, x₂ = {sol[1]:.2f}")
    print(f"Residuos: {residuals}")
    print("No hay solución exacta para las tres ecuaciones simultáneamente")
except np.linalg.LinAlgError:
    print("El sistema es incompatible - no tiene solución exacta")
```

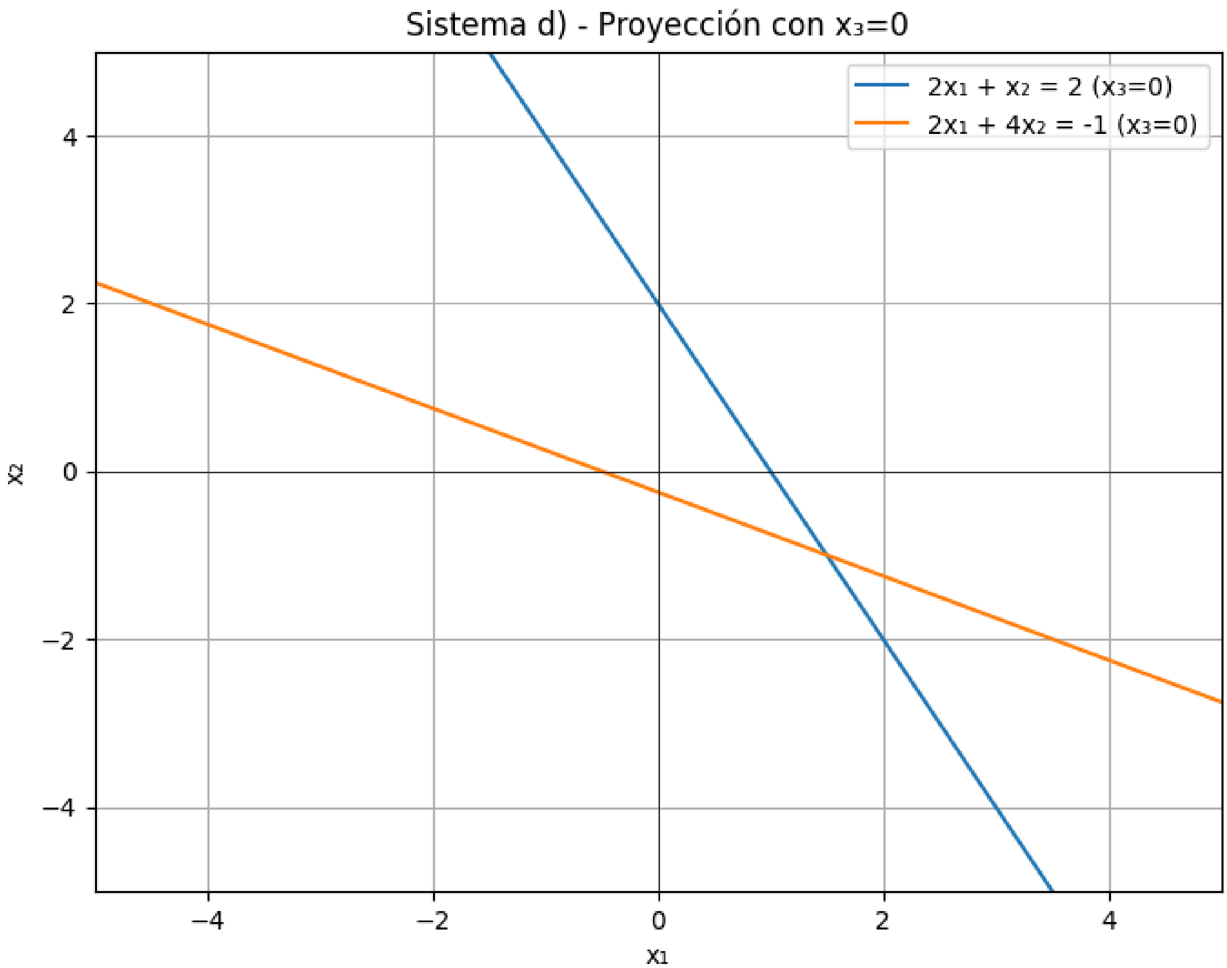



```
Solución aproximada: x₁ = 0.83, x₂ = -1.27
Residuos: [8.01515152]
No hay solución exacta para las tres ecuaciones simultáneamente
```

Literal d

In [8]:
```python
# Sistema d)
x = np.linspace(-5, 5, 100)
# Proyección con x₃=0
y1 = 2 - 2*x  # 2x₁ + x₂ = 2 → x₂ = 2 - 2x₁
y2 = (-1 - 2*x)/4  # 2x₁ + 4x₂ = -1 → x₂ = (-1 - 2x₁)/4

plt.figure(figsize=(8, 6))
plt.plot(x, y1, label='2x₁ + x₂ = 2 (x₃=0)')
plt.plot(x, y2, label='2x₁ + 4x₂ = -1 (x₃=0)')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.legend()
plt.title('Sistema d) - Proyección con x₃=0')
plt.xlabel('x₁')
plt.ylabel('x₂')
plt.xlim(-5, 5)
```

```python
plt.ylim(-5, 5)
plt.show()

# Solución paramétrica (asignando x₃ como parámetro)
print("Solución paramétrica del sistema d):")
print("Podemos expresar x₁ y x₂ en términos de x₃:")
print("x₁ = (7 - 5x₃)/6")
print("x₂ = (-1 - x₃)/3")
print("x₃ = x₃ (parámetro libre)")
print("Hay infinitas soluciones, correspondientes a la recta de intersección d
```




```
Solución paramétrica del sistema d):
Podemos expresar x₁ y x₂ en términos de x₃:
x₁ = (7 - 5x₃)/6
x₂ = (-1 - x₃)/3
x₃ = x₃ (parámetro libre)
Hay infinitas soluciones, correspondientes a la recta de intersección de los do
s planos
```

2. Utilice la eliminación gaussiana con sustitución hacia atrás y aritmética de redondeo de dos dígitos para resolver los siguientes sistemas lineales. No reordene las ecuaciones. (La solución exacta para cada sistema es �_1=−1, �_2=2, �_3=3.)

a. $$\begin{aligned} -x_1 + 4x_2 + \ \ x_3 &= 8, \\ \tfrac{5}{3}x_1 + \tfrac{2}{3}x_2 + \tfrac{2}{3}x_3 &= 1, \\ 2x_1 + \ \ x_2 + 4x_3 &= 11. \end{aligned}$$

b. $$\begin{aligned} 4x_1 + 2x_2 - x_3 &= -5, \\ \tfrac{1}{9}x_1 + \tfrac{1}{9}x_2 - \tfrac{1}{3}x_3 &= -1, \\ x_1 + 4x_2 + 2x_3 &= 9, \end{aligned}$$

Literal a

In [ ]:

