LECTURE 5:Systems of linear equations

October 3, 2011

1 Naive Gaussian elimination

Consider the following system of n linear equations with n unknowns:

$$\left\{\begin{array}{ccccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \ldots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & \ldots & + & a_{2n}x_n & = & b_2 \\
a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & + & \ldots & + & a_{3n}x_n & = & b_3 \\
\vdots & & \vdots & & \vdots & & & & \vdots & & \vdots \\
a_{i1}x_1 & + & a_{i2}x_2 & + & a_{i3}x_3 & + & \ldots & + & a_{in}x_n & = & b_i \\
\vdots & & \vdots & & \vdots & & & & \vdots & & \vdots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & a_{n3}x_3 & + & \ldots & + & a_{nn}x_n & = & b_n
\end{array}\right.$$

In compact form, this system can be written as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \quad (1 \leq i \leq n)$$

In these equations, a_{ij} and b_i are prescribed real numbers (data), and the unknowns x_j are to be determined.

The objective in this chapter is to find an efficient way for obtaining the numerical solution of such a set of linear equations. We begin by discussing the simplest form of a procedure known as Gaussian elimination. This form is not suitable for automatic computation, but serves as a starting point for developing a more robust algorithm.

Example

We demonstrate the simplest form of the Gaussian elimination procedure using a numerical example. In this form, the procedure is not suitable for numerical computation, but it can be modified as we see later on.

Consider the following four equations with four unknowns:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ 12x_1 - 8x_2 + 6x_3 + 10x_4 = 26 \\ 3x_1 - 13x_2 + 9x_3 + 3x_4 = -19 \\ -6x_1 + 4x_2 + x_3 - 18x_4 = -34 \end{cases}$$

In the first round of the procedure, we eliminate x_1 from the last three equations. This is done by *subtracting* multiples of the first equation from the other ones. The result is

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ - 4x_2 + 2x_3 + 2x_4 = -6 \\ - 12x_2 + 8x_3 + x_4 = -27 \\ 2x_2 + 3x_3 - 14x_4 = -18 \end{cases}$$

Note that the first equation was not changed in the elimination procedure. It is called the *pivot equation*.

The second step consists of eliminating x_2 from the last two equations. We ignore the first equation and use the second one as the pivot equation. We get

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ - 4x_2 + 2x_3 + 2x_4 = -6 \\ + 2x_3 + 5x_4 = -9 \\ 4x_3 - 13x_4 = -21 \end{cases}$$

Finally we eliminate x_3 from the last equation and obtain:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ - 4x_2 + 2x_3 + 2x_4 = -6 \\ + 2x_3 + 5x_4 = -9 \\ - 3x_4 = -3 \end{cases}$$

This system is said to be in the *upper triangular form*. It is equivalent to the system we started with. This completes the first phase of the Gaussian elimination, called *forward elimination.*

The second phase is called *back substitution.* We solve the system in the upper triangular form, starting from the bottom.

From the last equation we obtain

$$x_4 = \frac{-3}{-3} = 1$$

Substituting this to the second last equation gives

$$2x_3 - 5 = 9$$

Eventually we obtain the solution

$$x_1 = 3 \quad x_2 = 1 \quad x_3 = -2 \quad x_4 = 1$$

Algorithm

We now write the system of n linear equations in matrix form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix}$$

or

$$\mathbf{Ax} = \mathbf{b}$$

In the naive Gaussian elimination algorithm, the original data are overwritten with new computed values. In the forward elimination phase, there are $n-1$ principal steps.

The first step consists of subtracting appropriate multiples of the first equation from the others. The first equation is called the first *pivot equation* and a_{11} is called the first *pivot element*.

For each of the other equations ($2 \leq i \leq n$), we compute

$$\begin{cases} a_{ij} & \leftarrow a_{ij} - \left(\frac{a_{i1}}{a_{11}}\right) a_{1j} \quad 1 \leq j \leq n \\ \\ b_i & \leftarrow b_i - \left(\frac{a_{i1}}{a_{11}}\right) b_1 \end{cases}$$

The quantities a_{i1}/a_{11} are the *multipliers*. The new coefficient for x_1 in all of the equations $2 \leq i \leq n$ will be zero because $a_{i1} - (a_{i1}/a_{11})a_{11} = 0$.

