The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Computer Modelling Techniques

MMME3086 UNUK, 2023/24

Numerical Methods: Solution of linear systems of equations

Author: Mirco Magnini
Office: Coates B100a
Email: mirco.magnini@nottingham.ac.uk

$$\begin{pmatrix} a_{1,1} & a_{1,2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & a_{i-1,i-2} & a_{i-1,i-1} & a_{i-1,i} & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & a_{i,i-1} & a_{i,i} & a_{i,i+1} & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & a_{i+1,i} & a_{i+1,i+1} & a_{i+1,i+2} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix} \times \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_{i-1} \\ T_i \\ T_{i+1} \\ \vdots \\ T_{n-2} \\ T_{n-1} \\ T_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{i-1} \\ b_i \\ b_{i+1} \\ \vdots \\ b_{n-2} \\ b_{n-1} \\ b_n \end{pmatrix}$$

Contents

1 Introduction

In the previous lecture, we have seen that the finite-volume discretisation of diffusion problems, irrispective of the 1D/2D/3D nature of the problem and of the structured/unstructured grid used, leads to a system of linear algebraic equations that, in matricial form, can be written as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{B}, \tag{1}$$

where $\mathbf{A}$ is a square $n \times n$ matrix, $\mathbf{x}$ a $n \times 1$ vector which contains the n unknowns, and $\mathbf{B}$ is a $n \times 1$ vector of known elements; n is the number of control volumes employed to discretised the domain. The solution of the system of equations, Eq. (1), is the task of the linear solver.

Depending on the morphology or regularity of $\mathbf{A}$, e.g. full, band-diagonal, triangular, symmetric, positive definite, etc. etc., there exist optimal solution algorithms. These can be grouped into:

- **Direct methods:** compute the solution $x_1, x_2, ..., x_n$ within a finite number of operations.
- **Iterative methods:** are based on the sequential evaluation of approximate solutions that are supposed to converge to the exact solution after ∞ iterations.

The scope of this lecture is to outline the most popular direct and indirect methods adopted in the CFD practice to solve linear systems. We will start with a brief introduction on the desirable characteristics of the matrix of the coefficients $\mathbf{A}$ in Section 2, followed by the explanation of direct methods in Section 3. Iterative methods will be illustrated in Section 4, which will be followed by a comparison of direct and iterative methods in Section 5. We will conclude with a tutorial about the implementation of a direct and an iterative method in Matlab to solve linear systems, in Sec. 6.

2 System condition

There are certain features of the matrix $\mathbf{A}$ that make it easier, or impossible, to solve the linear system. Systems of equations can be classified into three main types:

- **Well-conditioned**: very small changes in one or more of the coefficients of $\mathbf{A}$ result in a very small or negligible effect on the solution for $\mathbf{x}$. This is the most desirable situation.
- **Ill-conditioned**: small changes in coefficients result in large changes in the solution. This happens when two or more of the equations are nearly identical, which means that the determinant of the matrix is close to zero. This is a very dangereous situation, as it will be explained below.
- **Singular**: Two or more of the equations are exactly identical (or differ by a constant multiple), so that the determinant of the matrix is exactly zero. Essentially, this means that we have only $n - 1$ (or less) independent equations in our system, whereas the number of variables is still n. As such, the system is underdetermined and no solution exists, therefore numerical methods cannot be used for singular systems.

Example 1. Consider the following system of 2 equations:

$$\begin{vmatrix} 1 & 2 \\ 1.1 & 2 \end{vmatrix} . \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} 10 \\ 10.4 \end{vmatrix},$$

The solution can be easily shown to be $x_1 = 4$ and $x_2 = 3$. However, if we change just one coefficient in the second equation from 1.1 to 1.09, i.e.:

$$\begin{vmatrix} 1 & 2 \\ 1.09 & 2 \end{vmatrix} . \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} 10 \\ 10.4 \end{vmatrix},$$

the solution changes to $x_1 = 4.444$ and $x_2 = 2.777$. If the same coefficient is changed to 1.08, the solution changes again to $x_1 = 5$ and $x_2 = 2.5$. Therefore, this is an ill-conditioned system since a small change in one coefficient has a large effect on the solution for $\mathbf{x}$.

