Chapter 6

Systems of Linear Equations

6.1 Introduction

Systems of linear equations hold a special place in computational physics, chemistry, and engineering. In fact, multiple computational problems in science and technology can be mathematically expressed as a linear system. Most methods in computational mathematics for solving systems of partial differential equations (PDE), integral equations, and boundary value problems in ordinary differential equations (ODE) utilize the Finite Difference Approximation, effectively replacing the original differential and integral equations on systems of linear equation. Additionally, other applications of systems of linear equations in numerical analysis include the linearization of systems of simultaneous nonlinear equations, and the fitting of polynomials to a set of data points.

A system of linear equations has the following form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{6.1}$$

where $x_j (j = 1, 2, \ldots, n)$ are unknown variables, $a_{ij} (i, j = 1, 2, \ldots, n)$ are the coefficients, and $b_i (i = 1, 2, \ldots, n)$ are the nonhomogeneous terms. The first subscript i identifies the row of the equation and the second subscript j identifies the column of the system of equations. The system (6.1) can also be written as a matrix equation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}, \tag{6.2}$$

or in a compact form as $Ax = b$

Methods for solving linear systems are normally taught in mathematical classes and courses of linear algebra. The standard syllabus includes the substitution method, Cramer's rule, and

the inverse matrix. Unfortunately, Cramer's rule is a highly inefficient method for solving real systems of linear equations: the number of equations in a system may run into the hundreds, thousands, or even millions (e.g. structure and spectra calculations for quantum systems). Since Cramer's rule is based on evaluations of multiple determinants, it needs about $n!$ multiplications and divisions for solving a system of n equations. Thus, solving a system of only 30 equations ($30! \sim 2 \cdot 10^{32}$) would take as much time as the age of the universe on a teraflop computer. Another common idea in standard linear algebra courses is that the solution to $Ax = b$ can be written as $x = A^{-1}b$, where A^{-1} is the inverse matrix of A. However, in most practical computational problems, it is not recommended to compute the inverse matrix to solve a system of linear equations. In fact, it normally takes more operations to compute the actual inverse matrix instead of simply finding the solution by one of the direct elimination methods. Finally, the method of substitution, well known for high school students, is the foundation for multiple methods in numerical analysis for solving real problems.

There are two classes of methods for solving systems of linear equations. In *direct methods*, a finite number of arithmetic operations leads to an "exact" (within round-off errors) solution. Examples of such direct methods include Gauss elimination, Gauss-Jordan elimination, the matrix inverse method, and LU factorization. The average number of operations to solve a system of linear equations for these methods is $\sim n^3$. *Iterative methods* achieve the solution asymptotically by an iterative procedure, starting from the trial solution. The calculations continue until the accepted tolerance ε is achieved. Jacobi, Gauss-Seidel, and successive over-relaxation, are all examples of iterative methods. Direct elimination methods are normally used when the number of equations is less than a few hundred, or if the system of equations is ill-conditioned. Iterative methods are more common for large and diagonally dominant systems of equations, especially when many non-diagonal coefficients equal zero or very small numbers.

At present, there are multiple algorithms and programs developed for solving systems of linear equations based on direct and iterative methods. Using a method that utilizes the most from the matrix shape (symmetric, sparse, tridiagonal, banded) results in higher efficiency and accuracy. The most common problems in matrix calculations are the results of round-off errors or the running out of memory and computational time for large systems of equations. It is also important to remember that various computer languages may handle the same data very differently. For example, in C/C++, the first element of an array starts from index 0, in Fortran (by default), from index 1. It is also useful to note that Fortran 90/95 has numerous intrinsic functions to do matrix calculations.

In this chapter, we will consider linear systems (6.1) to have real coefficients a_{ij}. We will also assume an existence of a unique solution (e.g. $\det A \neq 0$ if the right-hand coefficients $b_i \neq 0$, or $\det A = 0$ if $b_i = 0$).

Comments: 1) possible examples from physics: electric circuits, equilibrium problems

6.2 Direct elimination methods

Elimination methods use a simple idea that is well known from courses of algebra: a system of two equations worked out formally by solving one of the equations. Let's say we solve the first equation for the unknown x_1 in terms of the other unknown x_2. Substituting the solution for x_1 into the second equation gives us a single equation for one unknown x_2, thus x_1 is eliminated from the second equation. After x_2 is found, the other x_1 unknown can be found by back substitution.

In the general case of n linear equations, the elimination process employs operations on rows of linear equations that do not change the solution, namely, "scaling" - any equation may be multiplied by a constant, "pivoting" - the order of equations can be interchanged, "elimination" - any equation can be replaced by a linear combination of that equation with any other equation.