After the first step, the system is in the form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{i2} & a_{i3} & \dots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix}$$

From now on, we ignore the first row (and effectively also the first column). We take the second row as the pivot equation and compute for the remaining equations ($3 \leq i \leq n$)

$$\begin{cases} a_{ij} & \leftarrow a_{ij} - \left(\frac{a_{i2}}{a_{22}}\right) a_{2j} \quad 2 \leq j \leq n \\ \\ b_i & \leftarrow b_i - \left(\frac{a_{i2}}{a_{22}}\right) b_2 \end{cases}$$

This step eliminates x_2 from the equations $3 \leq i \leq n$.

In the kth step, we use the kth row as the pivot equation and for the remaining equations below the kth row ($k+1 \leq i \leq n$), we compute

$$\begin{cases} a_{ij} & \leftarrow a_{ij} - \left(\frac{a_{ik}}{a_{kk}}\right) a_{kj} \quad k \leq j \leq n \\ \\ b_i & \leftarrow b_i - \left(\frac{a_{ik}}{a_{kk}}\right) b_k \end{cases}$$

Note that we must assume that all the divisors in this algorithm are nonzero!!

In the beginning of the *back substitution* phase, the system is in the upper triangular form.

Back substitution starts by solving the nth equation for x_n:

$$x_n = \frac{b_n}{a_{nn}}$$

Then using the $(n-1)$th equation, x_{n-1} is solved:

$$x_{n-1} = \frac{1}{a_{n-1,n-1}}(b_{n-1} - a_{n-1,n}x_n)$$

The formula for the ith equation is

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right) \qquad (i = n-1, n-2, \ldots, 1)$$

Code

The following C program performs the Naive Gaussian elimination:

```
#define N 4

int main(void)
{
  int i, j, k, n;
  double sum, xmult;
  double a[N][N], b[N], x[N];
  FILE *outfile;

  /* Read input */
  GetEquations(a,b);

  /* Forward elimination */
  for(k=0; k<N-1; k++) {
    for(i=k+1; i<N; i++) {
      xmult = a[i][k]/a[k][k];
      a[i][k] = xmult;
      for(j=k+1; j<N; j++) {
        a[i][j] = a[i][j] - xmult*a[k][j];
      }
      b[i] = b[i] - xmult*b[k];
    }
  }

  /* Back substitution */
  x[N-1] = b[N-1]/a[N-1][N-1];
  for(i=N-2; i>=0; i--) {
    sum = b[i];
    for(j=i+1; j<N; j++) {
      sum = sum - a[i][j]*x[j];
    }
    x[i] = sum/a[i][i];
  }
}
```

If we apply the code to the example case discussed before, we get the following output (only recalculated values):

```
           aij                    bi
 -4.0000   2.0000   2.0000     -6.0000
-12.0000   8.0000   1.0000    -27.0000
  2.0000   3.0000 -14.0000    -18.0000

           2.0000  -5.0000     -9.0000
           4.0000 -13.0000    -21.0000

                   -3.0000     -3.0000

-----------------------------------------

x[1] =  3.000000
x[2] =  1.000000
x[3] = -2.000000
x[4] =  1.000000
```

The result is the same as was previously obtained by hand.

This simple algorithm does not work in many cases. The crucial computation is the replacement

$$a_{ij} \leftarrow a_{ij} - \left(\frac{a_{ik}}{a_{kk}}\right) a_{kj}$$

A *round-off error* in a_{kj} is multiplied by the factor (a_{ik}/a_{kk}). This factor is large if the pivot element $|a_{kk}|$ is small compared to $|a_{ik}|$.

The conclusion is that small pivot elements lead to worse round-off errors.

