Lab 2 : Linear Algebra

Solutions of the system of equations

There are missing fields in the code that you need to fill to get the results but note that you can write you own code to obtain the results

In [2]:

```
## Import the required Libraries here
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
```

Case 1 :

Consider an eqauation Ax=b where A is a Full rank and square martrix, then the solution is given as *x* $_{op}$=A^{-1}*b*, where *x* $_{op}$ is the optimal solution and the error is given as *b* - A*x* $_{op}$

Use the above information to solve the following equatation and compute the error :

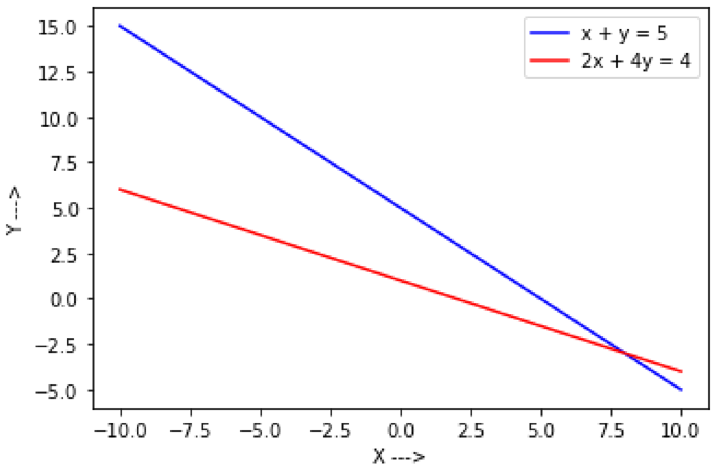
$$x + y = 5$$
$$2x + 4y$$
$$= 4$$

In [3]:

```
# Matrix A and B
A =  np.array([[1,1],[2,4]])
b = np.array([5,4]).reshape(-1,1)
print('A =\n',A,'\n')
print('b =\n',b,'\n')


# Determinant

Det =  np.linalg.det(A) # (A[0][0] * A[1][1] - A[0][1] * A[1][0])
print('Determinant =',Det,'\n')

# Determine the rank of the matrix A
# Since A is a full rank matix, rank = no.of independent rows = no.of rows and same with
the columns as well

rank =  np.linalg.matrix_rank(A) # = A.shape[0] = A.shape[1]
print('Matrix rank =',rank,'\n')

# Determine the Inverse of matrix A
A_inverse =  np.linalg.inv(A)
print('A_inverse =\n',A_inverse,'\n')

# Determine the optimal solution
x_op =  A_inverse @ b
print('x =\n',x_op,'\n')

# Validate the solution by obtaining the error
error =  b - (A @ x_op)
print('error =\n',error,'\n')

# Plotting graphs
x = np.linspace(-10,10,100)
```

```
y1 = b[0] - x
y2 = b[1]/4 - x/2
plt.plot(x,y1,'-b',label = "x + y = 5")
plt.plot(x,y2,'-r', label = '2x + 4y = 4')
plt.xlabel('X --->')
plt.ylabel('Y --->')
plt.legend(loc = "upper right")
plt.show()
```

```
A =
 [[1 1]
 [2 4]]

b =
 [[5]
 [4]]

Determinant = 2.0

Matrix rank = 2

A_inverse =
 [[ 2.  -0.5]
 [-1.   0.5]]

x =
 [[ 8.]
 [-3.]]

error =
 [[0.]
 [0.]]
```



For the following equation :

$$x + y + z = 5$$
$$2x + 4y + z = 4$$
$$x + 3y + 4z = 4$$

Write the code to :

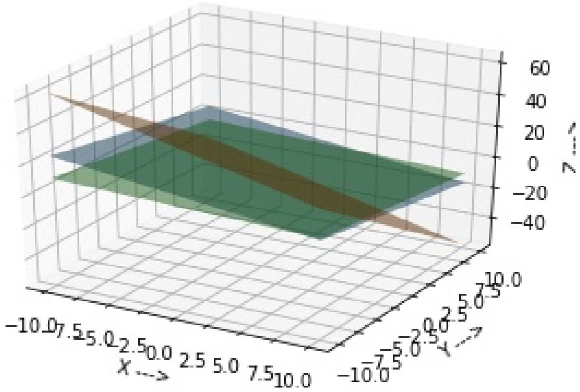
1. **Define Matrices A and B**
2. **Determine the determinant of A**
3. **Determine the rank of A**
4. **Determine the Inverse of matrix A**
5. **Determine the optimal solution**
6. **Plot the equations**
7. **Validate the solution by obataining error**

