Last update: November 17, 2023 Translated From: e-maxx.ru

# Gauss method for solving system of linear equations

Given a system of $n$ linear algebraic equations (SLAE) with $m$ unknowns. You are asked to solve the system: to determine if it has no solution, exactly one solution or infinite number of solutions. And in case it has at least one solution, find any of them.

Formally, the problem is formulated as follows: solve the system:

$$\begin{align}
a_{11} x_1 + a_{12} x_2 + &\dots + a_{1m} x_m = b_1\\
a_{21} x_1 + a_{22} x_2 + &\dots + a_{2m} x_m = b_2\\
&\vdots \\
a_{n1} x_1 + a_{n2} x_2 + &\dots + a_{nm} x_m = b_n
\end{align}$$

where the coefficients $a_{ij}$ (for $i$ from 1 to $n$, $j$ from 1 to $m$) and $b_i$ ($i$ from 1 to $n$) are known and variables $x_i$ ($i$ from 1 to $m$) are unknowns.

This problem also has a simple matrix representation:

$$Ax = b,$$

where $A$ is a matrix of size $n \times m$ of coefficients $a_{ij}$ and $b$ is the column vector of size $n$.

It is worth noting that the method presented in this article can also be used to solve the equation modulo any number p, i.e.:

$$\begin{align}
a_{11} x_1 + a_{12} x_2 + &\dots + a_{1m} x_m \equiv b_1 \pmod p\\
a_{21} x_1 + a_{22} x_2 + &\dots + a_{2m} x_m \equiv b_2 \pmod p\\
&\vdots \\
a_{n1} x_1 + a_{n2} x_2 + &\dots + a_{nm} x_m \equiv b_n \pmod p
\end{align}$$

## Gauss

Strictly speaking, the method described below should be called "Gauss-Jordan", or Gauss-Jordan elimination, because it is a variation of the Gauss method, described by Jordan in 1887.

## Overview

The algorithm is a `sequential elimination` of the variables in each equation, until each equation will have only one remaining variable. If $n = m$, you can think of it as transforming the matrix $A$ to identity matrix, and solve the equation in this obvious case, where solution is unique and is equal to coefficient $b_i$.

Gaussian elimination is based on two simple transformation:

- It is possible to exchange two equations
- Any equation can be replaced by a linear combination of that row (with non-zero coefficient), and some other rows (with arbitrary coefficients).

In the first step, Gauss-Jordan algorithm divides the first row by $a_{11}$. Then, the algorithm adds the first row to the remaining rows such that the coefficients in the first column becomes all zeros. To achieve this, on the i-th row, we must add the first row multiplied by $-a_{i1}$. Note that, this operation must also be performed on vector $b$. In a sense, it behaves as if vector $b$ was the $m+1$-th column of matrix $A$.

As a result, after the first step, the first column of matrix $A$ will consists of $1$ on the first row, and $0$ in other rows.

Similarly, we perform the second step of the algorithm, where we consider the second column of second row. First, the row is divided by $a_{22}$, then it is subtracted from other rows so that all the second column becomes $0$ (except for the second row).

We continue this process for all columns of matrix $A$. If $n = m$, then $A$ will become identity matrix.

## Search for the pivoting element

The described scheme left out many details. At the $i$th step, if $a_{ii}$ is zero, we cannot apply directly the described method. Instead, we must first `select a pivoting row`: find one row of the matrix where the $i$th column is non-zero, and then swap the two rows.

Note that, here we swap rows but not columns. This is because if you swap columns, then when you find a solution, you must remember to swap back to correct places. Thus, swapping rows is much easier to do.

In many implementations, when $a_{ii} \neq 0$, you can see people still swap the $i$th row with some pivoting row, using some heuristics such as choosing the pivoting row with maximum absolute value of $a_{ji}$. This heuristic is used to reduce the value range of the matrix in later steps. Without this heuristic, even for matrices of size about $20$, the error will be too big and can cause overflow for floating points data types of C++.

## Degenerate cases

In the case where $m = n$ and the system is non-degenerate (i.e. it has non-zero determinant, and has unique solution), the algorithm described above will transform $A$ into identity matrix.

Now we consider the `general case`, where $n$ and $m$ are not necessarily equal, and the system can be degenerate. In these cases, the pivoting element in $i$th step may not be found. This means that on the $i$th column, starting from the current line, all contains zeros. In this case, either there is no possible value of variable $x_i$ (meaning the SLAE has no solution), or $x_i$ is an independent variable and can take arbitrary value. When implementing Gauss-Jordan, you should continue the work for subsequent variables and just skip the $i$th column (this is equivalent to removing the $i$th column of the matrix).

So, some of the variables in the process can be found to be independent. When the number of variables, $m$ is greater than the number of equations, $n$, then at least $m - n$ independent variables will be found.