6.2.1 Basic elimination

For a good understanding of basic techniques of direct elimination, it is incredibly helpful to apply the elimination method to find solutions of a system of three linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{6.3}$$

Step 1a: Subtracting the first equation multiplied by a_{21}/a_{11} from the second equation, and multiplied by a_{31}/a_{11} from the third equation gives

$$\begin{array}{rcrcrcr} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & = & b_1 \\ (a_{21}-a_{21})x_1 & + & (a_{22}-\frac{a_{21}}{a_{11}}a_{12})x_2 & + & (a_{23}-\frac{a_{21}}{a_{11}}a_{13})x_3 & = & b_2-\frac{a_{21}}{a_{11}}b_1 \\ (a_{31}-a_{31})x_1 & + & (a_{32}-\frac{a_{31}}{a_{11}}a_{12})x_2 & + & (a_{33}-\frac{a_{31}}{a_{11}}a_{13})x_3 & = & b_3-\frac{a_{31}}{a_{11}}b_1 \end{array} \tag{6.4}$$

One can see that the coefficients by the unknown x_1 in the second and the third rows of the new system are zero

$$\begin{array}{rcrcrcr} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & = & b_1 \\ 0 & + & a'_{22}x_2 & + & a'_{23}x_3 & = & b'_2 \\ 0 & + & a'_{32}x_2 & + & a'_{33}x_3 & = & b'_3 \end{array} \tag{6.5}$$

where $a'_{ij} = a_{ij} - \frac{a_{i1}a_{1j}}{a_{11}}$ and $b'_i = b_i - \frac{a_{i1}}{a_{11}}b_1$. Thus, we eliminated the first unknown x_1 from the second and third equations.

Step 1b: Now, let's subtract the modified second equation multiplied by a'_{32}/a'_{22} from the third equation in (6.5)

$$0 + (a'_{32} - a'_{32})x_2 + (a'_{33} - a'_{23}\frac{a'_{32}}{a'_{22}})x_3 = b'_3 - b'_2\frac{a'_{32}}{a'_{22}} \tag{6.6}$$

After the two eliminations we have a new form for the system (6.3)

$$\begin{array}{rcrcrcl} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & = & b_1 \\ 0 & + & a'_{22}x_2 & + & a'_{23}x_3 & = & b'_2 \\ 0 & + & 0 & + & a''_{33}x_3 & = & b''_3 \end{array} \tag{6.7}$$

with $a''_{ij} = a'_{ij} - \frac{a'_{i2}a'_{2j}}{a'_{22}}$ and $b''_i = b'_i - \frac{a'_{i2}}{a'_{22}}b_2$. Thus, the original system $Ax = b$ is reduced to triangular form.

Step 2: The last equation in (6.7) can be solved for x_3, and the second for x_2, and finally the first for x_1

$$\begin{aligned} x_3 &= b''_3/a''_{33} \\ x_2 &= (b'_2 - a'_{23}x_3)/a'_{22} \\ x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \end{aligned} \tag{6.8}$$

The basic elimination method can be easily generalized for a general n by n system, $Ax = b$

Algorithm 6.1 *The basic elimination algorithm for solving a system of n linear equations.*
Step 1: Apply the elimination procedure to every column k ($k = 1, 2, \ldots, n-1$) for rows i ($i = k+1, k+2, \ldots, n$) to create zeros in column k below the pivot element $a_{k,k}$

$$a_{i,j} = a_{i,j} - (a_{i,k}/a_{k,k})\, a_{k,j} \qquad (i, j = k+1, k+2, \ldots, n) \tag{6.9}$$

$$b_i = b_i - (a_{i,k}/a_{k,k})\, b_k \qquad (i, j = k+1, k+2, \ldots, n) \tag{6.10}$$

Step 2: The solutions of the reduced triangular system can then be found using the backward substitution

$$x_n = (b_n/a_{n,n}) \tag{6.11}$$

$$x_j = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i.j}x_j \right) \qquad (i = n-1, n-2, \ldots, 1) \tag{6.12}$$

The total number of multiplications and divisions done by the basic elimination algorithm for a system of n equations is about $O(n^3)$. The back substitution takes approximately $O(n^2)$ multiplication and divisions.

Comments: Every next elimination uses results from the elimination before. For large systems of equations the round-off errors may quickly accumulate. Say again that it takes finite number of steps to get a true (within the round-off error solution)

The program below implements the basic elimination for a general n by n matrix A

Program 6.1 ***The basic elimination.***

```
  subroutine gauss_1(a,b,x,n)
!============================================================
! Solutions to a system of linear equations A*x=b
! Method: the basic elimination (simple Gauss elimination)
! Alex G. November 2009
!------------------------------------------------------------
! input ...
! a(n,n) - array of coefficients for matrix A
! b(n)   - vector of the right hand coefficients b
! n      - number of equations
! output ...
! x(n)   - solutions
! comments ...
! the original arrays a(n,n) and b(n) will be destroyed
! during the calculation
!============================================================
implicit none
integer n
double precision a(n,n), b(n), x(n)
double precision c
integer i, j, k

!step 1: forward elimination
do k=1, n-1
   do i=k+1,n
      c=a(i,k)/a(k,k)
      a(i,k) = 0.0
      b(i)=b(i)- c*b(k)
      do j=k+1,n
         a(i,j) = a(i,j)-c*a(k,j)
      end do
   end do
end do

!step 2: back substitution
x(n) = b(n)/a(n,n)
do i=n-1,1,-1
   c=0.0
   do j=i+1,n
     c= c + a(i,j)*x(j)
   end do
   x(i) = (b(i)- c)/a(i,i)
end do
end subroutine gauss_1
```

Example 6.1 ***Solution by the basic elimination.***

```
Basic elimination (Simple Gauss)

Matrix A and vector b
   3.000000    2.000000    4.000000    4.000000
   2.000000   -3.000000    1.000000    2.000000
   1.000000    1.000000    2.000000    3.000000

Matrix A and vector b after elimination
   3.000000    2.000000    4.000000    4.000000
   0.000000   -4.333333   -1.666667   -0.666667
   0.000000    0.000000    0.538462    1.615385

Solutions x(n)
  -2.000000   -1.000000    3.000000
```

