Final Report, Gaussian Elimination by LU Decomposition With Permutation Matrix

Zubair Mukhi
MATH 201, Case Western Reserve University

May 1, 2020

Contents

1 Introduction

Gaussian Elimination is a process by which the series $Ax = b$, where A is a square (nxn) matrix and b is an nx1 matrix (a column vector). For some pairings of A and b, there exist a group of solutions, x, for which the equation $Ax = b$ holds true. x is an nx1 matrix.
As an example, take an arbitrarily chosen 3x3 matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

b is then a 3x1 column vector:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

x is then a 3x1 column vector:

$$b = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

This can be rewritten via matrix multiplication:

$$Ax = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{bmatrix} = \begin{bmatrix} b \\ \end{bmatrix}$$

This can further be separated into a series of equations:
$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$
$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$
$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$
We thus have a set of, when abstracted, n equations with n unknowns. This can be represented as both a series and the above matrix equations. Systems of equations are useful, but challenges arise when the value of n is very large, namely that solving large systems by hand is inefficient compared to leveraging computer power to resolve these systems for us. However, were a computer to solve systems of equations in the same way a human does, the computer would take a long time. Transforming the system into a set of matrices address these issues.
Were we to solve the system of equations, we would do so by combining multiples of different equations together with the intent of isolating one or more variables, e.g. x_1 or x_3. Continuous combination continues until we reach one of three endpoints: the system has one solution, the system has infinitely many solutions, or there are no solutions. The same process is taken to solve the matrix equation Ax=b above. Using this approach, we move each equation into a row, separating the unknowns into the x column vector. To ensure visually that row operations are carried across both A and b, we can create the augmented

matrix $\left[A \mid b\right]$. Once we have the augmented matrix, we then manipulate its rows via elementary row operations, namely row swap, row addition, and row multiplication to create an upper triangular matrix, U and its solution c. The lower triangular matrix L is also created by storing the row operations used to create U. Thus, we have $A{=}LU$; $Ux{=}c$ is used to solve x.
In some cases, however, rows should be permuted, or reordered, to prevent division by zero. Thus, the same swaps made to A are made to an identity matrix before LU decomposition occurs, creating the equation $PA{=}LU$. This is the approach utilized by the solving code.

2 Process Explanation

To solve $Ax{=}b$, the matrices should be reordered to avoid zeroes on the diagonals. This initial step will mitigate the need for reordering in the middle of the solution process, although it generates $PA{=}LU$.
For reference, I will solve the following augmented ($\left[A \mid b\right]$) matrices to illustrate the three outcomes of Gaussian elimination, one solution, infinitely many solutions, and no solution. I will do so following the algorithm used by the solver even if the numbers are unfavorable to compute by hand:

1: $\left[A \mid b\right] = \left[\begin{array}{cccc|c} 1 & -1 & 2 & -1 & -8 \\ 2 & -2 & 3 & -3 & -20 \\ 1 & 1 & 1 & 0 & -2 \\ 1 & -1 & 4 & 3 & 4 \end{array}\right]$

2: $\left[A \mid b\right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 1 & 6 \\ 1 & 1 & 2 & 6 \end{array}\right]$

3: $\left[A \mid b\right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 1 & 4 \\ 1 & 1 & 2 & 6 \end{array}\right]$

2.1 Equation 1

2.1.1 Reordering

$$\left[A \mid b\right] = \left[\begin{array}{cccc|c} 1 & -1 & 2 & -1 & -8 \\ 2 & -2 & 3 & -3 & -20 \\ 1 & 1 & 1 & 0 & -2 \\ 1 & -1 & 4 & 3 & 4 \end{array}\right]$$

We can reorder this and generate the permutation matrix P

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

This reordering leads to:

$$[A \mid b] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & -3 & -20 \\ 1 & 1 & 1 & 0 & -2 \\ 1 & -1 & 4 & 3 & 4 \\ 1 & -1 & 2 & -1 & -8 \end{array}\right]$$