In general, if you find at least one independent variable, it can take any arbitrary value, while the other (dependent) variables are expressed through it. This means that when we work in the field of real numbers, the system potentially has infinitely many solutions. But you should remember that when there are independent variables, SLAE can have no solution at all. This happens when the remaining untreated equations have at least one non-zero constant term. You can check this by assigning zeros to all independent variables, calculate other variables, and then plug in to the original SLAE to check if they satisfy it.

## Implementation

Following is an implementation of Gauss-Jordan. Choosing the pivot row is done with heuristic: choosing maximum value in the current column.

The input to the function `gauss` is the system matrix $a$. The last column of this matrix is vector $b$.

The function returns the number of solutions of the system $(0, 1, \text{or } \infty)$. If at least one solution exists, then it is returned in the vector $ans$.

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity or a big number

int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
```

```
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

Implementation notes:

- The function uses two pointers - the current column $col$ and the current row $row$.
- For each variable $x_i$, the value $where(i)$ is the line where this column is not zero. This vector is needed because some variables can be independent.
- In this implementation, the current $i$th line is not divided by $a_{ii}$ as described above, so in the end the matrix is not identity matrix (though apparently dividing the $i$th line can help reducing errors).
- After finding a solution, it is inserted back into the matrix - to check whether the system has at least one solution or not. If the test solution is successful, then the function returns 1 or $\inf$, depending on whether there is at least one independent variable.

## Complexity

Now we should estimate the complexity of this algorithm. The algorithm consists of $m$ phases, in each phase:

- Search and reshuffle the pivoting row. This takes $O(n+m)$ when using heuristic mentioned above.
- If the pivot element in the current column is found - then we must add this equation to all other equations, which takes time $O(nm)$.

So, the final complexity of the algorithm is $O(\min(n,m).nm)$. In case $n=m$, the complexity is simply $O(n^3)$.

Note that when the SLAE is not on real numbers, but is in the modulo two, then the system can be solved much faster, which is described below.

## Acceleration of the algorithm

The previous implementation can be sped up by two times, by dividing the algorithm into two phases: forward and reverse:

- Forward phase: Similar to the previous implementation, but the current row is only added to the rows after it. As a result, we obtain a triangular matrix instead of diagonal.
- Reverse phase: When the matrix is triangular, we first calculate the value of the last variable. Then plug this value to find the value of next variable. Then plug these two values to find the next variables...

Reverse phase only takes $O(nm)$, which is much faster than forward phase. In forward phase, we reduce the number of operations by half, thus reducing the running time of the implementation.

## Solving modular SLAE

For solving SLAE in some module, we can still use the described algorithm. However, in case the module is equal to two, we can perform Gauss-Jordan elimination much more effectively using bitwise operations and C++ bitset data types:

```
int gauss (vector < bitset<N> > a, int n, int m, bitset<N> & ans) {
    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        for (int i=row; i<n; ++i)
            if (a[i][col]) {
                swap (a[i], a[row]);
                break;
            }
        if (! a[row][col])
            continue;
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row && a[i][col])
                a[i] ^= a[row];
        ++row;
    }
        // The rest of implementation is the same as above
}
```

Since we use bit compress, the implementation is not only shorter, but also 32 times faster.

## A little note about different heuristics of choosing pivoting row

There is no general rule for what heuristics to use.

The heuristics used in previous implementation works quite well in practice. It also turns out to give almost the same answers as "full pivoting" - where the pivoting row is search amongst all elements of the whose submatrix (from the current row and current column).

Though, you should note that both heuristics is dependent on how much the original equations was scaled. For example, if one of the equation was multiplied by $10^6$, then this equation is almost certain to be chosen as pivot in first step. This seems rather strange, so it seems logical to change to a more complicated heuristics, called `implicit pivoting`.

Implicit pivoting compares elements as if both lines were normalized, so that the maximum element would be unity. To implement this technique, one need to maintain maximum in each row (or maintain each line so that maximum is unity, but this can lead to increase in the accumulated error).

## Improve the solution

Despite various heuristics, Gauss-Jordan algorithm can still lead to large errors in special matrices even of size $50 - 100$.

Therefore, the resulting Gauss-Jordan solution must sometimes be improved by applying a simple numerical method - for example, the method of simple iteration.

Thus, the solution turns into two-step: First, Gauss-Jordan algorithm is applied, and then a numerical method taking initial solution as solution in the first step.

## Practice Problems

- Spoj - Xor Maximization
- Codechef - Knight Moving

- Lightoj - Graph Coloring
- UVA 12910 - Snakes and Ladders
- TIMUS1042 Central Heating
- TIMUS1766 Humpty Dumpty
- TIMUS1266 Kirchhoff's Law
- Codeforces - No game no life