6.2.2 Gauss elimination

The first immediate problem with the basic elimination method comes when one of diagonal elements is zero. For example, the following system

$$\begin{aligned} 0x_1 + 1x_1 + 2x_1 &= 4 \\ 2x_1 + 1x_2 + 4x_3 &= 3 \\ 2x_1 + 4x_2 + 6x_3 &= 7 \end{aligned} \tag{6.13}$$

has a unique solution of $x = \{-2.5, 0.0, 2.0\}$. However, basic elimination would fail on the first step since the a_{11} pivot element is zero. The procedure also fails when any of subsequent $a_{k,k}$ pivot elements during the elimination procedure are zero. However, the basic elimination procedure can be modified to push zero $a_{k,k}$ elements off the major diagonal. The order of equations in a linear system can be interchanged without changing the solution. This procedure is called "partial pivoting". "Full pivoting" includes interchanging both equations and variables, and it is rarely applied in practical calculations because of its complexity. Nevertheless, pivoting can remove divisions by zero during the elimination process.

The effect of round-off errors can be reduced by scaling before pivoting. Scaling selects an equation with the relatively largest pivot element a_{kk}. On every step k of the elimination procedure we a) look first for a largest element $a_{i,j}$ in every row $i = k, k+1, \ldots, n$ and scale (normalize) every element in that row on the largest element, b) look for the largest element $a_{i,k}$ in the column k to have it as a pivot element for the next elimination, c) interchange the current equation k with the equation with the largest pivot element.

Let's apply scaled pivoting to the system (6.13). The first scaling gives the following $a_{i,1}$ elements $\{0.00, 0.50, 0.33\}$. Therefore, we rearrange the system placing the second equation as

the first one, and the third equation into second place.

$$\begin{aligned} 2x_1 + 1x_2 + 4x_3 &= 3 \\ 2x_1 + 4x_2 + 6x_3 &= 7 \\ 0x_1 + 1x_1 + 2x_1 &= 4 \end{aligned} \tag{6.14}$$

After the first elimination, we have

$$\begin{aligned} 2x_1 + 1x_2 + 4x_3 &= 3 \\ 0x_1 + 3x_2 + 2x_3 &= 4 \\ 0x_1 + 1x_2 + 2x_3 &= 4 \end{aligned} \tag{6.15}$$

The second scaling gives $\{1.00, 0.50\}$ for $a_{i,2}$ elements where $i \geq 2$. Therefore, we keep the same order of equations. After the second elimination, the original matrix is transformed to the upper triangular form

$$\begin{aligned} 2.00x_1 + 1.00x_2 + 4.00x_3 &= 3.00 \\ 0.00x_1 + 3.00x_2 + 2.00x_3 &= 4.00 \\ 0.00x_1 + 0.00x_2 + 1.33x_3 &= 2.66 \end{aligned} \tag{6.16}$$

The backward substitution returns the solutions $\{2.0, 0.0, -2.5\}$

The Gauss elimination includes all three basic operations on rows of linear equations: scaling, pivoting and elimination.

Algorithm 6.2 *Gauss elimination for solving a system of n linear equations.*
Step 1: Apply the scaling, pivoting and elimination to every column k ($k = 1, 2, \ldots, n-1$) starting from $k = 1$
a). Find the largest element in every row $i = k, k+1, \ldots, n$ and divide other elements of those rows by the corresponding largest element.
b). Find the largest pivoting element $a_{i,k}$ in a given column k for $i = k, k+1, \ldots, n$. Let's say it was $a_{l,k}$
c). Interchange rows k and l to have the relatively largest a_{kk} into the pivot position.
d). Apply the elimination procedure to the column k for rows i ($i = k+1, k+2, \ldots, n$)

$$a_{i,j} = a_{i,j} - (a_{i,k}/a_{k,k})\, a_{k,j} \qquad (i, j = k+1, k+2, \ldots, n) \tag{6.17}$$

$$b_i = b_i - (a_{i,k}/a_{k,k})\, b_k \qquad (i, j = k+1, k+2, \ldots, n) \tag{6.18}$$

Step 2. Now it is time for backward substitution. At this point all the diagonal elements are non zero, if the matrix is not singular. From the last equation we have

$$x_n = (b_n/a_{n,n}) \tag{6.19}$$

Solving the other unknowns in the reverse order

$$x_j = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i.j} x_j \right) \qquad (i = n-1, n-2, \ldots, 1) \tag{6.20}$$

The solution is achieved in a finite number of steps determined by the size of the system. The partial pivoting takes a very small fraction of computational efforts comparing to the elimination calculations. The total number of operations is about $O(n^3)$. If all the potential pivots elements are zero, then the matrix A is singular. Linear systems with singular matrices either have no solutions, or do not have a unique solution.