2.1.2 LU Decomposition

We can now apply LU Decomposition to this matrix, as it has been reordered to avoid zeroes in the determinant. $L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$r_2 = r_2 - 0.5r_1$
$r_3 = r_3 - 0.5r_1$
$r_4 = r_4 - 0.5r_1$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0.5 & 0 & 1 & 0 \\ 0.5 & 0 & 0 & 1 \end{bmatrix}$$

$$[U \mid c] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & -3 & -20 \\ 0 & 2 & -0.5 & 1.5 & -12 \\ 0 & 0 & 2.5 & 4.5 & 14 \\ 0 & 0 & 0.5 & 0.5 & 2 \end{array}\right]$$

There is no cancellation needed for pivot at (2,2), so the step is ignored.
$r_4 = r_4 - 0.2r_3$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0.5 & 0 & 1 & 0 \\ 0.5 & 0 & 0.2 & 1 \end{bmatrix}$$

$$[U \mid c] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & -3 & -20 \\ 0 & 2 & -0.5 & 1.5 & 8 \\ 0 & 0 & 2.5 & 4.5 & 14 \\ 0 & 0 & 0 & -0.4 & -0.8 \end{array}\right]$$

At this point, L and U are both triangular, so we can move to the next step, backsolving.

2.1.3 Backsolving

$$[U \mid c] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & -3 & -20 \\ 0 & 2 & -0.5 & 1.5 & 8 \\ 0 & 0 & 2.5 & 4.5 & 14 \\ 0 & 0 & 0 & -0.4 & -0.8 \end{array}\right]$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Taking the last row, $-0.4x_4 = -0.8$

$x_4 = 2$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 2 \end{bmatrix}$$

Now, we propagate the change through U.

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & -3(2) & -20 \\ 0 & 2 & -0.5 & 1.5(2) & 8 \\ 0 & 0 & 2.5 & 4.5(2) & 14 \\ 0 & 0 & 0 & -0.4(2) & -0.8 \end{array}\right]$$

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & -2 & 3 & 0 & -14 \\ 0 & 2 & -0.5 & 0 & 5 \\ 0 & 0 & 2.5 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Now, we repeat for row 3

$2.5x_3 = 5$

$x_3 = 2$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ 2 \\ 2 \end{bmatrix}$$

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & -2 & 3(2) & 0 & -14 \\ 0 & 2 & -0.5(2) & 0 & 5 \\ 0 & 0 & 2.5(2) & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & -2 & 0 & 0 & -20 \\ 0 & 2 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Now, we repeat for row 2

$2x_2 = 6$

$x_2 = 3$

$$x = \begin{bmatrix} x_1 \\ 3 \\ 2 \\ 2 \end{bmatrix}$$

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & -2(3) & 0 & 0 & -20 \\ 0 & 2(3) & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\left[U \mid c\right] = \left[\begin{array}{cccc|c} 2 & 0 & 0 & 0 & -14 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

At this point, only x_1 remains to be solved. This is now trivial.

$$x = \begin{bmatrix} -7 \\ 3 \\ 2 \\ 2 \end{bmatrix}$$

2.2 Equation 2

2.2.1 Reordering

$$\left[A \mid b\right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 1 & 6 \\ 1 & 1 & 2 & 6 \end{array}\right]$$

We can reorder this and generate the permutation matrix P

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This reordering leads to:

$$\left[A \mid b\right] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 1 & 1 & 1 & 4 \\ 1 & 1 & 2 & 6 \end{array}\right]$$

2.2.2 LU Decomposition

We can now apply LU Decomposition to this matrix, as it has been reordered to avoid zeroes in the determinant. $L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

$r_2 = r_2 - 0.5r_1$

$r_3 = r_3 - 0.5r_1$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}$$

This is the only cancellation step needed.

$$\left[U \mid c\right] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 0 & 0.5 & 1 \\ 0 & 0 & 1.5 & 3 \end{array}\right]$$

At this point, L and U are both triangular, so we can move to the next step, backsolving.

