Chapter 10
Solving the System of Algebraic Equations

Abstract The result of the discretization process is a system of linear equations of the form $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$ where the unknowns $\boldsymbol{\phi}$, located at the centroids of the mesh elements, are the sought after values. In this system, the coefficients of the unknown variables constituting matrix $\mathbf{A}$ are the result of the linearization procedure and the mesh geometry, while vector $\mathbf{b}$ contains all sources, constants, boundary conditions, and non-linearizable components. Techniques for solving linear systems of equations are generally grouped into *direct* and *iterative* methods, with many sub-groups in each category. Since flow problems are highly non-linear, the coefficients resulting from their linearization process are generally solution dependent. For this reason and since an accurate solution is not needed at each iteration, direct methods have been rarely used in CFD applications. Iterative methods on the other hand have been more popular because they are more suited for this type of applications requiring lower computational cost per iteration and lower memory. The chapter starts by presenting few direct methods applicable to structured and/or unstructured grids (Gauss elimination, LU factorization, Tridiagonal and Pentadiagonal matrix algorithms) to set the ground for discussing the more widely used iterative methods in CFD applications. Then the performance and limitations of some of the basic iterative methods with and without preconditioning are reviewed. This include the Jacobi, Gauss-Siedel, Incomplete LU factorization, and the conjugate gradient methods. This is followed by an introduction to the multigrid method that is generally used in combination with iterative solvers to help addressing some of their important limitations.

10.1 Introduction

The starting point for any linear solver is the set of equations generated by the discretization process, which are written mathematically as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \tag{10.1}$$

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*, Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_10

where $\mathbf{A}$ is the matrix of coefficients of elements a_{ij}, $\boldsymbol{\phi}$ the vector of unknown variables ϕ_i, and $\mathbf{b}$ the vector of sources b_i. Using matrix numbering, the expanded form of Eq. (10.1) is given by

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N-1} & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN-1} & a_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} \tag{10.2}$$

Generally each row in the above matrix represents an equation defined over one element of the computational domain, and the non-zero coefficients are those related to the neighbors of that element. The coefficient a_{ij} measures the strength of the link between the value of ϕ_i at the centroid of the control volume and its neighbors. As a cell is connected to only few neighbors, with their number depending on the connectivity of the elements in the discretized domain, many of the coefficients are zeros and the resulting $\mathbf{A}$ matrix is always sparse (i.e., the non-zero coefficients are a very small fraction of the matrix). If in addition the method uses a structured grid system, the matrix $\mathbf{A}$ will be banded with all non-zero elements aligned along few diagonals. Therefore methods for efficiently solving such systems should exploit this characteristic.

As mentioned above techniques for solving algebraic systems of equations are broadly divided into two categories denoted by direct and iterative methods, respectively. In a direct method, matrix $\mathbf{A}$ is inverted and the solution $\boldsymbol{\phi}$ is computed in one step as $\boldsymbol{\phi} = \mathbf{A}^{-1}\mathbf{b}$. When the matrix $\mathbf{A}$ is large, computing its inverse is computationally very expensive requiring large memory. It is basically impractical to apply direct linear solvers in CFD applications as they generally involve non-linear systems of equations with their coefficients depending on the solution necessitating the use of an iterative process.

On the other hand, with iterative algebraic solvers the solution algorithm is repeatedly applied as many times as required until a pre-assigned level of convergence is reached without the need for a fully converged solution be attained at every iteration.

The presentation starts with few direct linear solvers applicable to structured and unstructured grid methods. This is followed by a description of solution algorithms that take advantage of the banded structure of the coefficient matrix in structured grid systems. The main focus of the chapter is, however, on a specific class of iterative linear algebraic solvers that has been identified to be generally very efficient and economical with the FVM, and has been exclusively implemented as the linear solver in almost all finite volume-based codes.

10.2 Direct or Gauss Elimination Method

Even though direct methods are not efficient at solving sparse systems of linear algebraic equations due to their high computational cost, their discussion will pave the way for introducing efficient iterative methods in the next section. The simplest direct method for finding solutions to the system of equations described by Eq. (10.1) is the Gauss elimination technique, which will be described first. The transformation of the system into an equivalent upper triangular system, when using the Gauss elimination method, has motivated the development of the Lower-Upper (LU) triangulation method, which will also be presented. In this approach, matrix **A** is decomposed into the product of two matrices **L** and **U** with **L** being a lower triangular matrix and **U** an upper triangular one. This procedure is also known as LU factorization. In addition, direct methods benefiting from the banded structure of **A**, applicable to structured grid methods, will be discussed.

10.2.1 Gauss Elimination

The best way to describe the Gauss elimination technique is to start with a simple example. For that purpose a linear system of equations in the two unknowns ϕ_1 and ϕ_2 is considered. The equations are given by

$$a_{11}\phi_1 + a_{12}\phi_2 = b_1 \tag{10.3}$$

$$a_{21}\phi_1 + a_{22}\phi_2 = b_2 \tag{10.4}$$

The system can be solved by eliminating one of the variables from one of the equations [say ϕ_1 from Eq. (10.4)]. This can be done by multiplying Eq. (10.3) by a_{21}/a_{11} and subtracting the resulting equation from Eq. (10.4). This yields

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)\phi_2 = b_2 - \frac{a_{21}}{a_{11}}b_1 \tag{10.5}$$

The obtained equation involves only one unknown. As such it can be used to solve for ϕ_2, which is obtained as

$$\phi_2 = \frac{b_2 - \dfrac{a_{21}}{a_{11}}b_1}{a_{22} - \dfrac{a_{12}a_{21}}{a_{11}}} \tag{10.6}$$

Knowing ϕ_2, its value can be substituted back into Eq. (10.3) to find ϕ_1. Performing this step, ϕ_1 is found to be

$$\phi_1 = \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} \frac{b_2 - \frac{a_{21}}{a_{11}} b_1}{a_{22} - \frac{a_{12}}{a_{11}} a_{21}} \tag{10.7}$$

The above procedure is composed of two steps. In the first step the equations are manipulated in order to eliminate one of the unknowns. The end result of this step is an equation with one unknown. In the second step, this equation is solved directly and the result back-substituted into one of the equations to solve for the remaining unknown. The same procedure can be generalized to a system of N equations described by Eq. (10.1) or (10.2), as detailed next.

10.2.2 Forward Elimination

In the derivations to follow, the first row of **A** refers to the discretized equation for ϕ_1, the second row represents the equation for ϕ_2, and in general the ith row refers to the equation for ϕ_i. The procedure starts by eliminating ϕ_1 from all equations below row 1 in **A**. To eliminate ϕ_1 from the ith row ($i = 2, 3, \ldots, N$), the coefficients of the first row are multiplied by a_{i1}/a_{11} and the resulting equation is subtracted from the ith row. The system of equations by the end of this step becomes

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N-1} & a_{1N} \\ 0 & a'_{22} & \ldots & a'_{2N-1} & a'_{2N} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & a'_{N2} & \ldots & a'_{NN-1} & a'_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ \vdots \\ \vdots \\ b'_N \end{bmatrix} \tag{10.8}$$

Then ϕ_2 is eliminated from all equations below row 2 in the modified **A**. To eliminate ϕ_2 from the ith row ($i = 3, 4, \ldots, N$), the coefficients of the second row are multiplied by a'_{i2}/a'_{22} and the resulting equation is subtracted from the ith row. Then ϕ_3 is eliminated from all rows below the third row in the modified coefficient matrix and the process is continued until ϕ_{N-1} is eliminated from the Nth row leading to the following equivalent system of equations with its matrix **A** transformed into an upper triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1N-1} & a_{1N} \\ 0 & a'_{22} & a'_{23} & \ldots & a'_{2N-1} & a'_{2N} \\ 0 & 0 & a''_{33} & \cdots & a''_{3N-1} & a''_{3N} \\ \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & a^{N-1}_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b^{N-1}_N \end{bmatrix} \tag{10.9}$$

The resulting algorithm is as described below.

10.2.3 Forward Elimination Algorithm

```
For $k = 1$ to $N-1$
  {
    For $i = k+1$ to $N$
      {
        $Ratio = \frac{a_{ik}}{a_{kk}}$
          {
            For $j = k+1$ to $N$
                $a_{ij} = a_{ij} - Ratio * a_{kj}$
          }
        $b_i = b_i - Ratio * b_k$
      }
  }
```

10.2.4 Backward Substitution

The modified system of equations given by Eq. (10.9) indicates that the only unknown in the Nth equation is ϕ_N. Therefore this equation can be used to obtain ϕ_N as

$$\phi_N = \frac{b_N^{N-1}}{a_{NN}^{N-1}} \tag{10.10}$$

The ($N - 1$)th equation is function of ϕ_{N-1} and ϕ_N. Having found ϕ_N then this equation can be used to find ϕ_{N-1} as

$$\phi_{N-1} = \frac{b_{N-1}^{N-2} - a_{N-1N}^{N-2}\phi_N}{a_{N-1N-1}^{N-2}} \tag{10.11}$$

The process continues moving backward and by the time the ith equation is reached, the values $\phi_{i+1}, \phi_{i+2}, \phi_{i+3}, \ldots, \phi_{N-1}, \phi_N$ would have become available and as such ϕ_i can be computed using

$$\phi_i = \frac{b_i^{i-1} - \sum_{j=i+1}^{N} a_{ij}^{i-1}\phi_j}{a_{ii}^{i-1}} \tag{10.12}$$

the process is continued until ϕ_1 is calculated. Algorithmically, this is represented as shown below.

10.2.5 Back Substitution Algorithm

```
$\phi_N = \frac{b_N}{a_{NN}}$
$For\ i = N-1\ to\ 1$
 {
    $Term = 0$
      {
         $For\ j = i+1\ to\ N$
              $Term = Term + a_{ij} * \phi_j$
      }
    $\phi_i = \frac{\phi_i - Term}{a_{ii}}$
 }
```

Techniques to improve the performance of the method to avoid division by zero through pivoting (interchanging rows in order to select the largest pivoting element) and to reduce roundoff errors in large systems are available but are not discussed here. Interested readers may consult specialized textbooks on the subject [1–4]. The presented algorithm shows that the method is expensive and the number of operations required to solve a linear system of N equations is proportional to $N^3/3$ of which only $N^2/2$ arithmetic operations are required for back substitution. This high computational cost has enticed researchers to look for more efficient specialized solvers for systems with sparse matrices.

10.2.6 LU Decomposition

Another direct method for solving linear algebraic systems of equations is the LU or more generally the PLU (where P refers to the pivoting process mentioned above), in this book reference is made only to LU factorization method, which are variants of the Gauss elimination method. The advantage of these methods over the Gauss elimination method is that once the (P)LU factorization is performed the linear system can be solved as many times as needed for different values of the right hand side vector **b** without performing any additional elimination, which would still be required with the Gauss method.

Based on the elimination performed in the previous section, Eq. (10.1) was transformed into an upper triangular matrix as given by Eq. (10.9), which can be written as

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1N-1} & u_{1N} \\ 0 & u_{22} & u_{23} & \dots & u_{2N-1} & u_{2N} \\ 0 & 0 & u_{33} & \dots & u_{3N-1} & u_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & u_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_N \end{bmatrix} \tag{10.13}$$

Using compact matrix notation Eq. (10.13) can be simplified to

$$\mathbf{U}\boldsymbol{\phi} - \mathbf{c} = 0 \tag{10.14}$$

Let **L** be a unit lower triangular matrix (diagonal elements are set to 1 in order to make the factorization unique) given by

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \ell_{21} & 1 & 0 & \dots & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ \ell_{N1} & \ell_{N2} & \ell_{N3} & \dots & \ell_{NN-1} & 1 \end{bmatrix} \tag{10.15}$$

such that if Eq. (10.14) is multiplied by **L**, Eq. (10.1) is recovered. If this holds, then the following can be written:

$$\mathbf{L}(\mathbf{U}\boldsymbol{\phi} - \mathbf{c}) = \mathbf{L}\mathbf{U}\boldsymbol{\phi} - \mathbf{L}\mathbf{c} = \mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.16}$$

Based on matrix properties it follows that

$$\mathbf{L}\mathbf{U} = \mathbf{A} \tag{10.17}$$

and

$$\mathbf{L}\mathbf{c} = \mathbf{b} \tag{10.18}$$

Equation (10.17) indicates that **A** is written as the product of a proper lower and an upper triangular matrix, known as LU factorization.

10.2.7 The Decomposition Step

The efficient procedure to find the **L** and **U** coefficients described next is denoted by the Crout decomposition [1–4]. In the original Crout algorithm a unit upper triangular matrix is used whereas here a unit lower triangular matrix is assumed. The procedure is based on multiplying **L** and **U** to obtain **A** such that

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \ell_{21} & 1 & 0 & \dots & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ \ell_{N1} & \ell_{N2} & \ell_{N3} & \dots & \ell_{NN-1} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1N-1} & u_{1N} \\ 0 & u_{22} & u_{23} & \dots & u_{2N-1} & u_{2N} \\ 0 & 0 & u_{33} & \dots & u_{3N-1} & u_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & u_{NN} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2N-1} & a_{2N} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N-1} & a_{3N} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN-1} & a_{NN} \end{bmatrix} \tag{10.19}$$

The calculation of the coefficients starts by multiplying the first row of **L** by all columns of **U**, and equating with the corresponding coefficients of **A** to yield

$$u_{1j} = a_{1j} \quad j = 1,2,3,\ldots,N \tag{10.20}$$

Then the second through *N*th rows of **L** are multiplied by the first column of **U** leading to

$$\ell_{i1} u_{11} = a_{i1} \Rightarrow \ell_{i1} = \frac{a_{i1}}{u_{11}} \quad i = 2,3,\ldots,N \tag{10.21}$$

The process is repeated by multiplying the second row of **L** by the second through *N*th columns of **U** to give

$$u_{2j} = a_{2j} - \ell_{21} u_{1j} \quad j = 2,3,\ldots,N \tag{10.22}$$

after that the third through *N*th rows of **L** are multiplied by the second column of **U** to give

$$\ell_{i2} u_{22} + \ell_{i1} u_{12} = a_{i2} \Rightarrow \ell_{i2} = \frac{a_{i2} - \ell_{i1} u_{12}}{u_{22}} \quad i = 3,4,\ldots,N \tag{10.23}$$

In general, the *i*th row of **L** is multiplied by the *i*th through *N*th columns of **U**, resulting in

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \quad j = i, i+1, \ldots, N \tag{10.24}$$

and the (*i* + 1)th through *N*th rows of **L** are multiplied by the *i*th column of **U**, giving

$$\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}} \quad k = i+1, i+2, \ldots, N \tag{10.25}$$

For the *N*th row of **L**, its coefficients are multiplied by the coefficients of the *N*th column of **U** from which u_{NN} is obtained as

$$u_{NN} = a_{NN} - \sum_{k=1}^{N-1} \ell_{Nk} u_{kN} \tag{10.26}$$

A summary of the LU factorization is shown algorithmically below.

10.2.8 LU Decomposition Algorithm

```
$u_{1j} = a_{1j} \quad j = 1 \; to \; N$
$\ell_{i1} = \frac{a_{i1}}{u_{11}} \quad i = 2 \; to \; N$
For $i = 2 \; to \; N-1$
  {
    $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \qquad j = i, i+1, \ldots, N$
    $\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}} \qquad k = i+1, i+2, \ldots, N$
  }
$u_{NN} = a_{NN} - \sum_{i=1}^{N-1} \ell_{Ni} u_{iN}$
```

10.2.9 The Substitution Step

Having decomposed the original matrix **A** into **L** and **U**, the system of equations can be solved in a two step procedure via Eqs. (10.18) and (10.14). Note that the two-step procedure is equivalent to solving two linear systems of equations but now simplified by the fact that **L** and **U** are of lower and upper triangular form, respectively.

In the first step the vector **c** is obtained from Eq. (10.18) by **forward substitution**. The process can be described as

$$\begin{aligned} c_1 &= b_1 \\ c_i &= b_i - \sum_{j=1}^{i-1} \ell_{ij} c_j \quad i = 2, 3, \ldots, N \end{aligned} \tag{10.27}$$

In the second step, the ϕ values are found from Eq. (10.14) by **back substitution**. The process is described by

$$\begin{aligned} \phi_N &= \frac{c_N}{u_{NN}} \\ \phi_i &= \frac{c_i - \sum_{j=i+1}^{N} u_{ij}\phi_j}{u_{ii}} \quad i = N-1, N-2, \ldots, 3, 2, 1 \end{aligned} \tag{10.28}$$

The elements of **L** and **U** can be directly stored in the original matrix **A** if it is no longer needed. This is because the elements of **A** are only needed when the corresponding elements of either **L** or **U** are calculated. The number of operations required to perform the LU factorization of a square matrix of size $N \times N$ is $2N^3/3$, which is double the number of operations required to solve the same system of equations by Gauss elimination. Again the advantage of using LU factorization is when the same matrix **A** applies to many systems with different **b** vectors. Nevertheless, the main reason for introducing the LU factorization is because it forms the basis for developing some of the more efficient iterative solvers of linear algebraic systems of equations, which will be introduced in the next section.