```python
def round_two_digits(x):
    return np.round(x, decimals=2)

# Matriz aumentada
A = np.array([
    [-1, 4, 1, 8],
    [5/3, 2/3, 2/3, 1],
    [2, 1, 4, 11]
], dtype=float)

# Aplicamos redondeo a 2 dígitos
A = round_two_digits(A)
print("Matriz inicial:\n", A)

# Paso 1: Eliminación debajo del primer pivote
m21 = round_two_digits(A[1,0]/A[0,0])
A[1] = round_two_digits(A[1] - m21*A[0])
m31 = round_two_digits(A[2,0]/A[0,0])
A[2] = round_two_digits(A[2] - m31*A[0])
print("\nDespués de eliminar columna 1:\n", A)

# Paso 2: Eliminación debajo del segundo pivote
m32 = round_two_digits(A[2,1]/A[1,1])
A[2] = round_two_digits(A[2] - m32*A[1])
print("\nMatriz triangularizada:\n", A)

# Sustitución hacia atrás
x3 = round_two_digits(A[2,3]/A[2,2])
x2 = round_two_digits((A[1,3] - A[1,2]*x3)/A[1,1])
x1 = round_two_digits((A[0,3] - A[0,1]*x2 - A[0,2]*x3)/A[0,0])

print("\nSolución con redondeo a 2 dígitos:")
print(f"x1 = {x1:.2f}, x2 = {x2:.2f}, x3 = {x3:.2f}")
print("Solución exacta: x1 = -1, x2 = 2, x3 = 3")
```

```
Matriz inicial:
 [[-1.    4.    1.    8.  ]
 [ 1.67  0.67  0.67  1.  ]
 [ 2.    1.    4.   11.  ]]

Después de eliminar columna 1:
 [[-1.    4.    1.    8.  ]
 [ 0.    7.35  2.34 14.36]
 [ 0.    9.    6.   27.  ]]

Matriz triangularizada:
 [[-1.    4.    1.    8.  ]
 [ 0.    7.35  2.34 14.36]
 [ 0.    0.03  3.15  9.48]]

Solución con redondeo a 2 dígitos:
x1 = -0.99, x2 = 1.00, x3 = 3.01
Solución exacta: x1 = -1, x2 = 2, x3 = 3
```

Literal b

In [ ]:

```python
# Matriz aumentada
B = np.array([
    [4, 2, -1, -5],
    [1/9, 1/9, -1/3, -1],
    [1, 4, 2, 9]
], dtype=float)

# Aplicamos redondeo a 2 dígitos
B = round_two_digits(B)
print("\n\nMatriz inicial:\n", B)

# Paso 1: Eliminación debajo del primer pivote
m21 = round_two_digits(B[1,0]/B[0,0])
B[1] = round_two_digits(B[1] - m21*B[0])
m31 = round_two_digits(B[2,0]/B[0,0])
B[2] = round_two_digits(B[2] - m31*B[0])
print("\nDespués de eliminar columna 1:\n", B)

# Paso 2: Eliminación debajo del segundo pivote
m32 = round_two_digits(B[2,1]/B[1,1])
B[2] = round_two_digits(B[2] - m32*B[1])
print("\nMatriz triangularizada:\n", B)

# Sustitución hacia atrás
x3 = round_two_digits(B[2,3]/B[2,2])
x2 = round_two_digits((B[1,3] - B[1,2]*x3)/B[1,1])
x1 = round_two_digits((B[0,3] - B[0,1]*x2 - B[0,2]*x3)/B[0,0])

print("\nSolución con redondeo a 2 dígitos:")
print(f"x1 = {x1:.2f}, x2 = {x2:.2f}, x3 = {x3:.2f}")
print("Solución exacta: x1 = -1, x2 = 2, x3 = 3")
```

```
Matriz inicial:
 [[ 4.    2.   -1.   -5.  ]
 [ 0.11  0.11 -0.33 -1.  ]
 [ 1.    4.    2.    9.  ]]

Después de eliminar columna 1:
 [[ 4.000e+00  2.000e+00 -1.000e+00 -5.000e+00]
 [-1.000e-02  5.000e-02 -3.000e-01 -8.500e-01]
 [ 0.000e+00  3.500e+00  2.250e+00  1.025e+01]]

Matriz triangularizada:
 [[ 4.000e+00  2.000e+00 -1.000e+00 -5.000e+00]
 [-1.000e-02  5.000e-02 -3.000e-01 -8.500e-01]
 [ 7.000e-01  0.000e+00  2.325e+01  6.975e+01]]

Solución con redondeo a 2 dígitos:
x1 = -1.00, x2 = 1.00, x3 = 3.00
Solución exacta: x1 = -1, x2 = 2, x3 = 3
```

3. Utilice el algoritmo de eliminación gaussiana para resolver, de ser posible, los siguientes sistemas lineales, y determine si se necesitan intercambios de fila:

a. $$\begin{aligned} x_1 - x_2 + 3x_3 &= 2, \\ 3x_1 - 3x_2 + x_3 &= -1, \\ x_1 + x_2 &= 3. \end{aligned}$$

b. $$\begin{aligned} 2x_1 - 1.5x_2 + 3x_3 &= 1, \\ -x_1 + 2x_3 &= 3, \\ 4x_1 - 4.5x_2 + 5x_3 &= 1, \end{aligned}$$

c. $$\begin{aligned} 2x_1 &= 3, \\ x_1 + 1.5x_2 &= 4.5, \\ -3x_2 + 0.5x_3 &= -6.6. \\ 2x_1 - 2x_2 + x_3 + x_4 &= 0.8. \end{aligned}$$

d. $$\begin{aligned} x_1 + x_2 + x_4 &= 2, \\ 2x_1 + x_2 - x_3 + x_4 &= 1, \\ 4x_1 - x_2 - 2x_3 + 2x_4 &= 0, \\ 3x_1 - x_2 - x_3 + 2x_4 &= -3. \end{aligned}$$

Literal a

In [11]:
```python
# Matriz aumentada
A = np.array([
    [1, -1, 3, 2],
    [3, -3, 1, -1],
    [1, 1, 0, 3]
], dtype=float)

print("Matriz inicial:\n", A)

# Paso 1: Pivote en A[0,0] = 1
m21 = A[1,0]/A[0,0]
A[1] = A[1] - m21*A[0]
m31 = A[2,0]/A[0,0]
A[2] = A[2] - m31*A[0]
print("\nDespués de eliminar columna 1:\n", A)
```

```python
# Paso 2: Pivote en A[1,1] es cero - necesita intercambio
if np.abs(A[1,1]) < 1e-10:
    print("\nSe necesita intercambio de filas (pivote cero)")
    A[[1,2]] = A[[2,1]]  # Intercambiamos filas 2 y 3
    print("\nDespués del intercambio:\n", A)

# Continuamos con el nuevo pivote
m32 = A[2,1]/A[1,1]
A[2] = A[2] - m32*A[1]
print("\nMatriz triangularizada:\n", A)

# Sustitución hacia atrás
x3 = A[2,3]/A[2,2]
x2 = (A[1,3] - A[1,2]*x3)/A[1,1]
x1 = (A[0,3] - A[0,1]*x2 - A[0,2]*x3)/A[0,0]

print("\nSolución:")
print(f"x1 = {x1:.4f}, x2 = {x2:.4f}, x3 = {x3:.4f}")
```

```
Matriz inicial:
 [[ 1. -1.  3.  2.]
 [ 3. -3.  1. -1.]
 [ 1.  1.  0.  3.]]