2.2.3 Backsolving

$$\left[U \mid c\right] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 0 & 0.5 & 1 \\ 0 & 0 & 1.5 & 3 \end{array}\right]$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$
Taking the last row, $1.5x_3 = 3$
$x_3 = 2$
$$x = \begin{bmatrix} x_1 \\ x_2 \\ 2 \end{bmatrix}$$
Now, we propagate the change through U.
$$\begin{bmatrix} U \mid c \end{bmatrix} = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$
Now, we repeat for row 2.
However, $0x_2 = 0$, so x_2 is considered "free," so there are infinitely many solutions.
$$x = \begin{bmatrix} x_1 \\ x_2 \\ 2 \end{bmatrix}$$
For our purposes, we will assign x_2 the values 1 and -1.
$$x_+ = \begin{bmatrix} x_1 \\ 1 \\ 2 \end{bmatrix}$$
$$x_- = \begin{bmatrix} x_1 \\ -1 \\ 2 \end{bmatrix}$$
This propagates through the matrix, leaving x_1.
This is a single variable, so I will skip the propagation.
$$x_+ = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$
$$x_- = \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix}$$

2.3 Equation 3

2.3.1 Reordering

$$\begin{bmatrix} A \mid b \end{bmatrix} = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 1 & 6 \\ 1 & 1 & 2 & 6 \end{array}\right]$$
We can reorder this and generate the permutation matrix P
$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
This reordering leads to:

$$[A \mid b] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 1 & 1 & 1 & 4 \\ 1 & 1 & 2 & 6 \end{array}\right]$$

2.3.2 LU Decomposition

We can now apply LU Decomposition to this matrix, as it has been reordered to avoid zeroes in the determinant. $L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

$r_2 = r_2 - 0.5r_1$
$r_3 = r_3 - 0.5r_1$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}$$

This is the only cancellation step needed.

$$[U \mid c] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 4 \\ 0 & 0 & 0.5 & 2 \\ 0 & 0 & 1.5 & 4 \end{array}\right]$$

At this point, L and U are both triangular, so we can move to the next step, backsolving.

2.3.3 Backsolving

$$[U \mid c] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 4 \\ 0 & 0 & 0.5 & 2 \\ 0 & 0 & 1.5 & 4 \end{array}\right]$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Taking the last row, $1.5x_3 = 4$
$x_3 = \frac{8}{3}$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \frac{8}{3} \end{bmatrix}$$

Now, we propagate the change through U.

$$[U \mid c] = \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 0 & 0.5 * \frac{8}{3} & 2 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

However, observe r_2. For r_2 to provide a solution, $\frac{4}{3} = 2$. This is simply false.
Thus, there are no solutions to the given $Ax{=}b$.

3 Implementation

To solve $Ax=b$, the solving code goes through several steps.
It first prompts the user to enter the size of A, taking one dimension and stipulating that A be a square matrix. It then requests b, validating that all sizes match the requirements for Gaussian elimination. It then prints these values
Next, it permutes A to make sure that the diagonals contain the largest values. It records the swaps in the permutation matrix P, initialized as the nxn identity matrix, and permutesb$A$ to match, returning updated P, A, and b. See the function

```
reorder(a, b, p, ptr)
```

at A.3 for the source code.
Next, it executes LU decomposition in the function

```
generate_u(arr_a, arr_b, arr_l, n_rows)
```

Refer to A.4 for source code. This function returns L and U of the L and U resulting from the *permutation*, given as $PA=LU$. Note that this code skips zero value pivots in order to avoid division by zero errors. It also returns c, the result of ensuring that the elimination steps taken on A to generate U also occur on b.
Last, the solving code actually solves the equation $Ux=c$ generated from the previous code (see A.5 for code). It starts from the lower right, moving up and left along the diagonal. If the currently focused pivot value is zero, it compares the value in c to the pivot and shifts into the `infinite_sols` flow to finish solving the series. If the value in c corresponding to the zero pivot is not zero, the function terminates, as it has determined that there are no solutions. If the pivot is not zero, then it solves the corresponding x by dividing the corresponding c value by the pivot and storing it as the corresponding x. It then propagates this value through the rest of the partially-solved U and c to ensure that the next processed row has only one unknown. The `infinite_sols` flow does the same thing in parallel on 2 arrays, one of which has the the free variable set to 1 and the other to -1.