10.2.10 LU Decomposition and Gauss Elimination

It may not be apparent, but Gauss elimination can be used to perform LU decomposition. It was shown that the forward elimination step results in an upper triangular matrix **U**. In the process however, **L** is actually produced. The elements of **L** are the factors (denoted by *ratio* in the Gauss elimination algorithm) by which the rows are multiplied during the various elimination steps. The below algorithm, which assumes a unit lower triangular matrix **L**, performs the LU decomposition of **A** by Gauss elimination.

10.2.11 LU Decomposition Algorithm by Gauss Elimination

```
$u_{1j} = a_{1j} \quad j = 1 \; to \; N$
$For \; k = 1 \; to \; N-1$
  {
    $For \; i = k+1 \; to \; N$
      {
        $\ell_{ik} = \frac{a_{ik}}{a_{kk}}$
          {
            $For \; j = k+1 \; to \; N$
                $u_{ij} = a_{ij} - \ell_{ik} * a_{kj}$
          }
      }
  }
```

Example 1

Solve the following system of linear algebraic equations using the LU decomposition technique:

$$\begin{bmatrix} 3 & -1 & 0 & 0 \\ -2 & 6 & -1 & 0 \\ 0 & -2 & 6 & -1 \\ 0 & 0 & -2 & 7 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ -3 \end{bmatrix}$$

Solution

The system is of the form $\mathbf{A\phi} = b$. The **L** and **U** should satisfy

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} 3 & -1 & 0 & 0 \\ -2 & 6 & -1 & 0 \\ 0 & -2 & 6 & -1 \\ 0 & 0 & -2 & 7 \end{bmatrix}$$

Following the procedure described above the elements are calculated as follows:

$$u_{1j} = a_{1j} \quad j = 1, 2, 3, 4 \Rightarrow \begin{cases} u_{11} = 3 \\ u_{12} = -1 \\ u_{13} = 0 \\ u_{14} = 0 \end{cases}$$

$$\ell_{i1} = \frac{a_{i1}}{u_{11}} \quad i = 2, 3, 4 \Rightarrow \begin{cases} \ell_{21} = \dfrac{a_{21}}{u_{11}} = -\dfrac{2}{3} \\ \ell_{31} = \dfrac{a_{31}}{u_{11}} = 0 \\ \ell_{41} = \dfrac{a_{41}}{u_{11}} = 0 \end{cases}$$

$$u_{2j} = a_{2j} - \ell_{21} u_{1j} \quad j = 2, 3, 4 \Rightarrow \begin{cases} u_{22} = a_{22} - \ell_{21} u_{12} = \dfrac{16}{3} \\ u_{23} = a_{23} - \ell_{21} u_{13} = -1 \\ u_{24} = a_{24} - \ell_{21} u_{14} = 0 \end{cases}$$

$$\ell_{i2} = \frac{a_{i2} - \ell_{i1} u_{12}}{u_{22}} \quad i = 3, 4 \Rightarrow \begin{cases} \ell_{32} = \frac{a_{32} - \ell_{31} u_{12}}{u_{22}} = -\dfrac{3}{8} \\ \ell_{42} = \frac{a_{42} - \ell_{41} u_{12}}{u_{22}} = 0 \end{cases}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \quad i = 3, j = 3, 4 \Rightarrow \begin{cases} u_{33} = a_{33} - \ell_{31} u_{13} - \ell_{32} u_{23} = \dfrac{45}{8} \\ u_{34} = a_{34} - \ell_{31} u_{14} - \ell_{32} u_{24} = -1 \end{cases}$$

$$\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}} \quad i = 3, k = 4 \Rightarrow \ell_{43} = \frac{a_{43} - \ell_{41} u_{13} - \ell_{42} u_{23}}{u_{33}} = -\frac{16}{45}$$

$$u_{NN} = a_{NN} - \sum_{k=1}^{N-1} \ell_{Nk} u_{kN} \Rightarrow u_{44} = a_{44} - \ell_{41} u_{14} - \ell_{42} u_{24} - \ell_{43} u_{34} = \frac{299}{45}$$

Therefore the **L** and **U** matrices are given by

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\dfrac{2}{3} & 1 & 0 & 0 \\ 0 & -\dfrac{3}{8} & 1 & 0 \\ 0 & 0 & -\dfrac{16}{45} & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 3 & -1 & 0 & 0 \\ 0 & \dfrac{16}{3} & -1 & 0 \\ 0 & 0 & \dfrac{45}{8} & -1 \\ 0 & 0 & 0 & \dfrac{299}{45} \end{bmatrix}$$

The **c** vector should satisfy

$$\mathbf{Lc} = \mathbf{b} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{2}{3} & 1 & 0 & 0 \\ 0 & -\frac{3}{8} & 1 & 0 \\ 0 & 0 & -\frac{16}{45} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ -3 \end{bmatrix}$$

with its solution obtained as

$$\begin{aligned} &c_1 = 3 \\ &-\frac{2}{3}c_1 + c_2 = 4 \Rightarrow c_2 = 6 \\ &-\frac{3}{8}c_2 + c_3 = 5 \Rightarrow c_3 = \frac{29}{4} \\ &-\frac{16}{45}c_3 + c_4 = -3 \Rightarrow c_4 = -\frac{19}{45} \end{aligned}$$

The solution to the original equation is obtained by solving

$$\mathbf{U\phi} = \mathbf{c} \Rightarrow \begin{bmatrix} 3 & -1 & 0 & 0 \\ 0 & \frac{16}{3} & -1 & 0 \\ 0 & 0 & \frac{45}{8} & -1 \\ 0 & 0 & 0 & \frac{299}{45} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ \frac{29}{4} \\ -\frac{19}{45} \end{bmatrix}$$

with the solution found as

$$\left.\begin{aligned} &\frac{299}{45}\phi_4 = -\frac{19}{45} \Rightarrow \phi_4 = -\frac{19}{299} \\ &\frac{45}{8}\phi_3 - \phi_4 = \frac{29}{4} \Rightarrow \phi_3 = \frac{382}{299} \\ &\frac{16}{3}\phi_2 - \phi_3 = 6 \Rightarrow \phi_2 = \frac{408}{299} \\ &3\phi_1 - \phi_2 = 3 \Rightarrow \phi_1 = \frac{435}{299} \end{aligned}\right\} \Rightarrow \mathbf{\phi} = \begin{bmatrix} \frac{435}{299} \\ \frac{408}{299} \\ \frac{382}{299} \\ -\frac{19}{299} \end{bmatrix}$$

10.2.12 Direct Methods for Banded Sparse Matrices

The Gauss elimination and LU decomposition methods are applicable to any system of equations. In specific it can be used for solving the system of equations resulting from the discretization of the conservation equations of interest in this book on structured or unstructured grid networks. When a structured grid method is used the discretization process results in a system of equations with the non-zero elements of its matrix of coefficients aligning along few diagonals. Depending on the discretization stencil used and the dimension of the problem being solved, tridiagonal or pentadiagonal matrices may arise, for which efficient algorithms have been developed as described next.

10.2.13 TriDiagonal Matrix Algorithm (TDMA)

The TriDiagonal Matrix Algorithm (TDMA), also known as Thomas algorithm [5, 6], solves the system of algebraic equations with a tridiagonal coefficient matrix written as

$$a_i\phi_i + b_i\phi_{i+1} + c_i\phi_{i-1} = d_i \quad i = 1, 2, 3, \ldots, N \quad c_1 = b_N = 0 \tag{10.29}$$

For the grid arrangement adopted in this book, i refers to the grid point location shown in Fig. 10.1.

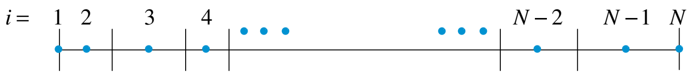


Fig. 10.1 One dimensional grid arrangement

For $i = 1$ the equation can be used to solve for ϕ_1 in term of ϕ_2 as

$$i = 1 \Rightarrow a_1\phi_1 = -b_1\phi_2 + d_1 \Rightarrow \phi_1 = -\frac{b_1}{a_1}\phi_2 + \frac{d_1}{a_1} \tag{10.30}$$

Similarly for $i = 2$ Eq. (10.29) with the help of Eq. (10.30) allows expressing ϕ_2 solely in term of ϕ_3 as

$$i = 2 \Rightarrow a_2\phi_2 = -b_2\phi_3 - c_2\phi_1 + d_2 \Rightarrow \phi_2 = -\frac{a_1 b_2}{a_1 a_2 - c_2 b_1}\phi_3 + \frac{d_2 a_1 - c_2 d_1}{a_1 a_2 - c_2 b_1} \tag{10.31}$$

The same can be repeated for ϕ_3 through ϕ_N suggesting that in general ϕ_i can be expressed as function of ϕ_{i+1} according to

$$\phi_i = P_i\phi_{i+1} + Q_i \quad i = 1, 2, 3, \ldots, N \tag{10.32}$$

Equation (10.32) for $i - 1$ when combined with Eq. (10.29) results in

$$\left.\begin{array}{r}\phi_{i-1}=P_{i-1}\phi_i+Q_{i-1}\\ a_i\phi_i+b_i\phi_{i+1}+c_i\phi_{i-1}=d_i\end{array}\right\}\Rightarrow \phi_i=-\frac{b_i}{a_i+c_iP_{i-1}}\phi_{i+1}+\frac{d_i-c_iQ_{i-1}}{a_i+c_iP_{i-1}} \tag{10.33}$$

Comparing Eq. (10.32) with Eq. (10.33) the following recurrence relations for P_i and Q_i are found:

$$P_i=-\frac{b_i}{a_i+c_iP_{i-1}} \quad Q_i=\frac{d_i-c_iQ_{i-1}}{a_i+c_iP_{i-1}} \quad i=1,2,\ldots,N \tag{10.34}$$

For $i = 1$ the values for P_1 and Q_1 are computed from Eq. (10.30) as

$$P_1=-\frac{b_1}{a_1} \quad Q_1=\frac{d_1}{a_1} \tag{10.35}$$

For $i = N$, since $b_N = 0$ the following is deduced:

$$b_N=0\Rightarrow P_N=0\Rightarrow \phi_N=Q_N \tag{10.36}$$

The TDMA solution algorithm can be summarized as follows:

```
1. Compute the values for $P_1$ and $Q_1$ using Eq. (10.35)
2. For $i=2,3,...,N$ use forward recursion to compute the values of $P_i$
   and $Q_i$ from Eq. (10.34)
3. Set $\phi_N=Q_N$ as given by Eq. (10.36)
4. For $i=N-1,N-2,...3,2,1$ use backward recursion to compute the values
   of $\phi_i$ from Eq. (10.32)
```

10.2.14 PentaDiagonal Matrix Algorithm (PDMA)

The PentaDiagonal Matrix Algorithm (PDMA) [7–10], solves the system of algebraic equations with a pentadiagonal coefficient matrix arising from discretization schemes that relate the value of ϕ_i at grid point i to the values at its two upstream ($i-1$ and $i-2$) and two downstream ($i+1$ and $i+2$) neighboring node values. For the notation illustrated schematically in Fig. 10.1, the general algebraic equation is written as

$$a_i\phi_i+b_i\phi_{i+2}+c_i\phi_{i+1}+d_i\phi_{i-1}+e_i\phi_{i-2}=f_i \quad i=1,2,3,\ldots,N \tag{10.37}$$

Subject to

$$\begin{array}{c}d_1=e_1=e_2=0\\ b_{N-1}=b_N=c_N=0\end{array} \tag{10.38}$$

For $i = 1$, Eq. (10.37) gives

$$\phi_1 = -\frac{b_1}{a_1}\phi_3 - \frac{c_1}{a_1}\phi_2 + \frac{f_1}{a_1} \tag{10.39}$$

while for $i = 2$ the value of ϕ_2 is found to be

$$\phi_2 = -\frac{a_1 b_2}{a_1 a_2 - d_2 c_1}\phi_4 - \frac{a_1 c_2 - b_1 d_2}{a_1 a_2 - d_2 c_1}\phi_3 + \frac{a_1 f_2 - d_2 f_1}{a_1 a_2 - d_2 c_1} \tag{10.40}$$

The process can be continued for other values of i and in general ϕ_i can be expressed as

$$\phi_i = P_i\phi_{i+2} + Q_i\phi_{i+1} + R_i \quad i = 1, 2, 3, \ldots, N \tag{10.41}$$

Computing ϕ_{i-1} and ϕ_{i-2} using Eq. (10.41) and substituting their values in Eq. (10.37), an equation for ϕ_i is derived as

$$\begin{aligned}\phi_i = &-\frac{b_i}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\phi_{i+2} \\ &-\frac{c_i + (d_i + e_i Q_{i-2})P_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\phi_{i+1} \\ &+\frac{f_i - e_i R_{i-2} - (d_i + e_i Q_{i-2})R_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\end{aligned} \tag{10.42}$$

Comparing Eqs. (10.41) and (10.42) P_i, Q_i, and R_i are found as

$$\begin{aligned}P_i &= -\frac{b_i}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}} \\ Q_i &= -\frac{c_i + (d_i + e_i Q_{i-2})P_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}} \\ R_i &= \frac{f_i - e_i R_{i-2} - (d_i + e_i Q_{i-2})R_{i-1}}{a_i + e_i P_{i-2} + (d_i + e_i Q_{i-2})Q_{i-1}}\end{aligned} \tag{10.43}$$

with their values for $i = 1$ and 2 given by

$$\begin{aligned}&P_1 = -\frac{b_1}{a_1} \quad Q_1 = -\frac{c_1}{a_1} \quad R_1 = \frac{f_1}{a_1} \\ &P_2 = -\frac{b_2}{a_2 + d_2 Q_1} \quad Q_2 = -\frac{c_2 + d_2 P_1}{a_2 + d_2 Q_1} \quad R_2 = \frac{f_2 - d_2 R_1}{a_2 + d_2 Q_1}\end{aligned} \tag{10.44}$$

Since $b_{N-1} = b_N = c_N = 0$ then $P_{N-1} = P_N = Q_N = 0$. Thus, the equations for ϕ_{N-1} and ϕ_N are found from

$$\begin{aligned}\phi_N &= R_N \\ \phi_{N-1} &= Q_{N-1}\phi_N + R_{N-1}\end{aligned} \tag{10.45}$$

The PDMA solution algorithm can be summarized as follows:

```
1. Compute the values for $P_1$, $Q_1$, $R_1$, $P_2$, $Q_2$, and $R_2$ using Eq.
   (10.44).
2. For $i = 3,4,...,N$ use forward recursion to compute the values of $P_i$,
   $Q_i$, and $R_i$ from Eq. (10.43).
3. Compute $\phi_N$ and $\phi_{N-1}$ from Eq. (10.45).
4. For $i = N-2,...3,2,1$ use backward recursion to compute the values of
   $\phi_i$ from Eq. (10.41).
```

10.3 Iterative Methods

Direct methods are generally not appropriate for solving large systems of equations particularly when the coefficient matrix is sparse, i.e., when most of the matrix elements are zero. This is more so when the linearized system of equations is nonlinear with solution dependent coefficients, or when dealing with time dependent problems. This is exactly the type of equations encountered when solving fluid flow problems.

In contrast, iterative methods are more appealing for these problems since the solution of the linearized system becomes part of the iterative solution process. Add to that the low computer storage and low computational cost requirements of this approach relative to the direct method.

There are many families of iterative methods and for a thorough review of this approach the reader is directed to dedicated books on the topic [11–14]. In this chapter a brief examination of basic iterative methods is provided along with an appraisal of multigrid algorithms that are generally used to address their deficiency. The Gauss elimination and LU decomposition direct methods were introduced for the sole purpose of clarifying some fundamental numerical processes needed for understanding iterative methods.

To unify the presentation of these methods, the coefficient matrix will be written in the following form:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U} \tag{10.46}$$

where **D**, **L**, and **U** refers to a diagonal, strictly lower, and strictly upper matrix, respectively.

Iterative methods for solving a linear system of the type $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$, compute a series of solutions $\boldsymbol{\phi}^{(n)}$ that, if certain conditions are satisfied, converge to the exact solution $\boldsymbol{\phi}$. Thus, for the solution, a starting point is chosen (i.e., $\boldsymbol{\phi}^{(0)}$ is selected as the initial condition or initial guess) and an iterative procedure that computes $\boldsymbol{\phi}^{(n)}$ from the previously computed $\boldsymbol{\phi}^{(n-1)}$ field is developed.

A "fixed-point" iteration can always be associated to the above system by decomposing matrix **A** as

$$\mathbf{A} = \mathbf{M} - \mathbf{N} \tag{10.47}$$

Using this decomposition, Eq. (10.1) is rewritten as

$$(\mathbf{M} - \mathbf{N})\boldsymbol{\phi} = \mathbf{b} \tag{10.48}$$

Applying a fixed point iteration solution procedure, Eq. (10.48) becomes

$$\mathbf{M}\boldsymbol{\phi}^{(n)} = \mathbf{N}\boldsymbol{\phi}^{(n-1)} + \mathbf{b} \tag{10.49}$$

which can be rewritten in the following form:

$$\boldsymbol{\phi}^{(n)} = \mathbf{B}\boldsymbol{\phi}^{(n-1)} + \mathbf{C}\mathbf{b} \quad n = 1, 2, \ldots \tag{10.50}$$

where $\mathbf{B} = \mathbf{M}^{-1}\mathbf{N}$ and $\mathbf{C} = \mathbf{M}^{-1}$. Different choices of these matrices define different iterative methods.

