

Solving Linear Equations

3.0 Gaussian Elimination

Last time we looked at computing the stresses in a simple truss. To solve for the stresses, we need to solve a set of equations with several unknowns. The number of unknowns increases as the number of elements and nodes in the truss increases. For a very complex truss there would be many equations and unknowns.

We need a method that is very fast and can be used on a wide range of problems. Gaussian elimination meets these requirements. It has the following attributes:

- a. It works for most reasonable problems
- b. It is computationally very fast
- c. It is not too difficult to program

The major drawback is that it can suffer from the accumulation of round off errors.

3.1 Upper Triangular Form (Forward Sweep)

We can illustrate how it works with the following set of equations.

$$\begin{aligned}a_{11}X_1 + a_{12}X_2 + a_{13}X_3 &= d_1 \\a_{21}X_1 + a_{22}X_2 + a_{23}X_3 &= d_2 \\a_{31}X_1 + a_{32}X_2 + a_{33}X_3 &= d_3\end{aligned}\tag{3.1}$$

We would like to solve this set of equations for X_1 , X_2 , and X_3 . We can rewrite the equations in matrix form as:

$$\begin{array}{ccc|c}a_{11} & a_{12} & a_{13} & d_1 \\a_{21} & a_{22} & a_{23} & d_2 \\a_{31} & a_{32} & a_{33} & d_3\end{array}\tag{3.2}$$

We want to multiply the first equation by some factor so that when we subtract the second equation the a_{21} is eliminated. We can do this by multiplying it by a_{21} / a_{11} . This yields:

$$\begin{array}{ccc|c}a_{21} & \frac{a_{12}a_{21}}{a_{11}} & \frac{a_{13}a_{21}}{a_{11}} & \frac{a_{21}d_1}{a_{11}} \\a_{21} & a_{22} & a_{23} & d_2 \\a_{31} & a_{32} & a_{33} & d_3\end{array}\tag{3.3}$$

Subtracting the first equation in 3.3 from the second equation in 3.3 then replacing the first equation with its original form yields:

$$\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & d_1 \\ 0 & a_{22} - \frac{a_{12}a_{21}}{a_{11}} & a_{23} - \frac{a_{13}a_{21}}{a_{11}} & d_2 - \frac{a_{21}d_1}{a_{11}} \\ a_{31} & a_{32} & a_{33} & d_3 \end{array} \quad (3.4)$$

We do the same thing for the third row by multiplying the first row by a_{31} / a_{11} and subtracting it from the third row.

$$\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & d_1 \\ 0 & a_{22} - \frac{a_{12}a_{21}}{a_{11}} & a_{23} - \frac{a_{13}a_{21}}{a_{11}} & d_2 - \frac{a_{21}d_1}{a_{11}} \\ 0 & a_{32} - \frac{a_{12}a_{31}}{a_{11}} & a_{33} - \frac{a_{13}a_{31}}{a_{11}} & d_3 - \frac{a_{31}d_1}{a_{11}} \end{array} \quad (3.5)$$

We can reduce the complexity of the terms symbolically by substituting new variable names for the complex terms. This yields

$$\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & d_1 \\ 0 & b_{22} & b_{23} & e_2 \\ 0 & b_{32} & b_{33} & e_3 \end{array} \quad (3.6)$$

Now multiply the second row by b_{32}/b_{22} and subtracting it from the third row.

$$\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & d_1 \\ 0 & b_{22} & b_{23} & e_2 \\ 0 & 0 & b_{33} - \frac{b_{23}b_{32}}{b_{22}} & e_3 - \frac{b_{32}e_2}{b_{22}} \end{array} \quad (3.7)$$

Again we simplify by substituting in for the complex terms.

$$\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & d_1 \\ 0 & b_{22} & b_{23} & e_2 \\ 0 & 0 & c_{33} & f_3 \end{array} \quad (3.8)$$

This is what we call upper triangular form. We started at the first equation and worked our way to the last equation in what is called a forward sweep. The forward

sweep results in a matrix with values on the diagonal and above and zeros below the diagonal.