Después de eliminar columna 1:
 [[ 1. -1.  3.  2.]
 [ 0.  0. -8. -7.]
 [ 0.  2. -3.  1.]]

Se necesita intercambio de filas (pivote cero)

Después del intercambio:
 [[ 1. -1.  3.  2.]
 [ 0.  2. -3.  1.]
 [ 0.  0. -8. -7.]]

Matriz triangularizada:
 [[ 1. -1.  3.  2.]
 [ 0.  2. -3.  1.]
 [ 0.  0. -8. -7.]]

Solución:
x1 = 1.1875, x2 = 1.8125, x3 = 0.8750
```

Literal b

In [12]:

```python
# Matriz aumentada
B = np.array([
    [2, -1.5, 3, 1],
    [-1, 0, 2, 3],
    [4, -4.5, 5, 1]
], dtype=float)

print("\n\nMatriz inicial:\n", B)
```

```python
# Paso 1: Pivote en B[0,0] = 2
m21 = B[1,0]/B[0,0]
B[1] = B[1] - m21*B[0]
m31 = B[2,0]/B[0,0]
B[2] = B[2] - m31*B[0]
print("\nDespués de eliminar columna 1:\n", B)

# Paso 2: Pivote en B[1,1] = -0.75
m32 = B[2,1]/B[1,1]
B[2] = B[2] - m32*B[1]
print("\nMatriz triangularizada:\n", B)

# Sustitución hacia atrás
x3 = B[2,3]/B[2,2]
x2 = (B[1,3] - B[1,2]*x3)/B[1,1]
x1 = (B[0,3] - B[0,1]*x2 - B[0,2]*x3)/B[0,0]

print("\nSolución:")
print(f"x1 = {x1:.4f}, x2 = {x2:.4f}, x3 = {x3:.4f}")
print("No se necesitaron intercambios de fila")
```

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

Después de eliminar columna 1:
 [[ 2.   -1.5   3.    1.  ]
 [ 0.   -0.75  3.5   3.5 ]
 [ 0.   -1.5  -1.   -1.  ]]

Matriz triangularizada:
 [[ 2.   -1.5   3.    1.  ]
 [ 0.   -0.75  3.5   3.5 ]
 [ 0.    0.   -8.   -8.  ]]

Solución:
x1 = -1.0000, x2 = -0.0000, x3 = 1.0000
No se necesitaron intercambios de fila
```

Literal c

In [21]:

```python
# Matriz aumentada del sistema c)
C = np.array([
    [2, 0, 0, 3],
    [1, 1.5, 0, 4.5],
    [0, -3, 0.5, -6.6],
    [2, -2, 1, 0.8]
], dtype=float)

print("Matriz inicial del sistema c):\n", C)
```

```python
# Paso 1: Pivote en C[0,0] = 2
m21 = C[1,0]/C[0,0]
C[1] = C[1] - m21*C[0]
m31 = C[2,0]/C[0,0]  # Es cero, no necesita operación
m41 = C[3,0]/C[0,0]
C[3] = C[3] - m41*C[0]
print("\nDespués de eliminar columna 1:\n", C)

# Paso 2: Pivote en C[1,1] = 1.5
m32 = C[2,1]/C[1,1]
C[2] = C[2] - m32*C[1]
m42 = C[3,1]/C[1,1]
C[3] = C[3] - m42*C[1]
print("\nDespués de eliminar columna 2:\n", C)

# Paso 3: Pivote en C[2,2] = 0.5
m43 = C[3,2]/C[2,2]
C[3] = C[3] - m43*C[2]
print("\nMatriz triangularizada:\n", C)

# Verificación de consistencia
if np.abs(C[3,2]) < 1e-10:
    if np.abs(C[3,3]) < 1e-10:
        print("\nSistema compatible indeterminado (infinitas soluciones)")
    else:
        print("\nSistema incompatible (no tiene solución)")
else:
    # Sustitución hacia atrás
    x3 = C[3,3]/C[3,2]
    x2 = (C[2,3] - C[2,2]*x3)/C[2,1]
    x1 = (C[1,3] - C[1,2]*x3 - C[1,1]*x2)/C[1,0]
    x0 = (C[0,3] - C[0,1]*x2 - C[0,2]*x3)/C[0,0]

    print("\nSolución:")
    print(f"x1 = {x1:.4f}, x2 = {x2:.4f}, x3 = {x3:.4f}")
```

```
Matriz inicial del sistema c):
 [[ 2.   0.   0.   3. ]
 [ 1.   1.5  0.   4.5]
 [ 0.  -3.   0.5 -6.6]
 [ 2.  -2.   1.   0.8]]

Después de eliminar columna 1:
 [[ 2.   0.   0.   3. ]
 [ 0.   1.5  0.   3. ]
 [ 0.  -3.   0.5 -6.6]
 [ 0.  -2.   1.  -2.2]]

Después de eliminar columna 2:
 [[ 2.   0.   0.   3. ]
 [ 0.   1.5  0.   3. ]
 [ 0.   0.   0.5 -0.6]
 [ 0.   0.   1.   1.8]]

Matriz triangularizada:
 [[ 2.   0.   0.   3. ]
 [ 0.   1.5  0.   3. ]
 [ 0.   0.   0.5 -0.6]
 [ 0.   0.   0.   3. ]]

Sistema incompatible (no tiene solución)
```