Before embarking on the description of the various iterative methods, a minimal set of characteristics that an iterative method should possess to guarantee convergence is first presented.

A. The iterative equation can be written at convergence as

$$\boldsymbol{\phi} = \mathbf{B}\boldsymbol{\phi} + \mathbf{C}\mathbf{b} \tag{10.51}$$

which, after rearranging, becomes

$$\mathbf{C}^{-1}(\mathbf{I} - \mathbf{B})\boldsymbol{\phi} = \mathbf{b} \tag{10.52}$$

Comparing Eq. (10.52) with Eq. (10.1), the coefficient matrix is obtained as

$$\mathbf{A} = \mathbf{C}^{-1}(\mathbf{I} - \mathbf{B}) \tag{10.53}$$

or, alternatively, as

$$\mathbf{B} + \mathbf{C}\mathbf{A} = \mathbf{I} \tag{10.54}$$

This relation between the various matrices ensures that once the exact solution is reached all consecutive iterations will not modify it.

B. Starting from some guess $\boldsymbol{\phi}^{(0)} \neq \boldsymbol{\phi}$, the method should guarantee that $\boldsymbol{\phi}^{(n)}$ will converge to $\boldsymbol{\phi}$ as $\boldsymbol{n}$ increases. Since $\boldsymbol{\phi}^{(n)}$ can be expressed in terms of $\boldsymbol{\phi}^{(0)}$ as

$$\boldsymbol{\phi}^{(n)} = \mathbf{B}^n \boldsymbol{\phi}^{(0)} + \sum_{i=0}^{n-1} \mathbf{B}^i \mathbf{C}\mathbf{b} \tag{10.55}$$

then, for the above to be true, **B** should satisfy

$$\lim_{n\to\infty} \mathbf{B}^n = \lim_{n\to\infty} \underbrace{\mathbf{B} * \mathbf{B} * \mathbf{B} \cdots * \mathbf{B}}_{n\ times} = 0 \tag{10.56}$$

Equation (10.56) implies that the spectral radius of **B** should be less than 1, i.e.,

$$\rho(\mathbf{B}) < 1 \tag{10.57}$$

This condition guaranties that the iterative method is self corrective, i.e., it is robust to any error adversely inserted into the solution vector $\boldsymbol{\phi}$.
More insight into the above condition can be obtained by defining the error $\mathbf{e}^{(n)}$ in the solution as the difference between the exact value and the value at any iteration (n), then

$$\mathbf{e}^{(n)} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi} \quad \text{and} \quad \mathbf{e}^{(n-1)} = \boldsymbol{\phi}^{(n-1)} - \boldsymbol{\phi} \tag{10.58}$$

Subtracting Eq. (10.51) from Eq. (10.50) and using the definitions in Eq. (10.58), a relation between the error at iteration n and $n - 1$ is obtained as

$$\mathbf{e}^{(n)} = \mathbf{B}\mathbf{e}^{(n-1)} \tag{10.59}$$

Thus, for the method to converge, the following should be satisfied:

$$\lim_{n\to\infty} \mathbf{e}^{(n)} = 0 \tag{10.60}$$

To translate Eq. (10.60) into something meaningful, the eigenvectors of **B** are assumed to be complete and to form a full set, meaning that they form a basis for $\mathbf{R}^N$. This being the case then **e** can be expressed as a linear combination of the N eigenvectors **v** of **B**. That is

$$\mathbf{e} = \sum_{i=1}^{N} \alpha_i \mathbf{v}_i \tag{10.61}$$

with each of the eigenvectors satisfying

$$\mathbf{B}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{10.62}$$

where λ_i is the eigenvalue corresponding to the eigenvector $\mathbf{v}_i$. Starting with the first iteration, Eq. (10.59) gives

$$\mathbf{e}^{(1)} = \mathbf{B}\mathbf{e}^{(0)} = \mathbf{B}\sum_{i=1}^{N}\alpha_i\mathbf{v}_i = \sum_{i=1}^{N}\alpha_i(\mathbf{B}\mathbf{v}_i) = \sum_{i=1}^{N}\alpha_i\lambda_i\mathbf{v}_i \tag{10.63}$$

For the second iteration, the error is obtained as

$$\mathbf{e}^{(2)} = \mathbf{B}\mathbf{e}^{(1)} = \mathbf{B}\sum_{i=1}^{N}\alpha_i\lambda_i\mathbf{v}_i = \sum_{i=1}^{N}\alpha_i\lambda_i(\mathbf{B}\mathbf{v}_i) = \sum_{i=1}^{N}\alpha_i\lambda_i^2\mathbf{v}_i \tag{10.64}$$

This procedure can be continued and it is easily shown by induction that

$$\mathbf{e}^{(n)} = \sum_{i=1}^{N}\alpha_i\lambda_i^n\mathbf{v}_i \tag{10.65}$$

Therefore for the iterative procedure to converge as n approaches infinity, all eigenvalues should be less than 1. If any of them is greater than 1 then the error will tend to infinity. This explains the importance of the spectral radius ρ of the matrix **B** defined as

$$\rho(\mathbf{B}) = \max_{i=1}^{N}(\lambda_i) \tag{10.66}$$

that was mentioned above. The convergence of iterative methods is accelerated by reducing the spectral radius of the iterative matrix. This is at the heart of iterative techniques.

C. Some type of a stopping criterion is needed with iterative methods. Many used criteria are based on a variation of the norm of the residual error defined as

$$\mathbf{r}^{(n)} = \mathbf{A}\boldsymbol{\phi}^{(n)} - \mathbf{b} \tag{10.67}$$

One criterion is to find the maximum residual in the domain and to require its value to become less than some threshold ε to declare a solution converged, i.e.,

$$\underset{i=1}{\overset{N}{Max}}\left|b_i - \sum_{j=1}^{N}a_{ij}\phi_j^{(n)}\right| \le \varepsilon \tag{10.68}$$

or that the root mean square residual be smaller than ε, i.e.,

$$\frac{\sum_{i=1}^{N}\left(b_i - \sum_{j=1}^{N}a_{ij}\phi_j^{(n)}\right)^2}{N} \le \varepsilon \tag{10.69}$$

Another possible criterion is for the maximum normalized difference between two consecutive iterations to drop below ε. This condition can be written as

$$\underset{i=1}{\overset{N}{Max}}\left|\frac{\phi_i^{(n)}-\phi_i^{(n-1)}}{\phi_i^{(n)}}\right|\times 100\leq\varepsilon \tag{10.70}$$

10.3.1 Jacobi Method

Perhaps the simplest of the iterative methods for solving a linear system of equations is the Jacobi method, which is graphically presented in Fig. 10.2.

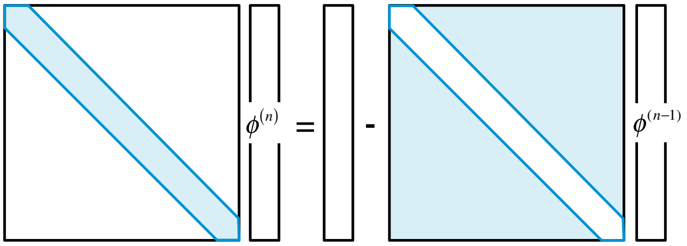


Fig. 10.2 A graphical representation of the Jacobi method

Considering the system of equations described by Eq. (10.1), if the diagonal elements are nonzero, then the first equation can be used to solve for ϕ_1, the second equation to solve for ϕ_2, and so on. The solution process starts by assigning guessed values to the unknown vector $\boldsymbol{\phi}$. These guessed values are used to calculate new estimates starting with ϕ_1, then ϕ_2, and computations proceed until a new estimate for ϕ_N is computed. This represents one iteration. Results obtained are treated as a new guess for the next iteration and the solution process is repeated. Iterations continue until the changes in the predictions between two consecutive iterations drop below a vanishing value or until a preset convergence criterion is satisfied. Once this happens the final solution is reached. In this method, given some current estimate $\phi^{(n-1)}$, an update is obtained using the following relation:

$$\phi_j^{(n)}=\frac{1}{a_{ii}}\left(b_i-\sum_{\substack{j=1\\ j\neq i}}^{N}a_{ij}\phi_j^{(n-1)}\right)\qquad i=1,2,3,\ldots,N \tag{10.71}$$

Equation (10.71) indicates that values obtained during an iteration are not used in the subsequent calculations during the same iteration but rather retained for the next iteration. Using matrices, the expanded form of Eq. (10.71) is given by

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 & 0 \\ 0 & a_{22} & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \dots & 0 & a_{N-1,N-1} & 0 \\ 0 & 0 & \dots & 0 & a_{NN} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & 0 & \dots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N-1,1} & a_{N-1,2} & \dots & 0 & a_{N-1,N} \\ a_{N1} & a_{N2} & a_{N3} & \dots & 0 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} \tag{10.72}$$

Solving for $\boldsymbol{\phi}^{(n)}$, Eq. (10.72) yields

$$\begin{bmatrix} \phi_1^{(n)} \\ \phi_2^{(n)} \\ \vdots \\ \phi_{N-1}^{(n)} \\ \phi_N^{(n)} \end{bmatrix} = \begin{bmatrix} a_{11} & 0 & \dots & 0 & 0 \\ 0 & a_{22} & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \dots & 0 & a_{N-1,N-1} & 0 \\ 0 & 0 & \dots & 0 & a_{NN} \end{bmatrix}^{-1} \left(\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_N \end{bmatrix} - \begin{bmatrix} 0 & a_{12} & \dots & a_{1N-1} & a_{1N} \\ a_{21} & 0 & \dots & a_{2N-1} & a_{2N} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{N-1,1} & a_{N-1,2} & \dots & 0 & a_{N-1,N} \\ a_{N1} & a_{N2} & a_{N3} & \dots & 0 \end{bmatrix} \begin{bmatrix} \phi_1^{(n-1)} \\ \phi_2^{(n-1)} \\ \vdots \\ \phi_{N-1}^{(n-1)} \\ \phi_N^{(n-1)} \end{bmatrix} \right) \tag{10.73}$$

Using Eqs. (10.46) and (10.73) can be more concisely written as

$$\phi^{(n)} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\phi^{(n-1)} + \mathbf{D}^{-1}\mathbf{b} \tag{10.74}$$

The Jacobi method converges as long as $\rho\left(-\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\right) < 1$. This condition is satisfied for a large class of matrices including diagonally dominant ones where their coefficients satisfy

$$\sum_{\substack{j=1\\j\neq i}}^{N} |a_{ij}| \leq |a_{ii}| \quad i = 1, 2, 3, \ldots, N \tag{10.75}$$

10.3.2 Gauss-Seidel Method

A more popular take on the Jacobi is the Gauss-Seidel method, which has better convergence characteristics. It is somewhat less expensive memory-wise since it does not require storing the new estimates in a separate array. Rather, it uses the latest available estimate of ϕ in its calculations. The iterative formula in the Gauss Seidel method, schematically displayed in Fig. 10.3, is given as

$$\phi_i^{(n)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\phi_j^{(n)} - \sum_{j=i+1}^{N} a_{ij}\phi_j^{(n-1)}\right) \quad i = 1, 2, 3, \ldots, N \tag{10.76}$$

In matrix form Eq. (10.76) is written as

$$\boldsymbol{\phi}^{(n)} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\boldsymbol{\phi}^{(n-1)} + (\mathbf{D} + \mathbf{L})^{-1}\mathbf{b} \tag{10.77}$$

In effect the Gauss-Seidel method uses the most recent values in its iteration, specifically all $\phi_j^{(n)}$ values for $j < i$ since by the time ϕ_i is to be calculated, the values of $\phi_1, \phi_2, \phi_3, \ldots, \phi_{i-1}$ at the current iteration are already calculated. This approach also saves memory since the newer value is always overwriting the previous one. The Gauss-Seidel iterations converge as long as

$$\rho\left(-(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\right) < 1 \tag{10.78}$$

Although in some cases the Jacobi method converges faster, Gauss-Seidel is the preferred method.

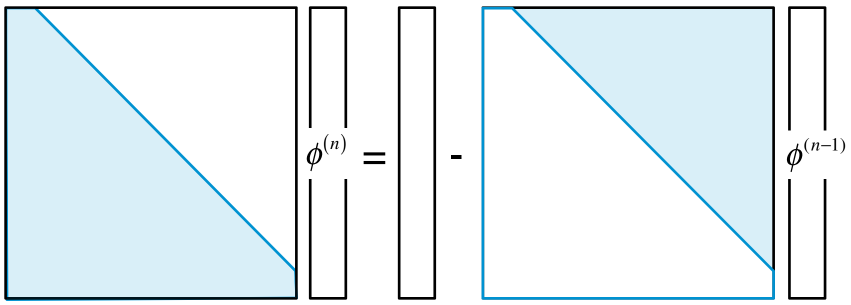


Fig. 10.3 A graphical representation of the Gauss-Seidel method

Example 2

Apply 5 iterations of the Gauss-Seidel and Jacobi methods to the system of equations in Example 1 and compute the errors at each iteration using the exact solution.

$$\boldsymbol{\phi} = \begin{bmatrix} \frac{435}{299} & \frac{408}{299} & \frac{382}{299} & -\frac{19}{299} \end{bmatrix}.$$

As an initial guess start with the field $\boldsymbol{\phi}^* = [0 \quad 0 \quad 0 \quad 0]$

Solution

Denoting with a superscript $(^*)$ values from the previous iteration, the equations to be solved in the Jacobi method are as follows:

$$\begin{aligned}
\phi_1 &= \frac{1}{3}\left(\phi_2^* + 3\right)\\
\phi_2 &= \frac{1}{6}\left(2\phi_1^* + \phi_3^* + 4\right)\\
\phi_3 &= \frac{1}{6}\left(2\phi_2^* + \phi_4^* + 5\right)\\
\phi_4 &= \frac{1}{7}\left(2\phi_3^* - 3\right)
\end{aligned}$$

with the error given as $\boldsymbol{\varepsilon} = \left|\boldsymbol{\phi}_{exact} - \boldsymbol{\phi}_{computed}\right|$. The solution for the first iteration is obtained as

$$\begin{aligned}
\phi_1 &= \frac{1}{3}(0+3) = 1 \Rightarrow \varepsilon_1 = |1.4548 - 1| = 0.4548\\
\phi_2 &= \frac{1}{6}(0+0+4) = 0.6667 \Rightarrow \varepsilon_2 = |1.3645 - 0.6667| = 0.6978\\
\phi_3 &= \frac{1}{6}(0+0+5) = 0.8333 \Rightarrow \varepsilon_3 = |1.2776 - 0.8333| = 0.4443\\
\phi_4 &= \frac{1}{7}(0-3) = -0.4286 \Rightarrow \varepsilon_4 = |-0.06354 + 0.4286| = 0.3650
\end{aligned}$$

Computations proceed in the same manner with solution obtained treated as the new guess. The results for the first five iterations are given in Table 10.1.

Table 10.1 Summary of results obtained using the Jacobi iterative method

Iter #	ϕ_1	ε_1	ϕ_2	ε_2	ϕ_3	ε_3	ϕ_4	ε_4
0	0	1.4548	0	1.3645	0	1.2776	0	0.06354
1	1	0.4548	0.6667	0.6978	0.8333	0.4443	−0.4286	0.3650
2	1.2222	0.2326	1.1389	0.2257	0.9841	0.2935	−0.1905	0.1269
3	1.3796	7.52E−02	1.2381	0.1265	1.1812	9.64E−02	−0.1474	8.38E−02
4	1.4127	4.22E−02	1.3234	4.11E−02	1.2215	5.61E−02	−9.11E−02	2.75E−02
5	1.4411	1.37E−02	1.3411	2.34E−02	1.2593	1.83E−02	−7.96E−02	1.6E−02

Denoting with a superscript (*) values from the previous iteration, the equations to be solved in the Gauss-Seidel method are as follows:

$$
\begin{aligned}
\phi_1 &= \frac{1}{3}\left(\phi_2^* + 3\right) \\
\phi_2 &= \frac{1}{6}\left(2\phi_1 + \phi_3^* + 4\right) \\
\phi_3 &= \frac{1}{6}\left(2\phi_2 + \phi_4^* + 5\right) \\
\phi_4 &= \frac{1}{7}(2\phi_3 - 3)
\end{aligned}
$$

with the error given as $\boldsymbol{\varepsilon} = \left|\boldsymbol{\phi}_{exact} - \boldsymbol{\phi}_{computed}\right|$. The solution for the first iteration is obtained as

$$
\begin{aligned}
\phi_1 &= \frac{1}{3}(0+3) = 1 \Rightarrow \varepsilon_1 = |1.4548 - 1| = 0.4548 \\
\phi_2 &= \frac{1}{6}(2*1+0+4) = 1 \Rightarrow \varepsilon_2 = |1.3645 - 1| = 0.3645 \\
\phi_3 &= \frac{1}{6}(2*1+0+5) = 1.1667 \Rightarrow \varepsilon_3 = |1.2776 - 1.1667| = 0.1109 \\
\phi_4 &= \frac{1}{7}(2*1.1667-3) = -0.09523 \Rightarrow \varepsilon_4 = |-0.06354 + 0.09523| = 0.03169
\end{aligned}
$$

Computations proceed in the same manner with solution obtained treated as the new guess. The results for the first five iterations are given in Table 10.2.

