Sets of linear equations

Sets of linear equations are discussed in the school mathematics course, although there they are not really needed; most examples look quite artificial.

In numerical analysis linear equations and more generally linear algebra, however, have many applications. They are needed e.g. in the linear least squares method or when solving differential equations.

Gaussian elimination

The method learned at schooll is known as the Gaussian elimination. The equations are transformed by eliminating unknowns. The elimination stage will eventually lead to an equation with one unknown only. In the back substitution stage this value is substituted to all other equations reducing the number of unknowns by one.

Elimination stage:

Subtract the first equation multiplied by suitable factors from the other ones in such a way that the first unknown is eliminated from the other equations.

$$\begin{aligned} x + y + z &= 1, \\ x - y - z &= 2, \\ 2x + y - z &= 2. \end{aligned}$$

Thus we get the equations

$$\begin{aligned} x + y + z &= 1, \\ -2y - 2z &= 1, \\ -y - 3z &= 0. \end{aligned}$$

We'll repeat the same procedure for the last two equations. This will leave only one unknown in the last equation:

$$\begin{aligned} x + y + z &= 1, \\ -2y - 2z &= 1, \\ -2z &= -0.5. \end{aligned}$$

Back substitution:

Last equation $\Rightarrow z = 0.25$.

Substitute this to the second equation: $-2y - 0.5 = 1 \Rightarrow y = -0.75$.

Substitute the values to the first equation: $x - 0.5 = 1 \Rightarrow x = 1.5$.

For programming the method is expressed formally in terms of matrices. At the elimination stage transform the matrix of coefficients to an upper triangular matrix having only zeroes below the diagonal.

```
! Gaussian elimination
! elimination stage
do i=1,n-1
 do k=i+1,n
  c=A(k,i)/A(i,i)
  do j=i,n
    A(k,j)=A(k,j)-c*A(i,j)
  end do
  b(k) = b(k)-c*b(i)
 end do
end do
```

Back substituion goes from bottom upwards:

```
! Gaussian elimination
! back substitution
do i=n,1,-1
  x(i)=b(i)/A(i,i)
  do k=i-1,1,-1
    b(k)=b(k)-A(k,i)*x(i)
  end do
end do
```

The coefficient matrix `A` is replaced by an upper triangular matrix; the original values are overwritten.

At the back substituion stage the elements of the vector `b` below the line being processed are not needed; thus the solution can be stored to the vector `b`.

```
! back substitution
! the solution is written to the constant vector
do i=n,1,-1
  b(i)=b(i)/A(i,i)
  do k=i-1,1,-1
    b(k)=b(k)-A(k,i)*b(i)
  end do
end do
```

If there are n equations, space is needed for $n^2 + n$ variables.

Elimination fails if the divisor is zero. Rearrange the equations so that the divisor is non-zero. Reordering the equations does not change the solution.

Reordering of lines is called **partial pivoting**.

It is advantageous to find a line that will make the divisor as big as possible.

The program using partial pivoting is:

```
! Gaussian elimination
! elimination using partial pivoting
do i=1,n
  ! find the largest element on column i
  m=i
  s=A(i,i)
  do k=i+1,n
    if (abs(A(k,i)) > s) then
      s=abs(A(k,i)); m=k
    end if
  end do

  ! the largest element is on line m
```

```
    ! exchange lines i and m
    do l=i,n
      x=A(i,l); A(i,l)=A(m,l); A(m,l)=x
    end do
    x=b(i); b(i)=b(m); b(m)=x

    ! eliminate element i
    if (A(i,i)==0) then exit
    do k=i+1,n
      c=A(k,i)/A(i,i)
      do j=i,n
        A(k,j)=A(k,j)-c*A(i,j)
      end do
      b(k) = b(k)-c*b(i)
    end do
  end do
```

Complete pivoting: maximise the divisor by rearranging also columns. The order of the unknowns will change; so extra bookkeeping is required so that the solution can be arranges to correspond to the original set of equations.