Literal d

In [22]:
```python
# Matriz aumentada
D = np.array([
    [1, 1, 0, 1, 2],
    [2, 1, -1, 1, 1],
    [4, -1, -2, 2, 0],
    [3, -1, -1, 2, -3]
], dtype=float)

print("\n\nMatriz inicial del sistema d):\n", D)

# Paso 1: Pivote en D[0,0] = 1
m21 = D[1,0]/D[0,0]
D[1] = D[1] - m21*D[0]
m31 = D[2,0]/D[0,0]
D[2] = D[2] - m31*D[0]
m41 = D[3,0]/D[0,0]
D[3] = D[3] - m41*D[0]
print("\nDespués de eliminar columna 1:\n", D)

# Paso 2: Pivote en D[1,1] = -1
# Verificamos si necesitamos intercambio
if np.abs(D[1,1]) < 1e-10:
    # Buscamos una fila debajo con elemento no nulo en columna 2
    for i in range(2,4):
        if np.abs(D[i,1]) > 1e-10:
```

```python
            D[[1,i]] = D[[i,1]]  # Intercambiamos filas
            print("\nIntercambio de filas 2 y", i+1)
            break

m32 = D[2,1]/D[1,1]
D[2] = D[2] - m32*D[1]
m42 = D[3,1]/D[1,1]
D[3] = D[3] - m42*D[1]
print("\nDespués de eliminar columna 2:\n", D)

# Paso 3: Pivote en D[2,2] = -4.0
m43 = D[3,2]/D[2,2]
D[3] = D[3] - m43*D[2]
print("\nMatriz triangularizada:\n", D)

# Verificación de consistencia
if np.abs(D[3,3]) < 1e-10:
    if np.abs(D[3,4]) < 1e-10:
        print("\nSistema compatible indeterminado (infinitas soluciones)")
        # Asignamos x4 como parámetro libre
        x4 = 1  # Valor arbitrario
        x3 = (D[2,4] - D[2,3]*x4)/D[2,2]
        x2 = (D[1,4] - D[1,2]*x3 - D[1,3]*x4)/D[1,1]
        x1 = (D[0,4] - D[0,1]*x2 - D[0,2]*x3 - D[0,3]*x4)/D[0,0]
        print(f"Solución paramétrica con x4 = {x4}:")
        print(f"x1 = {x1:.4f}, x2 = {x2:.4f}, x3 = {x3:.4f}, x4 = {x4:.4f}")
    else:
        print("\nSistema incompatible (no tiene solución)")
else:
    # Sustitución hacia atrás
    x4 = D[3,4]/D[3,3]
    x3 = (D[2,4] - D[2,3]*x4)/D[2,2]
    x2 = (D[1,4] - D[1,2]*x3 - D[1,3]*x4)/D[1,1]
    x1 = (D[0,4] - D[0,1]*x2 - D[0,2]*x3 - D[0,3]*x4)/D[0,0]

    print("\nSolución:")
    print(f"x1 = {x1:.4f}, x2 = {x2:.4f}, x3 = {x3:.4f}, x4 = {x4:.4f}")
```

```
Matriz inicial del sistema d):
 [[ 1.  1.  0.  1.  2.]
 [ 2.  1. -1.  1.  1.]
 [ 4. -1. -2.  2.  0.]
 [ 3. -1. -1.  2. -3.]]

Después de eliminar columna 1:
 [[ 1.  1.  0.  1.  2.]
 [ 0. -1. -1. -1. -3.]
 [ 0. -5. -2. -2. -8.]
 [ 0. -4. -1. -1. -9.]]

Después de eliminar columna 2:
 [[ 1.  1.  0.  1.  2.]
 [ 0. -1. -1. -1. -3.]
 [ 0.  0.  3.  3.  7.]
 [ 0.  0.  3.  3.  3.]]

Matriz triangularizada:
 [[ 1.  1.  0.  1.  2.]
 [ 0. -1. -1. -1. -3.]
 [ 0.  0.  3.  3.  7.]
 [ 0.  0.  0.  0. -4.]]

Sistema incompatible (no tiene solución)
```

4. Use el algoritmo de eliminación gaussiana y la aritmética computacional de precisión de 32 bits para resolver los siguientes sistemas lineales.

a. $$\begin{aligned} \tfrac{1}{4}x_1 + \tfrac{1}{5}x_2 + \tfrac{1}{6}x_3 &= 9, \\ \tfrac{1}{3}x_1 + \tfrac{1}{4}x_2 + \tfrac{1}{5}x_3 &= 8, \\ \tfrac{1}{2}x_1 + x_2 + 2x_3 &= 8. \end{aligned}$$

b. $$\begin{aligned} 3.333x_1 + 15920x_2 - 10.333x_3 &= 15913, \\ 2.222x_1 + 16.71x_2 + 9.612x_3 &= 28.544, \\ 1.5611x_1 + 5.1791x_2 + 1.6852x_3 &= 8.4254. \end{aligned}$$

c. $$\begin{aligned} x_1 + \tfrac{1}{2}x_2 + \tfrac{1}{3}x_3 + \tfrac{1}{4}x_4 &= \tfrac{1}{6}, \\ \tfrac{1}{2}x_1 + \tfrac{1}{3}x_2 + \tfrac{1}{4}x_3 + \tfrac{1}{5}x_4 &= \tfrac{1}{7}, \\ \tfrac{1}{3}x_1 + \tfrac{1}{4}x_2 + \tfrac{1}{5}x_3 + \tfrac{1}{6}x_4 &= \tfrac{1}{8}, \\ \tfrac{1}{4}x_1 + \tfrac{1}{5}x_2 + \tfrac{1}{6}x_3 + \tfrac{1}{7}x_4 &= \tfrac{1}{9}. \end{aligned}$$

d. $$\begin{aligned} 2x_1 + x_2 - x_3 + x_4 - 3x_5 &= 7, \\ x_1 + 2x_3 - x_4 + x_5 &= 2, \\ -2x_2 - x_3 + x_4 - x_5 &= -5, \\ 3x_1 + x_2 - 4x_3 + 5x_5 &= 6, \\ x_1 - x_2 - x_3 - x_4 + x_5 &= -3. \end{aligned}$$