1.1 Condition number

The *condition number* of a linear system $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\kappa(\mathbf{A}) = ||\mathbf{A}||\ ||\mathbf{A}^{-1}||$$

Here, $||\mathbf{A}||$ is the matrix norm. There are several norms, e.g. l_1-norm for an $n \times n$-matrix $||\mathbf{A}||$ is $||\mathbf{A}||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{ij}|$. $\kappa(\mathbf{A})$ gauges the transfer of error from the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ to the solution $\mathbf{x}$. The rule of thumb is that if $\kappa(\mathbf{A}) = 10^k$, then one can expect to lose at least k digits of precision in solving the system $\mathbf{Ax} = \mathbf{b}$.

A matrix $\mathbf{A}$ is called ill-conditioned if the linear system $\mathbf{Ax} = \mathbf{b}$ is sensitive to perturbations in the elements of $\mathbf{A}$, or to perturbations of the components of $\mathbf{b}$. The larger the condition number is, the more *ill-conditioned* is the system.

In computational studies, it is extremely important to estimate whether a numerical result can be trusted or not. The condition number provides some evidence regarding this question if one is solving linear systems.

In mathematical software systems such as Matlab, an estimate of the condition number is often available so that one can estimate the reliability of the numerical result.

Such a criterion may be used for example to choose a suitable solution method.

1.2 Residual vectors

Without having the aid of sophisticated mathematical software, how can we determine whether a numerical solution to a linear system is correct?

One check is to *substitute* the obtained numerical solution $\hat{\mathbf{x}}$ back into the original system. This means computing the *residual vector*

$$\hat{\mathbf{r}} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b}$$

The situation is, however, more complicated than one would tend to imagine. A solution with a small residual vector is not necessarily a good solution. This can be the case if the original problem is sensitive to roundoff errors - ill-conditioned.

The problem of whether a computed solution is good or not is difficult and mostly beyond the scope of this course.

2 Gaussian elimination with scaled partial pivoting

2.1 Failure of Naive Gaussian elimination

Consider the following simple example

$$\begin{cases} 0x_1 & + & x_2 & = & 1 \\ x_1 & + & x_2 & = & 2 \end{cases}$$

The Naive Gaussian elimination program would fail here because the pivot element $a_{11} = 0$.

The program is also unreliable for small values of a_{11}. Consider the system

$$\begin{cases} \varepsilon x_1 & + & x_2 & = & 1 \\ x_1 & + & x_2 & = & 2 \end{cases}$$

where ε is a small number other than zero.

The forward elimination produces the following system

$$\begin{cases} \varepsilon x_1 & + & x_2 & = & 1 \\ & & (1-1/\varepsilon)x_2 & = & 2-1/\varepsilon \end{cases}$$

In the back substitution, we obtain

$$x_2 = \frac{2-1/\varepsilon}{1-1/\varepsilon} \qquad x_1 = \frac{1-x_2}{\varepsilon}$$

If ε is small, then $1/\varepsilon$ is large, and in a computer with finite word length, both the values $(2-1/\varepsilon)$ and $(1-1/\varepsilon)$ would be computed as $-1/\varepsilon$.

Thus the computer calculates that $x_2 = 1$ and $x_1 = 0$. But the correct answer is $x_2 \approx 1$ and $x_1 \approx 1$. The relative error in the computed solution for x_1 is 100%!!

If we change the order of the equations, the Naive Gaussian algorithm works:

$$\begin{cases} x_1 & + & x_2 & = & 2 \\ \varepsilon x_1 & + & x_2 & = & 1 \end{cases}$$

This becomes

$$\begin{cases} x_1 & + & x_2 & = & 2 \\ & & (1-\varepsilon)x_2 & = & 1-2\varepsilon \end{cases}$$

And the back substitution gives

$$x_2 = \frac{1-2\varepsilon}{1-\varepsilon} \approx 1 \qquad x_1 = 2 - x_2 \approx 1$$

This idea of changing the order of the equations is used in a procedure called scaled partial pivoting.

2.2 Scaled partial pivoting

We have seen that the *order* in which we treat the equations significantly affects the accuracy of the algorithm.

We now describe an algorithm where the order of the equations is determined by the system itself. We call the original order in which the equations are given the *natural order* of the system: $\{1,2,3,\ldots,n\}$. The new order in which the elimination is performed is denoted by the row vector

$$\mathbf{l} = [l_1, l_2, \ldots, l_n]$$

This is called the *index vector*. The strategy of interchanging the rows is called *scaled partial pivoting*.