Program 6.2 ***Gauss elimination with scaling and pivoting.***

```
  subroutine gauss_2(a,b,x,n)
!============================================================
! Solutions to a system of linear equations A*x=b
! Method: Gauss elimination (with scaling and pivoting)
! Alex G. (November 2009)
!------------------------------------------------------------
! input ...
! a(n,n) - array of coefficients for matrix A
! b(n)   - array of the right hand coefficients b
! n      - number of equations (size of matrix A)
! output ...
! x(n)   - solutions
! coments ...
! the original arrays a(n,n) and b(n) will be destroyed
! during the calculation
!============================================================
implicit none
integer n
double precision a(n,n), b(n), x(n)
double precision s(n)
double precision c, pivot, store
integer i, j, k, l

! step 1: begin forward elimination
do k=1, n-1

! step 2: "scaling"
! s(i) will have the largest element from row i
  do i=k,n                       ! loop over rows
    s(i) = 0.0
```

```
    do j=k,n                    ! loop over elements of row i
      s(i) = max(s(i),abs(a(i,j)))
    end do
  end do

! step 3: "pivoting 1"
! find a row with the largest pivoting element
  pivot = abs(a(k,k)/s(k))
  l = k
  do j=k+1,n
    if(abs(a(j,k)/s(j)) > pivot) then
      pivot = abs(a(j,k)/s(j))
      l = j
    end if
  end do

! Check if the system has a sigular matrix
  if(pivot == 0.0) then
    write(*,*) ' The matrix is sigular '
    return
  end if

! step 4: "pivoting 2" interchange rows k and l (if needed)
if (l /= k) then
  do j=k,n
     store = a(k,j)
     a(k,j) = a(l,j)
     a(l,j) = store
  end do
  store = b(k)
  b(k) = b(l)
  b(l) = store
end if

! step 5: the elimination (after scaling and pivoting)
   do i=k+1,n
      c=a(i,k)/a(k,k)
      a(i,k) = 0.0
      b(i)=b(i)- c*b(k)
      do j=k+1,n
         a(i,j) = a(i,j)-c*a(k,j)
      end do
   end do
end do
```

```
! step 6: back substiturion
x(n) = b(n)/a(n,n)
do i=n-1,1,-1
   c=0.0
   do j=i+1,n
     c= c + a(i,j)*x(j)
   end do
   x(i) = (b(i)- c)/a(i,i)
end do

end subroutine gauss_2
```

Example 6.2 ***Solution by Gauss elimination.***

```
 Gauss elimination with scaling and pivoting

 Matrix A and vector b
    0.000000    1.000000    2.000000    4.000000
    2.000000    1.000000    4.000000    3.000000
    2.000000    4.000000    6.000000    7.000000

 Matrix A and vector b after elimination
    2.000000    1.000000    4.000000    3.000000
    0.000000    3.000000    2.000000    4.000000
    0.000000    0.000000    1.333333    2.666667

 Solutions x(n)
   -2.500000    0.000000    2.000000
```

It is useful to remember that there are variations of the Gauss elimination. For example, the Gauss-Jordan elimination transforms the matrix A to a diagonal form, with a subsequent reduction to the identity matrix I. As a result, the transformed vector b is a solution vector. Despite the fact that this method needs more computational time, it can be used to evaluate the inverse of matrix A^{-1}, so that $AA^{-1} = I$. On the other hand, LU factorization is very efficient for solving multiple systems with the same matrix A but with different vectors b. The Thomas algorithm treats tridiagonal systems of equations.

6.2.3 Computing inverse matrices and determinants

The inverse matrix A^{-1} can be computed using the same Gauss elimination procedure. Finding an inverse matrix is equivalent to finding matrix X such as

$$AX = I \tag{6.21}$$

This equation can be rewritten as

$$\sum_{k=1}^{n} a_{i,k}x_{k,j} = \delta_{i,j} \qquad (i,j = 1,2,\ldots,n), \tag{6.22}$$

where $\delta_{i,j}$ is the Kronecker delta. Then the system (6.22) is actually a set of n independent systems of equations with the same matrix A but different vectors b. Let's define the two following vectors

$$x^{(j)} = \{x_{i,j}\}, \qquad e^{(j)} = \{\delta_{i,j}\}, \qquad (i = 1,2,\ldots,n) \tag{6.23}$$

Now the the j-th column of the inverse matrix A^{-1} is the solution of the linear system

$$Ax^{(j)} = e^{(j)} \qquad (j = 1,2,\ldots,n) \tag{6.24}$$

The set of systems (6.24) can be solved with Gauss elimination. It is clear that finding the inverse matrix requires n-times more computational time than the elimination procedure.

For the illustration of this method, we consider a system of three equations. The first column of the inverse matrix X can be found from the following systems

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{6.25}$$

The next two columns of the inverse matrix X correspond to solutions of the linear equations with the same matrix A and the right sides as

$$b = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \text{ for the second column, and } \quad b = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ for the third column.} \tag{6.26}$$

Technically, we may use Gauss elimination algorithm for solving n systems of n linear equations to find the inverse matrix. However, computationally, it is time consuming since we have to do the elimination for the same matrix A over and over again.

On the other hand, the LU factorization algorithm is incredibly efficient for solving multiple linear equations with the same matrix but different right-hand vectors b. Any matrix can be written as a product of two other matrices, in particular as $A = LU$, where L and U are the lower triangular and upper triangular matrices. If the elements on the major diagonal of L are equal to one, the method is called the Doolittle method. For unity elements on the major diagonal of U, the method is called the Crout method. For $A = LU$, the linear system of equations $Ax = b$ becomes $LUx = b$. Multiplying both sides of the system by L^{-1} gives $L^{-1}LUx = L^{-1}b$, and then $Ux = L^{-1}b = d$, where d is a solution of $Ld = b$. Now it should be easy to see that the following algorithm would lead to a solution of the linear system. First, we calculate U and L

matrices using the Gaussian elimination. While getting U is the goal of the elimination, the L matrix consists of the elimination multipliers with unity elements of the main diagonal (it would correspond to the Dolittle method). For every vector b we solve $Ld = b$ to find d, namely

$$d_i = b_i - \sum_{k=1}^{i-1} l_{i,k} d_k \quad (i = 2, 3, \ldots, n), \text{ note that } d_1 = b_1 \tag{6.27}$$

Then, $x_n = d_n / U(n, n)$, and other solutions for the linear system $Ux = d$ are

$$x_i = d_i - \sum_{k=i+1}^{n} u_{i,k} x_k / u_{i,i} \quad (i = n-1, n-2, \ldots, 1) \tag{6.28}$$

Since the number of multiplications to find solutions from the last two equations are of the order $O(n^2)$, we can see that the LU decomposition method is exceptionally helpful for computing inverse matrices.