Literal a

In [23]:
```
# Configurar precisión de 32 bits
np.set_printoptions(precision=7, suppress=True)

# Matriz aumentada
A = np.array([
    [1/4, 1/5, 1/6, 9],
    [1/3, 1/4, 1/5, 8],
```

```
    [1/2, 1, 2, 8]
], dtype=np.float32)

print("Matriz inicial:\n", A)

# Eliminación gaussiana
for k in range(2):
    for i in range(k+1, 3):
        factor = np.float32(A[i,k] / A[k,k])
        A[i,k:] = np.subtract(A[i,k:], np.multiply(factor, A[k,k:]), dtype=np.
    print(f"\nPaso {k+1}:\n", A)

# Sustitución hacia atrás
x = np.zeros(3, dtype=np.float32)
x[2] = A[2,3] / A[2,2]
x[1] = (A[1,3] - A[1,2]*x[2]) / A[1,1]
x[0] = (A[0,3] - A[0,1]*x[1] - A[0,2]*x[2]) / A[0,0]

print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}")
```

```
Matriz inicial:
 [[0.25      0.2       0.1666667 9.       ]
 [0.3333333 0.25      0.2       8.       ]
 [0.5       1.        2.        8.       ]]

Paso 1:
 [[  0.25        0.2         0.1666667   9.       ]
 [  0.         -0.0166667  -0.0222222  -4.       ]
 [  0.          0.6         1.6666666 -10.       ]]

Paso 2:
 [[   0.25         0.2          0.1666667    9.       ]
 [   0.          -0.0166667   -0.0222222   -4.       ]
 [   0.          -0.0000001    0.8666667 -153.9999   ]]

Solución:
x1 = -227.0766602, x2 = 476.9226379, x3 = -177.6921692
```

Literal b

In [24]:

```
# Matriz aumentada
B = np.array([
    [3.333, 15920, -10.333, 15913],
    [2.222, 16.71, 9.612, 28.544],
    [1.5611, 5.1791, 1.6852, 8.4254]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", B)

# Pivoteo parcial para mejorar estabilidad
for k in range(2):
    max_row = np.argmax(np.abs(B[k:,k])) + k
    B[[k,max_row]] = B[[max_row,k]]
```

```python
    for i in range(k+1, 3):
        factor = np.float32(B[i,k] / B[k,k])
        B[i,k:] = np.subtract(B[i,k:], np.multiply(factor, B[k,k:]), dtype=np.
    print(f"\nPaso {k+1}:\n", B)

# Sustitución hacia atrás
x = np.zeros(3, dtype=np.float32)
x[2] = B[2,3] / B[2,2]
x[1] = (B[1,3] - B[1,2]*x[2]) / B[1,1]
x[0] = (B[0,3] - B[0,1]*x[1] - B[0,2]*x[2]) / B[0,0]

print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}")
```

```
Matriz inicial:
 [[    3.333  15920.       -10.333  15913.    ]
 [    2.222     16.71       9.612     28.544 ]
 [    1.5611     5.1791     1.6852     8.4254]]

Paso 1:
 [[     3.333      15920.            -10.333      15913.        ]
 [     0.        -10596.623         16.500668  -10580.122    ]
 [     0.         -7451.3804         6.5249376  -7444.8555   ]]

Paso 2:
 [[     3.333      15920.            -10.333      15913.        ]
 [     0.        -10596.623         16.500668  -10580.122    ]
 [     0.             0.            -5.0780745     -5.0786133]]

Solución:
x1 = 0.9997431, x2 = 1.0000001, x3 = 1.0001061
```

Literal c

In [25]:
```python
# Matriz aumentada
C = np.array([
    [1, 1/2, 1/3, 1/4, 1/6],
    [1/2, 1/3, 1/4, 1/5, 1/7],
    [1/3, 1/4, 1/5, 1/6, 1/8],
    [1/4, 1/5, 1/6, 1/7, 1/9]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", C)

# Eliminación gaussiana
for k in range(3):
    for i in range(k+1, 4):
        factor = np.float32(C[i,k] / C[k,k])
        C[i,k:] = np.subtract(C[i,k:], np.multiply(factor, C[k,k:]), dtype=np.
    print(f"\nPaso {k+1}:\n", C)

# Sustitución hacia atrás
```

```python
x = np.zeros(4, dtype=np.float32)
x[3] = C[3,4] / C[3,3]
x[2] = (C[2,4] - C[2,3]*x[3]) / C[2,2]
x[1] = (C[1,4] - C[1,2]*x[2] - C[1,3]*x[3]) / C[1,1]
x[0] = (C[0,4] - C[0,1]*x[1] - C[0,2]*x[2] - C[0,3]*x[3]) / C[0,0]

print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}, x4 = {x[3]:.7f}")
```

```
Matriz inicial:
 [[1.         0.5        0.3333333 0.25       0.1666667]
 [0.5        0.3333333 0.25       0.2        0.1428571]
 [0.3333333 0.25       0.2        0.1666667 0.125    ]
 [0.25       0.2        0.1666667 0.1428571 0.1111111]]

Paso 1:
 [[1.         0.5        0.3333333 0.25       0.1666667]
 [0.         0.0833333 0.0833333 0.075      0.0595238]
 [0.         0.0833333 0.0888889 0.0833333 0.0694444]
 [0.         0.075      0.0833333 0.0803571 0.0694444]]

Paso 2:
 [[1.         0.5        0.3333333 0.25       0.1666667]
 [0.         0.0833333 0.0833333 0.075      0.0595238]
 [0.         0.         0.0055556 0.0083333 0.0099206]
 [0.         0.         0.0083333 0.0128572 0.015873 ]]

Paso 3:
 [[1.         0.5        0.3333333 0.25       0.1666667]
 [0.         0.0833333 0.0833333 0.075      0.0595238]
 [0.         0.         0.0055556 0.0083333 0.0099206]
 [0.         0.         0.         0.0003571 0.0009921]]

Solución:
x1 = -0.0317476, x2 = 0.5952595, x3 = -2.3810065, x4 = 2.7778137
```