3.2 Backward Sweep

We can solve each equation of the upper triangular form by moving through the list starting with the bottom equation and working our way up to the top.

$$X_3 = f_3 / c_{33} \quad (3.9)$$

$$X_2 = (e_2 - b_{22}X_3) / b_{22} \quad (3.10)$$

$$X_1 = (d_1 - a_{12}X_2 - a_{13}X_3) / a_{11} \quad (3.11)$$

This is called back substitution and the process as a whole is called the backward sweep.

3.3 Equation Normalization

You can see from the equations that it is very important for the diagonal coefficients to be non-zero. A zero valued diagonal term will lead to a divide by zero error.

In fact, there will be fewer problems with accumulated round off error if the diagonal coefficients have a larger magnitude than the off diagonal terms. The reason for this is that computers have limited precision. If the off diagonal terms are much larger than the diagonal term, a very large number will be created when the right-hand-side of the equation is divided by the diagonal term. The size of this number dominates the precision of the computer and causing smaller values to be ignored when added to the larger value. For example, adding 1.3×10^{11} to 94.6 in a computer that has 8 digits of precision does results in a value of 1.3×10^{11} . The 94.6 is completely lost in the process. This can be seen in the previous example

$$a_{22} - \frac{a_{12}a_{21}}{a_{11}} \quad (3.12)$$

If a_{11} is small compared to a_{22} then we could end up with

Small – large

And loose the value of small (a_{22}) altogether.

This problem is solved by using two steps in the processing.

- a. Normalize the rows so that the largest term in the equation is 1.
- b. Swap the rows around so that the largest term in any equation occurs on the diagonal. This is called partial pivoting.

Diagonal dominance can usually be improved by moving the rows or the columns in the set of equations. Moving the columns is somewhat more difficult than moving the rows because the position of X_1, X_2, \dots changes when you move the columns. If you move the columns you must keep track of the change in position of the X s. When the rows are moved, the X s stay in the same place so rows can be moved with the added complication.

Moving both rows and columns is called full pivoting and it is the best method for achieving diagonal dominance. We are going to look at partial pivoting because it works for many problems and is much simpler to implement. With partial pivoting, we are only going to move the rows.

Before we can perform either full or partial pivoting, we must normalize the rows. The process is illustrated in the following example. We start with the equations

$$\begin{array}{ccc|c} 0.5 & -8.9 & 3.2 & 7.1 \\ 4.9 & -7.2 & 22.3 & 4.2 \\ 0.2 & -.03 & .06 & 2.2 \end{array} \quad (3.13)$$

We normalize each row of the equations by dividing through by the largest term in magnitude in that row. This results in:

$$\begin{array}{ccc|c} -.00562 & 1 & -.35955 & -.79775 \\ .21973 & -.32287 & 1 & .18834 \\ 1 & -.15 & .3 & 11 \end{array} \quad (3.14)$$

3.4 Partial Pivoting

Now we can move the equations around (partial pivoting) so that the ones are on the diagonal. It is important to notice that normalization must be done first because without normalization it would be difficult to know how to rearrange the equation. There would be no basis for the row by row comparison.

$$\begin{array}{ccc|c} 1 & -.15 & .3 & 11 \\ -.00562 & 1 & -.35955 & -.79775 \\ .21973 & -.32287 & 1 & .18834 \end{array} \quad (3.15)$$

These are the equations we solve using Gaussian elimination.

3.5 Overall Process

The overall process becomes

- a. Normalize each equation
- b. Move the equations to achieve diagonal dominance (partial pivoting).

- c. Do a forward sweep that transforms the matrix to upper triangular form
- d. Do a backwards sweep that solves for the values of the unknowns.

This algorithm can be written into very compact and efficient computer algorithms and is one of the more common ways of solving large numbers of simultaneous linear equations.

The following program reads a data file containing a matrix defining a set of linear simultaneous linear equations. The program then executes 4 functions to solve the system. The first function normalizes the equations, the second does partial pivoting to achieve diagonal dominance, the third puts the matrix in upper triangular form, and the fourth function does the back substitution to solve the system of equation. The program is sized to handle up to 100 equations and 100 unknowns.