In [4]:

```python
from numpy.core.function_base import linspace
# Defining matrices A and B
A = np.array([[1,1,1],[2,4,1],[1,3,4]])
b = np.array([5,4,4]).reshape(-1,1)
print("Matrix A:\n\n",A,'\n')
print("Matrix b:\n\n",b,'\n')

# Determining the determinant of A
det = np.linalg.det(A)
print("Determinant of matrix A is = ",det,'\n')

# Determining the rank of matrix A
rank = np.linalg.matrix_rank(A)
print("rank of the matrix A is = ",rank,'\n')

# Determining inverse of matrix A
A_inv = np.linalg.inv(A)
print("Inverse of matrix A is: \n\n",A_inv,'\n')

# Determining the optimal solution
x_op = A_inv @ b
print("Optimal solution of x is ",x_op,'\n')

# Validating the solution by obataining error
err = b - (A @ x_op)
print("Error obtained in the above method is ",err,'\n')


# Plotting the equations
fig = plt.figure(facecolor='w')
ax = fig.gca(projection='3d')
plt.rcParams["figure.figsize"] = [6.00, 3.50]
plt.rcParams["figure.autolayout"] = True

x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X,Y = np.meshgrid(x,y)
Z1 = b[0] - X - Y
Z2 = b[1] - 2*X - 4*Y
Z3 = b[2]/4 - X/4 - 3*Y/4
ax.plot_surface(X,Y,Z1, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z2, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z3, alpha=0.5, rstride=100, cstride=100)

ax.set_xlabel("X --->", linespacing = 3.2)
ax.set_ylabel("Y --->", linespacing = 3.1)
ax.set_zlabel("Z --->", linespacing = 3.5)
ax.dist = 10
plt.show()
```

```
Matrix A:

 [[1 1 1]
 [2 4 1]
 [1 3 4]]

Matrix b:

 [[5]
 [4]
 [4]]

Determinant of matrix A is =  7.999999999999998

rank of the matrix A is =  3

Inverse of matrix A is:

 [[ 1.625 -0.125 -0.375]
 [-0.875  0.375  0.125]
 [ 0.25  -0.25   0.25 ]]
```

```
Optimal solution of x is  [[ 6.125]
 [-2.375]
 [ 1.25 ]]

Error obtained in the above method is  [[0.]
 [0.]
 [0.]]
```



Case 2 :

Consider an eqauation Ax=b where A is a Full rank but it is not a square matrix ($m > n$, dimension of A is $m * n$) , Here if b lies in the span of columns of A then there is unique solution and it is given as *x* $_u$=A^{-1}*b* (here A^{-1} is the pseudo inverse of matrix A), where *x* $_u$ is the unique solution and the error is given as *b* - A*x* $_u$, If b does not lie in the span of columns of A then there are no solutions and the least square solution is given as *x* $_{ls}$=A^{-1}*b* (here A^{-1} is the pseudo inverse of matrix A) and the error is given as *b* - A*x* $_{ls}$

Use the above information solve the following equations and compute the error :

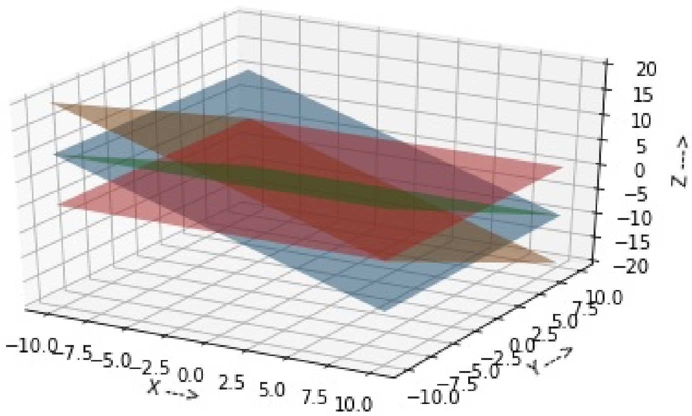
$$x + z = 0$$
$$x + y + z = 0$$
$$y + z = 0$$
$$z = 0$$