Literal d

In [26]:
```python
# Matriz aumentada
D = np.array([
    [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]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", D)

# Eliminación gaussiana con pivoteo parcial
for k in range(4):
    max_row = np.argmax(np.abs(D[k:,k])) + k
    D[[k,max_row]] = D[[max_row,k]]
```

```
    for i in range(k+1, 5):
        factor = np.float32(D[i,k] / D[k,k])
        D[i,k:] = np.subtract(D[i,k:], np.multiply(factor, D[k,k:]), dtype=np.
    print(f"\nPaso {k+1}:\n", D)

# Sustitución hacia atrás
x = np.zeros(5, dtype=np.float32)
x[4] = D[4,5] / D[4,4]
x[3] = (D[3,5] - D[3,4]*x[4]) / D[3,3]
x[2] = (D[2,5] - D[2,3]*x[3] - D[2,4]*x[4]) / D[2,2]
x[1] = (D[1,5] - D[1,2]*x[2] - D[1,3]*x[3] - D[1,4]*x[4]) / D[1,1]
x[0] = (D[0,5] - D[0,1]*x[1] - D[0,2]*x[2] - D[0,3]*x[3] - D[0,4]*x[4]) / D[0,

print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}, x4 = {x[3]:.7f}, x5
```

```
Matriz inicial:
 [[ 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.]]

Paso 1:
 [[ 3.          1.         -4.          0.          5.          6.        ]
 [ 0.         -0.3333333  3.3333335 -1.         -0.6666667  0.        ]
 [ 0.         -2.         -1.          1.         -1.         -5.        ]
 [ 0.          0.3333333  1.6666667  1.         -6.3333335  3.        ]
 [ 0.         -1.3333334  0.3333334 -1.         -0.6666667 -5.        ]]

Paso 2:
 [[ 3.          1.         -4.          0.          5.          6.        ]
 [ 0.         -2.         -1.          1.         -1.         -5.        ]
 [ 0.          0.          3.5000002 -1.1666666 -0.5000001  0.8333334]
 [ 0.          0.          1.5000001  1.1666666 -6.5         2.1666667]
 [ 0.          0.          1.         -1.6666667 -0.0000001 -1.6666665]]

Paso 3:
 [[ 3.          1.         -4.          0.          5.          6.        ]
 [ 0.         -2.         -1.          1.         -1.         -5.        ]
 [ 0.          0.          3.5000002 -1.1666666 -0.5000001  0.8333334]
 [ 0.          0.          0.          1.6666666 -6.285714   1.8095238]
 [ 0.          0.          0.         -1.3333335  0.1428571 -1.9047618]]

Paso 4:
 [[ 3.          1.         -4.          0.          5.          6.        ]
 [ 0.         -2.         -1.          1.         -1.         -5.        ]
 [ 0.          0.          3.5000002 -1.1666666 -0.5000001  0.8333334]
 [ 0.          0.          0.          1.6666666 -6.285714   1.8095238]
 [ 0.          0.          0.          0.         -4.885715  -0.4571425]]

Solución:
x1 = 1.8830409, x2 = 2.8070173, x3 = 0.7309940, x4 = 1.4385961, x5 = 0.0935672
```

5. Dado el sistema lineal:

$$\begin{aligned} x_1 - \quad x_2 + \alpha x_3 &= -2, \\ -x_1 + 2x_2 - \alpha x_3 &= 3, \\ \alpha x_1 + \quad x_2 + \quad x_3 &= 2. \end{aligned}$$

a. Encuentre el valor(es) de ⯑ para los que el sistema no tiene soluciones.

In [33]:
```python
import sympy as sp
```

```python
# Definimos símbolos
α = sp.symbols('α')

# Matriz de coeficientes
A = sp.Matrix([
    [1, -1, α],
    [-1, 2, -α],
    [α, 1, 1]
])

# Términos independientes
b = sp.Matrix([-2, 3, 2])

# Matriz aumentada
Ab = A.row_join(b)

print("Matriz aumentada del sistema:")
sp.pretty_print(Ab)

# Calculamos el determinante de A
det_A = A.det()
print("\nDeterminante de la matriz de coeficientes:")
sp.pretty_print(det_A)

# Encontramos los valores de α que hacen det(A) = 0
α_values = sp.solve(det_A, α)
print("\nValores críticos de α:", α_values)

# Para α = 1
print("\nPara α = 1:")
A_1 = Ab.subs(α, 1)
sp.pretty_print(A_1.rref())

# Para α = -2
print("\nPara α = -2:")
A_minus2 = Ab.subs(α, -2)
sp.pretty_print(A_minus2.rref())

print("\nRespuesta parte a):")
print("El sistema no tiene solución cuando α = 1")
```

```
Matriz aumentada del sistema:
⎡1   -1  α   -2⎤
⎢              ⎥
⎢-1  2   -α  3 ⎥
⎢              ⎥
⎣α   1   1   2 ⎦

Determinante de la matriz de coeficientes:
     2
1 - α

Valores críticos de α: [-1, 1]

Para α = 1:
⎛⎡1  0  1  0⎤            ⎞
⎜⎢          ⎥            ⎟
⎜⎢0  1  0  0⎥, (0, 1, 3)⎟
⎜⎢          ⎥            ⎟
⎝⎣0  0  0  1⎦            ⎠

Para α = -2:
⎛⎡1  0  0  -1/3⎤            ⎞
⎜⎢             ⎥            ⎟
⎜⎢0  1  0   1  ⎥, (0, 1, 2)⎟
⎜⎢             ⎥            ⎟
⎝⎣0  0  1  1/3 ⎦            ⎠

Respuesta parte a):
El sistema no tiene solución cuando α = 1
```

b. Encuentre el valor(es) de ⯑ para los que el sistema tiene un número infinito de soluciones.

In [34]:
```python
print("\nParte b):")
print("Para α = -2 el sistema tiene infinitas soluciones")
print("(Se observa en la forma escalonada reducida para α = -2)")
```

```
Parte b):
Para α = -2 el sistema tiene infinitas soluciones
(Se observa en la forma escalonada reducida para α = -2)
```

c. Suponga que existe una única solución para una a determinada, encuentre la solución.

In [35]:
```python
# Solución general para α ≠ 1 y α ≠ -2
print("\nParte c): Solución para valores de α con solución única")
solution = sp.solve_linear_system(Ab, *sp.symbols('x1 x2 x3'))
sp.pretty_print(solution)

# Ejemplo con α = 0
print("\nEjemplo con α = 0:")
sol_α0 = {k: v.subs(α, 0) for k, v in solution.items()}
sp.pretty_print(sol_α0)
```

```python
# Ejemplo con α = 2
print("\nEjemplo con α = 2:")
sol_α2 = {k: v.subs(α, 2) for k, v in solution.items()}
sp.pretty_print(sol_α2)
```

```
Parte c): Solución para valores de α con solución única
⎧      1               -1  ⎫
⎨x₁: ─────, x₂: 1, x₃: ─────⎬
⎩    α - 1             α - 1⎭

Ejemplo con α = 0:
{x₁: -1, x₂: 1, x₃: 1}

Ejemplo con α = 2:
{x₁: 1, x₂: 1, x₃: -1}
```

EJERCICIOS APLICADOS

6. Suponga que en un sistema biológico existen n especies de animales y m fuentes de alimento. Si �� representa la población de las j-ésimas especies, para cada �=1,⋯,�; ��; representa el suministro diario disponible del i-ésimo alimento y ��� representa la cantidad del i-ésimo alimento.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ \vdots \qquad \vdots \qquad\qquad \vdots \qquad &\ \ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m, \end{aligned}$$

representa un equilibrio donde existe un suministro diario de alimento para cumplir con precisión con el promedio diario de consumo de cada especie.

a. Si

$$A = [a_{ij}] = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

x=(��)=[1000,500,350,400], y b=(��)=[3500,2700,900]. ¿Existe suficiente alimento para satisfacer el consumo promedio diario?