Four functions are used to solve the equations. They are:

normal	which normalizes each row of the matrix
pivot	attempts to improve diagonal dominance by exchanging the position of rows.
forelm	the forward sweep that puts the matrix into upper triangular form
baksub	which does the back substitution to solve for the unknowns

These routines are very short and efficient.

```
// gauss.C - This program tests a series of routines that solve multiple
//           linear equations using Gaussian elimination.

#include <fstream.h>
#include <iomanip.h>
#include <math.h>

//      ROUTINES

void baksub (double a[100][100], double b[100], double x[100], int m);
void gauss (double a[100][100], double b[100], double x[100], int m);
void normal (double a[100][100], double b[100], int m);
void pivot (double a[100][100], double b[100], int m);
void forelm (double a[100][100], double b[100], int m);
void baksub (double a[100][100], double b[100], double x[100], int m);

void main (void)
{
    int      size, i, j;
    double   a[100][100], b[100], x[100];
    ifstream input;

// Open the file and read the input data. The equations in the file are
// written in an augmented matrix format

    input.open ("gauss.dat");
    input >> size;

    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size; j++)
            input >> a[i][j];
    }
}
```

```

        input >> b[i];
    }

    gauss (a, b, x, size);

// print out the results

    for (i = 0; i < size; i++)
        cout << "x[" << i << "] = " << x[i] << endl;
}

// Here are a series of routines that solve multiple linear equations
// using the Gaussian Elimination technique

void gauss (double a[100][100], double b[100], double x[100], int m)
{
    // Normalize the matix

    normal (a, b, m);

    // Arrange the equations for diagonal dominance

    pivot (a, b, m);

    // Put into upper triangular form

    forelm (a, b, m);

    // Do the back substitution for the solution

    baksub (a, b, x, m);
}

// This routine normalizes each row of the matrix so that the largest
// term in a row has an absolute value of one

void normal (double a[100][100], double b[100], int m)
{
    int    i, j;
    double big;

    for (i = 0; i < m; i++)
    {
        big = 0.0;
        for (j = 0; j < m; j++)
            if (big < fabs(a[i][j])) big = fabs(a[i][j]);

        for (j = 0; j < m; j++)
            a[i][j] = a[i][j] / big;

        b[i] = b[i] / big;
    }
}

// This routine attempts to rearrange the rows of the matrix so
// that the maximum value in each row occurs on the diagonal.

void pivot (double a[100][100], double b[100], int m)
{
    int    i, j, ibig;
    double temp;

```

```

    for (i = 0; i < m-1; i++)
    {
        ibig = i;
        for (j = i+1; j < m; j++)
            if (fabs (a[ibig][i]) < fabs (a[j][i])) ibig = j;

        if (ibig != i)
        {
            for (j = 0; j < m; j++)
            {
                temp = a[ibig][j];
                a[ibig][j] = a[i][j];
                a[i][j] = temp;
            }

            temp = b[ibig];
            b[ibig] = b[i];
            b[i] = temp;
        }
    }
}

// This routine does the forward sweep to put the matrix in to upper
// triangular form

void    forelm (double a[100][100], double b[100], int m)
{
    int    j, k, i;
    double    fact;

    for (i = 0; i < m-1; i++)
    {
        for (j = i+1; j < m; j++)
        {
            if (a[i][i] != 0.0)
            {
                fact = a[j][i] / a[i][i];
                for (k = 0; k < m; k++)
                    a[j][k] -= a[i][k] * fact;

                b[j] -= b[i]*fact;
            }
        }
    }
}

// This routine does the back substitution to solve the equations

void    baksub (double a[100][100], double b[100], double x[100], int m)
{
    int    i, j;
    double    sum;

    for (j = m-1; j >= 0; j--)
    {
        sum = 0.0;
        for (i = j+1; i < m; i++)
            sum += x[i] * a[j][i];

        x[j] = (b[j] - sum) / a[j][j];
    }
}

```