Table 10.2 Summary of results obtained using the Gauss-Siedel iterative method

Iter #	ϕ_1	ε_1	ϕ_2	ε_2	ϕ_3	ε_3	ϕ_4	ε_4
0	0	1.4548	0	1.3645	0	1.2776	0	0.06354
1	1	0.4548	1	0.3645	1.1667	0.1109	−0.09523	0.03169
2	1.3333	0.1215	1.3056	5.90E−02	1.2526	2.49E−02	−7.07E−02	7.13E−03
3	1.4352	1.97E−02	1.3538	1.07E−02	1.2728	4.76E−03	−6.49E−02	1.36E−03
4	1.4513	3.57E−03	1.3626	1.98E−03	1.2767	8.88E−04	−6.38E−02	2.54E−04
5	1.4542	6.61E−04	1.3642	3.68E−04	1.2774	1.65E−04	−6.36E−02	4.72E−05

10.3.3 Preconditioning and Iterative Methods

The rate of convergence of iterative methods depends on the spectral properties of the iteration matrix **B**, which is contingent on the matrix of coefficients. Based on that an iterative method looks for a transformation of the system of equations into an equivalent one that has the same solution, but of better spectral properties. Under these conditions the eigenvalues of the equivalent system are more clustered allowing the iterative solution to be obtained faster than with the original system. A preconditioner is defined as a matrix that effects such a transformation.

A preconditioning matrix **P** is defined such that the system

$$\mathbf{P}^{-1}\mathbf{A}\boldsymbol{\phi} = \mathbf{P}^{-1}\mathbf{b} \tag{10.79}$$

has the same solution as the original system $\mathbf{A}\boldsymbol{\phi} = \mathbf{b}$, but the spectral properties of its coefficient matrix $\mathbf{P}^{-1}\mathbf{A}$ are more conducive. In defining the preconditioner **P**, the difficulty is to find a matrix that approximates $\mathbf{A}^{-1}$ and is easy to invert (i.e., to find $\mathbf{P}^{-1}$) at a reasonable cost.

Writing again Eq. (10.47), but now with **P** replacing **M** (i.e., $\mathbf{M} = \mathbf{P}$ and $\mathbf{A} = \mathbf{P} - \mathbf{N}$) the associated fixed point iteration system is given by

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= \mathbf{B}\boldsymbol{\phi}^{(n-1)} + \mathbf{C}\mathbf{b}\\ &= \mathbf{P}^{-1}\mathbf{N}\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b}\\ &= \mathbf{P}^{-1}(\mathbf{P} - \mathbf{A})\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b}\\ &= \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b}\end{aligned} \tag{10.80}$$

which in residual form can be written as

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{b}\\ &= \boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right)\\ &= \boldsymbol{\phi}^{(n-1)} + \mathbf{P}^{-1}\mathbf{r}^{(n-1)}\end{aligned} \tag{10.81}$$

From both equations it is now clear that the iterative procedure is just a fixed-point iteration on a preconditioned system associated with the decomposition $\mathbf{A} = \mathbf{P} - \mathbf{N}$ where the spectral properties are now

$$\rho\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right) < 1 \tag{10.82}$$

By comparison, the preconditioning matrix for the Jacobi (J) and Gauss-Seidel (GS) methods are simply

$$\begin{aligned}\mathbf{P}_J &= \mathbf{D}\\ \mathbf{P}_{GS} &= \mathbf{D} + \mathbf{L}\end{aligned} \tag{10.83}$$

where **D** and **L** are respectively the diagonal and lower triangular part of matrix **A**.

Thus, preconditioning is a manipulation of the original system to improve its spectral properties with the preconditioning matrix **P** used in the associated iterative procedure. As will be described in the following sections, it is possible to develop more advanced preconditioning matrices in which the coefficients are defined in a more complex way.

10.3.4 Matrix Decomposition Techniques

The low rate of convergence of the Gauss-Seidel and Jacobi methods was the prime motivator for the development of faster iterative techniques. One approach to accelerate the convergence rate of solvers and to develop iterative methods is through the use of more advanced preconditioners. A simple, yet efficient, approach for that purpose is to perform an *incomplete* factorization of the original matrix of coefficients **A**. The stress on incomplete is essential since a complete factorization of **A** into a lower **L** and an upper triangular matrix **U** is tantamount to a direct solution and is very expensive in term of memory requirements (fill in and loss of sparsity) and computational cost.

10.3.5 Incomplete LU (ILU) Decomposition

As can be seen in Example 1, the **L** and **U** matrices result in non-zero elements at locations that were 0 in the original matrix **A** (this is known as fill-in). So if an incomplete LU (ILU) factorization of **A** is performed such that the resulting lower **L** and upper **U** matrices have the same nonzero structure as the lower and upper parts of **A**, then

$$\mathbf{A} = \mathbf{L}\mathbf{U} + \mathbf{R} \tag{10.84}$$

where **R** is the residual of the factorization procedure. The matrices **L** and **U** being sparse (same structure as **A**) are easier to deal with then if they were obtained from a complete factorization. However, their product being an approximation to **A**, necessitates the use of an iterative solution procedure to solve the system of equations. The first step in the solution process is to rewrite Eq. (10.1) as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \Rightarrow 0 = \mathbf{b} - \mathbf{A}\boldsymbol{\phi} \Rightarrow (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} + (\mathbf{b} - \mathbf{A}\boldsymbol{\phi}) \tag{10.85}$$

Denoting values obtained from the previous iteration with a superscript $(n-1)$, and values obtained at the current iteration with superscript (n), the iterative process is obtained by rewriting Eq. (10.85) in the following form:

$$(\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n)} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n-1)} + \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right) \tag{10.86}$$

Therefore values of $\boldsymbol{\phi}^{(n)}$ at the current iteration can be obtained from knowledge of $\boldsymbol{\phi}^{(n-1)}$ values obtained at the previous iteration. Equation (10.86) is usually solved in residual form whereby the solution $\boldsymbol{\phi}^{(n)}$ at iteration (n) is expressed in terms of the solution $\boldsymbol{\phi}^{(n-1)}$ at iteration $(n-1)$ plus a correction $\boldsymbol{\phi}'^{(n)}$, i.e.,

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{\phi}^{(n-1)} + \boldsymbol{\phi}'^{(n)} \tag{10.87}$$

Thus Eq. (10.86) becomes

$$(\mathbf{A} - \mathbf{R})\boldsymbol{\phi}'^{(n)} = \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}\right) \tag{10.88}$$

Once $\boldsymbol{\phi}'^{(n)}$ is found, Eq. (10.87) is used to update $\boldsymbol{\phi}$ at every iteration.

The ILU factorization can be performed using Gaussian elimination while dropping some non diagonal elements at preset locations. The locations where elements are to be dropped give rise to different ILU approximations.

10.3.6 Incomplete LU Factorization with no Fill-in ILU(0)

Many variants of the ILU factorization technique exist and the simplest is the one denoted by ILU(0) [15–17]. In ILU(0) the pattern of zero elements in the combined **L** and **U** matrices is taken to be precisely the pattern of zero elements in the original matrix **A**. Using Gaussian elimination, computations are performed as in the case of a full LU factorization, but any new nonzero element (ℓ_{ij} and u_{ij}) arising in the process is dropped if it appears at a location where a zero element exists in the original matrix **A**. Hence, the combined **L** and **U** matrices have together the same number of non zeros as the original matrix **A**. With this approach, the fill-in problem that usually arises when factorizing sparse matrices (i.e., the creation of nonzero elements at locations where the original matrix has zeros) is eliminated. In the process however, the accuracy is reduced thereby increasing the number of required iterations for convergence to be reached. To remedy this shortcoming, more accurate ILU factorization methods, which are often more efficient and more reliable, have been developed. These methods, differing by the level of fill-in allowed, are denoted by ILU(p) where p represents the order of fill-ins. The higher the level of fill-ins, the more expensive the ILU decomposition step becomes. Moreover, when used within a multigrid approach (to be explained in a later section), the ILU(0) method is more than adequate as a smoother. For this reason higher level methods are not presented and for more information interested readers are referred, among others, to the book by Saad [12].

An ILU(0) factorization algorithm which assumes **L** to be a unit lower triangular matrix and for which the same matrix **A** is used to store the elements of the unit lower and upper triangular matrices **L** and **U** is as given next.

10.3.7 ILU(0) Factorization Algorithm

```
$For\ k = 1\ to\ N - 1$
  {
    $For\ i = k + 1\ to\ N\ and\ if\ a_{ik} \neq 0\ \ Do:$
      {
        $a_{ik} = \frac{a_{ik}}{a_{kk}}\ \ (\ell\ values)$
          {
            $For\ j = k + 1\ to\ N\ and\ if\ a_{ij} \neq 0\ Do:$
            $a_{ij} = a_{ij} - a_{ik} * a_{kj}\ \ (u\ values)$
          }
      }
  }
```

It should be mentioned that the ILU decomposition of symmetric positive definite matrices is denoted by *Incomplete Cholesky* decomposition. In this case the factorization is made just of the lower (or upper) triangular part and the approximation to the original matrix is written as

$$\overline{\mathbf{L}}\,\overline{\mathbf{L}}^{\mathrm{T}} \approx \mathbf{A} \tag{10.89}$$

where $\bar{\mathbf{L}}$ is the factorized sparse lower triangular matrix (approximation of $\mathbf{L}$), with the preconditioning matrix $\mathbf{P}$ given by

$$\mathbf{P} = \overline{\mathbf{L}}\,\overline{\mathbf{L}}^{\mathrm{T}} \approx \mathbf{A} \tag{10.90}$$

10.3.8 ILU Factorization Preconditioners

A very popular class of preconditioners is based on incomplete factorizations. In the discussions of direct methods it was shown that decomposing a sparse matrix $\mathbf{A}$ into the product of a lower and an upper triangular matrices may lead to substantial fill-in. Because a preconditioner is only required to be an approximation to $\mathbf{A}^{-1}$, it is sufficient to look for an approximate decomposition of $\mathbf{A}$ such that $\mathbf{A} \approx \overline{\mathbf{L}}\,\overline{\mathbf{U}}$. Choosing $\mathbf{P} = \overline{\mathbf{L}}\,\overline{\mathbf{U}}$ leads also to an efficient evaluation of the inverse of the preconditioned matrix $\mathbf{P}^{-1}$ since the inversion can easily be performed by the forward and backward substitution, as described above, in which the exact **L** and **U** are now replaced by the approximations $\overline{\mathbf{L}}$ and $\overline{\mathbf{U}}$, respectively.

For the ILU(0) method, the incomplete factorization mimics the nonzero elements sparsity of the original matrix such that the pre-conditioner has exactly the size of the original matrix. In order to reduce the storage needed, Pommerell introduced a simplified version of the ILU called diagonal ILU (DILU) [18]. In the DILU the fill-in of the off-diagonal elements is eliminated (i.e., the upper and lower parts of the matrix are kept unchanged) and only the diagonal elements are modified.

In this case it is possible to write the preconditioner in the form

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}(\mathbf{D}^* + \mathbf{U}) \tag{10.91}$$

where **L** and **U** are the lower and upper triangular decomposition of **A**, and $\mathbf{D}^*$ is now a proper diagonal matrix, different from the diagonal of **A**. The $\mathbf{D}^*$ matrix is thus defined, as shown below, in a way that the diagonal of the product of the matrices in Eq. (10.91) equals the diagonal of **A**.

10.3.9 Algorithm for the Calculation of $\mathbf{D}^*$ in the DILU Method

```
$For\ i = 1\ to\ N\ \ Do:$
  {
    $d_{ii} = a_{ii}$
  }
$For\ i = 1\ to\ N\ \ Do:$
  {
    $For\ j = i + 1\ to\ N\ and\ if\ a_{ij} \neq 0, a_{ji} \neq 0\ Do:$
      {
        $d_{jj} = d_{jj} - \frac{a_{ji}}{d_{ii}} * a_{ij}$
      }
  }
```

In this case the inverse of the preconditioner, which is defined as

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}(\mathbf{D}^* + \mathbf{U}) = \overline{\mathbf{L}}\,\overline{\mathbf{U}}, \quad \overline{\mathbf{L}} = (\mathbf{D}^* + \mathbf{L})\mathbf{D}^{*-1}, \quad \overline{\mathbf{U}} = (\mathbf{D}^* + \mathbf{U})$$

or

$$\mathbf{P} = (\mathbf{D}^* + \mathbf{L})\left(\mathbf{I} + \mathbf{D}^{*-1}\mathbf{U}\right) = \overline{\mathbf{L}}\,\overline{\mathbf{U}}, \quad \overline{\mathbf{L}} = (\mathbf{D}^* + \mathbf{L}), \quad \overline{\mathbf{U}} = \left(\mathbf{I} + \mathbf{D}^{*-1}\mathbf{U}\right) \tag{10.92}$$

needed in the solution of $\mathbf{P}\boldsymbol{\phi}'^{(n+1)} = \mathbf{r}^{(n)}$ to find the correction field $\boldsymbol{\phi}'^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n)}$, can easily be calculated using the following forward and backward substitution algorithm.

10.3.10 Forward and Backward Solution Algorithm with the DILU Method

```
For $i = 1$ to $N$ Do:
{
  For $j = 1$ to $i-1$ Do:
  {
      $t_i = d_{ii}^{-1}(r_i - \ell_{ij} * t_j)$
  }
}
For $i = N$ to $1$ Do:
{
  For $j = i+1$ to $N$ Do:
  {
      $\phi'_i = t_i - d_{ii}^{-1}(u_{ij} * t_j)$
  }
}
```

The clear advantage of the DILU, apart from its recursive formulation, is that it requires only one extra diagonal of storage.

10.3.11 Gradient Methods for Solving Algebraic Systems

Another group of iterative procedures for solving linear algebraic systems of equations is the Gradient Methods, which include the Steepest Descent and the Conjugate Gradient methods. They were initially developed for cases where the coefficient matrix $\mathbf{A}$ is symmetric positive definite (SPD) to reformulate the problem as a minimization problem for the quadratic vector function $\mathbf{Q}(\boldsymbol{\phi})$ given by

$$\mathbf{Q}(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} + \mathbf{c} \tag{10.93}$$

where $\mathbf{c}$ is a vector of scalars, and other variables are as defined in Eq. (10.1). The minimum of $\mathbf{Q}(\boldsymbol{\phi})$ is obtained when its gradient with respect to $\boldsymbol{\phi}$ is zero. The gradient $\mathbf{Q}'(\boldsymbol{\phi})$ of a vector field $\mathbf{Q}(\boldsymbol{\phi})$, at a given $\boldsymbol{\phi}$, points in the direction of greatest increase of $\mathbf{Q}(\boldsymbol{\phi})$. Through mathematical manipulations the gradient is found as

$$\mathbf{Q}'(\boldsymbol{\phi}) = \frac{1}{2}\mathbf{A}^{\mathrm{T}}\boldsymbol{\phi} + \frac{1}{2}\mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.94}$$

If $\mathbf{A}$ is symmetric $\left(\mathbf{A} = \mathbf{A}^{\mathrm{T}}\right)$, then Eq. (10.94) implies

$$\mathbf{Q}'(\boldsymbol{\phi}) = \mathbf{A}\boldsymbol{\phi} - \mathbf{b} \tag{10.95}$$

The minimum is obtained when $\mathbf{Q}'(\boldsymbol{\phi}) = 0$, leading to

$$\mathbf{Q}'(\boldsymbol{\phi}) = 0 \Rightarrow \mathbf{A}\boldsymbol{\phi} = \mathbf{b} \tag{10.96}$$

Therefore minimizing $\mathbf{Q}(\boldsymbol{\phi})$ is equivalent to solving Eq. (10.1) and the solution of the minimization problem yields the solution of the system of linear equations.