Program 6.3 ***Compute Inverse matrix using LU Doolittle factorization***

```
  subroutine inverse(a,c,n)
!============================================================
! Inverse matrix
! Method: Based on Doolittle LU factorization for Ax=b
! Alex G. December 2009
!------------------------------------------------------------
! input ...
! a(n,n) - array of coefficients for matrix A
! n      - dimension
! output ...
! c(n,n) - inverse matrix of A
! comments ...
! the original matrix a(n,n) will be destroyed
! during the calculation
!============================================================
implicit none
integer n
double precision a(n,n), c(n,n)
double precision L(n,n), U(n,n), b(n), d(n), x(n)
double precision coeff
integer i, j, k

! step 0: initialization for matrices L and U and b
! Fortran 90/95 aloows such operations on matrices
L=0.0
U=0.0
```

```
b=0.0

! step 1: forward elimination
do k=1, n-1
   do i=k+1,n
      coeff=a(i,k)/a(k,k)
      L(i,k) = coeff
      do j=k+1,n
         a(i,j) = a(i,j)-coeff*a(k,j)
      end do
   end do
end do

! Step 2: prepare L and U matrices
! L matrix is a matrix of the elimination coefficient
! + the diagonal elements are 1.0
do i=1,n
  L(i,i) = 1.0
end do
! U matrix is the upper triangular part of A
do j=1,n
  do i=1,j
    U(i,j) = a(i,j)
  end do
end do

! Step 3: compute columns of the inverse matrix C
do k=1,n
  b(k)=1.0
  d(1) = b(1)
! Step 3a: Solve Ld=b using the forward substitution
  do i=2,n
    d(i)=b(i)
    do j=1,i-1
      d(i) = d(i) - L(i,j)*d(j)
    end do
  end do
! Step 3b: Solve Ux=d using the back substitution
  x(n)=d(n)/U(n,n)
  do i = n-1,1,-1
    x(i) = d(i)
    do j=n,i+1,-1
      x(i)=x(i)-U(i,j)*x(j)
    end do
```

```
    x(i) = x(i)/u(i,i)
  end do
! Step 3c: fill the solutions x(n) into column k of C
  do i=1,n
    c(i,k) = x(i)
  end do
  b(k)=0.0
end do
end subroutine inverse
```

Example 6.3 ***Inverse matrix***

```
 Computing Inverse matrix

 Matrix A
    3.000000    2.000000    4.000000
    2.000000   -3.000000    1.000000
    1.000000    1.000000    2.000000

 Inverse matrix A^{-1}
    1.000000    0.000000   -2.000000
    0.428571   -0.285714   -0.714286
   -0.714286    0.142857    1.857143
```

The elimination method can be easily applied to compute matrix determinants. At the end of the elimination procedure, the original matrix A is transformed to the upper triangular form. For such matrices, the determinant is a product of diagonal elements.

$$\det(A) = \pm \prod_{i=1}^{n} a_{ii} = a_{11} a_{22} a_{33} \dots a_{nn}, \tag{6.29}$$

where the sign depends on the number of interchanges. Let's remember that pivoting changes the value of the determinant (interchanging any two equations changes the sign of the determinant). However, counting the number of equation interchanges would give us the proper sign for the determinant.

6.2.4 Tridiagonal systems

When a system of linear equations has a special shape (symmetric, or tridiagonal), then it is recommended to use a method specifically developed for this kind of equation. Such methods are not only more efficient in term of computational time and computer memory, but also accumulate smaller round-off errors.

Here is an example of a tridiagonal system of five equations

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}, \tag{6.30}$$

It is clear to see that one only element is to be eliminated in every row, namely $a_{i-1,i}$, affecting only the diagonal elements and the right hand vector. Subsequently, the elimination procedure for a tridiagonal matrix

$$a_{i,i} = a_{i,i} - (a_{i,i-1}/a_{i-1,i-1})\, a_{i-1,i} \qquad (i = 2, \ldots, n) \tag{6.31}$$

and

$$b_i = b_i - (a_{i,i-1}/a_{i-1,i-1})\, b_{i-1} \qquad (i = 2, \ldots, n) \tag{6.32}$$

However, it is possible to improve the efficiency of this method even further. Instead of storing all $n \times n$ elements of the matrix A, since there is no need to keep the zero elements, we may use a smaller matrix such $n \times 3$:

$$\begin{pmatrix} - & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & a_{32} & c_{33} \\ \cdots & \cdots & \cdots \\ c_{n-1,1} & c_{n-1,2} & c_{n-1,3} \\ c_{n,1} & c_{n,2} & - \end{pmatrix} \tag{6.33}$$

where the coefficients c_{ij} are related to the coefficients of the original matrix A as

$$c_{i,1} = a_{i,i-1}, \quad c_{i,2} = a_{i,i}, \text{ and } c_{i,3} = a_{i,i+1}. \tag{6.34}$$