In [5]:

```
# Define matrix A and B
A =  np.array([[1,0,1],[1,1,1],[0,1,1],[0,0,1]])
b =  np.zeros((4,1))

print('A=\n',A,'\n')
print('b=\n',b,'\n')

# Determine the rank of matrix A
rank =  np.linalg.matrix_rank(A)
print('Matrix rank=',rank,'\n')

# Determine the pseudo-inverse of A (since A is not Square matrix)
A_inv =  np.linalg.pinv(A)
print('A_inverse=\n',A_inv,'\n')

# Determine the optimal solution
x_u =  A_inv @ b
print('x=\n',x_u,'\n')


# Validate the solution by computing the error
error =  b - (A @ x_u)
print('error=\n',error,'\n')
```

```
# Plot the equations
fig = plt.figure(facecolor='w')
ax = fig.gca(projection='3d')
plt.rcParams["figure.figsize"] = [6.00, 3.50]
plt.rcParams["figure.autolayout"] = True

x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X,Y = np.meshgrid(x,y)
Z1 = b[0] - X
Z2 = b[1] - X - Y
Z3 = b[2] - Y
Z4 = np.zeros(Y.shape)
ax.plot_surface(X,Y,Z1, alpha=0.5, rstride=100, cstride=100,fc = 'r')
ax.plot_surface(X,Y,Z2, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z3, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z4, alpha=0.5, rstride=100, cstride=100)
ax.set_xlabel("X --->", linespacing = 3.2)
ax.set_ylabel("Y --->", linespacing = 3.1)
ax.set_zlabel("Z --->", linespacing = 3.5)
ax.dist = 10
plt.show()
```

```
A=
 [[1 0 1]
 [1 1 1]
 [0 1 1]
 [0 0 1]]

b=
 [[0.]
 [0.]
 [0.]
 [0.]]

Matrix rank= 3

A_inverse=
 [[ 0.5   0.5  -0.5  -0.5 ]
 [-0.5   0.5   0.5  -0.5 ]
 [ 0.25 -0.25  0.25  0.75]]

x=
 [[0.]
 [0.]
 [0.]]

error=
 [[0.]
 [0.]
 [0.]
 [0.]]
```



For the following equation :

$$x + y + z = 35$$
$$2x + 4y + z = 94$$
$$x + 3y + 4z = 4$$
$$x + 9y + 4z = -230$$

Write the code to :

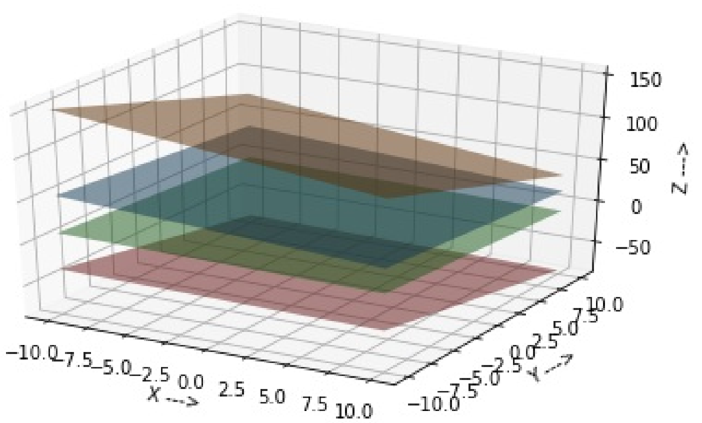
1. **Define Matrices A and B**
2. **Determine the rank of A**
3. **Determine the Pseudo Inverse of matrix A**
4. **Determine the optimal solution**
5. **Plot the equations**
6. **Validate the solution by obataining error**