In practice, it is not necessary to exchange the lines of the matrix. Instead we can use an index vector that will tell the order of the lines. (The index jungle is now getting rather dense ...)

```
! Gaussian elimination
...
! initialise the index vector
do i=1,n
  ind(i)=i
end do

! elimination using partial pivoting
do i=1,n

 ! find the largest element on column i
 m=i
 s=A(ind(m),i)
 do k=i+1,n
  if (abs(A(ind(k),i)) > s) then
    s=abs(A(ind(k),i)); m=k
  end if
 end do
```

```
  ! largest value on line m;
  ! exchange indices of lines i and m
  l=ind(i); ind(i)=ind(m); ind(m)=l

  ! eliminate variable i
  ii=ind(i)
  if (abs(a(ii,i)) < limit) then exit
  do k=i+1,n
   kk=ind(k)
   c=A(kk,i)/A(ii,i)
   do j=i,n
      A(kk,j)=A(kk,j)-c*A(ii,j)
   end do
   b(kk) = b(kk)-c*b(ii)
  end do
end do
```

```
! back substitution
! solution will replace the constant vector
do i=n,1,-1
  ii=ind(i)
  b(ii)=b(ii)/A(ii,i)
  do k=i-1,1,-1
    kk=ind(kk)
    b(kk)=b(kk)-A(kk,i)*b(ii)
  end do
end do

! write the solution
do i=1,n
   write (6,*) b(ind(i))
end do
```

Estimating execution time

The innermost `j` loop in algorithm 1 is executed $n - i + 1$ times; each time one multiplication and one subtraction is computed. The `k` loop is executed $n - i$ times, and each time the `j` loop + three other operations are execued. Thus the number of floating point operations is

$$(3 + 2(n - i + 1))(n - i) = 2(n - i)^2 + 5(n - i).$$

These are executed with the values of `i` running from 1 to $n - 1$. The total number of operations is

$$\begin{aligned} N_1 &= \sum_{i=1}^{n-1} \left[2(n-i)^2 + 5(n-i)\right] \\ &= \sum_{i=1}^{n-1} \left[2n^2 + 5n + 2i^2 - (4n+5)i\right] \\ &= (n-1)(2n^2 + 5n) + 2\sum_{i=1}^{n-1} i^2 - (4n+5)\sum_{i=1}^{n-1} i. \end{aligned}$$

The sum $\sum i^p$ can be estimated by an integral:

$$\sum_{i=1}^{n} i^p \approx \int_0^n i^p \, di = \frac{n^{p+1}}{p+1}.$$

Hence $\sum_{i=1}^{n} i \propto n^2$ and $\sum_{i=1}^{n} i^2 \propto n^3$.

$$N_1 \propto n^3.$$

In the back substitution (algorithm 2) the inner loop is executed $i - 1$ times, and each time two operations are needed. This loop and one division are executed with the values of `i` running from i to n.

$$N_2 = \sum_{i=1}^{n} (1 + 2(i-1)) = \sum_{i=1}^{n} (2i - 1) \propto n^2.$$

The total number of floating point operations is

$$N = N_1 + N_2 \propto n^3$$

If the number of operations in method 1 is $N_1 = 10n^3$ and in method 2 $N_2 = 1000n^2$, the methods are equally efficient when $n = 100$. Because of its smaller exponent, in bigger problems method 2 is better than method 1. The advantage will increase as the size of the problem increases. But if n is almost always less than 100, it is better to use method 1, sinve obviously it is very simple and therefore efficient in small problems.

If the upper limit of the operations can be expressed as a polynomial of the problem size, the problem can be solved in polynomial time.

If the time needed increases faster than any polynomial the problem is called **NP complete**. If the time dependence of even the most efficient method is e.g. exponential, the problem is NP complete. There are many problems for which no algorithm is known that will solve the problem in polynomial time. E.g. some combinatorial problems seem to be NP complete.

LU decomposition

A disadvantage of the simple Gaussian eliminitoin is that the original matrix will disappear. However, below the diagonal there is space that is not used.

The original matrix can be decomposed to a product of upper and lower triangular matrices in several ways. These matrices can be stored in the space of the original matrix.