We begin by calculating a *scale factor* for each equation:

$$s_i = \max_{1 \le j \le n} |a_{ij}| \quad (1 \le i \le n)$$

These n numbers are recorded in a *scale vector*: $\mathbf{s} = [s_1, s_2, \ldots, s_n]$. The scale vector is not updated during the procedure because according to a common view, this extra computation is not worth the effort.

Next we choose the pivot equation based on the scale factors. The equation for which the ratio $|a_{i1}|/s_i$ is the greatest is chosen to be the first pivot equation.

Initially we set for the index vector $\mathbf{l} = [l_1, l_2, \ldots, l_n] = [1,2,\ldots,n]$.

If j is selected to be the index of the first pivot equation, we interchange l_j with l_1 in the index vector. Thus we get $\mathbf{l} = [l_j, l_2, \ldots, l_1, \ldots, l_n]$.

Next we use the multipliers

$$\frac{a_{l_i 1}}{a_{l_1 1}}$$

times the row l_1 and subtract these from the equations l_i for $2 \le i \le n$. (Note that now $l_1 \leftrightarrow l_j$.) This is the first step of the forward elimination.

Note that only the index vector is updated, *not* the equations!

In the next step, we go through the ratios

$$\left\{ \frac{|a_{l_i 2}|}{s_{l_1}} \ : \ 2 \le i \le n \right\}$$

If j is now the index corresponding to the largest ratio, then interchange l_j and l_2 in the index vector.

Continue by subtracting the multipliers

$$\frac{a_{l_i 1}}{a_{l_2 1}}$$

times equation l_2 from the remaining equations l_i for which $3 \le i \le n$.

The same procedure is repeated until $n-1$ steps have been completed.

Numerical example

Consider the system:

$$\begin{bmatrix} 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \\ 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -19 \\ -34 \\ 16 \\ 26 \end{bmatrix}$$

The index vector is initially $\mathbf{l} = [1,2,3,4]$.

The scale vector is $\mathbf{s} = [13,18,6,12]$.

Step 1.

In order to determine the first pivot row, we look at the ratios:

$$\left\{ \frac{|a_{l_i 1}|}{s_{l_i}} \ : \ i = 1,2,3,4 \right\} = \left\{ \frac{3}{13}, \frac{6}{18}, \frac{6}{6}, \frac{12}{12} \right\} \approx \{0.23, 0.33, 1.0, 1.0\}$$

The largest value of these ratios occurs *first* for the index $j = 3$. So the third row is going to be our pivot equation in step 1 of the elimination process.

The entries l_1 and l_j are interchanged in the index vector. Thus we have $\mathbf{l} = [3,2,1,4]$.

Now appropriate multiples of the pivot equation ($l_1 = 3$) are subtracted from the other equations so that zeros are created as coefficients of x_1. The result is

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \\ 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -18 \\ 16 \\ -6 \end{bmatrix}$$

Step 2.

In the next step ($k = 2$), we use the index vector $\mathbf{l} = [3,2,1,4]$ and scan the rows $l_2 = 2$, $l_3 = 1$ and $l_4 = 4$ for the largest ratio:

$$\left\{ \frac{|a_{l_i 2}|}{s_{l_i}} \ : \ i = 2,3,4 \right\} = \left\{ \frac{2}{18}, \frac{12}{13}, \frac{4}{12} \right\} \approx \{0.11, 0.92, 0.33\}$$

The largest is the second value (with index $l_3 = 1$). Thus the first row is our new pivot equation and the index vector becomes $\mathbf{l} = [3,1,2,4]$ after interchanging l_3 and l_2.

Next multiples of the first equation are subtracted from the two remaining equations $l_3 = 2$ and $l_4 = 4$. The result is

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & 13/3 & -83/6 \\ 6 & -2 & 2 & 4 \\ 0 & 0 & -2/3 & 5/3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -45/2 \\ 16 \\ 3 \end{bmatrix}$$

Step 3.