In [6]:

```
from numpy.core.function_base import linspace
# Defining matrices A and B
A = np.array([[1,1,1],[2,4,1],[1,3,4],[1,9,4]])
b = np.array([35,94,4,-230]).reshape(-1,1)
print("Matrix A:\n\n",A,'\n')
print("Matrix b:\n\n",b,'\n')


# Determining the rank of matrix A
rank = np.linalg.matrix_rank(A)
print("rank of the matrix A is = ",rank,'\n')

# Determining Pseudo inverse of matrix A
A_inv = np.linalg.pinv(A)
print("Inverse of matrix A is: \n\n",A_inv,'\n')

# Determining the optimal solution
x_ls = A_inv @ b
print("Optimal solution of x is ",x_ls,'\n')

# Validating the solution by obataining error
err = b - (A @ x_ls)
print("Error obtained in the above method is ",err,'\n')


# Plotting the equations

fig = plt.figure(facecolor='w')
ax = fig.gca(projection='3d')
plt.rcParams["figure.figsize"] = [6.00, 3.50]
plt.rcParams["figure.autolayout"] = True

x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X,Y = np.meshgrid(x,y)
Z1 = b[0] - X - Y
Z2 = b[1] - 2*X - 4*Y
Z3 = b[2]/4 - X/4 - 3*Y/4
Z4 = (b[3] - X - 9*Y)/4

ax.plot_surface(X,Y,Z1, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z2, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z3, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z4, alpha=0.5, rstride=100, cstride=100)

ax.set_xlabel("X --->", linespacing = 3.2)
ax.set_ylabel("Y --->", linespacing = 3.1)
ax.set_zlabel("Z --->", linespacing = 3.5)
ax.dist = 10
```

```
plt.show()
```

```
Matrix A:

 [[1 1 1]
 [2 4 1]
 [1 3 4]
 [1 9 4]]

Matrix b:

 [[  35]
 [  94]
 [   4]
 [-230]]

rank of the matrix A is =  3

Inverse of matrix A is:

 [[ 0.27001704  0.45570698  0.07666099 -0.25809199]
 [-0.06558773  0.02810903 -0.14480409  0.15417376]
 [ 0.04429302 -0.16183986  0.31856899 -0.03918228]]

Optimal solution of x is  [[111.9548552 ]
 [-35.69250426]
 [ -3.37649063]]

Error obtained in the above method is  [[-37.88586031]
 [ 16.23679727]
 [ 12.6286201 ]
 [ -7.21635434]]
```



Case 3 :

Consider an eqauation Ax=b where A is not a Full rank matrix, Here if b lies in the span of columns of A then there are multiple solutions and one of the solution is given as *x*$_u$=A^{-1}*b* (here A^{-1} is the pseudo inverse of matrix A), the error is given as *b* - A*x*$_u$, If b does not lie in the span of columns of A then there are no solutions and the least square solution is given as *x*$_{ls}$=A^{-1}*b* (here A^{-1} is the pseudo inverse of matrix A) and the error is given as *b* - A*x*$_{ls}$

Use the above information solve the following equations and compute the error :

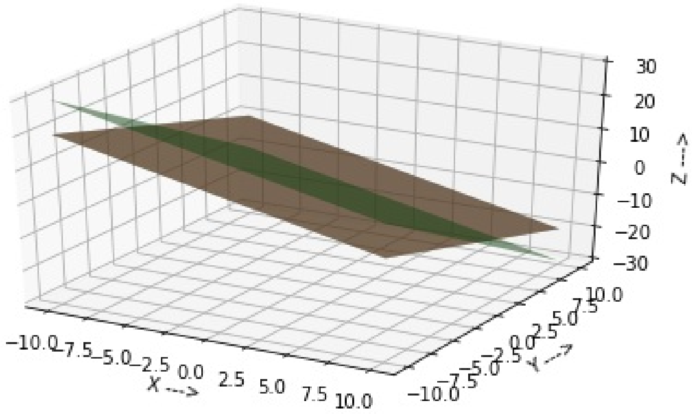
$$x + y + z = 0$$
$$3x + 3y + 3z = 0$$
$$x + 2y + z = 0$$