Now for the function $\mathbf{Q}(\boldsymbol{\phi})$ to have a global minimum it is necessary for the coefficient matrix $\mathbf{A}$ to be positive definite, i.e., it should satisfy the inequality $\boldsymbol{\phi}^{T}\mathbf{A}\boldsymbol{\phi} > 0$ for all $\boldsymbol{\phi} \neq \mathbf{0}$. This requirement can be established by considering the relationship between the exact solution $\boldsymbol{\phi}$ and its current estimate $\boldsymbol{\phi}^{(n)}$. If $\mathbf{e} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi}$ denotes the difference between the exact solution and the current estimate, then Eq. (10.93) gives

$$\begin{aligned}
\mathbf{Q}(\boldsymbol{\phi} + \mathbf{e}) &= \frac{1}{2}(\boldsymbol{\phi} + \mathbf{e})^{\mathrm{T}}\mathbf{A}(\boldsymbol{\phi} + \mathbf{e}) - \mathbf{b}^{\mathrm{T}}(\boldsymbol{\phi} + \mathbf{e}) + \mathbf{c} \\
&= \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} + \frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\mathbf{e} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\mathbf{e} + \mathbf{c} \\
&= \underbrace{\frac{1}{2}\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi} - \mathbf{b}^{\mathrm{T}}\boldsymbol{\phi} + \mathbf{c}}_{\mathbf{Q}(\boldsymbol{\phi})} + \frac{1}{2}\left(\underbrace{\mathbf{e}^{\mathrm{T}}\mathbf{A}\boldsymbol{\phi}}_{\mathbf{e}^{\mathrm{T}}\mathbf{b}=\mathbf{b}^{\mathrm{T}}\mathbf{e}} + \underbrace{\boldsymbol{\phi}^{\mathrm{T}}\mathbf{A}\mathbf{e}}_{\mathbf{b}^{\mathrm{T}}\mathbf{e}}\right) - \mathbf{b}^{\mathrm{T}}\mathbf{e} + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e} \\
&= \mathbf{Q}(\boldsymbol{\phi}) + \frac{1}{2}\mathbf{e}^{\mathrm{T}}\mathbf{A}\mathbf{e}
\end{aligned} \tag{10.97}$$

indicating that if $\mathbf{A}$ is positive definite, the second term will be always positive except when $\mathbf{e} = \mathbf{0}$, in which case the required solution would have been obtained. Moreover, when $\mathbf{A}$ is positive definite, all its eigenvalues are positive and the function $\mathbf{Q}(\boldsymbol{\phi})$ has a unique minimum.

Thus with a **symmetric** and **positive definite** matrix a converging series of $\boldsymbol{\phi}^{(n)}$ can be derived such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\left(\boldsymbol{\delta\phi}^{(n)}\right) \tag{10.98}$$

where $\alpha^{(n)}$ is some relaxation factor, and $\boldsymbol{\delta\phi}^{(n)}$ is related to the correction needed to minimize the said function at each iteration. This can be accomplished in a variety of ways leading to different methods.

10.3.12 The Method of Steepest Descent

The method of steepest descent for solving linear systems of equations of the form given by Eq. (10.1) is based on minimizing the quadratic form given by Eq. (10.93). If $\boldsymbol{\phi}$ is a one dimensional vector with its components given by the scalar ϕ, then $Q(\phi)$ will represent a parabola. Finding the minimum of a parabolic function iteratively starting at some point ϕ_0, involves moving down along the parabola until hitting the minimum.

The same idea is used in N dimensions. In this case $\mathbf{Q}(\boldsymbol{\phi})$ may be depicted as a paraboloid and the solution is iteratively found starting from an initial position $\boldsymbol{\phi}^{(0)}$ and moving down the paraboloid until the minimum is reached. For quick convergence the sequence of steps $\boldsymbol{\phi}^{(0)}, \boldsymbol{\phi}^{(1)}, \boldsymbol{\phi}^{(2)}, \ldots$ should be selected such that the fastest rate of descent occurs, i.e., in the direction of $-\mathbf{Q}'(\boldsymbol{\phi})$. According to Eq. (10.95) this direction is also given by

$$-\mathbf{Q}'(\boldsymbol{\phi}) = \mathbf{b} - \mathbf{A}\boldsymbol{\phi} \tag{10.99}$$

The exact solution being $\boldsymbol{\phi}$, the error and residual at any step n, denoted respectively by $\mathbf{e}^{(n)}$ and $\mathbf{r}^{(n)}$, are computed as

$$\left.\begin{array}{r}\mathbf{e}^{(n)} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi}\\ \mathbf{r}^{(n)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)} = -\mathbf{Q}'\left(\boldsymbol{\phi}^{(n)}\right)\end{array}\right\} \Rightarrow \mathbf{r}^{(n)} = -\mathbf{A}\mathbf{e}^{(n)} \tag{10.100}$$

Moving linearly in the direction of the steepest descent, the value of $\boldsymbol{\phi}$ at step $n + 1$ can be expressed in terms of the value of $\boldsymbol{\phi}$ at step n according to

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)} \tag{10.101}$$

The value of $\alpha^{(n)}$ that minimizes $\mathbf{Q}(\boldsymbol{\phi})$ should satisfy

$$\frac{d}{d\alpha^{(n)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right) = 0 \tag{10.102}$$

This can be expanded into

$$\frac{d}{d\alpha^{(n)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right) = 0 \Rightarrow \left[\frac{d}{d\boldsymbol{\phi}^{(n+1)}}\mathbf{Q}\left(\boldsymbol{\phi}^{(n+1)}\right)\right]^{\mathrm{T}}\frac{d\boldsymbol{\phi}^{(n+1)}}{d\alpha^{(n)}} = 0 \Rightarrow \left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \tag{10.103}$$

indicating that the new step should be in a direction normal to the old step. The value of $\alpha^{(n)}$ is calculated by using Eq. (10.103) as follows:

$$\begin{aligned}
&\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0 \Rightarrow \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = 0\\
&\Rightarrow \left[\mathbf{b} - \mathbf{A}\left(\boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}\right)\right]^{\mathrm{T}}\mathbf{r}^{(n)} = 0\\
&\Rightarrow \left(\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = \alpha^{(n)}\left(\mathbf{A}\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}\\
&\Rightarrow \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} = \alpha^{(n)}\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)}\\
&\Rightarrow \alpha^{(n)} = \frac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)}}
\end{aligned} \tag{10.104}$$

The steepest descent algorithm can be summarized as follows:

```
$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose residual as starting direction)
iterate starting at $(n)$ until convergence
$\mathbf{r}^{(n)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n)}$ (Compute the residual vector)
$\alpha^{(n)} = \frac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(n)}}$ (Compute the factor in the orthogonal direction)
$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}$ (Obtain new $\boldsymbol{\phi}$ )
```

As presented above, the algorithm necessitates performing two matrix-vector multiplications per iteration. One of them can be eliminated by multiplying both sides of Eq. (10.101) by $-\mathbf{A}$ and adding $\mathbf{b}$ to obtain

$$\begin{aligned}\boldsymbol{\phi}^{(n+1)} &= \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)} \Rightarrow \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n+1)} \\ &= \mathbf{b} - \mathbf{A}\left(\boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{r}^{(n)}\right) \Rightarrow \mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{r}^{(n)}\end{aligned} \tag{10.105}$$

The equation for $\mathbf{r}^{(n)}$ in step 1 is needed only to calculate $\mathbf{r}^{(0)}$, while Eq. (10.105) can be used afterwards. With this formulation there will be no need to compute $\mathbf{A}\boldsymbol{\phi}^{(n)}$, as it was replaced by $\mathbf{A}\mathbf{r}^{(n)}$. However a shortcoming of this approach, is the lack of feedback from the value of $\boldsymbol{\phi}^{(n)}$ into the residual, which may cause the solution to converge to a value different from the exact one due to accumulation of roundoff errors. This deficiency can be resolved by periodically computing the residual using the original equation.

10.3.13 The Conjugate Gradient Method

While the steepest descent method guarantees convergence, its rate of convergence is low. This slow convergence is caused by oscillations around local minima forcing the method to search in the same direction repeatedly. To avoid this undesirable behavior every new search should be in a direction different from the directions of previous searches [19]. This can be accomplished by selecting a set of search directions $\mathbf{d}^{(0)}, \mathbf{d}^{(1)}, \mathbf{d}^{(2)}, \ldots, \mathbf{d}^{(N-1)}$ that are $\mathbf{A}$-orthogonal. Two vectors $\mathbf{d}^{(n)}$ and $\mathbf{d}^{(m)}$ are said to be $\mathbf{A}$-orthogonal if they satisfy the following condition:

$$\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(m)} = 0 \tag{10.106}$$

If in each search direction the right step size is taken, the solution will be found after N steps. Step $n + 1$ is chosen such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)} \tag{10.107}$$

Subtracting $\boldsymbol{\phi}$ from both sides of the above equation, an equation for the error is obtained as

$$\mathbf{e}^{(n+1)} = \mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)} \tag{10.108}$$

Combining Eq. (10.100) with Eq. (10.108), an equation for the residual is found as

$$\begin{aligned}\mathbf{r}^{(n+1)} &= -\mathbf{A}\mathbf{e}^{(n+1)} \\ &= -\mathbf{A}\left(\mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}\right) \\ &= \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}\end{aligned} \tag{10.109}$$

Equation (10.109) shows that each new residual $\mathbf{r}^{(n+1)}$ is just a linear combination of the previous residual and $\mathbf{Ad}^{(n)}$.

It is further required that $\mathbf{e}^{(n+1)}$ be **A**-orthogonal to $\mathbf{d}^{(n)}$. This new condition is equivalent to finding the minimum point along the search direction $\mathbf{d}^{(n)}$. Using this **A**-orthogonality condition between $\mathbf{e}^{(n+1)}$ and $\mathbf{d}^{(n)}$ along with Eq. (10.108) an expression for $\alpha^{(n)}$ can be derived as

$$\begin{aligned}\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{Ae}^{(n+1)} = 0 \Rightarrow \left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\left(\mathbf{e}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}\right)\\ = 0 \Rightarrow \alpha^{(n)} = \frac{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{Ad}^{(n)}}\end{aligned} \tag{10.110}$$

The above requirement also implies that

$$\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{Ae}^{(n+1)} = 0 \Rightarrow \left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)} = 0 \tag{10.111}$$

If the search directions are known then $\alpha^{(n)}$ can be calculated.

To derive the search direction, it is assumed to be governed by an equation of the form

$$\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n)}\mathbf{d}^{(n)} \tag{10.112}$$

The **A**-orthogonality requirement of the **d** vectors implies that

$$\left(\mathbf{d}^{(n+1)}\right)^{\mathrm{T}}\mathbf{Ad}^{(n)} = 0 \tag{10.113}$$

Substituting the value of $\mathbf{d}^{(n+1)}$ from Eq. (10.112) in Eq. (10.113) yields

$$\beta^{(n)} = -\frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{Ad}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{Ad}^{(n)}} \tag{10.114}$$

From Eq. (10.109) an expression for $\mathbf{Ad}^{(n)}$ is obtained as

$$\mathbf{Ad}^{(n)} = -\frac{1}{\alpha^{(n)}}\left(\mathbf{r}^{(n+1)} - \mathbf{r}^{(n)}\right) \tag{10.115}$$

Combining Eqs. (10.110), (10.114), and (10.115) leads to

$$\begin{aligned}\beta^{(n)} &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\left(\mathbf{r}^{(n+1)} - \mathbf{r}^{(n)}\right)}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \\ &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)} - \underbrace{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}_{=0}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \\ &= \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}\end{aligned} \tag{10.116}$$

The denominator of the above equation can be further expressed as

$$\begin{aligned}\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} &= \left(\mathbf{r}^{(n)} + \beta^{(n-1)}\mathbf{d}^{(n-1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} \\ &= \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)} + \beta^{(n-1)}\underbrace{\left(\mathbf{d}^{(n-1)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}}_{=0} \\ &= \left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}\end{aligned} \tag{10.117}$$

Using Eqs. (10.116) and (10.117), the final expression for $\beta^{(n)}$ is obtained as

$$\beta^{(n)} = \frac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{r}^{(n)}} \tag{10.118}$$

The Conjugate Gradient algorithm becomes

```
$\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose residual as starting direction)
iterate starting at $(n)$ until convergence
    $\alpha^{(n)} = \frac{\mathbf{d}^{(n)T}\mathbf{r}^{(n)}}{\mathbf{d}^{(n)T}\mathbf{A}\mathbf{d}^{(n)}}$ (Choose factor in $\mathbf{d}$ direction)
    $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)
    $\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new residual)
    $\beta^{(n)} = \frac{\mathbf{r}^{(n+1)T}\mathbf{r}^{(n+1)}}{\mathbf{r}^{(n)T}\mathbf{r}^{(n)}}$ (Calculate coefficient to conjugate residual)
    $\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n)}\mathbf{d}^{(n)}$ (obtain new conjugated search direction)
```

The convergence rate of the CG method may be increased by preconditioning. This can be done by multiplying the original system of equations by the inverse of the preconditioned matrix $\mathbf{P}^{-1}$, where $\mathbf{P}$ is a symmetric positive-definite matrix, to yield Eq. (10.79). The problem is that $\mathbf{P}^{-1}\mathbf{A}$ is not necessarily symmetric even if $\mathbf{P}$ and $\mathbf{A}$ are symmetric. To circumvent this problem the Cholesky decomposition is used to write $\mathbf{P}$ in the form

$$\mathbf{P} = \mathbf{L}\mathbf{L}^{\mathrm{T}} \tag{10.119}$$

To guarantee symmetry, the system of equations is written as

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-\mathrm{T}}\mathbf{L}^{\mathrm{T}}\boldsymbol{\phi} = \mathbf{L}^{-1}\mathbf{b} \tag{10.120}$$

where $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-\mathrm{T}}$ is symmetric and positive-definite. The CG method can be used to solve for $\mathbf{L}^{\mathrm{T}}\boldsymbol{\phi}$, from which $\boldsymbol{\phi}$ is found. However, by variable substitutions, $\mathbf{L}$ can be eliminated from the equations without disturbing symmetry or affecting the validity of the method. Performing this step and adopting the terminology used with the CG method, the various steps in the preconditioned CG method are obtained.

The preconditioned CG method can be summarized as follows:

```
$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ and $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$ (choose starting direction)
iterate starting at $(n)$ until convergence
$\alpha^{(n)} = \dfrac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}$ (Choose factor in $\mathbf{d}$ direction)
$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$ )
$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new residual)
$\beta^{(n+1)} = \dfrac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}$ (Calculate coefficient to conjugate residual)
$\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new conjugated search direction)
```

Many pre-conditioners have been developed with a wide spectrum of sophistication varying from a simple diagonal matrix whose elements are the diagonal elements of the original matrix $\mathbf{A}$ (Jacobi pre-conditioner) to more involved ones using incomplete Cholesky factorization. Nonetheless, the CG method should always be used with a pre-conditioner when solving large systems of equations.

10.3.14 The Bi-conjugate Gradient Method (BiCG) and Preconditioned BICG

The matrix of coefficients resulting from the discretization of the diffusion equation presented in Chap. 8 and some other equations like the incompressible pressure or pressure correction equation that will be presented in Chap. 15 are symmetrical leading to symmetrical systems that can be solved using the CG method discussed above. However, the matrix **A** obtained from the discretization of the general conservation equation arising in CFD applications is unsymmetrical yielding an unsymmetrical system of equations. To be able to solve this system using the CG method, it should be transformed into a symmetrical one [20]. One way to do that is to rewrite Eq. (10.1) as

$$\begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\phi}} \\ \boldsymbol{\phi} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \tag{10.121}$$

where $\widehat{\boldsymbol{\phi}}$ is a dummy variable added in order to convert the original unsymmetrical system into a symmetrical one amenable to a solution by the CG method. When applied to this system, the CG method results in two sequences of CG-like vectors, the ordinary sequence based on the original system with the coefficient matrix **A**, from which $\boldsymbol{\phi}$ is calculated, and the shadow sequence for the unneeded system with the coefficient matrix $\mathbf{A}^{\mathrm{T}}$, from which $\widehat{\boldsymbol{\phi}}$ can be calculated if desired. Because of the two series of vectors the name bi-conjugate gradient (BiCG) is coined to the method. The same terminology used with the CG method is used here. The series of ordinary vectors for the residuals and search directions are denoted by **r** and **d**, respectively, and their shadow equivalent forms by $\widehat{\mathbf{r}}$ and $\widehat{\mathbf{d}}$. The bi-orthogonality of the residuals is guaranteed by forming them such that

$$\left(\widehat{\mathbf{r}}^{(m)}\right)^{\mathrm{T}} \mathbf{r}^{(n)} = \left(\widehat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(m)} = 0 \quad m < n \tag{10.122}$$

and the bi-conjugacy of the search directions is fulfilled by requiring that

$$\left(\widehat{\mathbf{d}}^{(n)}\right)^{T} \mathbf{A}\mathbf{d}^{(m)} = \left(\mathbf{d}^{(n)}\right)^{T} \mathbf{A}^{T} \widehat{\mathbf{d}}^{(m)} = 0 \quad m < n \tag{10.123}$$

Moreover, the sequences of residuals and search directions are constructed such that the ordinary form of one is orthogonal to the shadow form of the other. Mathematically this is written as

$$\left(\widehat{\mathbf{r}}^{(n)}\right)^{T} \mathbf{d}^{(m)} = \left(\mathbf{r}^{(n)}\right)^{T} \widehat{\mathbf{d}}^{(m)} \quad m < n \tag{10.124}$$

Several variants of the method, which has irregular convergence with the possibility of breaking down, have been developed and the algorithm described next is due to Lanczos [21, 22].