In [36]:
```python
# Definir matriz A, vector x y vector b
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])

# Calcular consumo
b_calculado = A @ x

# Verificar si hay suficiente alimento
suficiente = np.allclose(b_calculado, b)

print("Consumo calculado:", b_calculado)
print("Suministro disponible:", b)
print("\n¿Existe suficiente alimento?", "Sí" if suficiente else "No")
```

```
Consumo calculado: [3200 2500  750]
Suministro disponible: [3500 2700  900]

¿Existe suficiente alimento? No
```

b. ¿Cuál es el número máximo de animales de cada especie que se podría agregar de forma individual al sistema con el suministro de alimento que cumpla con el consumo?

In [40]:
```python
# Calcular excedente de alimento
excedente = b - (A @ x)

# Calcular máximo incremento para cada especie
incrementos = []
for j in range(A.shape[1]):
    col = A[:, j]
    with np.errstate(divide='ignore', invalid='ignore'):
        ratios = np.where(col != 0, excedente / col, np.inf)
    incremento_max = np.min(ratios[ratios > 0]) if any(ratios > 0) else 0
    incrementos.append(incremento_max)

# Mostrar resultados
print("Máximo incremento individual por especie:")
for i, inc in enumerate(incrementos, 1):
    print(f"Especie {i}: {inc:.2f} animales")
```

```
Máximo incremento individual por especie:
Especie 1: 200.00 animales
Especie 2: 150.00 animales
Especie 3: 100.00 animales
Especie 4: 100.00 animales
```

c. Si la especie 1 se extingue, ¿qué cantidad de incremento individual de las especies restantes se podría soportar?

In [41]:
```python
# Eliminar especie 1 (columna 0)
A_sin_1 = np.delete(A, 0, axis=1)
x_sin_1 = np.delete(x, 0)

# Calcular nuevo consumo y excedente
excedente_sin_1 = b - (A_sin_1 @ x_sin_1)

# Calcular incrementos para especies restantes
incrementos_sin_1 = []
for j in range(A_sin_1.shape[1]):
    col = A_sin_1[:, j]
    with np.errstate(divide='ignore', invalid='ignore'):
        ratios = np.where(col != 0, excedente_sin_1 / col, np.inf)
    incremento_max = np.min(ratios[ratios > 0]) if any(ratios > 0) else 0
    incrementos_sin_1.append(incremento_max)

# Mostrar resultados
print("Incremento posible si la especie 1 se extingue:")
for esp, inc in zip([2, 3, 4], incrementos_sin_1):
    print(f"Especie {esp}: {inc:.2f} animales")
```

```
Incremento posible si la especie 1 se extingue:
Especie 2: 650.00 animales
Especie 3: 150.00 animales
Especie 4: 150.00 animales
```

d. Si la especie 2 se extingue, ¿qué cantidad de incremento individual de las especies restantes se podría soportar?

In [42]:
```python
# Eliminar especie 2 (columna 1)
A_sin_2 = np.delete(A, 1, axis=1)
x_sin_2 = np.delete(x, 1)

# Calcular nuevo consumo y excedente
excedente_sin_2 = b - (A_sin_2 @ x_sin_2)

# Calcular incrementos para especies restantes
incrementos_sin_2 = []
for j in range(A_sin_2.shape[1]):
    col = A_sin_2[:, j]
    with np.errstate(divide='ignore', invalid='ignore'):
        ratios = np.where(col != 0, excedente_sin_2 / col, np.inf)
    incremento_max = np.min(ratios[ratios > 0]) if any(ratios > 0) else 0
    incrementos_sin_2.append(incremento_max)

# Mostrar resultados
print("Incremento posible si la especie 2 se extingue:")
for esp, inc in zip([1, 3, 4], incrementos_sin_2):
    print(f"Especie {esp}: {inc:.2f} animales")
```

```
Incremento posible si la especie 2 se extingue:
Especie 1: 200.00 animales
Especie 3: 100.00 animales
Especie 4: 100.00 animales
```

EJERCICIOS TEÓRICOS

7. Repita el ejercicio 4 con el método Gauss-Jordan.

Literal a

In [27]:
```
# Configurar precisión de 32 bits
np.set_printoptions(precision=7, suppress=True)

# Matriz aumentada
A = np.array([
    [1/4, 1/5, 1/6, 9],
    [1/3, 1/4, 1/5, 8],
    [1/2, 1, 2, 8]
], dtype=np.float32)

print("Matriz inicial:\n", A)

# Gauss-Jordan
n = len(A)
for k in range(n):
    # Normalizar fila pivote
    pivot = A[k,k]
    A[k] = np.divide(A[k], pivot, dtype=np.float32)

    # Eliminación hacia arriba y abajo
    for i in range(n):
        if i != k:
            factor = A[i,k]
            A[i] = np.subtract(A[i], np.multiply(factor, A[k]), dtype=np.float
    print(f"\nPaso {k+1}:\n", A)

# Extraer solución
x = A[:, -1]
print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}")
```

```
Matriz inicial:
 [[0.25      0.2       0.1666667 9.       ]
 [0.3333333 0.25      0.2       8.       ]
 [0.5       1.        2.        8.       ]]

Paso 1:
 [[  1.          0.8         0.6666667  36.       ]
 [  0.         -0.0166667  -0.0222222  -4.       ]
 [  0.          0.6         1.6666666 -10.       ]]

Paso 2:
 [[   1.           0.          -0.3999998 -155.99985  ]
 [  -0.           1.           1.333333   239.9998   ]
 [   0.           0.           0.8666668 -153.9999   ]]

Paso 3:
 [[   1.          0.          0.       -227.07668]
 [  -0.          1.          0.        476.9226 ]
 [   0.          0.          1.       -177.69215]]

Solución:
x1 = -227.0766754, x2 = 476.9226074, x3 = -177.6921539
```

Literal b

In [28]:

```python
# Matriz aumentada
B = np.array([
    [3.333, 15920, -10.333, 15913],
    [2.222, 16.71, 9.612, 28.544],
    [1.5611, 5.1791, 1.6852, 8.4254]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", B)

# Gauss-Jordan con pivoteo parcial
n = len(B)
for k in range(n):
    # Pivoteo parcial
    max_row = np.argmax(np.abs(B[k:,k])) + k
    B[[k,max_row]] = B[[max_row,k]]

    # Normalizar fila pivote
    pivot = B[k,k]
    B[k] = np.divide(B[k], pivot, dtype=np.float32)

    # Eliminación hacia arriba y abajo
    for i in range(n):
        if i != k:
            factor = B[i,k]
            B[i] = np.subtract(B[i], np.multiply(factor, B[k]), dtype=np.float
    print(f"\nPaso {k+1}:\n", B)

# Extraer solución
```

```python
x = B[:, -1]
print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}")
```

```
Matriz inicial:
 [[    3.333  15920.       -10.333  15913.    ]
 [    2.222     16.71       9.612     28.544 ]
 [    1.5611     5.1791     1.6852     8.4254]]

Paso 1:
 [[     1.          4776.4775        -3.1002102   4774.3774   ]
 [     0.        -10596.623         16.500668  -10580.122    ]
 [     0.         -7451.38           6.5249386  -7444.8555   ]]

Paso 2:
 [[ 1.          0.          4.337543   5.3378906]
 [-0.          1.        -0.0015572  0.9984428]
 [ 0.          0.        -5.0780735 -5.0786133]]

Paso 3:
 [[ 1.          0.          0.          0.9998865]
 [-0.          1.         0.          1.0000001]
 [-0.         -0.         1.          1.0001063]]

Solución:
x1 = 0.9998865, x2 = 1.0000001, x3 = 1.0001063
```

