Gauss-Jordan Method

In the Gauss-Jordan method both the all elements above and below the main diagonal are eliminated to transform the coefficient matrix to the identity matrix, I_n, and, accordingly, transform the constant terms vector to the solution vector. In other words, the system

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

is transformed to

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1^* \\ b_2^* \\ \vdots \\ b_n^* \end{Bmatrix}$$

where the values b_1^*, b_2^*, …, b_n^* are the values of constant terms after the n-th transformation.

Its algorithm is quite similar to that of Gauss elimination with one main difference that is, for each pivot row, elimination operations are performed for all other rows above and below the pivot row. No back-substitution is applied since at the end of the elimination, the constant term vector will be already transformed to the solution vector.

The following steps are applied the reduction of each pivot column.

Step 1: Performing partial pivoting by rearranging the rows of the system at each transformation to guarantee non-zero elements in the main diagonal of the coefficients matrix.

Step 2: Division of every row of the system (including constant term) by the pivot element of that row ($a_{k,k}$). This step will make all element on the main diagonal equal 1.0.

Step 3: Elimination of all elements above and below the main diagonal with applying the same operations on the constant terms vector. This algorithm requires the following three nested loops:

1- The main loop of k from row 1 to n to index the pivot rows. In this loop each element in the pivot row is divided by the pivot;

$$a_{k,j}^* = \frac{a_{k,j}}{a_{k,k}}, \quad b_k^* = \frac{b_k}{a_{k,k}}, \quad k = 1,2,\dots,n, \quad j = k,n$$

The asterisk (*) superscript refers to that the element with a new value. For more concise code, the **step 2** is included in this loop.

2- The loop of i from row 1 to n to index the subtraction rows. In order to avoid the subtraction of the pivot row from itself when i is equal to k **or** if the coefficient is already is equal to zero, the subtraction for current i should be skipped.
3- The loop of j from k to n to index the columns for element subtraction starting from the pivot element to the last column of the coefficients matrix. Thus, the elimination statement is written as

$$a^*_{i,j} = a_{i,j} - a_{i,k}\, a^*_{k,j}$$

The new constant vector is calculated within the i-loop as

$$b^*_i = b_i - a_{i,k}\, b^*_k$$

where, $k = 1,2,\ldots,n$, $i = 1,2,\ldots,n$, $i \neq k$

In this process, it should be noticed that the row and column numbers in Python arrays, including NumPy arrays, always start from 0 to n-1 although in theoretical speaking indices are from 1 to n.

Example

Solve the following system

$$a = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 2 & 3 & 2 \\ 4 & -3 & 0 & 1 \\ 6 & 1 & -6 & -5 \end{bmatrix}, \quad b = \begin{Bmatrix} 0 \\ -2 \\ -7 \\ 6 \end{Bmatrix}$$

Answer

$$\{x\} = \begin{Bmatrix} -0.5 \\ 1 \\ 0.3333 \\ -2 \end{Bmatrix}$$

Solution

The system can be solved by the following Gauss-Jordan function

```
import numpy as np
def gssjrdn(a,b):
    a = np.array(a, float) # to ensure NumPy arrays
    b = np.array(b, float)
    n = len(b)
    # Partial pivoting
    if np.fabs(a[k, k]) < 1.0e-12:
            for i in range(k+1,n):
                if np.fabs(a[i,k]) > np.fabs(a[k,k]):
                    for j in range(k,n):
                        a[k,j],a[i,j] = a[i,j],a[k,j]
                    b[k],b[i] = b[i],b[k]
                    break
    # main loop of pivot rows
    for k in range(n):
        pivot = a[k,k]    # saved to keep its value in j-loop
        for j in range(n):
            a[k,j] /= pivot
        b[k] /= pivot
        # Elimination loops
        for i in range(n):
            if i == k  or a[i,k] == 0 : continue
            factor = a[i,k]   # saved to keep its value in j-loop
            for j in range(k,n):
                a[i,j] -= factor * a[k,j]
            b[i] -= factor * b[k]
    return b, a

a = a = [[0,2,0,1],[2,2,3,2],[4,-3,0,1],[6,1,-6,-5]] # Python list
b = [0,-2,-7,6]   # Python list

X, A = gssjrdn(a,b)

print('The solution of the system:')
print(X)
print('The coefficients matrix after transformation:')
print(A)
```

The output of the code is

```
The solution of the system:
```

```
[-0.5         1.          0.33333333 -2.        ]
The coefficients matrix after transformation:
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [-0. -0.  1.  0.]
 [-0. -0. -0.  1.]]
```

In this example, the identity matrix resulting from the elimination operations has been returned to show the

Although the back-substitution stage is not applied in Gauss-Jordan method, its number of operations is larger with about 50% than that in Gauss elimination because of the entire system elimination. That is why Gauss-Jordan method is not used extensively in the numerical applications.

NumPy Implicit Array Operations

In this code, we used the traditional method of computer languages to carry out elementwise operations on rows by using the j-loop. Actually, NumPy arrays can perform the same operations for entire rows internally. Therefore, code can be modified as following,

For interchanging two rows, instead of explicit tuple swap method

```
for j in range(n):
      a[k,j],a[i,j] = a[i,j],a[k,j]
b[k],b[i] = b[i],b[k]
```

use the double brackets implicit swapping in the same location

```
a[[k,i]] = a[[i,k]]
b[[k,i]] = b[[i,k]]
```

For division an entire row of in a matrix, replace the lines

```
for j in range(k,n):
      a[k,j] /= pivot
```

with

```
a[k] /= pivot
```

Finally, for element subtraction step, replace the loop

```
for j in range(k,n):
      a[i,j] -= factor * a[k,j]
```

with

```
a[i] -= factor * a[k]
```

The difference between using for-loops and the alternatives is mainly the number of elements of the rows involved in the computations. The loops use the elements in columns which have not been transformed to ones or zeros yet while the implicit operations include the entire rows which may require noticeably more execution time for large systems.

Exercise

Solve the following system by using the code of Gauss-Jordan method.

$$\begin{bmatrix} 0 & 7 & -1 & 3 & 1 \\ 2 & 3 & 4 & 1 & 7 \\ 6 & 2 & 0 & 2 & -1 \\ 2 & 1 & 2 & 0 & 2 \\ 3 & 4 & 1 & -2 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 7 \\ 2 \\ 3 \\ 4 \end{Bmatrix}$$

Answer: $x = \{\ 0.021705 \ \ 0.792248 \ \ 1.051163 \ \ 0.158140 \ \ 0.031008\}^T$