The BiCG algorithm of Lanczos can be summarized as follows:

```
$\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \hat{\mathbf{d}}^{(0)} = \hat{\mathbf{r}}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$ (choose starting directions)
iterate starting at $(n)$ until convergence
    $\alpha^{(n)} = \dfrac{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(n)}}{\left(\hat{\mathbf{d}}^{(n)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{d}^{(n)}}$ (Choose factor in $\mathbf{d}$ direction)
    $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)
    $\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new $\mathbf{r}$ residual)
    $\hat{\mathbf{r}}^{(n+1)} = \hat{\mathbf{r}}^{(n)} - \alpha^{(n)}\mathbf{A}^{\mathrm{T}}\hat{\mathbf{d}}^{(n)}$ (calculate new $\hat{\mathbf{r}}$ residual)
    $\beta^{(n+1)} = \dfrac{\left(\hat{\mathbf{r}}^{(n+1)}\right)^{\mathrm{T}} \mathbf{r}^{(n+1)}}{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{r}^{(n)}}$ (Calculate coefficient to conjugate residual)
    $\mathbf{d}^{(n+1)} = \mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new search $\mathbf{d}$ direction)
    $\hat{\mathbf{d}}^{(n+1)} = \hat{\mathbf{r}}^{(n+1)} + \beta^{(n+1)}\hat{\mathbf{d}}^{(n)}$ (obtain new search $\hat{\mathbf{d}}$ direction)
```

The BiCG method necessitates a multiplication with the coefficient matrix and with its transpose at each iteration resulting in almost double the computational effort required by the CG method per iteration.

Preconditioning may also be used with the BiCG method. For the same terminology as for the preconditioned CG method, a robust variant of the method developed by Fletcher [23] is summarized next.

The preconditioned algorithm for the BiCG of Fletcher is as follows (**P** representing the preconditioning matrix):

```
$\mathbf{r}^{(0)} = \hat{\mathbf{r}}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$, $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$, $\hat{\mathbf{d}}^{(0)} = \mathbf{P}^{-\mathrm{T}}\hat{\mathbf{r}}^{(0)}$ (choose starting directions)
iterate starting at $(n)$ until convergence
    $\alpha^{(n)} = \dfrac{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\hat{\mathbf{d}}^{(n)}\right)^{\mathrm{T}} \mathbf{A}\mathbf{d}^{(n)}}$ (Choose factor in $\mathbf{d}$ direction)
    $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)
    $\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$ (calculate new $\mathbf{r}$ residual)
    $\hat{\mathbf{r}}^{(n+1)} = \hat{\mathbf{r}}^{(n)} - \alpha^{(n)}\mathbf{A}^{\mathrm{T}}\hat{\mathbf{d}}^{(n)}$ (calculate new $\hat{\mathbf{r}}$ residual)
    $\beta^{(n+1)} = \dfrac{\left(\hat{\mathbf{r}}^{(n+1)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\hat{\mathbf{r}}^{(n)}\right)^{\mathrm{T}} \mathbf{P}^{-1}\mathbf{r}^{(n)}}$ (Calculate coefficient to conjugate residual)
    $\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$ (obtain new search $\mathbf{d}$ direction)
    $\hat{\mathbf{d}}^{(n+1)} = \mathbf{P}^{-\mathrm{T}}\hat{\mathbf{r}}^{(n+1)} + \beta^{(n+1)}\hat{\mathbf{d}}^{(n)}$ (obtain new search $\hat{\mathbf{d}}$ direction)
```

Other variants of the BiCG method that are more stable and robust have been reported such as the conjugate gradient squared (CGS) method of Sonneveld [24], the bi-conjugate gradient stabilized (Bi-CGSTAB) method of Van Der Vorst [25] and the generalized minimal residual method GMRES [13, 26–29]. These methods are useful for solving large systems of equations arising in CFD applications as they are applicable to non-symmetrical matrices and to both structured and unstructured grids.

10.4 The Multigrid Approach

The rate of convergence of iterative methods drastically deteriorates as the size of the algebraic system increases, with the drop in convergence rate even observed in medium to large systems after the initial errors have been eliminated. This has constituted a severe limitation for iterative solvers. Luckily it was very quickly found that the combination of multigrid and iterative methods can practically remedy this weakness.

Developments in multigrid methods started with the work of Fedorenko [30] (Geometric Multigrid), Poussin [31] (Algebraic Multigrid), and Settari and Azziz [32], and gained more interest with the theoretical work of Brandt [33]. While high-frequency or oscillatory errors are easily eliminated with standard iterative solvers (Jacobi, Gauss-Seidel, ILU), these solution techniques cannot easily remove the smooth or low frequency error components [34]. Because of that these solution

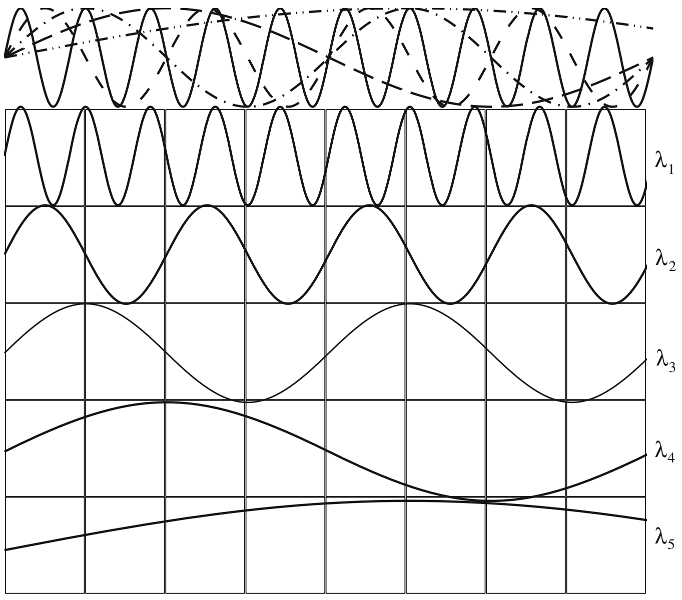


Fig. 10.4 Schematic of different error modes in a one dimensional grid

methods are denoted by smoothers in the context of multigrid methods. An illustration of error frequency is shown in Fig. 10.4 where the variations of error frequency modes for a one dimensional problem are plotted.

The error modes shown in Fig. 10.4 vary from high frequency of short wavelength λ_1 to low frequency of long wavelength λ_5 and are plotted collectively on the top of the figure. The one dimensional domain is discretized using the one dimensional grid shown and the various modes are separately plotted over the same grid. As can be seen, the high frequency error appears oscillatory over an element and is easily sensed by the iterative method. As the frequency of the error decreases or as the wavelength (λ) increases, the error becomes increasingly smoother over the grid as only a small portion of the wavelength lies within any cell. This gets worse as the grid is further refined, leading to a higher number of equations and explaining the degradation in the rate of convergence as the size of the system increases.

Multigrid methods improve the efficiency of iterative solvers by ensuring that the resulting low frequency errors that arise from the application of a smoother at any one grid level are transformed into higher frequency errors at a coarser grid level. By using a hierarchy of coarse grids (Fig. 10.5), multigrid methods are able to overcome the convergence degradation.

Generally the coarse mesh can be formed using either the topology and geometry of the finer mesh, this is akin to generating a new mesh for each coarse level on top of the finer level mesh or by direct agglomeration of the finer mesh elements [35–40]; this approach is also known as the Algebraic MultiGrid Method (AMG). In the AMG no geometric information is directly needed or used, and the agglomeration process is purely algebraic, with the equations at each coarse level reconstructed from those of the finer level, again through the agglomeration process. This approach can be used to build highly efficient and robust linear solvers

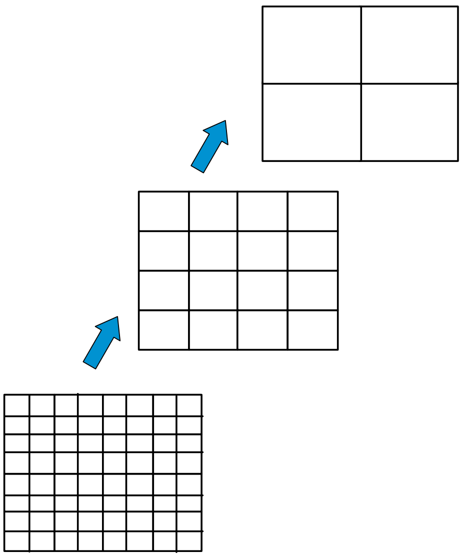


Fig. 10.5 A schematic of the hierarchy of grid systems used with the multi grid approach

for both highly anisotropic grids and/or problems with large changes in the coefficients of their equations.

In either approach, a multigrid cycling procedure is used to guide the traversal of the various grid hierarchies. Each traversal from a fine grid to a coarse one involves: (i) a restriction procedure, (ii) the setup or update of the system of equations for the coarse grid level, and (iii) the application of a number of smoother iterations. A traversal from a coarse grid to a finer one requires: (i) a prolongation procedure, (ii) the correction of the field values at the finer level, and (iii) the application of a number of smoother iterations on the equations constructed during restriction. The various steps needed are detailed next.

10.4.1 Element Agglomeration/Coarsening

The first step in the solution process is to generate the coarse/fine grid levels by an agglomeration/coarsening algorithm. Three different approaches can be adopted for that purpose. In the first approach, the coarse mesh is initially generated and the fine levels are obtained by refinement [41, 42]. This facilitates the definitions of the coarse-fine grid relations and is attractive in an adaptive grid setup [41–43]. A major drawback however, is the dependence of the fine grid distribution on the coarse grid. In the second method, non-nested grids are used [44] rendering the transfer of information between grid levels very expensive. In addition, both approaches do not allow good resolution of complex domains. In the third approach, recommended here, the process starts with the generation of the finest mesh that will be used in solving the problem. Then coarse grid levels are developed through agglomeration of the fine-grid elements [45, 46], as shown in Fig. 10.6, with the agglomeration process based either on the elements geometry or on a criterion to be satisfied by the coefficients of neighboring elements. The discussions to follow are pertinent to the third approach.

Coarse grid levels are generated by fusing fine grid elements through an agglomeration algorithm. For each coarse grid level, the algorithm is repeatedly applied until all grid cells of the finer level become associated with coarse grid cells.

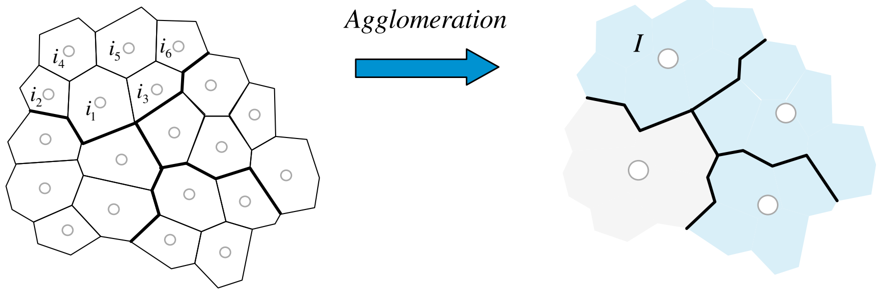


Fig. 10.6 Agglomeration of a fine grid level to form a coarse grid level

During this heuristic agglomeration process, fine grid points are individually visited. A cell is selected as the seed element into which a certain number of neighboring elements satisfying the set criteria are fused to form a coarse element. The maximum number of fine elements to be fused into a coarse element is decided a priori. If the chosen seed element fails to form a coarse element, it is added to the least populated coarse element among its neighbors.

An efficient agglomeration algorithm is the directional agglomeration (DA) algorithm developed by Mavriplis [47]. In the DA, agglomeration is performed by starting with a seed element and merging with it the neighboring fine grid elements based on the strength of their geometric connectivity. The procedure needs to be performed only once at the start of the solution.

10.4.2 The Restriction Step and Coarse Level Coefficients

The solution starts at the fine grid level. After performing few iterations, the error is transferred or restricted to a coarser grid level and the solution is found at that level. Then after performing few iterations at that level the error is restricted again to a higher level and the sequence of events repeated until the highest or coarsest grid level is reached. Let (k) denotes some level at which the solution has been found by solving the following system of equations in correction form:

$$\mathbf{A}^{(k)}\mathbf{e}^{(k)} = \mathbf{r}^{(k)} \tag{10.125}$$

The next coarser level is $(k+1)$ to which the error will be restricted. Let G_I represents the set of cells i on the fine grid level (k) that are agglomerated to form cell I of the coarse grid level $(k+1)$. Then, the system to be solved on the coarse grid at level $(k+1)$ is

$$\mathbf{A}^{(k+1)}\mathbf{e}^{(k+1)} = \mathbf{r}^{(k+1)} \tag{10.126}$$

with the residuals on the RHS of Eq. (10.126) computed as

$$\mathbf{r}^{(k+1)} = I_k^{k+1}\mathbf{r}^{(k)} \tag{10.127}$$

where I_k^{k+1} is the restriction operator (i.e., the interpolation matrix) from the fine grid to the coarse grid as defined by the agglomeration process. In AMG the restriction operator is defined in a linear manner to yield a summation of the fine grid residuals as

$$\mathbf{r}_I^{(k+1)} = \sum_{i \in G_I} \mathbf{r}_i^{(k)} \tag{10.128}$$

Moreover, the coefficients of the coarse element are constructed by adding the appropriate coefficients of the constituting fine elements. Recalling that a linear equation after discretization has the form

$$a_C\phi_C + \sum_{F=NB(C)} a_F\phi_F = b_C \tag{10.129}$$

which, for the current purpose, is written for the fine grid level in a more suitable form as

$$a_i^{(k)}\phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)}\phi_j^{(k)} = b_i^{(k)} \tag{10.130}$$

where $NB(i)$ refers to the neighbors of element i. Initially Eq. (10.130) is not satisfied resulting in the following residual:

$$r_i^{(k)} = b_i^{(k)} - \left(a_i^{(k)}\phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)}\phi_j^{(k)}\right) \tag{10.131}$$

Denoting by $\phi_I^{(k+1)}$ the solution on the coarse mesh element I that is parent to the fine mesh element i, the correction on the fine mesh from the coarse mesh can be written as

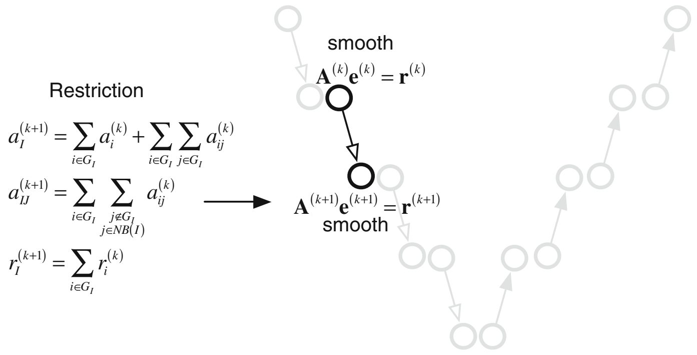
$$\phi_i'^{(k)} = \phi_I^{(k+1)} - \phi_i^{(k)} \tag{10.132}$$

It is desired for the correction to result in zero residuals over the coarse mesh element I. These new residuals denoted by $\tilde{r}_i^{(k)}$ are calculated as

$$\tilde{r}_i^{(k)} = b_i^{(k)} - \left(a_i^{(k)}\left(\phi_i^{(k)} + \phi_i'^{(k)}\right) + \sum_{j=NB(i)} a_{ij}^{(k)}\left(\phi_j^{(k)} + \phi_j'^{(k)}\right)\right) \tag{10.133}$$

or equivalently as

$$\begin{aligned}\tilde{r}_i^{(k)} &= \underbrace{b_i^{(k)} - \left(a_i^{(k)}\phi_i^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)}\phi_j^{(k)}\right)}_{r_i^{(k)}} - \left(a_i^{(k)}\phi_i'^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)}\phi_j'^{(k)}\right)\\ &= r_i^{(k)} - \left(a_i^{(k)}\phi_i'^{(k)} + \sum_{j=NB(i)} a_{ij}^{(k)}\phi_j'^{(k)}\right)\end{aligned} \tag{10.134}$$



Restriction
$a_I^{(k+1)} = \sum_{i\in G_I} a_i^{(k)} + \sum_{i\in G_I}\sum_{j\in G_I} a_{ij}^{(k)}$
$a_{IJ}^{(k+1)} = \sum_{i\in G_I} \sum_{\substack{j\notin G_I\\ j\in NB(I)}} a_{ij}^{(k)}$
$r_I^{(k+1)} = \sum_{i\in G_I} r_i^{(k)}$
smooth
$\mathbf{A}^{(k)}\mathbf{e}^{(k)} = \mathbf{r}^{(k)}$
$\mathbf{A}^{(k+1)}\mathbf{e}^{(k+1)} = \mathbf{r}^{(k+1)}$
smooth

Fig. 10.7 The restriction step and assembly of coarse grid level coefficients

Enforcing the residual sum in I to zero, i.e.,

$$\sum_{i\in G_I} \tilde{\boldsymbol{r}}_i^{(k)} = 0 \tag{10.135}$$

and substituting Eq. (10.134) in Eq. (10.135) yields

$$0 = \sum_{i\in G_I} \boldsymbol{r}_i^{(k)} - \left(\sum_{i\in G_I} a_i^{(k)} \phi_i'^{(k)} + \sum_{i\in G_I} \sum_{j=NB(i)} a_{ij}^{(k)} \phi_j'^{(k)} \right) \tag{10.136}$$

Rewriting Eq. (10.136) using coarse mesh numbering, the coarse mesh correction equation becomes

$$a_I^{(k+1)} \phi_I'^{(k+1)} + \sum_{J=NB(I)} a_{IJ}^{(k+1)} \phi_J'^{(k+1)} = r_I^{(k+1)} \tag{10.137}$$

where $a_I^{(k+1)}, a_{IJ}^{(k+1)}$, and $r_I^{(k+1)}$ are derived directly from fine grid coefficients as

$$\begin{aligned} a_I^{(k+1)} &= \sum_{i\in G_I} a_i^{(k)} + \sum_{i\in G_I}\sum_{j\in G_I} a_{ij}^{(k)} \\ a_{IJ}^{(k+1)} &= \sum_{i\in G_I} \sum_{\substack{j\notin G_I \\ j\in NB(I)}} a_{ij}^{(k)} \\ r_I^{(k+1)} &= \sum_{i\in G_I} r_i^{(k)} \end{aligned} \tag{10.138}$$

This is illustrated in Fig. 10.7.