Then the elimination procedure for the new matrix C

$$c_{i,2} = c_{i,2} - (c_{i,1}/c_{i-1,2})\, c_{i-1,3} \qquad (i = 2, 3, \ldots, n) \tag{6.35}$$

and

$$b_i = b_i - (c_{i,1}/c_{i-1,2})\, b_{i,i-1} \qquad (i = 2, 3, \ldots, n) \tag{6.36}$$

After the forward elimination, the back substitution gives the solutions of the tridiagonal system

$$x_n = b_n/c_{n,2} \tag{6.37}$$

$$x_i = (b_i - c_{i,3}x_{i+1})/c_{i,2} \qquad (i = n-1, n-2, \ldots, 1) \tag{6.38}$$

This algorithm for solving tridiagonal systems is called the Thomas algorithm. Thus algorithm is widely used in solving 3-point partial and ordinary differential equations (more details?)

Program 6.4 ***the Thomas method for tridiagonal systems***

```
  subroutine thomas(c,b,x,n)
!============================================================
! Solutions to a system of tridiagonal linear equations C*x=b
! Method: the Thomas method
! Alex G. November 2009
!------------------------------------------------------------
! input ...
! c(n,3) - array of coefficients for matrix C
! b(n)   - vector of the right hand coefficients b
! n      - number of equations
! output ...
! x(n)   - solutions
! comments ...
! the original arrays c(n,3) and b(n) will be destroyed
! during the calculation
!============================================================
implicit none
integer n
double precision c(n,3), b(n), x(n)
double precision coeff
integer i

!step 1: forward elimination
do i=2,n
  coeff=c(i,1)/c(i-1,2)
  c(i,2)=c(i,2)-coeff*c(i-1,3)
  b(i)=b(i)-c(i,1)*b(i-1)
end do

!step 2: back substitution
x(n) = b(n)/c(n,2)
do i=n-1,1,-1
   x(i) = (b(i)- c(i,3)*x(i+1))/c(i,2)
end do
end subroutine thomas
```

Example 6.4 ***Solution by the Thomas method***

```
 The Thomas method for tridiagonal systems

 Matrix A and vector b
    0.000000    4.000000   -1.000000    0.000000
   -1.000000    4.000000   -1.000000    0.000000
```

```
  -1.000000    4.000000   -1.000000    0.000000
  -1.000000    4.000000   -1.000000    0.000000
  -1.000000    4.000000   -1.000000    0.000000
  -1.000000    4.000000   -1.000000    0.000000
  -1.000000    4.000000   -1.000000    0.000000
  -1.000000    4.000000    0.000000   16.000000

Solutions x(n)
   0.000395    0.001578    0.005919    0.022099
   0.082476    0.307806    1.148748    4.287187
```

Pivoting destroys the tridiagonality, and cannot be used ... (*more?*) However, as a rule, tridiagonal systems representing real physical systems are diagonally dominant, and pivoting is unnecessary. more ... the number of multiplicative operations $\sim 5n$, that makes it much more efficient comparing to Gauss elimination by a factor of $\sim n^2$.

6.2.5 Round-off errors and ill-conditioned systems

In the elimination methods, each elimination step uses results from the step before. For linear systems with large numbers of equations, the round-off errors may strongly affect the solution. Round-off errors can be minimized by using double precision calculations and scaled pivoting. Therefore, for matrix calculations, it is vital to use high precision arithmetic. Unfortunately, it takes additional computational resources (memory and time), but it is better then having unreliable solutions.

The effect of round-off errors is especially dangerous for ill-conditioned systems, when doing "everything right", you may in fact get "everything wrong". Ill-conditioned systems are very sensitive to small variations in the equation coefficient. There are no methods for solving this problem other than increasing precision. If we cannot fix the problem, it is at least good to know if we are dealing with an ill-conditioned system. The Ill-conditioned system has a matrix similar to a singular form, and their determinant is close to zero. A commonly used measure of the condition of a matrix is its condition number. In fact, the norm of a matrix can be used to evaluate the condition number: there are several ways to define the norm of a matrix, but the most widely accepted is the Euclidean norm

$$\|A\| = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j}^2 \right)^{1/2} . \tag{6.39}$$

For a matrix equation $Ax = b$ it follows from the norm definition that

$$\|A\|\|x\| \geq \|b\|. \tag{6.40}$$

A small change in the right-hand vector b results in a change in the solution x as

$$A(x + \delta x) = b + \delta b, \tag{6.41}$$

or subtracting the original equation for this one

$$A\delta x = \delta b \text{ or } \delta x = a^{-1}\delta b \tag{6.42}$$

Using norm's properties we may write

$$\|\delta x\| \leq \|A^{-1}\|\|\delta b\| \tag{6.43}$$

Combining together equations (6.40) and (6.43)

$$\|b\|\|\delta x\| \leq \|A\|\|x\|\|A^{-1}\|\|\delta b\| \tag{6.44}$$

or

$$\frac{|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{|\delta b\|}{\|b\|} = C(A)\frac{|\delta b\|}{\|b\|}, \tag{6.45}$$

where the product of two norms

$$C(A) = \|A\|\|A^{-1}\| \tag{6.46}$$

is the condition number of matrix A. The condition number is always ≥ 1. Logically, the condition number is a factor by which a small variation in the coefficients is amplified during the elimination procedure. Since computing the inverse matrix takes more time than solving the system itself, it is common to use estimations for $\|A^{-1}\|$ without actually calculating the inverse matrix. The most sophisticated codes in numerical libraries estimate the condition number along with the solution, giving users an idea about the accuracy of the returned result. For ill-conditioned systems it is advisable to check the final solution by a direct substitution in the original equation.