Literal c

In [29]:
```python
# Matriz aumentada
C = np.array([
    [1, 1/2, 1/3, 1/4, 1/6],
    [1/2, 1/3, 1/4, 1/5, 1/7],
    [1/3, 1/4, 1/5, 1/6, 1/8],
    [1/4, 1/5, 1/6, 1/7, 1/9]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", C)

# Gauss-Jordan
n = len(C)
for k in range(n):
    # Normalizar fila pivote
    pivot = C[k,k]
    C[k] = np.divide(C[k], pivot, dtype=np.float32)

    # Eliminación hacia arriba y abajo
    for i in range(n):
        if i != k:
            factor = C[i,k]
            C[i] = np.subtract(C[i], np.multiply(factor, C[k]), dtype=np.float
    print(f"\nPaso {k+1}:\n", C)
```

```python
# Extraer solución
x = C[:, -1]
print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}, x4 = {x[3]:.7f}")
```

```
Matriz inicial:
 [[1.        0.5       0.3333333 0.25      0.1666667]
 [0.5       0.3333333 0.25      0.2       0.1428571]
 [0.3333333 0.25      0.2       0.1666667 0.125    ]
 [0.25      0.2       0.1666667 0.1428571 0.1111111]]

Paso 1:
 [[1.        0.5       0.3333333 0.25      0.1666667]
 [0.        0.0833333 0.0833333 0.075     0.0595238]
 [0.        0.0833333 0.0888889 0.0833333 0.0694444]
 [0.        0.075     0.0833333 0.0803571 0.0694444]]

Paso 2:
 [[ 1.         0.        -0.1666666 -0.2       -0.1904762]
 [ 0.         1.         0.9999998  0.8999999  0.7142857]
 [ 0.         0.         0.0055556  0.0083333  0.0099206]
 [ 0.         0.         0.0083333  0.0128572  0.015873 ]]

Paso 3:
 [[ 1.         0.         0.         0.0499997  0.107142 ]
 [ 0.         1.         0.        -0.5999985 -1.0714242]
 [ 0.         0.         1.         1.4999987  1.7857102]
 [ 0.         0.         0.         0.0003571  0.0009921]]

Paso 4:
 [[ 1.         0.         0.         0.        -0.0317472]
 [ 0.         1.         0.         0.         0.5952536]
 [ 0.         0.         1.         0.        -2.380991 ]
 [ 0.         0.         0.         1.         2.7778032]]

Solución:
x1 = -0.0317472, x2 = 0.5952536, x3 = -2.3809910, x4 = 2.7778032
```

Literal d

In [30]:

```python
# Matriz aumentada
D = np.array([
    [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]
], dtype=np.float32)

print("\n\nMatriz inicial:\n", D)

# Gauss-Jordan con pivoteo parcial
n = len(D)
```

```python
for k in range(n):
    # Pivoteo parcial
    max_row = np.argmax(np.abs(D[k:,k])) + k
    D[[k,max_row]] = D[[max_row,k]]

    # Normalizar fila pivote
    pivot = D[k,k]
    D[k] = np.divide(D[k], pivot, dtype=np.float32)

    # Eliminación hacia arriba y abajo
    for i in range(n):
        if i != k:
            factor = D[i,k]
            D[i] = np.subtract(D[i], np.multiply(factor, D[k]), dtype=np.float
    print(f"\nPaso {k+1}:\n", D)

# Extraer solución
x = D[:, -1]
print("\nSolución:")
print(f"x1 = {x[0]:.7f}, x2 = {x[1]:.7f}, x3 = {x[2]:.7f}, x4 = {x[3]:.7f}, x5
```

```
Matriz inicial:
 [[ 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.]]

Paso 1:
 [[ 1.         0.3333333 -1.3333334  0.         1.6666666  2.       ]
 [ 0.        -0.3333333  3.3333335 -1.        -0.6666666  0.       ]
 [ 0.        -2.        -1.         1.        -1.        -5.       ]
 [ 0.         0.3333333  1.6666667  1.        -6.333333   3.       ]
 [ 0.        -1.3333334  0.3333334 -1.        -0.6666666 -5.       ]]

Paso 2:
 [[ 1.         0.        -1.5        0.1666667  1.5        1.1666666]
 [-0.         1.         0.5       -0.5        0.5        2.5      ]
 [ 0.         0.         3.5000002 -1.1666666 -0.4999999  0.8333334]
 [ 0.         0.         1.5000001  1.1666666 -6.4999995  2.1666667]
 [ 0.         0.         1.        -1.6666667  0.0000001 -1.6666665]]

Paso 3:
 [[ 1.         0.         0.        -0.3333333  1.2857144  1.5238094]
 [-0.         1.         0.        -0.3333333  0.5714285  2.3809524]
 [ 0.         0.         1.        -0.3333333 -0.1428571  0.2380952]
 [ 0.         0.         0.         1.6666666 -6.2857137  1.8095238]
 [ 0.         0.         0.        -1.3333335  0.1428572 -1.9047618]]

Paso 4:
 [[ 1.         0.         0.         0.         0.0285717  1.8857142]
 [ 0.         1.         0.         0.        -0.6857142  2.7428572]
 [ 0.         0.         1.         0.        -1.3999997  0.6      ]
 [ 0.         0.         0.         1.        -3.7714283  1.0857143]
 [ 0.         0.         0.         0.        -4.8857145 -0.4571425]]

Paso 5:
 [[ 1.         0.         0.         0.         0.         1.8830408]
 [ 0.         1.         0.         0.         0.         2.8070176]
 [ 0.         0.         1.         0.         0.         0.730994 ]
 [ 0.         0.         0.         1.         0.         1.4385962]
 [-0.        -0.        -0.        -0.         1.         0.0935672]]

Solución:
x1 = 1.8830408, x2 = 2.8070176, x3 = 0.7309940, x4 = 1.4385962, x5 = 0.0935672
```