In the third and final step, we first examine the ratios

$$\left\{ \frac{|a_{l_i 3}|}{s_{l_i}} \ : \ i = 3,4 \right\} = \left\{ \frac{13/3}{18}, \frac{2/3}{12} \right\} \approx \{0.24, 0.06\}$$

The largest value is for $l_3 = 2$, and thus the 2nd row is used as the pivot equation. The index vector is unchanged because $l_k = l_j$ (index of step is the same as the index of pivot equation). We have $\mathbf{l} = [3,1,2,4]$.

The remaining task is to subtract the appropriate multiple of equation 2 from equation 4. This gives

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & 13/3 & -83/6 \\ 6 & -2 & 2 & 4 \\ 0 & 0 & 0 & -6/13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -45/2 \\ 16 \\ -6/13 \end{bmatrix}$$

Back substitution

The final index vector gives the order in which the *back substitution* is performed. The solution is obtained by first using $l_4 = 4$ to determine x_4, then $l_3 = 2$ to determine x_3, then $l_2 = 1$ to determine x_2 and finally $l_1 = 3$ to determine x_1.

After the back substitution, we obtain the result

$$\mathbf{x} = \begin{bmatrix} 3 \\ 1 \\ -2 \\ 1 \end{bmatrix}$$

Code

In the following C implementation, the algorithm is programmed in such a way that it first carries out the forward elimination phase on the coefficient array (a_{ij}) only (procedure `Gauss`).

The right-hand-side array (b_i) is treated in the next phase (procedure `Solve`).

Since we are treating the array (b_i) later, we need to store the multipliers. These are easily stored in the array (a_{ij}) in positions where 0 entries would have been created otherwise.

Here is the procedure `Gauss` for forward elimination with scaled partial pivoting.

```
void Gauss(int n, double a[N][N], int l[N])
{
  int i, j, k, temp;
  double aa, r, rmax, smax, xmult;
  double *s;

  /* Allocate memory */
  s = (double *) malloc((size_t) (n*sizeof(double)));

  /* Create the scaling vector */
  for(i=0; i<n; i++) {
    l[i] = i;
    smax = 0;
    for(j=0; j<n; j++) {
      aa = fabs(a[i][j]);
      if(aa > smax)
        smax = aa;
    }
    s[i] = smax;
  }

  /* Loop over steps */
  for(k=0; k<n-1; k++) {
    /* Choose pivot equation */
    rmax = 0;
    j=k;
    for(i=k; i<n; i++) {
      r = fabs(a[l[i]][k]/s[l[i]]);
      if(r > rmax) {
        rmax = r;
        j = i;
      }
    }
    /* Interchange indeces */
    temp = l[j];
    l[j] = l[k];
    l[k] = temp;
    /* Elimination for step k */
    for(i=k+1; i<n; i++) {
      xmult = a[l[i]][k]/a[l[k]][k];
      a[l[i]][k] = xmult;
      for(j=k+1; j<n; j++) {
        a[l[i]][j] = a[l[i]][j] - xmult*a[l[k]][j];
      }
    }
  }
  free(s);
}
```

And here is the procedure `Solve` for determining the array (b_i) and for the back substitution phase:

```
void Solve(int n, double a[N][N], int l[N], double b[N], double x[N])
{
  int i, j, k;
  double sum;

  /* Determine array bi */
  /* Elements a[l[i]][k] are the multipliers */
  for(k=0; k<n-1; k++) {
    for(i=k+1; i<n; i++) {
      b[l[i]] =  b[l[i]]-a[l[i]][k]*b[l[k]];
    }
  }
  /* Back substitution phase */
  x[n-1]= b[l[n-1]]/a[l[n-1]][n-1];
  for(i=n-2; i>=0; i--) {
    sum = b[l[i]];
    for(j=i+1; j<n; j++) {
      sum = sum - a[l[i]][j]*x[j];
    }
    x[i] = sum/a[l[i]][i];
  }
}
```

If we use the program on the same problem as solved before by hand and using the naive Gaussian elimination program, we get the following results (note that in C the indexing goes from 0 to $n-1$, here it has been changed to run from 1 to n:

```
Scale vector:
s[1] = 6.000000
s[2] = 12.000000
s[3] = 13.000000
s[4] = 18.000000

Final index vector:
l[1] = 1
l[2] = 3
l[3] = 4
l[4] = 2

Arrays (aij) and (bi) (multipliers in place of 0's):
    ai1       ai2       ai3      ai4      |      bi
  6.0000   -2.0000    2.0000    4.0000    |   16.0000
  2.0000    0.3333   -0.1538 | -0.4615    |   -0.4615
  0.5000 | -12.0000   8.0000    1.0000    |  -27.0000
 -1.0000   -0.1667  | 4.3333   -13.8333   |  -22.5000

Solution:
x[1] =  3.000000
x[2] =  1.000000
x[3] =  -2.000000
x[4] =  1.000000
```

We obtain the same solution as before but the arrays (aij) and (*bi*) are different (due to different sequence of pivot equations).

2.3 Long operation count

Solving large systems of linear equations on a computer can be expensive. We will now count the number of *long operations* involved in the Gaussian elimination with partial pivoting.

By long operations we mean multiplications and divisions. We make no distinction between the two, although division is more time consuming than multiplication.

i. Procedure *Gauss*

Step 1:
- Choice of pivot element: n divisions
- Multipliers (one for each row): $n-1$ divisions
- New elements: $n-1$ multiplications / row $\rightarrow (n-1)\times(n-1)$
- Total: $n+(n-1)+(n-1)(n-1)=n+n(n-1)=n^2$ long operations

Step 2:
- Involves $n-1$ rows and $n-1$ columns
- Total: $(n-1)^2$ long operations

Whole procedure:
-Total:

$$n^2+(n-1)^2+(n-2)^2+\ldots+3^2+2^2=\frac{n}{6}(n+1)(2n+1)-1\approx\frac{n^3}{3}$$

ii. Procedure *Solve*

Forward processing of (b_i): - Involves $n-1$ steps
- First step: $n-1$ multiplications
- Second step: $n-2$ multiplications, etc.
- Total: $(n-1)+(n-2)+\ldots+3+2+1=\frac{n}{2}(n-1)$

Back substitution: - One operation in first step, two in the second and so on.
- Total: $1+2+3+\ldots+n=\frac{n}{2}(n+1)$

Whole procedure: -Total:

$$\frac{n}{2}(n-1)+\frac{n}{2}(n+1)=\frac{n}{2}(n-1)(2n)=n^2$$

Theorem on long operations.
The forward elimination phase of the Gaussian elimination algorithm with scaled partial pivoting, if applied only to the $n\times n$ coefficient array, involves approximately $n^3/3$ long operations (multiplications or divisions). Solving for x requires an additional n^2 long operations.

---

3 LU factorization

An $n \times n$ system of linear equations can be written in matrix form

$$\mathbf{Ax} = \mathbf{b}$$

where the coefficient matrix $\mathbf{A}$ has the form

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

We now want to show that the naive Gaussian algorithm applied to $\mathbf{A}$ yields a factorization of $\mathbf{A}$ into two products:

$$\mathbf{A} = \mathbf{LU}$$

where $\mathbf{L}$ has a *lower triangular* form:

$$\mathbf{L} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{bmatrix}$$

and $\mathbf{U}$ has a *upper triangular* form:

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix}$$

This is called the **LU factorization**.

3.1 Benefit of LU factorization

Using the LU factorization of **A** we can write

$$\mathbf{Ax} = (\mathbf{LU})\mathbf{x} = \mathbf{L}(\mathbf{Ux}) = \mathbf{b}$$

This means that we can solve the system of linear equations in two parts:

$$\mathbf{L}\,\mathbf{z} = \mathbf{b}$$
$$\mathbf{U}\,\mathbf{x} = \mathbf{z}$$

Solving these sets of equations is trivial since the matrices are in triangular form.

We first solve for **z** using the first equation. Since **L** is lower triangular, we can use *forward substitution*:

$$z_1 = \frac{b_1}{l_{11}}$$
$$z_i = \frac{1}{l_{ii}}\left[b_i - \sum_{j=1}^{i-1} l_{ij} z_j\right] \quad (i = 2,3,\ldots,n)$$

Once we have **z**, the second equation can be solved using *back substitution*:

$$x_n = \frac{z_n}{u_{nn}}$$
$$x_i = \frac{1}{u_{ii}}\left[z_i - \sum_{j=i+1}^{N} u_{ij} x_j\right] \quad (i = n-1, n-2, \ldots, 1)$$

The benefit of this method is that once we have the LU decomposition of **A**, we can use the same matrices **U** and **L** to solve for *several* different vectors **b**.

The CPU time requirement for the above forward and back substitutions is $O(n^2)$, whereas the Gauss elimination algorithm is $O(n^3)$.

Numerical example

The first example (in Naive Gaussian elimination) can be written in matrix form as:

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 16 \\ 26 \\ -19 \\ -34 \end{bmatrix}$$

We obtained the following upper triangular form for the system as a result of the forward elimination phase:

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 16 \\ -6 \\ -9 \\ -3 \end{bmatrix}$$

The forward elimination phase can be interpreted as starting from the first equation, $\mathbf{Ax} = \mathbf{b}$, and proceeding to

$$\mathbf{MAx} = \mathbf{Mb}$$

where $\mathbf{M}$ is a matrix chosen so that $\mathbf{MA}$ is the coefficient matrix in upper triangular form:

$$\mathbf{MA} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix} \equiv \mathbf{U}$$

The forward elimination phase of the naive Gaussian elimination algorithm consists of a series of steps. The first step gives us the following system:

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 16 \\ -6 \\ -27 \\ -18 \end{bmatrix}$$

The first step can be accomplished by multiplying by a lower triangular matrix $\mathbf{M_1}$:

$$\mathbf{M_1 Ax} = \mathbf{M_1 b}$$

where

$$\mathbf{M_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Notice the special form of this matrix: the diagonal elements are all 1's and the only other nonzero elements are in the first column. These are the *negatives of the multipliers*.

The second step results in the system:

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 4 & -13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 16 \\ -6 \\ -9 \\ -21 \end{bmatrix}$$

which is equivalent to

$$\mathbf{M_2 M_1 A x = M_2 M_1 b}$$

where

$$\mathbf{M_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & \frac{1}{2} & 0 & 1 \end{bmatrix}$$

Finally, step 3 gives the system in the upper triangular form

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 16 \\ -6 \\ -9 \\ -3 \end{bmatrix}$$

which is equivalent to

$$\mathbf{M_3 M_2 M_1 A x = M_3 M_2 M_1 b}$$

where

$$\mathbf{M_3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix}$$

Thus we obtain the matrix $\mathbf{M}$ as a product of the three multiplier matrices:

$$\mathbf{M = M_3 M_2 M_1}$$

We can now derive a different interpretation of the forward elimination phase. We see that

$$\begin{aligned}\mathbf{A} &= \mathbf{M}^{-1}\mathbf{U} \\ &= \mathbf{M_1}^{-1}\mathbf{M_2}^{-1}\mathbf{M_3}^{-1}\mathbf{U} \\ &= \mathbf{LU}\end{aligned}$$

The inverse of the $\mathbf{M_k}$'s is obtained easily by changing the signs of the multipliers. Thus we have

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & -\frac{1}{2} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ \frac{1}{2} & 3 & 1 & 0 \\ -1 & -\frac{1}{2} & 2 & 1 \end{bmatrix}$$

This is a unit lower triangular matrix composed of the multipliers!

We can easily verify that

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ \frac{1}{2} & 3 & 1 & 0 \\ -1 & -\frac{1}{2} & 2 & 1 \end{bmatrix} \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix} = \mathbf{A}$$

In LU factorization, the matrix **A** is *factored* or *decomposed* into a unit lower triangular matrix **L** and an upper triangular matrix **U**.

- The lower triangular elements of **L** are the multipliers located in the positions of the elements they annihilated from **A**.
- **U** is the final coefficient matrix obtained after the forward elimination phase.

Note that the naive Gaussian elimination algorithm replaces the original coefficient matrix **A** with its **LU** factorization.

3.2 LU factorization theorem

Theorem.
Let $\mathbf{A} = (a_{ij})$ be an $n \times n$ matrix. Assume that the forward elimination phase of the naive Gaussian algorithm is applied to $\mathbf{A}$ without encountering any 0 divisors. Let the resulting matrix be denoted by $\widetilde{\mathbf{A}} = (\widetilde{a}_{ij})$. If

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \widetilde{a}_{21} & 1 & 0 & \dots & 0 \\ \widetilde{a}_{31} & \widetilde{a}_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \widetilde{a}_{n1} & \widetilde{a}_{n2} & \dots & \widetilde{a}_{n,n-1} & 1 \end{bmatrix}$$

and

$$\mathbf{U} = \begin{bmatrix} \widetilde{a}_{11} & \widetilde{a}_{12} & \widetilde{a}_{13} & \dots & \widetilde{a}_{1n} \\ 0 & \widetilde{a}_{22} & \widetilde{a}_{23} & \dots & \widetilde{a}_{2n} \\ 0 & 0 & \widetilde{a}_{33} & \dots & \widetilde{a}_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \widetilde{a}_{n,n} \end{bmatrix}$$

then $\mathbf{A} = \mathbf{LU}$.

Notice that we rely here on *not* encountering any zero divisors in the algorithm.

3.3 Solving linear systems using LU factorization

Once the LU factorization of $\mathbf{A}$ is available, we can solve the system

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

by writing

$$\mathbf{L}\,\mathbf{U}\,\mathbf{x} = \mathbf{b}$$

Then we solve two triangular systems:

$$\mathbf{L}\,\mathbf{z} = \mathbf{b}$$

for $\mathbf{z}$ and

$$\mathbf{U}\,\mathbf{x} = \mathbf{z}$$

for $\mathbf{x}$.

This is particularly useful if we have a problem that involves the same coefficient matrix $\mathbf{A}$ and several different right-hand side vectors $\mathbf{b}$.

$\mathbf{z}$ is obtained by the pseudocode (forward substitution):

real array $(b_i)_n$, $(l_{ij})_{n\times n}$, $(z_i)_n$
integer i, n
$z_1 \leftarrow b_1$
for $i = 2$ **to** n **do**
    $z_i \leftarrow b_i - \sum_{j=1}^{i-1} l_{ij} z_j$
end for

And similarly $\mathbf{x}$ is obtained by the pseudocode (back substitution):

real array $(x_i)_n$, $(u_{ij})_{n\times n}$, $(z_i)_n$
integer i, n
$x_n \leftarrow z_n / u_{nn}$
for $i = n-1$ **to** 1 **step-1 do**
    $x_i \leftarrow \frac{1}{u_{ii}} \left(z_i - \sum_{j=i+1}^{n} u_{ij} x_j \right)$
end for

3.4 Software packages

Matlab and Maple produce factorizations of the form

$$\mathbf{P}\,\mathbf{A} = \mathbf{L}\,\mathbf{U}$$

where **P** is a **permutation matrix** corresponding to the pivoting strategy used. Example of such a matrix is

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

A permutation matrix is an $n \times n$ matrix which is obtained from the unit matrix by permuting its rows. Permuting the rows of any $n \times n$ matrix **A** can be accomplished by multiplying **A** on the left by **P**.

When Gaussian elimination with row pivoting is performed on **A**, the result is expressible as $\mathbf{P}\,\mathbf{A} = \mathbf{L}\,\mathbf{U}$.

The matrix $\mathbf{P}\,\mathbf{A}$ is **A** with rows rearranged.

We can use the LU factorization of $\mathbf{P}\,\mathbf{A}$ to solve the system $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ in the following way:

$$\begin{aligned} \mathbf{P}\,\mathbf{A}\,\mathbf{x} &= \mathbf{P}\,\mathbf{b} \\ \mathbf{L}\,\mathbf{U}\,\mathbf{x} &= \mathbf{P}\,\mathbf{b} \\ \mathbf{L}\,\mathbf{y} &= \mathbf{P}\,\mathbf{b} \end{aligned}$$

This is easily solved for **y**. The solution is obtained by solving

$$\mathbf{U}\,\mathbf{x} = \mathbf{y}$$

for **x**.