Here is an example. Consider the equation

$$\begin{aligned} 3.000000x_1 + 2.00x_2 + 4.000000x_3 &= 4.00 \\ 3.000001x_1 + 2.00x_2 + 4.000002x_3 &= 4.00 \\ 1.000000x_1 + 1.00x_2 + 2.000000x_3 &= 3.00 \end{aligned} \tag{6.47}$$

The condition number of the matrix$A$ is $1.3264 \cdot 10^7$. The single precision solution by the basic elimination is $x = \{-2.000, 2.750, 1.125\}$, and the double precision solution is $x = \{-2.0, 3.0, 1.0\}$ (that is the true solution).

6.3 Iterative methods

Iterative methods cannot compete with direct elimination methods for arbitrary matrix A. However, in certain types of problems, systems of linear equations have many $a_{i,j}$ elements as zero, or close to zero (sparse systems). Under those circumstances, iterative methods can be

extremely fast. Iterative methods are also efficient for solving Partial Differential Equations by finite difference or finite element methods.

The idea of the iterative solution of a linear system is based on assuming an initial (trial) solution that can be used to generate an improved solution. The procedure is repeated until convergence with an accepted accuracy solution occurs. However, for an iterative method to succeed/converge, the linear system of equations needs to be diagonally dominant.

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|. \tag{6.48}$$

Iterative methods are less sensitive to round-off errors in comparison to direct elimination methods.

Let's consider a system of linear equations

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \qquad (i = 1, 2, \ldots, n). \tag{6.49}$$

Every equation can be formally solved for a diagonal element

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j - \sum_{j=i+1}^{n} a_{i,j} x_j \right) \qquad (i = 1, 2, \ldots, n). \tag{6.50}$$

Choosing an initial solution we may calculate the next iteration

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n} a_{i,j} x_j^k \right) \qquad (i = 1, 2, \ldots, n). \tag{6.51}$$

Equation (6.51) can be rewritten in the iterative form

$$x_i^{k+1} = x_i^k + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^k \right) \qquad (i = 1, 2, \ldots, n). \tag{6.52}$$

Equation (6.51) defines the Jacobi iterative method, which is also called the method of simultaneous iterations. It is possible to prove that if A is diagonally dominant, then the Jacobi iteration will converge. The number of iterations is either predetermined by a maximum number of allowed iterations, or by one of conditions for absolute errors

$$\max_{1 \leq i \leq n} \left| x_i^{k+1} - x_i^k \right| \leq \varepsilon, \text{ or } \sum_{i=1}^{n} \left| x_i^{k+1} - x_i^k \right| \leq \varepsilon, \text{ or } \left(\sum_{i=1}^{n} \left(x_i^{k+1} - x_i^k \right)^2 \right)^{1/2} \leq \varepsilon, \tag{6.53}$$

where ε is a tolerance. It is also possible to use another condition

$$\frac{\|Ax_k - b\|}{\|b\|} < \varepsilon \tag{6.54}$$

Since efforts to evaluate the norms above are comparable with the iterative calculations, it is recommended to check the convergence based on equation (6.54) after every tenth iteration.

In the Jacobi method, all values of x^{k+1} are calculated using x^k values. In the Gauss-Seidel method, the most recently computed values of x_i are used in calculations for $j > i$ solutions

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i+1}^{n} a_{i,j} x_j^k \right) \qquad (i = 1, 2, \ldots, n), \tag{6.55}$$

or

$$x_i^{k+1} = x_i^k + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i}^{n} a_{i,j} x_j^k \right) \qquad (i = 1, 2, \ldots, n). \tag{6.56}$$

The Gauss-Seidel iterations generally converge faster than Jacobi iterations.

Quite often, the iterative solution to a linear system approaches the true solution in the same direction. Then it is possible to accelerate the iterative process by introducing the over-relaxing factor ω

$$x_i^{k+1} = x_i^k + \omega \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i}^{n} a_{i,j} x_j^k \right) \qquad (i = 1, 2, \ldots, n). \tag{6.57}$$

For $\omega = 1$ the system (6.57) is the Gauss-Seidel method, for $1.0 < \omega < 2.0$ the system is over-relaxed, and for $\omega < 1.0$ the system is under-relaxed. The optimum value of ω depends on the size of the system and the nature of the equations. The iterative process (6.57) is called the successive-over-relaxation (SOR) method.