Here we will discuss the LU decomposition.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & \dots & u_{1n} \\ 0 & 1 & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}.$$

Example. Let's try to find a decomposition:

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

Calculate the product of the triangular matrices:

$$\begin{pmatrix} l_{11} & l_{11}u_{12} & l_{11}u_{13} \\ l_{21} & l_{21}u_{12} + l_{22} & l_{21}u_{13} + l_{22}u_{23} \\ l_{31} & l_{31}u_{12} + l_{32} & l_{31}u_{13} + l_{32}u_{23} + l_{33} \end{pmatrix}.$$

The first column of the matrix **L** is exactly the same as in the original matrix:

$$\begin{aligned} l_{11} &= a_{11} = 1, \\ l_{21} &= a_{21} = 1, \\ l_{31} &= a_{31} = 2. \end{aligned}$$

The first line of the product matrix will give the equations

$$\begin{aligned} l_{11}u_{12} &= a_{12}, \\ l_{11}u_{13} &= a_{13}. \end{aligned}$$

These contain l_{11} that has already been evaluated. Thus we can solve the elements of the first line of **U**:

$$\begin{aligned} u_{12} &= a_{12}/l_{11} = 2/1 = 2, \\ u_{13} &= a_{13}/l_{11} = 2/1 = 2. \end{aligned}$$

In the second column the following equations still remain:

$$\begin{aligned} l_{21}u_{12} + l_{22} &= a_{22}, \\ l_{31}u_{12} + l_{32} &= a_{32}. \end{aligned}$$

Everything else is known except the elements of the second column of **L**.

$$\begin{aligned} l_{22} &= a_{22} - l_{21}u_{12} = 1 - 1 \times 2 = -1, \\ l_{32} &= a_{32} - l_{31}u_{12} = -1 - 2 \times 2 = -5. \end{aligned}$$

Next, we'll evaluate the second line:

$$l_{21}u_{13} + l_{22}u_{23} = a_{23}.$$

This gives

$$u_{23} = (a_{23} - l_{21}u_{13})/l_{22} = (0 - 1 \times 2)/(-1) = 2.$$

Finally, the third column is

$$l_{31}u_{13} + l_{32}u_{23} + l_{33} = a_{33},$$

from which

$$l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} = 1 - 2 \times 2 - (-5) \times 2 = 7.$$

The decomposition is then

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 2 & -5 & 7 \end{pmatrix} \begin{pmatrix} 1 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

The order of evaluation is essential. In finding the decomposition the columns of **L** and lines of **U** alternate.

By inspecting the equations we notice that in order to compute the elements l_{ij} and u_{ij} only the element a_{ij} of the original matrix is needed. Hence this element can be replaced with the new evaluated value of l_{ij} or u_{ij}.

All other values needed to calculate l_{ij} and u_{ij} are previously evaluated elements of the matrices **L** and **U**, stored already in place of the elements of the original matrix.

The decomposition can be stored in the form

$$\begin{pmatrix} 1 & 2 & 2 \\ 1 & -1 & 2 \\ 2 & -5 & 7 \end{pmatrix}.$$

```
!  LU decomposition of a matrix

! first column
do i=1,n
  L(i,1)=A(i,1)
end do
! first line
U(1,1)=1
do j=2,n
  U(1,j)=A(1,j)/L(1,1)
end do

! other columns and lines
do m=2,n
  ! column of L
  do i=m,n
    s=0.0
    do k=1,m-1
      s=s+L(i,k)*U(k,m)
    end do
    L(i,m)=A(i,m)-s
  end do
```

```
    ! line of U
    U(m,m)=1
    do j=m+1,n
      s=0.0
      do k=1,m-1
        s=s+L(m,k)*U(k,j)
      end do
      U(m,j)=(A(m,j)-s)/L(m,m)
    end do
  end do
```

In this we can replace L and U with A, in which case the original matrix will be overwritten by the decomposition.

In terms of the decomposition the equation $\mathbf{Ax} = \mathbf{b}$ becomes

$$\mathbf{LUx} = \mathbf{b}.$$

This is equivalent to the equations

$$\mathbf{Ly} = \mathbf{b},$$
$$\mathbf{Ux} = \mathbf{y}.$$

In both equations the coefficient matris is triangular, and to solve the equations only back substitution is needed.

Begin by solving the vector $\mathbf{y}$ from

$$\mathbf{L}\mathbf{y} = \mathbf{b}.$$

$$\begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix},$$

which is equivalent to the equations

$$\begin{aligned} l_{11}y_1 &= b_1, \\ l_{21}y_1 + l_{22}y_2 &= b_2, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + \cdots + l_{nn}y_n &= b_n. \end{aligned}$$

$$
\begin{aligned}
y_1 &= b_1/l_{11},\\
y_2 &= (b_2 - l_{21}y_1)/l_{22},\\
&\vdots\\
y_n &= (b_n - l_{n1}y_1 - l_{n2}y_2 - \cdots - l_{n,n-1}y_{n-1})/l_{nn}.
\end{aligned}
$$

In computing y_i the previous values b_j, $j < i$, are no more needed, and we can write the solution to the vector $\mathbf{b}$.

