Function to implement Gauss Elimination With and Without pivoting.

In [47]:

```
import math
from functools import partial
import random

def round_d_significant(value, significant_digits):
    if value == 0:
        return 0
    return round(value, significant_digits - int(math.floor(math.log10(abs(value))))


def rank_of_matrix(matrix):
    rank = 0
    for row in matrix:
        if sum(row) != 0:
            rank += 1
    return rank


def gauss_elimination(A: list, b: list, partial_pivoting: bool = False, d: int = 3):
    """
    A: A square matrix n*n
    b: A vecor of size n
    partial_pivoting: Whether to perform partial pivoting or not
    d: number of significant digits to be rounded to
    """
    assert len(A) == len(b) # No of equations is equal to length of vector
    for row in A:
        assert len(row) == len(A) # Assert square matrix

    rounding = partial(round_d_significant, significant_digits = d)

    # Create augmented matrix
    for i in range(len(A)):
        A[i].append(b[i])

    # Bringing the matrix to reduced echelon form (REF)
    for index_row in range(0, len(A)-1):

        # Partial pivoting
        if partial_pivoting:
            cur_max = abs(A[index_row][index_row])
            cur_max_row = index_row
            # Check if pivoting need to be done
            for i in range(index_row + 1, len(A)):
                if abs(A[i][index_row]) > cur_max:
                    cur_max = abs(A[i][index_row]) # Absolute value considered for p
                    cur_max_row = i
            # If pivoting need to be done
            if cur_max_row != index_row:
                temp = A[index_row]
                A[index_row] = A[cur_max_row]
                A[cur_max_row] = temp

        for i in range(index_row + 1, len(A)):
            # Skip the rwo transform if the value is already zero
            if A[i][index_row] == 0:
                continue

            # Calculate the coefficient to mutiply with the index row
            scaler = rounding(A[i][index_row] / A[index_row][index_row])
```

```
            if (A[i][index_row] < 0 and A[index_row][index_row] < 0) or ((A[i][index
                scaler = -scaler

            # Assign all the prior values to zero
            for j in range(0, index_row + 1):
                A[i][j] = 0

            # Compute rest of the values in the row
            for j in range(index_row + 1, len(A[i])):
                A[i][j] = rounding(A[i][j] + (scaler * A[index_row][j]))


    # Back substitution

    # Initialize None for all variables
    variable_values = {}
    for i in range(len(A)):
        variable_values[f'x{i}'] = None

    # Assign arbitrary values if needed
    no_aug_A = [[A[i][j] for j in range(len(A[i]) - 1)] for i in range(len(A))]
    if rank_of_matrix(no_aug_A) != rank_of_matrix(A):
        print("The system is inconsistent!!")
        return
    else:
        if rank_of_matrix(A) < len(A):
            for i in range(len(A)-1, rank_of_matrix(A)-1, -1):
                variable_values[f'x{i}'] = rounding(random.random())

        # Solve different equations for different variables
        for i in range(rank_of_matrix(no_aug_A) - 1, -1, -1):
            known_coeffs = [A[i][j] * variable_values[f'x{j}'] for j in range(i+1, l
            known_coeffs = [rounding(v) for v in known_coeffs]
            rhs = rounding((A[i][-1] - rounding(sum(known_coeffs))))
            variable_values[f'x{i}'] =  rounding(rhs / A[i][i])

    return variable_values

print(gauss_elimination(A=[[3, 7, 2], [4, 1, 5], [1, 3, 2]], b=[5, 4, 2], partial_pi
print(gauss_elimination(A=[[3, 7, 2], [4, 1, 5], [1, 3, 2]], b=[5, 4, 2], partial_pi
```

```
{'x0': 0.2903, 'x1': 0.4536, 'x2': 0.4771}
{'x0': 0.8157, 'x1': 0.3421, 'x2': 0.07895}
```

In [ ]:

```

```

Code to count number of additions, multiplications & divisions

In [45]:

```
def gauss_operation_count(n: int):
    """
    n: Number of row / columns in a square matrix
    """
    # Operation count for addition
    ref_addition = (n * (n+1) * (2 * n + 1)) / 6 # Number of addition for REF
    backsub_addition = (n * (n - 1)) / 2
    total_addition = ref_addition + backsub_addition

    # Operation count for multiplication
    total_multiplication = total_addition # Same as addition

    # Operation count for division
    ref_division = (n * (n-1)) / 2
```

```
    backsub_division = n
    total_division = ref_division + backsub_division

    # Summarize
    operation_count = {
        'addition': total_addition,
        'multiplication': total_multiplication,
        'division': total_division
    }

    return operation_count

gauss_operation_count(3)
```

Out[45]:
```
{'addition': 17.0, 'multiplication': 17.0, 'division': 6.0}
```

In [ ]: