Programming Assignment - Gaussian Elimination

Welcome to the programming assignment on Gaussian Elimination! In this assignment, you will implement the Gaussian elimination method, a foundational algorithm for solving systems of linear equations.

Linear algebra is fundamental to machine learning, serving as the basis for numerous algorithms. Gaussian elimination, while not the most advanced method used today, is a classical and essential technique for solving systems of linear equations. It provides valuable insights into the core principles of linear algebra and lays the groundwork for more advanced numerical methods.

Why should you care?

- **Foundational Skills**: Strengthen your understanding of key linear algebra concepts.
- **Programming Practice**: Enhance your coding skills by implementing a classical mathematical algorithm.
- **Historical Significance**: Gaussian elimination, though not the most cutting-edge method today, is historically significant and provides a solid starting point for understanding the evolution of linear algebra techniques.

Outline

1 - Introduction

Gaussian Elimination Algorithm

Gaussian elimination offers a systematic approach to solving systems of linear equations by transforming an augmented matrix into row-echelon form, thereby enabling the determination of variables. The algorithm comprises several essential steps:

Step 1: Augmented Matrix

Consider a system of linear equations:

$$\begin{aligned} 2x_1 + 3x_2 + 5x_3 &= 12 \\ -3x_1 - 2x_2 + 4x_3 &= -2 \\ x_1 + x_2 - 2x_3 &= 8 \end{aligned}$$

Create the augmented matrix ([A | B]), where (A) represents the coefficient matrix and (B) denotes the column vector of constants:

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

$$B = \begin{bmatrix} 12 \\ -2 \\ 8 \end{bmatrix}$$

Thus, ([A | B]) is represented as:

$$\begin{bmatrix} 2 & 3 & 5 & | & 12 \\ -3 & -2 & 4 & | & -2 \\ 1 & 1 & -2 & | & 8 \end{bmatrix}$$

Note: For this assignment, matrix (A) is **always square**, accommodating scenarios with (n) equations and (n) variables.

Step 2: Transform Matrix into Reduced Row Echelon Form

Initiate row operations to convert the augmented matrix into row-echelon form. The objective is to introduce zeros below the leading diagonal.

- **Row Switching:** Rearrange rows to position the leftmost non-zero entry at the top.

- **Row Scaling:** Multiply a row by a non-zero scalar.
- **Row Replacement:** Substitute a row with the sum of itself and a multiple of another row.

Step 3: Solution Check

Examine for a row of zeros in the square matrix (excluding the augmented part).

Consider the following cases:

1. If no row comprises zeros, a **unique solution** exists.
2. If one row contains zeros with a non-zero augmented part, the system has **no solutions**.
3. If every row of zeros has a zero augmented part, the system boasts **infinitely many solutions**.

Special attention is required to address conditions 2 and 3. A matrix might contain one row of zeros with an augmented part of 0 and another row with zeros and a non-zero augmented part, making the system impossible.

Step 5: Back Substitution

After attaining the reduced row-echelon form, solve for variables starting from the last row and progressing upwards.

Remember, the aim is to simplify the system for easy determination of solutions!

Step 6: Compile the Gaussian Elimination Algorithm

Combine each function related to the aforementioned steps into a single comprehensive function.

2 - Necessary imports

Next codeblock will import the necessary libraries to run this assignment. Please do not add nor remove any value there.

In [1]:
```
import numpy as np
```

In [2]:
```
import w2_unittest
```

3 - Auxiliary functions

This section introduces five auxiliary functions crucial for facilitating your assignment. These functions have already been coded, eliminating the need for your concern regarding their implementation. However, it's essential to examine them carefully to grasp their appropriate usage.

Note: In Python, indices commence at 0 rather than 1. Therefore, a matrix with n rows is indexed as $0, 1, 2, \ldots, n-1$.

3.1 - Function swap rows

This function has as input a numpy array and two indexes to swap the rows corresponding to those indexes. It **does not change the original matrix**, but returns a new one.

In [3]:
```
def swap_rows(M, row_index_1, row_index_2):
    """
    Swap rows in the given matrix.

    Parameters:
    - matrix (numpy.array): The input matrix to perform row swaps on.
    - row_index_1 (int): Index of the first row to be swapped.
    - row_index_2 (int): Index of the second row to be swapped.
    """

    # Copy matrix M so the changes do not affect the original matrix.
    M = M.copy()
    # Swap indexes
    M[[row_index_1, row_index_2]] = M[[row_index_2, row_index_1]]
    return M
```

Let's practice with some examples. Consider the following matrix M.

In [4]:
```
M = np.array([
[1, 3, 6],
[0, -5, 2],
[-4, 5, 8]
])
print(M)
```

```
[[ 1  3  6]
 [ 0 -5  2]
 [-4  5  8]]
```

Swapping row 0 with row 2:

In [5]:
```
M_swapped = swap_rows(M, 0, 2)
print(M_swapped)
```

```
[[-4  5  8]
 [ 0 -5  2]
 [ 1  3  6]]
```

3.2 - Finding the first non-zero value in a column starting from a specific value

This function becomes essential when encountering a 0 value during row operations. It determines whether a non-zero value exists below the encountered zero, allowing for potential row swaps. Consider the following scenario within a square matrix (non-augmented):

Let's say, during a specific step of the row-echelon form process, you've successfully reduced the first 2 rows, but you encounter a zero pivot (highlighted in red) in the third row. The task is to search, **solely in entries below the pivot**, for a potential row swap.

$$\begin{bmatrix} 6 & 4 & 8 & 1 \\ 0 & 8 & 6 & 4 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 5 & 9 \end{bmatrix}$$

Performing a row swap between indexes 2 and 3 (remember, indexing starts at 0!), the matrix transforms into:

$$\begin{bmatrix} 6 & 4 & 8 & 1 \\ 0 & 8 & 6 & 4 \\ 0 & 0 & 5 & 9 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

Resulting in the matrix achieving the row-echelon form.

In [14]:

```
def get_index_first_non_zero_value_from_column(M, column, starting_row):
    """
    Retrieve the index of the first non-zero value in a specified column of the given ma

    Parameters:
    - matrix (numpy.array): The input matrix to search for non-zero values.
    - column (int): The index of the column to search.
    - starting_row (int): The starting row index for the search.

    Returns:
    int or None: The index of the first non-zero value in the specified column, starting
                 Returns None if no non-zero value is found.
    """
    # Get the column array starting from the specified row
    column_array = M[starting_row:,column]

    for i, val in enumerate(column_array):
        # Iterate over every value in the column array.
        # To check for non-zero values, you must always use np.isclose instead of doing
        if not np.isclose(val, 0, atol = 1e-5):
            # If one non zero value is found, then adjust the index to match the correc
            index = i + starting_row
            return index
    # If no non-zero value is found below it, return None.
    return None
```

Let's practice with this function. Consider the following matrix.

In [15]:

```
N = np.array([
[0, 5, -3 ,6 ,8],
[0, 6, 3, 8, 1],
[0, 0, 0, 0, 0],
[0, 0, 0 ,1 ,7],
[0, 2, 1, 0, 4]
]
)
print(N)
```

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

If you search for a value below the first column starting at the first row, the function should return None:

In [16]:
```
print(get_index_first_non_zero_value_from_column(N, column = 0, starting_row = 0))
```

```
None
```

Searching for the first non zero value in the last column starting from row with index 2, it should return 3 (index corresponding to the value 7).

In [17]:
```
print(get_index_first_non_zero_value_from_column(N, column = -1, starting_row = 2))
```

```
3
```

3.3 - Find the pivot for any row

This function aids in locating the pivot within a designated row of a matrix. It identifies the index of the first non-zero element in the desired row. If no non-zero value is present, it returns None.

In [18]:
```
def get_index_first_non_zero_value_from_row(M, row):
    """
    Find the index of the first non-zero value in the specified row of the given matrix

    Parameters:
    - matrix (numpy.array): The input matrix to search for non-zero values.
    - row (int): The index of the row to search.

    Returns:
    int or None: The index of the first non-zero value in the specified row.
                Returns None if no non-zero value is found.
    """
    # Get the desired row
    row_array = M[row]
    for i, val in enumerate(row_array):
        # If finds a non zero value, returns the index. Otherwise returns None.
        if not np.isclose(val, 0, atol = 1e-5):
            return i
    return None
```

Let's practice with the same matrix as before:

In [19]:
```
print(N)
```

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

Looking for the first non-zero index in row 2 must return None whereas in row 3, the value returned must be 3 (the index for the value 1 in that row).

In [20]:

```
print(f'Output for row 2: {get_index_first_non_zero_value_from_row(N, 2)}')
print(f'Output for row 3: {get_index_first_non_zero_value_from_row(N, 3)}')
```

```
Output for row 2: None
Output for row 3: 3
```

3.4 Moving one row to the bottom

This function facilitates the movement of a specific row to the bottom. Such an operation becomes necessary when confronted with a row entirely populated by zeroes. In reduced row-echelon form, rows filled with zeroes must be positioned at the bottom.

In [21]:

```
def move_row_to_bottom(M, row_index):
    """
    Move the specified row to the bottom of the given matrix.

    Parameters:
    - M (numpy.array): Input matrix.
    - row_index (int): Index of the row to be moved to the bottom.

    Returns:
    - numpy.array: Matrix with the specified row moved to the bottom.
    """

    # Make a copy of M to avoid modifying the original matrix
    M = M.copy()

    # Extract the specified row
    row_to_move = M[row_index]

    # Delete the specified row from the matrix
    M = np.delete(M, row_index, axis=0)

    # Append the row at the bottom of the matrix
    M = np.vstack([M, row_to_move])

    return M
```

One small example:

In [22]:

```
M = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

print(f'Matrix before:\n{M}')
print(f'Matrix after moving index 1:\n{move_row_to_bottom(M, 1)}')
```

```
Matrix before:
[[1 2 3]
```

```
 [4 5 6]
 [7 8 9]]
Matrix after moving index 1:
[[1 2 3]
 [7 8 9]
 [4 5 6]]
```

3.5 - Constructing the Augmented Matrix

This function constructs the augmented matrix by combining a square matrix of size $n \times n$, representing n equations with n variables each, with an $n \times 1$ matrix that denotes its constant values. The function concatenates both matrices to form the augmented matrix and returns the result.

In [23]:

```
def augmented_matrix(A, B):
    """
    Create an augmented matrix by horizontally stacking two matrices A and B.

    Parameters:
    - A (numpy.array): First matrix.
    - B (numpy.array): Second matrix.

    Returns:
    - numpy.array: Augmented matrix obtained by horizontally stacking A and B.
    """
    augmented_M = np.hstack((A,B))
    return augmented_M
```

In [24]:

```
A = np.array([[1,2,3], [3,4,5], [4,5,6]])
B = np.array([[1], [5], [7]])

print(augmented_matrix(A,B))
```

```
[[1 2 3 1]
 [3 4 5 5]
 [4 5 6 7]]
```

4 - Row echelon form and reduced row echelon form

4.1 - Row Echelon Form

As discussed in the lectures, a matrix in row echelon form adheres to the following conditions:

- Rows consisting entirely of zeroes should be positioned at the bottom.
- Each non-zero row must have its left-most non-zero coefficient (termed as a **pivot**) located to the right of any row above it. Consequently, all elements below the pivot within the same column should be 0.

This form ensures a structured arrangement facilitating subsequent steps in the Gaussian elimination process.

Example of matrix **in row echelon form**

$$M = \begin{bmatrix} 7 & 2 & 3 \\ 0 & 9 & 4 \\ 0 & 0 & 3 \end{bmatrix}$$

Examples of matrices that **are not in row echelon form**

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 0 & 5 & 3 \\ 1 & 0 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 7 \end{bmatrix}$$

Matrix A fails to satisfy the criteria for row echelon form as there exists a non-zero element below the first pivot (located in row 0). Similarly, matrix B does not meet the requirements as the second pivot (in row 1 with a value of 4) has a non-zero element below it.

4.2 - worked example

In this section, you'll revisit an example from the lecture to facilitate the implementation of an algorithm. If you feel confident in proceeding with the algorithm, you may skip this section.

Consider matrix M given by:

$$M = \begin{bmatrix} * & * & * \\ 0 & \text{pivot} & * \\ 0 & \text{value} & * \end{bmatrix}$$

Here, the asterisk (*) denotes any number. To nullify the last row (row 2), two steps are required:

- Scale R_1 by the inverse of the pivot:

$$\text{Row 1} \to \frac{1}{\text{pivot}} \cdot \text{Row}$$

Resulting in the updated matrix with the pivot for row 1 set to 1:

$$M = \begin{bmatrix} * & * & * \\ 0 & 1 & * \\ 0 & \text{value} & * \end{bmatrix}$$

Next, to eliminate the value below the pivot in row 1, apply the following formula:

$$\text{Row 2} \to \text{Row 2} - \text{value} \cdot \text{Row 1}$$

This transformation yields the modified matrix:

$$M = \begin{bmatrix} * & * & * \\ 0 & 1 & * \\ 0 & 0 & * \end{bmatrix}$$

Note that the square matrix A needs to be in reduced row-echelon form. However, every row operation conducted must also affect the augmented (constant) part. This ensures that you are effectively preserving the solutions for the entire system!

Let's review the example covered in the lecture.

$$\begin{aligned} 2x_2 + x_3 &= 3 \\ x_1 + x_2 + x_3 &= 6 \\ x_1 + 2x_2 + 1x_3 &= 12 \end{aligned}$$

Consequently, the square matrix A is formulated as:

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

The column vector (a matrix of size $n \times 1$) is represented by:

$$B = \begin{bmatrix} 3 \\ 6 \\ 12 \end{bmatrix}$$

Combining matrices A and B yields the augmented matrix M:

$$M = \begin{bmatrix} 0 & 2 & 1 & | & 3 \\ 1 & 1 & 1 & | & 6 \\ 1 & 2 & 1 & | & 12 \end{bmatrix}$$

Step 1:

Commencing with row 0: The initial candidate for the pivot is always the value in the main diagonal of the matrix. Denoting row 0 as R_0:

$$R_0 = \begin{bmatrix} 0 & 2 & 1 & | & 3 \end{bmatrix}$$

The value in the main diagonal is the element $M[0,0]$ (the first element of the first column). The first row can be accessed by performing $M[0]$, i.e., $M[0] = R_0$.

The first row operation involves **scaling by the pivot's inverse**. Since the value in the main diagonal is 0, necessitating a non-zero value for scaling by its inverse, you must switch rows in this case. Note that R_1 has a value different from 0 in the required index. Consequently, switching rows 0 and 1:

$$R_0 \to R_1$$

$$R_1 \to R_0$$

Resulting in the updated augmented matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & | & 6 \\ 0 & 2 & 1 & | & 3 \\ 1 & 2 & 1 & | & 12 \end{bmatrix}$$

Now, the pivot is already 1, eliminating the need for row scaling. Following the formula:

$$R_1 \to R_1 - 0 \cdot R_0 = R_1$$

Therefore, the second row remains unchanged. Moving to the third row (R_2), the value in the augmented matrix below the pivot from R_0 is $M[2,0]$, which is 1.

$$R_2 = R_2 - 1 \cdot R_0 = \begin{bmatrix} 0 & 1 & 0 & | & 6 \end{bmatrix}$$

Resulting in the modified augmented matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & | & 6 \\ 0 & 2 & 1 & | & 3 \\ 0 & 1 & 0 & | & 6 \end{bmatrix}$$

Progressing to the second row (R_1), the value in the main diagonal is 2, different from zero. Scaling it by $\frac{1}{2}$:

$$R_1 = \frac{1}{2} R_1$$

Resulting in the augmented matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & | & 6 \\ 0 & 1 & \frac{1}{2} & | & \frac{3}{2} \\ 0 & 1 & 0 & | & 6 \end{bmatrix}$$

Now, there's only one row below it for row replacement. The value just below the pivot is located at $M[2,1]$, which is 1. Thus:

$$R_2 = R_2 - 1 \cdot R_1 = \begin{bmatrix} 0 & 0 & -\frac{1}{2} & | & \frac{9}{2} \end{bmatrix}$$

Resulting in the augmented matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & | & 6 \\ 0 & 1 & \frac{1}{2} & | & \frac{3}{2} \\ 0 & 0 & -\frac{1}{2} & | & \frac{9}{2} \end{bmatrix}$$

Thus, the matrix is now in reduced row echelon form.

4.3 Handling Non-Diagonal Pivots

In some cases, matrices may lack pivots exclusively in the main diagonal, necessitating special handling within the algorithm. To simplify, let's focus solely on the square matrix, disregarding the augmented part:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 7 \\ 0 & 0 & 5 \end{bmatrix}$$

The process initiates typically, beginning with $R_0 = M[0]$ and the pivot as $M[0,0]$. Notably, the pivot is already 1, and all values below it are already 0, indicating that after row normalization and reduction, the outcome remains unchanged. Moving to $R_1 = M[1]$, the pivot candidate, $M[1,1]$, is 0, rendering the standard procedure impossible due to division by zero during row scaling. Thus, the following steps are necessary:

1. Explore another row below the current row to identify a row with a non-zero value in the same column as the value $M[1,1]$. In the given example, there's only one such row, so examining the value right below it, $M[2,1]$, reveals another 0, impeding row swapping. Consequently, the next step becomes vital.

2. Given the failure of step 1, search within the row for the first non-zero number, which becomes the new pivot. If no such number exists, it signifies a row entirely populated by zeroes, to be shifted to the matrix's last row (recall that in reduced row echelon form, rows filled with 0's reside at the bottom). In the current case, the first non-zero value in R_1 is 7, i.e., position 2 in that row. So the new pivot index is not in the diagonal but after it, i.e., $M[1,2]$.

In the current scenario, the initial non-zero value in R_1 is 7, specifically in position 2 within that row. Thus, the new pivot index lies beyond the diagonal, at $M[1,2]$.

$$R_1 = \frac{1}{7} R_1$$

Resulting in the matrix after normalization:

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

Next, scrutinize every value below this new pivot. In row 2, the considered value is $M[2,2]$, derived from the pivot position in the preceding step:

$$R_2 = R_2 - 5 \cdot R_1$$

Leading to:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Hence, the matrix is now in reduced row echelon form.

Now you are ready to go! You will implement such algorithm in the following exercise.

Exercise 1

This exercise involves implementing the elimination method to convert a matrix into row-echelon form. As discussed in lectures, the primary approach involves inspecting the values along the diagonal. If they equate to 0, an attempt to swap rows should be made to obtain a non-zero value.

In [ ]:
```
# GRADED FUNCTION: reduced_row_echelon_form

def reduced_row_echelon_form(A, B):
    """
    Utilizes elementary row operations to transform a given set of matrices,
    which represent the coefficients and constant terms of a linear system,
    into reduced row echelon form.

    Parameters:
    - A (numpy.array): The input square matrix of coefficients.
    - B (numpy.array): The input column matrix of constant terms

    Returns:
    numpy.array: A new augmented matrix in reduced row echelon form.
    """
    # Make copies of the input matrices to avoid modifying the originals
    A = A.copy()
    B = B.copy()


    # Convert matrices to float to prevent integer division
    A = A.astype('float64')
    B = B.astype('float64')

    # Number of rows in the coefficient matrix
    num_rows = len(A)

    # List to store rows that should be moved to the bottom (rows of zeroes)
    rows_to_move = []

    ### START CODE HERE ###

    # Transform matrices A and B into the augmented matrix M
    M = None

    # Iterate over the rows.
    for i in None:

        # Find the first non-zero entry in the current row (pivot)
        pivot = None
        # This variable stores the pivot's column index, it starts at i, but it may cha
        column_index = None


        # CASE PIVOT IS ZERO
        if np.isclose(pivot, 0):
            # PART 1: Look for rows below current row to swap, you may use the function
            index = get_index_first_non_zero_value_from_column(None, None, None)

            # If there is a non-zero pivot
            if index is not None:
                # Swap rows if a non-zero entry is found below the current row
                M = swap_rows(None, None, None)
                # Update the pivot after swapping rows
                pivot = None

            # PART 2 - NOT GRADED. This part deals with the case where the pivot isn't
            # If no non-zero entry is found below it to swap rows, then look for a non-
            if index is None:
                index_new_pivot = get_index_first_non_zero_value_from_row(M, i)
                # If there is no non-zero pivot, it is a row with zeroes, save it into
                # The reason in not moving right away is that it would mess up the inde
                # The second condition i >= num_rows is to avoid taking the augmented p
                if index_new_pivot is None or index_new_pivot >= num_rows:
                    rows_to_move.append(i)
                    continue
                # If there is another non-zero value outside from diagonal, it will be
                else:
                    pivot = None
                    # Update the column index to agree with the new pivot position
                    column_index = None

        # END HANDLING FOR PIVOT 0


        # Divide the current row by the pivot, so the new pivot will be 1. (reduced row
        M[i] = None * M[i]

        # Perform row reduction for rows below the current row
        for j in None:
            # Get the value in the row that is below the pivot value. Remember that the
            value_below_pivot = None

            # Perform row reduction using the formula:
            # row_to_reduce -> row_to_reduce - value_below_pivot * pivot_row
            M[j] = None

    ### END CODE HERE ###

    # Move every rows of zeroes to the bottom
    for row_index in rows_to_move:
        M = move_row_to_bottom(M,row_index)
    return M
```

In [ ]:

```
A = np.array([[1,2,3],[0,0,0], [0,0,5]])
B = np.array([[1], [2], [4]])
reduced_row_echelon_form(A,B)
```

In [ ]:

```
w2_unittest.test_reduced_row_echelon_form(reduced_row_echelon_form)
```

5 - Finding solutions

5.1 - Determining the Existence of Solutions

Before proceeding to find the solutions from the matrix in reduced row echelon form, it's crucial to determine if the linear system has viable solutions. In the process of transforming an augmented matrix to reduced row echelon form, a row of zeros within the coefficient matrix indicates a possible scenario.

If this row of zeros extends across the matrix of coefficients (excluding the augmented column), the system is termed **singular**. This singularity implies the likelihood of either having no solutions or an infinite number of solutions. The distinction lies in the values within the augmented column.

Consider two examples:

$$A = \begin{bmatrix} 1 & 3 & 1 & \| & 7 \\ 0 & 1 & 8 & \| & 5 \\ 0 & 0 & 0 & \| & 0 \end{bmatrix}$$

This system has infinitely many solutions.

$$B = \begin{bmatrix} 1 & 3 & 1 & 5 & \| & 7 \\ 0 & 1 & 8 & 4 & \| & 5 \\ 0 & 0 & 0 & 0 & \| & 0 \\ 0 & 0 & 0 & 0 & \| & 8 \end{bmatrix}$$

Unlike A, the system related to matrix B has no solutions. In this case, even though the third row contains all zeros with a zero in the augmented column, the last row has a non-zero value in the augmented column.

It's crucial to handle cases like matrix B in code implementation, considering scenarios where multiple rows contain zeros but have a non-zero value in their augmented columns.

Exercise 2

In this exercise you will implement a function to check whether an augmented matrix **in reduced row echelon form** has unique solution, no solutions or infinitely many solutions.

In [ ]:

```
# GRADED FUNCTION: check_solution

def check_solution(M):
    """
    Given an augmented matrix in reduced row echelon form, determine the nature of the 

    Parameters:
    - M (numpy.array): An (n x n+1) matrix representing the augmented form of a linear
      where n is the number of equations and variables

    Returns:
    - str: A string indicating the nature of the linear system:
      - "Unique solution." if the system has one unique solution,
      - "No solution." if the system has no solution,
      - "Infinitely many solutions." if the system has infinitely many solutions.

    This function checks for singularity and analyzes the constant terms to determine t
    """
    # Make a copy of the input matrix to avoid modifying the original
    M = M.copy()

    # Get the number of rows in the matrix
    num_rows = len(M)

    # Define the square matrix associated with the linear system
    coefficient_matrix = M[:,:-1]

    # Define the vector associated with the constant terms in the linear system
    constant_vector = M[:,-1]


    # Flag to indicate if the matrix is singular
    singular = False

    ### START CODE HERE ###

    # Iterate over the rows of the coefficient matrix
    for i in None:

        # Test if the row from the square matrix has only zeros (do not replcae the par
        if get_index_first_non_zero_value_from_row(None, None) is None:
            # The matrix is singular, analyze the corresponding constant term to determ
            singular = True

            # If the constant term is non-zero, the system has no solution
            if not np.isclose(None, None):
                return "No solution."

    ### END CODE HERE ###

    # Determine the type of solution based on the singularity condition
    if singular:
        return "Infinitely many solutions."
    else:
        return "Unique solution."
```

In [ ]:

```
w2_unittest.test_check_solution(check_solution)
```

5.2 Back substitution

The final step of the algorithm involves back substitution, a crucial process in obtaining solutions for the linear system. As discussed in the lectures, this method initiates from the bottom and moves upwards. Utilizing elementary row operations, it aims to convert every element above the pivot (which is already zero due to the reduced row echelon form) into zeros. The formula employed is:

$$\text{Row above} \rightarrow \text{Row above} - \text{value} \cdot \text{Row pivot}$$

In this equation, value denotes the value above the pivot, which initially equals 1. To illustrate this process, let's consider the matrix discussed previously:

$$M = \begin{bmatrix} 1 & -1 & \frac{1}{2} & | & \frac{1}{2} \\ 0 & 1 & 1 & | & -1 \\ 0 & 0 & 1 & | & -1 \end{bmatrix}$$

Starting from bottom to top:

- R_2:

-
  - $R_1 = R_1 - 1 \cdot R_2 = \begin{bmatrix} 0 & 1 & 0 & | & 0 \end{bmatrix}$
-
  - $R_0 = R_0 - \frac{1}{2} \cdot R_1 = \begin{bmatrix} 1 & -\frac{1}{2} & 0 & | & 1 \end{bmatrix}$

The resulting matrix is then

$$M = \begin{bmatrix} 1 & -\frac{1}{2} & 0 & | & 1 \\ 0 & 1 & 0 & | & 0 \\ 0 & 0 & 1 & | & -1 \end{bmatrix}$$

Moving to R_1:

- R_1:

-
  - $R_0 = R_0 - \left(-\frac{1}{2} R_1\right) = \begin{bmatrix} 1 & 0 & 0 & | & 1 \end{bmatrix}$

And the final matrix is

$$M = \begin{bmatrix} 1 & 0 & 0 & | & 1 \\ 0 & 1 & 0 & | & 0 \\ 0 & 0 & 1 & | & -1 \end{bmatrix}$$

Note that after back substitution, the solution is just the values in the augmented column! In this case,

$$x_0 = 1$$
$$x_1 = 0$$
$$x_2 = -1$$

Exercise 3

In this exercise you will implement a function to perform back substitution in an **augmented matrix with unique solution**. You may suppose that all checks for solutions were already done.

In [ ]:

```python
# GRADED FUNCTION: back_substitution

def back_substitution(M):
    """
    Perform back substitution on an augmented matrix (with unique solution) in reduced

    Parameters:
    - M (numpy.array): The augmented matrix in reduced row echelon form (n x n+1).
    Returns:
    numpy.array: The solution vector of the linear system.
    """
    # Make a copy of the input matrix to avoid modifying the original
    M = M.copy()

    # Get the number of rows (and columns) in the matrix of coefficients
    num_rows = len(M)

    ### START CODE HERE ####

    # Iterate from bottom to top
    for i in None:
        # Get the substitution row
        substitution_row = None

        # Iterate over the rows above the substitution_row
        for j in None:
            # Get the row to be reduced
            row_to_reduce = None

            # Get the index of the first non-zero element in the substitution row
            index = None

            # Get the value of the element at the found index
            value = row_to_reduce[None]


            # Perform the back substitution step using the formula row_to_reduce = None
            row_to_reduce = None

            # Replace the updated row in the matrix
            None = row_to_reduce

    ### END CODE HERE ####

     # Extract the solution from the last column
    solution = M[:,-1]

    return solution
```

In [ ]:

```
w2_unittest.test_back_substitution(back_substitution)
```

6 - The Gaussian Elimination

6.1 - Bringing it all together

Your task now is to integrate all the steps achieved thus far. Start with a square matrix A of size $n \times n$ and a column matrix B of size $n \times 1$ and transform the augmented matrix $[A|B]$ into reduced row echelon form. Subsequently, verify the existence of solutions. If solutions are present, proceed to perform back substitution to obtain the values. In scenarios where there are no solutions or an infinite number of solutions, handle and indicate these outcomes accordingly.

Exercise 4

In this exercise you will combine every function you just wrote to finish the Gaussian Elimination algorithm.

In [ ]:
```
# GRADED FUNCTION: gaussian_elimination

def gaussian_elimination(A, B):
    """
    Solve a linear system represented by an augmented matrix using the Gaussian elimina

    Parameters:
    - A (numpy.array): Square matrix of size n x n representing the coefficients of the
    - B (numpy.array): Column matrix of size 1 x n representing the constant terms.

    Returns:
    numpy.array or str: The solution vector if a unique solution exists, or a string in
    """

    ### START CODE HERE ###

    # Get the matrix in row echelon form
    reduced_row_echelon_M = None

    # Check the type of solution (unique, infinitely many, or none)
    solution = None

    # If the solution is unique, perform back substitution
    if solution == "Unique solution.":
        solution = None

    ### END SOLUTION HERE ###

    return solution
```

In [ ]:
```
w2_unittest.test_gaussian_elimination(gaussian_elimination)
```

7 - Test with any system of equations!

The code below will allow you to write any equation in the format it is given below (any unknown lower case variables are accepted, in any order) and transform it in its respective augmented matrix so you can solve it using the functions you just wrote in this assignment!

You just need to change the equations variable, always keeping * to indicate product between unknowns and variables and one equation in each line!

In [ ]:
```
from utils import string_to_augmented_matrix
```

In [ ]:
```
equations = """
3*x + 6*y + 6*w + 8*z = 1
5*x + 3*y + 6*w = -10
4*y - 5*w + 8*z = 8
4*w + 8*z = 9
"""

variables, A, B = string_to_augmented_matrix(equations)

sols = gaussian_elimination(A, B)

if not isinstance(sols, str):
    for variable, solution in zip(variables.split(' '),sols):
        print(f"{variable} = {solution:.4f}")
else:
    print(sols)
```

Congratulations! You have finished the first assignment of this course! You built from scratch a linear system solver!