4 Code Execution Samples

4.1 Equation 1

```
How many rows?
>>> 4
please enter row 1 of A
>>> 1,-1,2,-1
please enter row 2 of A
>>> 2,-2,3,-3
please enter row 3 of A
```

```
>>> 1,1,1,0
please enter row 4 of A
>>> 1,-1,4,3
please enter b
>>> -8,-20,-2,4
Ax=b
[[ 1. -1.  2. -1.]
 [ 2. -2.  3. -3.]
 [ 1.  1.  1.  0.]
 [ 1. -1.  4.  3.]]x=[ -8. -20.  -2.   4.]

P:
[[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [1. 0. 0. 0.]]

L:
[[1.  0.  0.  0. ]
 [0.5 1.  0.  0. ]
 [0.5 0.  1.  0. ]
 [0.5 0.  0.2 1. ]]

U:
[[ 2.  -2.   3.  -3. ]
 [ 0.   2.  -0.5  1.5]
 [ 0.   0.   2.5  4.5]
 [ 0.   0.   0.  -0.4]]

c:
[-20.    8.   14.   -0.8]

Next, we solve Ux=c
x is the following:
-6.999999999999998
2.9999999999999996
1.9999999999999993
2.0000000000000004
```

Note that here, there are some rounding issues. This stems from the variable forms I used in the solver, since I use a determinant check to switch between single-solution and the no-solution/infinitely-many mode, as `numpy` doesn't take the determinant of `longdouble` format variables and so I used the `double` datatype instead.

4.2 Equation 2

```
How many rows?
>>> 3
please enter row 1 of A
>>> 1,1,1
please enter row 2 of A
>>> 2,2,1
please enter row 3 of A
>>> 1,1,2
please enter b
>>> 4,6,6
Ax=b
[[1. 1. 1.]
 [2. 2. 1.]
 [1. 1. 2.]]x=[4. 6. 6.]

P:
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]

L:
[[1.  0.  0. ]
 [0.5 1.  0. ]
 [0.5 0.  1. ]]

U:
[[2.  2.  1. ]
 [0.  0.  0.5]
 [0.  0.  1.5]]

c:
[6. 1. 3.]

Next, we solve Ux=c

infinitely many solutions
[1. 1. 2.]
[ 3. -1.  2.]
```

4.3 Equation 3

```
How many rows?
>>> 3
please enter row 1 of A
```

```
>>> 1,1,1
please enter row 2 of A
>>> 2,2,1
please enter row 3 of A
>>> 1,1,2
please enter b
>>> 4,4,6
Ax=b
[[1. 1. 1.]
 [2. 2. 1.]
 [1. 1. 2.]]x=[4. 4. 6.]

P:
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]

L:
[[1.  0.  0. ]
 [0.5 1.  0. ]
 [0.5 0.  1. ]]

U:
[[2.  2.  1. ]
 [0.  0.  0.5]
 [0.  0.  1.5]]

c:
[4. 2. 4.]

Next, we solve Ux=c
x is the following:
No solution
```

5 Known Limitations

Currently, this code is not tested for multiple zeroes in the code. However, this could be resolved via recursive calls to the `infinite_sols` code. There is a possibility that this would explode the solution set, though. A less memory-intensive solution may be to replace any future zeroes found on pivots with a zero in the corresponding x.

Appendices

A Source Code

A.1 Imports

```
import numpy as np
```

A.2 Runner and Wrapper Code

```
def solver():
    print('this is a solver for Ax=b via Gaussian elimination, providing x, L, and U.\n'
          'It currently only accepts square matrices.\n'
          'The solver will move the largest value in the first row to the diagonal.\n'
          'Following that, it will repeat for each remaining row.\n'
          'it holds the permutations in an internal matrix, p.')
    print('please enter rows as numbers separated by commas, no spaces.')
    n_rows = int(input("How many rows?\n"))
    arr_p = np.identity(n_rows)
    arr_l = np.identity(n_rows)
    arr_a = []
    for i in range(0, n_rows):
        row = (input('please enter row {} of A\n'.format(i+1)))
        row = row.split(',')
        while len(row) != n_rows:
            print('invalid input, please re-enter row.')
            row = (input('please enter row {} of A\n'.format(i + 1)))
            row = row.split(',')
        for r in range(0, len(row)):
            row[r] = float(row[r])
        arr_a.append(row)
    arr_a = np.array(arr_a, dtype=np.double)
    b_string = input('please enter b\n')
    arr_b = b_string.split(',')
    while len(arr_b) != n_rows:
        print('invalid b, please try again.')
        b_string = input('please enter b\n')
        arr_b = b_string.split(',')
    for e in range(0, len(arr_b)):
        arr_b[e] = float(arr_b[e])
    arr_b = np.array(arr_b, dtype=np.double)
    print('Ax=b\n{}x={}\n'.format(arr_a, arr_b))
    for i in range(0, arr_a.shape[0]):
        reorder(arr_a, arr_b, arr_p, i)
    arr_u, arr_c, arr_l = generate_u(arr_a, arr_b, arr_l, n_rows)
```

```
    print('P:\n{}\n\nL:\n{}\n\nU:\n{}\n\nc:\n{}\n'.format(arr_p,arr_l, arr_u, arr_c))
    print('Next, we solve Ux=c\nx is the following:')
    arr_x = solve(arr_u, arr_c)
    for x in arr_x:
        print(x)


if __name__ == '__main__':
    solver()
```