Next solve the equation **Ux**=**y**:

$$\begin{pmatrix} 1 & u_{12} & \dots & u_{1n} \\ 0 & 1 & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

or

$$\begin{aligned} x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &= y_1, \\ &\vdots \\ x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, \\ x_n &= y_n. \end{aligned}$$

$$\begin{aligned} x_n &= y_n, \\ x_{n-1} &= y_{n-1} - u_{n-1,n} x_n, \\ &\vdots \\ x_1 &= y_1 - u_{12} x_2 - \cdots - u_{1n} x_n. \end{aligned}$$

Again the values y_i can be replaced by the solution.

Example

$$\begin{pmatrix} 1 & 2 & 2 \\ 1 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}.$$

Using the previously found LU decomposition this is equivalent to the set of equations

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 2 & -5 & 7 \end{pmatrix} \begin{pmatrix} 1 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}.$$

$$\begin{aligned} y_1 &= 1/1 = 1, \\ y_2 &= (2-1)/(-1) = -1, \\ y_3 &= (0-2-5)/7 = -1. \end{aligned}$$

$$\begin{aligned} x_3 &= -1, \\ x_2 &= -1+2 = 1, \\ x_1 &= 1-2+2 = 1. \end{aligned}$$

The decomposition method is handy if one has to solve several sets of equations with the same coefficient matrix. The LU decomposition has to be evaluated only once.

One application is the calculation of the inverse matrix.

Determinant

$$\det \mathbf{A} = \det \mathbf{LU} = \det \mathbf{L} \det \mathbf{U}.$$

L and **U** are triangular matrices. The determinant of a triangular matrix is simply the product of the diagonal elements. The diagonal of **U** contains only ones; thus its determinant is 1. Therefore, the determinant is simply

$$\det \mathbf{A} = l_{11} l_{22} \cdots l_{nn} = \prod_{i=1}^{n} l_{ii}.$$

Inverse matrix

The inverse matrix $\mathbf{A}^{-1}$ of $\mathbf{A}$ is a matrix satisfying the equation

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}.$$

There is no point in evaluating the inverse matrix unless it is really needed for something.

$\mathbf{X}$ is the required inverse matrix if

$$\mathbf{A}\mathbf{X} = \mathbf{I}.$$

This is equivalent to the equations

$$\mathbf{A}\mathbf{x}_i = \mathbf{e}_i,\ i = 1, \ldots, n,$$

where $\mathbf{x}_i$ is the column i of matrix $\mathbf{X}$ and $\mathbf{e}_i$ is the column i of a unit matrix.

Thus the columns of the inverse matrix are found by solving n sets on linear equations with the same coefficient matrix.

Iterative methods

Also sets of linear equations can be solved using iterative methods. The iteration can be performed in many different ways.

1 How to find the initial value of the solution vector?

2 How to evaluate the next iterate?

3 How to update the solution vector (and possibly some other quantities)?

The equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ can always be written in an equivalent form

$$\mathbf{M}\mathbf{x} = (\mathbf{M} - \mathbf{A})\mathbf{x} + \mathbf{b},$$

where $\mathbf{M}$ is an arbitrary matrix. If $\mathbf{x}_i$ is the approximation obtained in iteration i, the next iterate will be

$$\mathbf{M}\mathbf{x}_{i+1} = (\mathbf{M} - \mathbf{A})\mathbf{x}_i + \mathbf{b}.$$

This is a set of linear equations, from which $\mathbf{x}_{i+1}$ has to be solved. Naturally the matrix $\mathbf{M}$ must be such that it is essentially easier to solve this set than the original one.

The simplest case is to take **M** to be a unit matrix. The new iterate of the solution is then just the vector on the right hand side.

If **M** is the diagonal of the original matrix **A** we get another very simple method, known as Jacobi's method.