In [7]:

```
# Define matrix A and B
A =  np.array([[1,1,1],[3,3,3],[1,2,1]])
b =  np.zeros((3,1))
print('A=\n',A,'\n')
print('b=\n',b,'\n')

# Determine the rank of matrix A
rank =  np.linalg.matrix_rank(A)
print('Matrix rank=',rank,'\n')

# Determine the pseudo-inverse of A (since A is not Square matrix)
A_inv =  np.linalg.pinv(A)
print('A_inverse=\n',A_inv,'\n')

# Determine the optimal solution
x_u =  A_inv @ b
print('x=\n',x_u,'\n')

# Validate the solution by computing the error
error =  b - (A_inv @ x_u)
print('error=\n',error,'\n')

# Plot the equations

fig = plt.figure(facecolor='w')
ax = fig.gca(projection='3d')
plt.rcParams["figure.figsize"] = [6.00, 3.50]
plt.rcParams["figure.autolayout"] = True

x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X,Y = np.meshgrid(x,y)
Z1 = b[0] - X - Y
Z2 = b[1] - X - Y
Z3 = b[2] - X - 2*Y

ax.plot_surface(X,Y,Z1, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z2, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z3, alpha=0.5, rstride=100, cstride=100)

ax.set_xlabel("X --->", linespacing = 3.2)
ax.set_ylabel("Y --->", linespacing = 3.1)
ax.set_zlabel("Z --->", linespacing = 3.5)
ax.dist = 10
plt.show()
```

```
A=
 [[1 1 1]
 [3 3 3]
 [1 2 1]]

b=
 [[0.]
 [0.]
 [0.]]

Matrix rank= 2

A_inverse=
 [[ 0.1  0.3 -0.5]
 [-0.1 -0.3  1. ]
 [ 0.1  0.3 -0.5]]

x=
 [[0.]
 [0.]
 [0.]]

error=
 [[0.]
```

```
 [0.]
 [0.]]
```



For the following equation :

$$x + y + z = 0$$
$$3x + 3y + 3z = 2$$
$$x + 2y + z = 0$$

Write the code to :

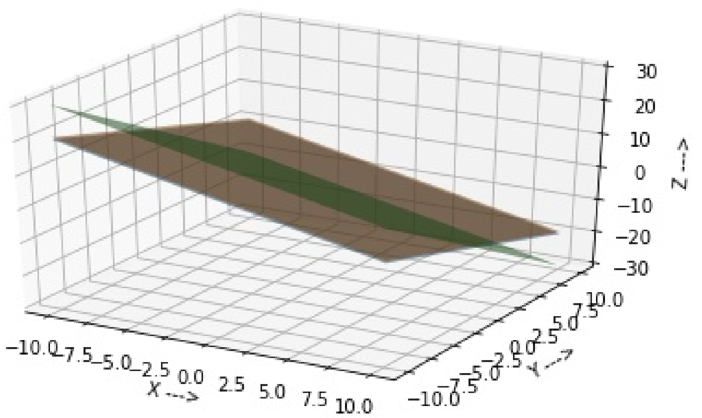
1. **Define Matrices A and B**
2. **Determine the rank of A**
3. **Determine the Pseudo Inverse of matrix A**
4. **Determine the optimal solution**
5. **Plot the equations**
6. **Validate the solution by obataining error**

In [8]:

```
from numpy.core.function_base import linspace
# Defining matrices A and B
A =  np.array([[1,1,1],[3,3,3],[1,2,1]])
b =  np.array([0,2,0]).reshape(-1,1)
print("Matrix A:\n\n",A,'\n')
print("Matrix b:\n\n",b,'\n')


# Determining the rank of matrix A
rank = np.linalg.matrix_rank(A)
print("rank of the matrix A is = ",rank,'\n')

# Determining Pseudo inverse of matrix A
A_inv = np.linalg.pinv(A)
print("Inverse of matrix A is: \n\n",A_inv,'\n')

# Determining the optimal solution
x_ls = A_inv @ b
print("Optimal solution of x is ",x_ls,'\n')

# Validating the solution by obataining error
err = b - (A @ x_ls)
print("Error obtained in the above method is ",err,'\n')


# Plotting the equations
fig = plt.figure(facecolor='w')
ax = fig.gca(projection='3d')
plt.rcParams["figure.figsize"] = [6.00, 3.50]
plt.rcParams["figure.autolayout"] = True
```

```
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X,Y = np.meshgrid(x,y)
Z1 = b[0]- X - Y
Z2 = b[1]/3 - X - Y
Z3 = b[2]- X - 2*Y

ax.plot_surface(X,Y,Z1, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z2, alpha=0.5, rstride=100, cstride=100)
ax.plot_surface(X,Y,Z3, alpha=0.5, rstride=100, cstride=100)

ax.set_xlabel("X --->", linespacing = 3.2)
ax.set_ylabel("Y --->", linespacing = 3.1)
ax.set_zlabel("Z --->", linespacing = 3.5)
ax.dist = 10
plt.show()
```

```
Matrix A:

 [[1 1 1]
 [3 3 3]
 [1 2 1]]

Matrix b:

 [[0]
 [2]
 [0]]

rank of the matrix A is =  2

Inverse of matrix A is:

 [[ 0.1  0.3 -0.5]
 [-0.1 -0.3  1. ]
 [ 0.1  0.3 -0.5]]

Optimal solution of x is  [[ 0.6]
 [-0.6]
 [ 0.6]]

Error obtained in the above method is  [[-6.0000000e-01]
 [ 2.0000000e-01]
 [-8.8817842e-16]]
```