A.3 Reordering Code

```
def reorder(a, b, p, ptr):
    pointer = ptr
    for i in range(ptr, a.shape[0]):
        if a[i][ptr] > a[pointer][ptr]:
            pointer = i
    a[[ptr, pointer]] = a[[pointer, ptr]]
    b[[ptr, pointer]] = b[[pointer, ptr]]
    p[[ptr, pointer]] = p[[pointer, ptr]]
```

A.4 LU Decomposer

```
def generate_u(arr_a, arr_b, arr_l, n):
    u = np.copy(arr_a)
    c = np.copy(arr_b)
    for diagonal in range(0, n):
        pivot = u[diagonal][diagonal]
        for i in range(diagonal+1, u.shape[0]):
            if pivot != 0:
                prod = u[i][diagonal] / pivot
                u[i] -= prod * u[diagonal]
                c[i] -= prod * c[diagonal]
                arr_l[i][diagonal] = prod
            else:
                pass  # avoids division by zero
    return u, c, arr_l
```

A.5 Solver code

```
def solve(arr_u, arr_c):
    m = arr_u.shape[0]
    u = arr_u
    arr_x = np.zeros_like(arr_c)
    det = np.linalg.det(arr_u)
    if det != 0:  # this matrix has one solution
```

```
            for ptr in reversed(range(0, m)):
                arr_x[ptr] = arr_c[ptr] / u[ptr][ptr]
                for i in range(0, ptr):
                    arr_c[i] = arr_c[i] - arr_x[ptr] * u[i][ptr]
            return arr_x
        else:
            for ptr in reversed(range(0, m)):
                if u[ptr][ptr] != 0:
                    arr_x[ptr] = arr_c[ptr] / u[ptr][ptr]
                    for i in range(0, ptr):
                        arr_c[i] = arr_c[i] - arr_x[ptr] * u[i][ptr]
                elif arr_c[ptr] != 0:
                    return ('No solution',)
                else:
                    return infinite_sols(u, arr_x, arr_c, ptr)
            return arr_x


    def infinite_sols(arr_u, arr_x, arr_c, p):
        arr_x_neg = np.copy(arr_x)
        arr_c_neg = np.copy(arr_c)
        arr_x[p] = np.double(1)
        arr_x_neg[p] = np.double(-1)
        for i in range(0, p):
            arr_c[i] = arr_c[i] - arr_x[p] * arr_u[i][p]
            arr_c_neg[i] = arr_c_neg[i] - arr_x_neg[p] * arr_u[i][p]
        for ptr in reversed(range(0, p)):
            arr_x[ptr] = arr_c[ptr] / arr_u[ptr][ptr]
            for i in range(0, ptr):
                arr_c[i] = arr_c[i] - arr_x[ptr] * arr_u[i][ptr]
        for ptr in reversed(range(0, p)):
            arr_x_neg[ptr] = arr_c_neg[ptr] / arr_u[ptr][ptr]
            for i in range(0, ptr):
                arr_c_neg[i] = arr_c_neg[i] - arr_x_neg[ptr] * arr_u[i][ptr]
        return 'Infinitely many solutions', arr_x, arr_x_neg
```