The method will converge for all initial values if the diagonal of the coefficient matrix is dominant:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \;\; i = 1, \ldots, n.$$

```
program jacobi
! solving set of linear equations
! using Jacobi's iteration
integer, parameter :: n=3
real, parameter :: epsilon = 0.00001
integer i, j, iter
real d, s, x(n), y(n), A(n,n), M(n), b(n)

A(1,:) = (/ 3, 1, 1 /)
A(2,:) = (/ 1, -3, -1 /)
A(3,:) = (/ 2, 1, -4 /)
b = (/1, 2, 2/)

! matrix M; only diagonal elements needed
do i=1,n
  M(i)=A(i,i)
end do

! replace A by A-M
do i=1,n
  A(i,i)=0
end do
```

```
! initial solution
x=0.0

! iterate until successive approximations
! x and y do not differ too much
d=1.0
iter=0
 do while (d > epsilon)

 ! right hand side matrix of the iteration formula
 do i=1,n
   s=0.0
   do j=1,n
     s=s-A(i,j)*x(j)
   end do
  y(i) = (s+b(i))/M(i)
 end do
 ! difference of successive iterates
 d=0.0
 do i=1,n
   d=d+(x(i)-y(i))**2
 end do
 ! update the solution vector
 x=y

 write (6,*) x, d

 iter = iter+1
 if (iter > 10) exit

end do
end

0.3333333   -0.6666667   -0.5000000    0.8055556
0.7222223   -0.3888889   -0.5000000    0.2283951
0.6296296   -0.2592592   -0.2361111    9.5014609E-02
0.4984568   -0.3780864   -0.2500000    3.1519126E-02
0.5426955   -0.4171811   -0.3452932    1.2566250E-02
0.5874915   -0.3706704   -0.3329476    4.3223356E-03
0.5678726   -0.3598537   -0.2989219    1.6596466E-03
0.5529252   -0.3777352   -0.3060271    5.9365941E-04
0.5612541   -0.3803492   -0.3179712    2.1886613E-04
0.5661068   -0.3735916   -0.3144603    8.1541584E-05
0.5626839   -0.3731443   -0.3103445    2.8855728E-05
```

Gauss–Seidel method

In Jacobi's method the whole solution vector is computed using only the values found in the previous iteration cycle. Often the convergence is faster if the components of the solution vector are updated as soon as they have been calculated. This is known as the Gauss–Seidel method.

Possible problems

Sensitivity to errors. The values in the set of equations may not be absolutely precise. If the coefficient matrix is almost singular, even small errors of the coefficients can change the solution cosiderably.

Scaling. Very small or very big numbers may appear as divisors changing the magnitude of the coefficients considerably.

Convergence. In iterative methods one has to keep track of the convergence of the approximations. It may be necessary to have some upper limit to the number of iteration steps.

Over- and underflows. Floating point operations may produce a number that is too big to be presented. Usually the problem can be avoided by proper scaling.

Example: Kahan's equation:

$$\begin{pmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \begin{pmatrix} 0.8642 \\ 0.1440 \end{pmatrix}.$$

If this is solved using 8 decimals, we find

$$x = 1.33317912, \ y = -1.0$$

The components of the residual

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

are of the order of 10^{-8}, comparable to the procision of the calculations.

However, the correct solution is $x = 2, y = -2$!

Condition number

Sensitivity to errors is given by the condition number

$$\mathrm{cond}(\mathbf{A}) = \parallel \mathbf{A} \parallel \parallel \mathbf{A}^{-\mathbf{1}} \parallel .$$

When the equations are solved, the errors of the initial values will increase roughly by a factor of cond($\mathbf{A}$).

An easily evaluated matrix norm is

$$\parallel \mathbf{A} \parallel_\infty = \max_i \sum_j |a_{ij}|.$$

In Kahan's equation we have

$$\mathbf{A} = \begin{pmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{pmatrix}$$

$$\mathbf{A}^{-1} = 10^8 \begin{pmatrix} 0.1441 & -0.8648 \\ -0.2161 & 1.2969 \end{pmatrix}$$

$\| \mathbf{A} \| \approx 2$ and $\| \mathbf{A}^{-1} \| = 2 \times 10^8$, thus $\text{cond}(\mathbf{A}) \approx 4 \times 10^8$.

This is a sign that the solution is very sensitive to errors.