10.4.3 The Prolongation Step and Fine Grid Level Corrections

The prolongation operator is used to transfer the correction from a coarse to a fine grid level. Many options may be used. One possibility, depicted in Fig. 10.8, is a zero order prolongation operator that yields the same value of the error on the fine grid, i.e., the error at a coarse grid cell will be inherited by all the children of this cell on the fine grid level.

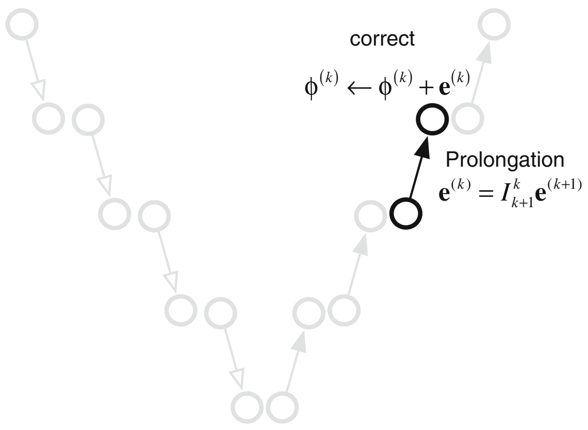

correct
$\phi^{(k)} \leftarrow \phi^{(k)} + \mathbf{e}^{(k)}$
Prolongation
$\mathbf{e}^{(k)} = I_{k+1}^{k}\mathbf{e}^{(k+1)}$

Fig. 10.8 The prolongation step and fine grid level corrections

The correction is basically obtained from the solution of the system of equations at the coarse grid. The interpolation or prolongation to the fine grid level is denote as

$$\mathbf{e}^{(k)} = I_{k+1}^{k}\mathbf{e}^{(k+1)} \tag{10.139}$$

where I_{k+1}^{k} is an interpolation matrix from the coarse grid to the fine grid. Finally, the fine grid solution is corrected as

$$\boldsymbol{\phi}^{(k)} \leftarrow \boldsymbol{\phi}^{(k)} + \mathbf{e}^{(k)} \tag{10.140}$$

The number of grid levels used depends on the size of the grid. For a larger number of grid levels, the procedure is the same as sketched in Fig. 10.8.

10.4.4 Traversal Strategies and Algebraic Multigrid Cycles

Traversal strategies refer to the way by which coarse grids are visited during the solution process, which are also known as multigrid cycles [48]. The usual cycles used in the AMG method are the V cycle, the W cycle, and the F cycle [35, 36, 49, 50] displayed in Fig. 10.9.

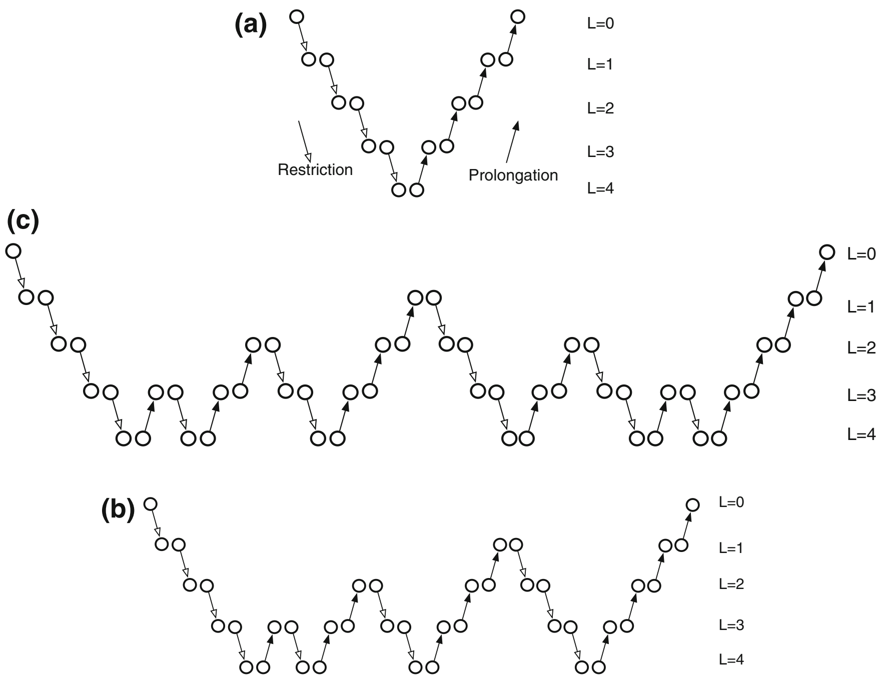


Fig. 10.9 **a** V, **b** W, and **c** F multigrid cycles

The simplest AMG cycle, illustrated in Fig. 10.9a, is the V cycle [49, 50] and consists of visiting each of the grid levels only once. The usual practice is to perform few iterative sweeps in the restriction phase and then to inject the residual to a coarser grid until reaching the coarsest level. For very stiff systems, the V cycle may not be sufficient for accelerating the solution and therefore, more iterations on the coarse level are required. The W cycle is based on applying smaller V cycles on each visited coarse grid level. In this manner, the W cycle (Fig. 10.9b) consists of nested coarse and fine grid level sweeps with the complexity increasing as the number of AMG levels increases. The F cycle is a variant of the W cycle and can be thought of as splitting the W cycle in half as shown in Fig. 10.9c. The F cycle requires less coarse level sweeps than the W cycle but more sweeps than the V cycle. Therefore, it lies in between the V and W cycling strategies.

10.5 Computational Pointers

10.5.1 uFVM

In uFVM, two linear algebraic solvers are implemented. The successive over-relaxation method (SOR) and the ILU(0) method. The implementation of these methods follows the procedures described earlier. The SOR is located in the file "cfdSORSolver.m" while the ILU(0) implementation can be found in "cfdILUSolver.m".

10.5.2 *OpenFOAM®*

The organizational structure of iterative linear algebraic solvers in OpenFOAM® [51] follows the usual approach. It starts by defining the base classes from which each type of algebraic matrix solvers is established. These algebraic solvers are grouped under three main categories denoted by solvers, preconditioners, and smoothers. Smoothers and preconditioners are differentiated by relating to smoothers the fixed point relation and embedding them within the preconditioners framework. Recalling Eq. (10.81), the preconditioners classes implement the product $\mathbf{P}^{-1}\mathbf{r}$ while the smoothers classes advance the solution. Moreover, the solvers category collects the necessary information related to the implementation of the conjugate gradient and multigrid algorithms.

The source codes of the linear algebraic solvers reside within the lduMatrix folder located at "…/src/OpenFOAM/matrices/lduMatrix/" in the following three subfolders:

- *solvers*
- *preconditioners*
- *smoothers*

The name of each subfolder reflects its functionality. The folder *solvers* contains the main codes of the iterative solvers implemented in OpenFOAM®, which are

- **diagonalSolver**: a diagonal solver for both symmetric and asymmetric problems.
- **GAMG**: a geometric agglomerated algebraic multigrid solver (also named Generalized geometric- algebraic multi-grid in the manual).
- **ICC**: an incomplete Cholesky preconditioned conjugate gradient solver.
- **PBiCG**: a preconditioned bi-conjugate gradient solver for asymmetric matrices.
- **PCG**: a preconditioned conjugate gradient solver for symmetric matrices.
- **smoothSolver**: an iterative solver using smoother for symmetric and asymmetric matrices based on preconditioners.

The folder *preconditioners* contains various implementations of the diagonal ILU denoted by

- **diagonalPreconditioner**: a diagonal preconditioner.
- **DICPreconditioner, DILUPreconditioner**: a diagonal Incomplete Cholesky preconditioner for symmetric and asymmetric matrices respectively.
- **FDICPreconditioner**: a faster version of the DICPreconditioners diagonal-based incomplete Cholesky preconditioner for symmetric matrices in which the reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored.

- **GAMGPreconditioner**: a geometric agglomerated algebraic multigrid preconditioner. It uses a mutigrid cycle as preconditioner to execute the second part of Eq. (10.81).
- **noPreconditioner**: a null preconditioner for both symmetric and asymmetric matrices.

Finally the *smoothers* folder contains the following:

- **DIC**, **DILU**: a diagonal-based incomplete Cholesky smoother for symmetric and asymmetric matrices.
- **DICGaussSeidel**, **DILUGaussSeidel**: a combined DIC, DILU/Gauss-Seidel smoother for symmetric and asymmetric matrices in which DIC, DILU smoothing is followed by Gauss-Seidel to ensure that any "spikes" created by the DIC, DILU sweeps are smoothed out.
- **DILU**: a diagonal-based incomplete LU smoother for asymmetric matrices.
- **GaussSeidel**: The Gauss-Seidel method for both symmetric and asymmetric matrices.

Furthermore OpenFOAM® defines inside the lduMatrix class three additional base classes that wrap the three corresponding categories. Thus the lduMatrix.H file reads (Listing 10.1)

```
class lduMatrix
{
    // private data

        //- LDU mesh reference
        const lduMesh& lduMesh_;

        //- Coefficients (not including interfaces)
        scalarField *lowerPtr_, *diagPtr_, *upperPtr_;

…
public:

    //- Abstract base-class for lduMatrix solvers
    class solver
    {
    protected:
…

    class smoother
    {
    protected:
…
    class preconditioner
    {
    protected:
…
```

Listing 10.1 The three base classes (solver, smoother, and preconditioner) defined with the lduMatrix class

So each smoother, solver, and preconditioner has to be derived from these three base classes. For example, the DILU preconditioner is declared as shown in Listing 10.2,

```
class DILUPreconditioner
:
    public lduMatrix::preconditioner
{
…
```

Listing 10.2 Syntax used to declare the DILU preconditioner

while the Conjugate Gradient solver is declared as shown in Listing 10.3.

```
class PCG
:
    public lduMatrix::solver
```

Listing 10.3 Syntax used to declare the PCG solver

In either case the preconditioner or the solver is evidently derived from the base class defined under the lduMatrix class.

Having clarified the basic concepts and organizational structure of algebraic solvers in OpenFOAM®, an example investigating the details of implementing the preconditioned CG method is now provided. The files are located in the directory "**$FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/PCG**".

The class derived from the lduMatrix::solver class, as shown in Listing 10.3, defines the main member function "solve" using the script in Listing 10.4.

```
    // Member Functions

        //- Solve the matrix with this solver
        virtual solverPerformance solve
        (
            scalarField& psi,
            const scalarField& source,
            const direction cmpt=0
        ) const;
```

Listing 10.4 Script used to define the member function "solve"

The function "solve" implements the solution algorithm of the chosen linear algebraic solver in file "PCG.C". Recalling the preconditioned conjugate gradient algorithm, its sequence of events are given by

1. Calculate $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(0)}$
2. Calculate $\mathbf{d}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$
3. Calculate $\alpha^{(n)} = \dfrac{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}{\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}}$
4. Calculate $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\mathbf{d}^{(n)}$
5. Calculate $\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} - \alpha^{(n)}\mathbf{A}\mathbf{d}^{(n)}$
6. If solution has converged stop
7. Calculate $\beta^{(n+1)} = \dfrac{\left(\mathbf{r}^{(n+1)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n+1)}}{\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}}$
8. Calculate $\mathbf{d}^{(n+1)} = \mathbf{P}^{-1}\mathbf{r}^{(n+1)} + \beta^{(n+1)}\mathbf{d}^{(n)}$
9. Go to step 3

The algorithm is directly implemented in "PCG.C" following the same procedure as described next.

In step 1, depicted in Listing 10.5, the residual is evaluated and stored in the rA variable while the wA variable stores the matrix-solution product $\mathbf{A}\boldsymbol{\phi}^{(0)}$.

```
// --- Calculate A.psi
matrix_.Amul(wA, psi, interfaceBouCoeffs_, interfaces_, cmpt);
// --- Calculate initial residual field
scalarField rA(source - wA);
```

Listing 10.5 Script used to calculate the residuals

Step 2 involves preconditioning. Thus, first the type of preconditioner used is defined with the object preconPtr as (Listing 10.6).

```
// --- Select and construct the preconditioner
autoPtr<lduMatrix::preconditioner> preconPtr =
lduMatrix::preconditioner::New
(
    *this,
    controlDict_
);
```

Listing 10.6 Defining the type of preconditioner used

The constructor used is a generic one based on the base class and the "New" constructor. The preconditioner type is chosen from the dictionary at run time. Then the preconditioning operation, $\mathbf{P}^{-1}\mathbf{r}^{(n)}$, is applied to the residual rA (according to the equation in step 2 of the algorithm). The result is stored in the same variable wA to reduce memory usage and then used in the evaluation of $\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}$ by simply performing the scalar product of the two vectors wA and rA using the gSumProd function, as shown in Listing 10.7. Moreover the old value of wArA from the previous iteration $(n-1)$ is stored in the variable wArAold.

```
            wArAold = wArA;

            // --- Precondition residual
            preconPtr->precondition(wA, rA, cmpt);

            // --- Update search directions:
            wArA = gSumProd(wA, rA, matrix().mesh().comm());
```

Listing 10.7 Syntax used to calculate $\left(\mathbf{r}^{(n)}\right)^{\mathrm{T}}\mathbf{P}^{-1}\mathbf{r}^{(n)}$ and to store its old value

Following the preconditioned conjugate gradient algorithm, steps 3, 4, and 5 are performed as displayed in Listing 10.8.

```
            // --- Update preconditioned residual
                matrix_.Amul(wA, pA, interfaceBouCoeffs_, interfaces_,
cmpt);

            scalar wApA = gSumProd(wA, pA, matrix().mesh().comm());


            // --- Update solution and residual:

            scalar alpha = wArA/wApA;

            for (register label cell=0; cell<nCells; cell++)
            {
                psiPtr[cell] += alpha*pAPtr[cell];
                rAPtr[cell] -= alpha*wAPtr[cell];
            }
```

Listing 10.8 Script used to calculate α and to update the values of the dependent variable and the residuals

Now the variable wA stores the product $\mathbf{A}\mathbf{d}^{(n)}$ while pA represents the $\mathbf{d}^{(n)}$ vector. Again the product $\left(\mathbf{d}^{(n)}\right)^{\mathrm{T}}\mathbf{A}\mathbf{d}^{(n)}$ is performed with the gSumProd function and stored in the wApA variable. Once alpha is evaluated, the update of residuals and solution of steps 4 and 5 is performed in the for loop with the variables psiPtr and rAPtr shown in Listing 10.8 representing $\boldsymbol{\phi}$ and $\mathbf{r}$, respectively.

The iteration in the algorithm is completed by executing steps 7 and 8 using the script in Listing 10.9.

```
scalar beta = wArA/wArAold;

for (register label cell=0; cell<nCells; cell++)
{
    pAPtr[cell] = wAPtr[cell] + beta*pAPtr[cell];
}
```

Listing 10.9 Script used for calculating new values for β and **d** to be used in the next iteration

For practical use in test cases, the definitions of the linear solver are made inside the *system* directory in the *fvSolution* under the syntax *solvers* {} as shown in Listing 10.10.

```
solvers
{
    T
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
}
```

Listing 10.10 Definition of the linear solver

The meaning of the various entries in Listing 10.10 are

- *solver*: defines the type of solver (in this case PCG is the preconditioned conjugate gradient for symmetric matrices), with the various options being as follows:
  - "PCG": preconditioned conjugate gradient (for symmetric matrices only).
  - "PBiCG": preconditioned biconjugate gradient (for asymmetric matrices only).
  - "smoothSolver": solver used only as a smoother to reduce residuals.
  - "GAMG": generalised geometric algebraic multigrid. It should be used for the pressure equation on large grids.

- preconditioner: defines the type of the preconditioner to be used. The two available options are
  - "DIC": diagonal incomplete-cholesky preconditioner for symmetric matrices.
  - "DILU": diagonal incomplete-lower-upper preconditioner for asymmetric matrices.
- tolerance: the maximum allowable value of the absolute residual for the linear solver to stop iterating.
- relTol: the ratio between the initial residual and the actual residual for the linear solver to stop iterating.