Examples

Find the solution for the below equations and justify the case that they belong to

1. $2x + 3y$
$+ 5z$
$= 2, 9x$

$+3y$
$+2z$
$=5,5x$
$+9y$
$+z=7$

2.$2x+3y$
$=1,5x$
$+9y$
$=4,x$
$+y=0$

3.$2x+5y$
$+10z$
$=0,9x$
$+2y$
$+z$
$=1,4x$
$+10y$
$+20z$
$=5$

4.$2x+3y$
$=0,5x$
$+9y$
$=2,x$
$+y=$
-2

5.$2x+5y$
$+3z$
$=0,9x$
$+2y$
$+z$
$=0,4x$
$+10y$
$+6z$
$=0$

In [9]:

```
def is_square_matrix(A):
    if A.shape[0] == A.shape[1]:
        return True
    else:
        return False

def is_Full_rank_matrix(A):
    rank = np.linalg.matrix_rank(A)
    if rank == min(A.shape[0],A.shape[1]):
        return True
    else:
        return False

def which_case(A):
    if is_square_matrix(A) and is_Full_rank_matrix(A):
        return 1
    elif is_Full_rank_matrix(A):
        return 2
    if not(is_Full_rank_matrix(A)):
        return 3
```

```python
def solve_equation(A,b):
    if which_case(A) == 1:
        x = np.linalg.solve(A,b)
        return x
    else:
        x = np.linalg.pinv(A) @ b
        return x
```

In [10]:

```python
# 1.
A = np.array([[2,3,5],[9,3,2],[5,9,1]])
b = np.array([2,5,7])
print(f'This matrix A belongs to case {which_case(A)}')
print(f'The solution for the above system of equations is {solve_equation(A,b)}')
```

```
This matrix A belongs to case 1
The solution for the above system of equations is [ 0.38613861  0.57425743 -0.0990099 ]
```

In [11]:

```python
# 2.
A = np.array([[2,3],[5,9],[1,1]])
b = np.array([1,4,0])
print(f'This matrix A belongs to case {which_case(A)}')
print(f'The solution for the above system of equations is {solve_equation(A,b)}')
```

```
This matrix A belongs to case 2
The solution for the above system of equations is [-1.  1.]
```

In [12]:

```python
# 3.
A = np.array([[2,5,10],[9,2,1],[4,10,20]])
b = np.array([0,1,5])
print(f'This matrix A belongs to case {which_case(A)}')
print(f'The solution for the above system of equations is {solve_equation(A,b)}')
```

```
This matrix A belongs to case 3
The solution for the above system of equations is [0.07720207 0.08041451 0.14435233]
```

In [13]:

```python
# 4.
A = np.array([[2,3],[5,9],[1,1]])
b = np.array([0,2,-2])
print(f'This matrix A belongs to case {which_case(A)}')
print(f'The solution for the above system of equations is {solve_equation(A,b)}')
```

```
This matrix A belongs to case 2
The solution for the above system of equations is [-4.          2.46153846]
```

In [14]:

```python
# 5.
A = np.array([[2,5,3],[9,2,1],[4,10,6]])
b = np.array([0,0,0])
print(f'This matrix A belongs to case {which_case(A)}')
print(f'The solution for the above system of equations is {solve_equation(A,b)}')
```

```
This matrix A belongs to case 3
The solution for the above system of equations is [0. 0. 0.]
```