Program 6.5 ***Gauss-Seidel: The successive-over-relaxation***

```
  subroutine gs_sor(a,b,x,omega,eps,n,iter)
!=========================================================
! Solutions to a system of linear equations A*x=b
! Method: The successive-over-relaxation (SOR)
! Alex G. (November 2009)
!---------------------------------------------------------
! input ...
! a(n,n) - array of coefficients for matrix A
! b(n)   - array of the right hand coefficients b
! x(n)   - solutions (initial guess)
! n      - number of equations (size of matrix A)
! omega  - the over-ralaxation factor
! eps    - convergence tolerance
! output ...
! x(n)   - solutions
```

```
! iter   - number of iterations to achieve the tolerance
! coments ...
! kmax   - max number of allowed iterations
!=========================================================
implicit none
integer, parameter::kmax=100
integer n
double precision a(n,n), b(n), x(n)
double precision c, omega, eps, delta, conv, sum
integer i, j, k, iter, flag

! check if the system is diagonally dominant
flag = 0
do i=1,n
  sum = 0.0
  do j=1,n
    if(i == j) cycle
    sum = sum+abs(a(i,j))
  end do
  if(abs(a(i,i)) < sum) flag = flag+1
end do
if(flag >0) write(*,*) 'The system is NOT diagonally dominant'

do k=1,kmax
  conv = 0.0
  do i=1,n
    delta = b(i)
    do j=1,n
      delta = delta - a(i,j)*x(j)
    end do
    x(i) = x(i)+omega*delta/a(i,i)
    if(abs(delta) > conv) conv=abs(delta)
  end do
  if(conv < eps) exit
end do
iter = k
if(k == kmax) write (*,*)'The system failed to converge'

end subroutine gs_sor
```

Example 6.5 ***Solution by successive-over-relaxation***

```
 The successive-over-relaxation (SOR)

 Matrix A and vector b
```

```
     8.00000     2.00000     4.00000     2.00000
     2.00000     6.00000     1.00000     6.00000
     1.00000     1.00000     8.00000     4.00000

 Trial solutions x(n)
     0.00000     0.00000     0.00000

 Solutions x(n)
    -0.20000     1.00000     0.40000

 iterations =    10
```

Consider here to have an example from PDE

The Jacobi and Gauss-Seidel iterative methods are one step iterative methods since the x_i^{k+1} solution is defined through x_i^k. In multi-step iterative methods, the x_i^{k+1} solution is determined in accordance with past iterations $x_i^{k+1} = f(x_i^k, x_i^{k-1}, \ldots, x_i^{k-m})$.

There are multiple variations or iterative methods, like explicit and implicit iterative methods, the method of upper relaxation, ... more?

6.4 Practical notes

There are multiple methods, and various computer packages available for solving systems of linear equations. A researcher (or a student) faces this question - what would be a good way to solve my problem? Should I invest my time in writing a program, buy software that could do this job for me, learn how to use a sophisticated numerical package, or attempt to find a matrix calculator on the Web?

The answer depends on the following factors: a) the complexity of the system of equations (the size, conditioning, a general or sparse matrix), b) whether the problem is a part of a larger computational problem or a standing alone task, c) whether a one-time solution is needed, or multiple systems are to be solved.

A simple student problem can instantly be solved even with Excel. Excel has a number of functions to work with matrices, in particularly `MINVERSE` to find an inverse matrix, and `MMULT` for matrix multiplication. With Excel a solution can be just a few clicks away using $x = A^{-1}b$. Software packages such as Mathematica, Maple, or MathCad have libraries for solving various systems of equations. If the problem is part of a larger computational project, and a system of equations is not very large (less than a few hundreds of equation), yet well-conditioned, then using the quick and efficient programs of this chapter would be best. However, for serious computational projects, it is advisable to use sophisticated packages developed by experts in numerical analysis. The most well known commercial general libraries are NAG (Numerical Algorithmic Group), and IMSL (International Mathematical and Statistical Library), both available in Fortran 90/5 and C/C++. The NAG package also includes libraries for parallel calculations. The

IMSL library now is a part of compilers such as Intel Fortran, and Intel C++ *(check it!)*.

Additionally, there are also various special packages to solve multiple problems of linear algebra that are absolutely free. LAPACK (Linear Algebra PACKage) is the most advanced linear algebra library. It provides routines for solving systems of linear equations and eigenvalue problems. LAPACK was originally written in Fortran 77, and was the successor of LINPAC (routines for the linear equations) and EISPACK (set of routines for solving the eigenvalue problem). LAPACK has routines to handle both real and complex matrices in single and double precision. The present core version of LAPACK is written in Fortran 90. It has several implementations: LAPACK95 uses features of Fortran 95, CLAPACK in for C, LAPACK++ for C++ (it is being superseded by the Template Numerical Toolkit (TNT), JLAPACK for Java.

There are also two large numerical libraries that have multiple routines for linear algebra problems. SLATEC is a comprehensive library of routines having over 1400 general purpose mathematical and statistical programs. The code was developed by a group from few National Laboratories (US), and is therefore public domain. The library was written in Fortran 77, but some routines are translated to Fortran 90, and there is a possibility to use SLATEC routines from a C++ code. The other large library is the GNU Scientific Library (or GSL). It is written in the C. The GSL is part of the GNU project and is distributed under the GNU General Public License. GAMS - Guide to Available Mathematical Software from the National Institute of Standards and Technology (NIST) is a practical starting point to find the best routine for your problem. GAMS provides an excellent reference place and orientation for available programs.

6.5 Problems

1. Modify the Gauss 2 program above to calculate the determinant of a matrix A.
2. Using the routines from this chapter, write a program that evaluates the conditional number of a linear system (*place eq number*)
3. Modify the GS SOR program above based on Gauss-Seidel successive over-relaxation, to change the convergence condition from (6.53) to (6.54).
4. Study on a diagonally dominant linear system how the choice of the factor ω affects the convergence of the solution.
5. Implement a program from the LAPACK library to solve a system of linear equations(select one or two)
6. Calculations: Compare accuracy of the program implementing the Gauss elimination method with a program from a standard library for solutions of the following system of equations

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

7. *consider some physics problems*