Other examples are shown in Listing 10.11.

```
solvers
{
    T
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }
}
solvers
{
    T
    {
        solver           smoothSolver;
        smoother         GaussSeidel;
        tolerance        1e-8;
        relTol           0.1;
        nSweeps          1;
    }
    T
    {
        solver           GAMG;
        tolerance        1e-7;
        relTol           0.01;
        smoother         GaussSeidel;
        nPreSweeps       0;
        nPostSweeps      2;
        cacheAgglomeration on;
        agglomerator     faceAreaPair;
        nCellsInCoarsestLevel 10;
        mergeLevels      1;
    }
}
```

Listing 10.11 Examples of defining the linear solver

10.6 Closure

The chapter introduced the direct and iterative approaches for solving algebraic systems of equations. In each category a number of methods were described. The algebraic multigrid technique was also discussed. The next chapter will proceed with the discretization of the conservation equation and will detail the discretization of the convection term.

10.7 Exercises

Exercise 1

In each of the following cases obtain the LU factorization of matrix **A** and use it to solve the system of equation **Ax** = **b** by performing the backward and forward substitution, i.e., **Ly** = **b**, **Ux** = **y**:

$$(a) \qquad \mathbf{A} = \begin{pmatrix} 13.0 & 5.0 & 6.0 & 7.0 & 5.0 \\ 2.0 & 12.0 & 6.0 & 3.0 & 4.0 \\ 4.0 & 2.0 & 15.0 & 4.0 & 5.0 \\ 3.0 & 3.0 & 5.0 & 9.0 & 5.0 \\ 4.0 & 6.0 & 0 & 5.0 & 13.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 3.0 \\ 12.0 \\ 13.0 \\ 21.0 \\ 13.0 \end{pmatrix}$$

$$(b) \qquad \mathbf{A} = \begin{pmatrix} 14.0 & 1.0 & 6.0 & 7.0 & 3.0 \\ 7.0 & 10.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 1.0 & 10.0 & 7.0 & 6.0 \\ 4.0 & 7.0 & 6.0 & 11.0 & 1.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 14.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 13.0 \\ 13.0 \\ 16.0 \\ 13.0 \\ 1.0 \end{pmatrix}$$

$$(c) \qquad \mathbf{A} = \begin{pmatrix} 12.0 & 6.0 & 1.0 & 5.0 & 4.0 \\ 4.0 & 11.0 & 1.0 & 0 & 0 \\ 3.0 & 6.0 & 14.0 & 1.0 & 6.0 \\ 6.0 & 2.0 & 7.0 & 10.0 & 7.0 \\ 4.0 & 1.0 & 1.0 & 6.0 & 12.0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 13.0 \\ 19.0 \\ 7.0 \\ 22.0 \\ 30.0 \end{pmatrix}$$

Exercise 2

Using the Gauss-Seidel and Jacobi methods and starting with a zero initial guess solve the following systems of equations while adopting as a stopping criteria $\max(\|\mathbf{Ax} - \mathbf{b}\|) < 0.1$:

(*a*) $$\mathbf{A} = \begin{pmatrix} 6 & 1 & 0 & 3 & 3 \\ 0 & 6 & 0 & 4 & 3 \\ 0 & 5 & 6 & 1 & 0 \\ 5 & 5 & 0 & 6 & 0 \\ 2 & 0 & 2 & 4 & 10 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 22 \\ 21 \\ 20 \\ 10 \\ 22 \end{pmatrix}$$

(*b*) $$\mathbf{A} = \begin{pmatrix} 36 & 4 & 5 & 4 & 0 & 2 \\ 0 & 40 & 4 & 0 & 0 & 3 \\ 0 & 0 & 37 & 0 & 2 & 4 \\ 3 & 2 & 0 & 36 & 1 & 5 \\ 3 & 3 & 1 & 0 & 36 & 0 \\ 5 & 0 & 2 & 0 & 2 & 40 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 24 \\ 14 \\ 19 \\ 29 \\ 20 \\ 8 \end{pmatrix}$$

(*c*) $$\mathbf{A} = \begin{pmatrix} 27 & 0 & 0 & 1 \\ 0 & 27 & 1 & 0 \\ 0 & 0 & 26 & 1 \\ 1 & 2 & 5 & 26 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 21 \\ 3 \\ 10 \\ 29 \end{pmatrix}$$

Exercise 3
For the systems of equations in Exercise 2 find the preconditioned matrix **P** for both the Gauss-Seidel and Jacobi methods. Use Eq. (10.81) with a zero initial guess to resolve the systems of equations subject to the same stopping criteria. Compare solutions with those obtained in Exercise 2.

Exercise 4
Perform the ILU(0) factorization of the following M matrices:

(*a*) $$\mathbf{M} = \begin{pmatrix} 6 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 3 & 1 & 3 \\ 0 & 0 & 6 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 1 & 1 & 6 & 0 \\ 0 & 0 & 5 & 0 & 3 & 0 & 6 \end{pmatrix}$$

(*b*) $$\mathbf{M} = \begin{pmatrix} 6 & 0 & 0 & 0 & 0 & 3 & 4 & 0 \\ 3 & 7 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 6 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 \\ 4 & 1 & 0 & 0 & 1 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 6 \end{pmatrix}$$

(c)
$$\mathbf{M}=\begin{pmatrix}6&0&0&2&0\\0&10&0&1&4\\0&1&6&0&4\\3&0&5&6&5\\0&4&0&5&10\end{pmatrix}$$

Exercise 5
Based on the factorizations in Exercise 4 and knowing that the lower and upper parts of the factorization correspond to $\overline{\mathbf{L}}$ and $\overline{\mathbf{U}}$, respectively, find for each of the cases the error matrix defined as $\mathbf{R} = \mathbf{M} - \mathbf{P}$.

Exercise 6
Perform the DILU factorization of the following M matrices:

(a)
$$\mathbf{M}=\begin{pmatrix}9&4&0&0&2&5&0\\3&7&0&0&1&2&1\\0&5&10&0&5&5&4\\0&1&1&6&3&0&0\\4&0&0&1&6&1&1\\0&5&2&0&0&6&3\\4&0&0&0&0&2&8\end{pmatrix}$$

(b)
$$\mathbf{M}=\begin{pmatrix}6&2&2&3&4&3&4&0\\1&6&0&0&1&4&3&0\\1&0&10&0&0&3&4&0\\2&1&4&8&0&0&1&0\\0&0&1&4&8&2&0&0\\1&0&3&0&1&8&0&1\\2&2&1&1&1&0&6&0\\4&0&4&0&0&4&1&6\end{pmatrix}$$

(c)
$$\mathbf{M}=\begin{pmatrix}6&2&3&0&5&0\\1&6&0&3&0&1\\3&3&8&3&4&0\\4&4&1&7&1&0\\3&4&0&0&8&0\\3&0&4&0&0&6\end{pmatrix}$$

Exercise 7
Based on the diagonal factorizations in exercise 6 and using Eq. (10.91), find for each of the cases the error matrix defined as $\mathbf{R} = \mathbf{M} - \mathbf{P}$.

Exercise 8
Solve the following systems of equations using the ILU(0) and DILU methods. Perform three iterations only starting with a zero initial guess.

$$(a)\qquad \mathbf{A}=\begin{pmatrix}1&3&0&0&0&0\\5&5&0&0&0&2\\3&2&7&0&1&2\\4&1&2&7&0&1\\4&2&0&0&4&0\\3&3&2&0&2&10\end{pmatrix},\quad \mathbf{b}=\begin{pmatrix}26\\28\\18\\26\\11\\26\end{pmatrix}$$

$$(b)\qquad \mathbf{A}=\begin{pmatrix}12&3&1&4&4&1\\4&7&1&0&3&0\\4&5&7&0&0&0\\3&2&0&6&2&0\\3&0&2&1&7&1\\0&0&0&0&0&-1\end{pmatrix},\quad \mathbf{b}=\begin{pmatrix}25\\24\\4\\4\\2\\25\end{pmatrix}$$

Exercise 9
Solve the systems of equations in Exercise 8 using the Gauss-Seidel and Jacobi methods (starting with a zero initial guess) and compare the errors after three iterations with the errors obtained in exercise 8 and with the exact solution. Comment on the results.

Exercise 10
Solve the following symmetric systems of equations by performing two iterations of the preconditioned conjugate gradient method (start with a zero field):

$$(a)\qquad \mathbf{A}=\begin{pmatrix}6&0&-1&0&-1\\0&6&1&1&-1\\-1&1&6&0&0\\0&1&0&4&-1\\-1&-1&0&-1&6\end{pmatrix},\quad \mathbf{b}=\begin{pmatrix}7\\9\\27\\23\\1\end{pmatrix}$$

$$(b)\qquad \mathbf{A}=\begin{pmatrix}4&1&0&-1&0\\1&5&2&-1&3\\0&2&5&0&0\\-1&-1&0&6&0\\0&3&0&0&5\end{pmatrix},\quad \mathbf{b}=\begin{pmatrix}8\\2\\12\\11\\30\end{pmatrix}$$

$$(c)\qquad \mathbf{A}=\begin{pmatrix}3&1&-1&2&-1&0\\1&7&1&1&2&-2\\-1&1&7&0&0&0\\2&1&0&6&0&2\\-1&2&0&0&6&-1\\0&-2&0&2&-1&7\end{pmatrix},\quad \mathbf{b}=\begin{pmatrix}29\\9\\3\\13\\28\\27\end{pmatrix}$$

Exercise 11
List all the available linear solvers inside OpenFOAM®.
After choosing the smoothSolver in OpenFOAM®, list all implemented smoothers.

Exercise 12
Find in OpenFOAM® the implementation of the BiCG linear solver (PBiCG) and compare it with the BiCG algorithm of Lanczos.

Exercise 13
Find in OpenFOAM® the implementation of the multigrid V-cycle ($FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/GAMG/GAMGSolverSolve.C) and compare it with the theoretical V-cycle algorithm.

Exercise 14
Verify the correct implementation in OpenFOAM® of the diagonal version of the ILU(0) developed by Pommerell.

References

1. Duff I, Erisman A, Reid J (1986) Direct methods for sparse matrices. Clarendon Press, Oxford
2. Westlake JR (1968) A handbook of numerical matrix inversion and solution of linear equations. Wiley, New York
3. Stoer J, Bulirsch R (1980) Introduction to numerical analysis. Springer, New York
4. Press WH (2007) Numerical recipes, 3rd edn. The art of scientific computing. Cambridge University Press, Cambridge
5. Thomas LH (1949) Elliptic problems in linear differential equations over a network. Watson Sci. Comput. Lab Report, Columbia University, New York
6. Conte SD, deBoor C (1972) Elementary numerical analysis. McGraw-Hill, New York
7. Pozrikidis C (1998) Numerical computation in science and engineering. Oxford University Press, Oxford
8. Sebben S, Baliga BR (1995) Some extensions of tridiagonal and pentadiagonal matrix algorithms. Numer Heat Transfer, Part B, 28:323–351
9. Zhao X-L, Huang T-Z (2008) On the inverse of a general pentadiagonal matrix. Appl Math Comput 202(2):639–646
10. Karawia AA (2010) Two algorithms for solving general backward pentadiagonal linear systems. Int J Comput Math 87(12):2823–2830
11. Hageman L, Young D (1981) Applied iterative methods. Academic Press, New York
12. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics
13. Golub G, Van Loan C (2012) Matrix computations, 4th edn. The Johns Hopkins University Press, Baltimore
14. Dongarra J, Van Der Vorst H (1993) Performance of various computers using standard sparse linear equations solving techniques. In: Computer benchmarks. Elsevier Science Publishers BV, New York, pp 177–188
15. Van Der Vorst H (1981) Iterative solution methods for certain sparse linear systems with a nonsymmetric matrix arising from PDEProblems. J Comput Phys 44:1–19
16. Meijerink J, Van Der Vorst H (1977) An iterative solution method for linear systems of which the coefficient matrix is a symmetric M matrix. Math Comput 31:148–162

17. Beauwens R, Quenon L (1976) Existence criteria for partial matrix factorizations in iterative methods. SIAM J Numer Anal 13:615–643
18. Pommerell C (1992) Solution of large unsymmetric systems of linear equations. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland
19. Van Der Sluis A, Van Der Vorst H (1986) The rate of convergence of conjugate gradients. Numer Math 48(5):543–560
20. Faber V, Manteuffel T (1984) Necessary and sufficient conditions for the existence of a conjugate gradient method. SIAM J Numer Anal 21:315–339
21. Lanczos C (1950) An iteration method for the solution of eigenvalue problem of linear differential and integral operators. J Res Natl Bur Stand 45:255–282 (RP 2133)
22. Lanczos C (1952) Solution of systems of linear equations by minimized iterations. J Res Natl Bur Stand 49(1):33–53 (RP 2341)
23. Fletcher R (1976) Conjugate gradient methods for indefinite systems. In: Watson G (ed) Numerical analysis Dundee 1975. Springer, Berlin, pp 73–89
24. Sonneveld P (1989) CGS, AFast Lanczostype solver for nonsymmetric linear systems. SIAM J Sci Stat Comput 10:36–52
25. Van Der Vorst H (1992) BiCGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. SIAM J Sci Stat Comput 13:631–644
26. Van Der Vorst H, Vuik C (1991) GMRESR: a family of nested GMRES methods. Technical report 9180, Delft University of Technology, Faculty of Tech. Math, Delft, The Netherlands
27. Southwell R (1946) Relaxation methods in theoretical physics. Clarendon Press, Oxford
28. Demmel J, Heath M, Van Der Vorst H (1993) Parallel numerical linear algebra. Acta Numerica 2:111–198
29. Saad Y, Schultz M (1986) A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J Sci Stat Comput 7:856–869
30. Fedorenko P (1962) A relaxation method for solving elliptic difference equations. USSR Comput Math Math Phys 1(4):1092–1096
31. Poussin FV (1968) An accelerated relaxation algorithm for iterative solution of elliptic equations. SIAM J Numer Anal 5:340–351
32. Settari A, Aziz K (1973) A generalization of the additive correction methods for the iterative solution of matrix equations. SIAM J Numer Anal 10(3):506–521
33. Brandt A (1977) Multi-level adaptive solutions to boundary value problems. Math Comput 31 (138):333–390
34. Briggs WL (1987) A multigrid tutorial. Society of Industrial and Applied Mathematics, Philadelphia, PA
35. Shome B (2006) An enhanced additive correction multigrid method. Numer Heat Transfer B 49:395–407
36. Hutchinson BR, Raithby GD (1986) A multigrid method based on the additive correction strategy. Numer Heat Transfer 9:511–537
37. Elias SR, Stubley GD, Raithby GD (1997) An adaptive agglomeration method for additive correction multigrid. Int J Numer Meth Eng 40:887–903
38. Mavriplis D, Venkatakfrishnan V (1994) Agglomeration multigrid for viscous turbulent flows. AIAA paper 94-2332
39. Phillips RE, Schmidt FW (1985) A multilevel-multigrid technique for recirculating flows. Numer Heat Transfer 8:573–594
40. Lonsdale RD (1991) An algebraic multigrid scheme for solving the Navier-Stokes equations on unstructured meshes. In: Taylor C, Chin JH, Homsy GM (eds) Numerical methods in laminar and turbulent flow, vol 7(2). Pineridge Press, Swansea, pp 1432–1442
41. Perez E (1985) A 3D finite element multigrid solver for the euler equations. INRIA report 442
42. Connell SD, Braaten DG (1994) A 3D unstructured adaptive multigrid scheme for the Euler equations. AIAA J 32:1626–1632
43. Parthasarathy V, Kallinderis Y (1994) New multigrid approach for three-dimensional unstructured adaptive grids. AIAA J 32:956–963

44. Mavriplis DJ (1995) Three-dimensional multigrid reynolds-averaged Navier-Stokes solver for unstructured meshes. AIAA J 33(12):445–453
45. Qinghua W, Yogendra J (2006) Algebraic multigrid preconditioned Krylov subspace methods for fluid flow and heat transfer on unstructured meshes. Numer Heat Transfer B 49:197–221
46. Lallemand M, Steve H, Dervieux A (1992) Unstructured multigridding by volume agglomeration: current status. Comput Fluids 21:397–433
47. Mavriplis DJ (1999) Directional agglomeration multigrid techniques for high Reynolds Number viscous flow solvers. AIAA J 37:393–415
48. Trottenberg U, Oosterlee C, Schüller A (2001) Multigrid. Academic Press, London
49. Phillips RE, Schmidt FW (1985) Multigrid techniques for the solution of the passive scalar advection-diffusion equation. Numer Heat Transfer 8:25–43
50. Ruge JW, Stüben K (1987) Algebraic multigrid (AMG). In: McCormick SF (ed) Multigrid methods, frontiers in applied mathematics, vol 3. SIAM, Philadelphia, pp 73–130
51. OpenFOAM, 2015 Version 2.3.x. http://www.openfoam.org