Example 2. Consider the following equations:

$$\begin{aligned} 0.9999x - 1.0001y &= 1 \\ x - y &= 1, \end{aligned}$$

the solution is $x = 0.5$ and $y = -0.5$. If we add a very small number, say β to the right-hand side of the second equation, the equations become:

$$\begin{aligned} 0.9999x - 1.0001y &= 1 \\ x - y &= 1 + \beta, \end{aligned}$$

and the solution becomes $x = 0.5 + 5000.5\beta$ and $y = -0.5 + 4999.5\beta$. Here, a small change in the value of β in the right-hand side of one equation results in a change in the solution of approximately 5000 times the value of β. This indicates that the equations are ill-conditioned.

Example 3. Similarly, an ill-conditioned system can be sensitive to changes in the right-hand side vector $\mathbf{B}$. To demonstrate this, consider the following set of 4 equations:

$$\begin{vmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{vmatrix} . \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} 32 \\ 23 \\ 33 \\ 31 \end{vmatrix}.$$

Assume that the following computed solution is obtained:

$$\begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} 6 \\ -7.2 \\ 2.0 \\ -0.1 \end{vmatrix}.$$

To check whether this computed solution is acceptable, we evaluate the residual vector:

$$\mathbf{R} = \mathbf{B} - \mathbf{A} \cdot \mathbf{x} = \begin{vmatrix} 32 \\ 23 \\ 33 \\ 31 \end{vmatrix} - \begin{vmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{vmatrix} \times \begin{vmatrix} 6 \\ -7.2 \\ 2.0 \\ -0.1 \end{vmatrix} = \begin{vmatrix} -0.1 \\ 0.1 \\ 0.1 \\ -0.1 \end{vmatrix}.$$

Since the residual vector $\mathbf{R}$ appears to be small, this may indicate that the computed solution is accurate. However, this is not true, since the exact solution is:

$$\begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \\ 1 \end{vmatrix}.$$

This example demonstrates that even if the residual vector is small, the computed solution may be inaccurate.

The system condition can be verified beforehand by calculating the condition number of the matrix:

$$cond(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}||, \tag{2}$$

where $||\mathbf{A}||$ is a scalar and is any opportune norm of $||\mathbf{A}||$. For a well-conditioned system, $cond(\mathbf{A})$ will be close to 1, whereas for an ill-conditioned system we will have $cond(\mathbf{A}) \gg 1$.

Ill-conditioned systems represent a danger for the calculation, because any small change of the coefficient matrix may yield drastically different solutions as shown in examples 1 and 2 (and this will affect direct methods), and because a small residual vector may not be sufficient to guarantee that the solution is accurate, as emphasised by example 3 (this will impact indirect methods).

Further reading: Burden and Douglas Faires [1], Ch. 7.5; Chapra and Canale [2], Ch. 9.1.

3 Direct methods

Direct methods for the solution of the linear system of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ compute the solution $x_1, x_2, ..., x_n$ within a finite number of operations. The basic idea is to add or subtract linear combinations of the given equations until (at least) one of the equations contains only one of the unknowns.

This is probably the way you learned to solve linear equations in high school. For example, you may be already familiar with the **Cramer**'s rule. Take the system of 3 equations:

$$a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 = b_1$$
$$a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 = b_2$$
$$a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 = b_3.$$

According to this rule, x_1 would be computed as:

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{1,2} & a_{1,3} \\ b_2 & a_{2,2} & a_{2,3} \\ b_3 & a_{3,2} & a_{3,3} \end{vmatrix}}{det(\mathbf{A})}$$

where the numerator is obtained from $\mathbf{A}$ by replacing the column of coefficients of the unknown in question by the constants $b_1, ..., b_n$, and $det(\mathbf{A})$ is the determinant of $\mathbf{A}$. Unfortunately, the *number of operations* required by the Cramer's rule is on the order of $n!$ (factorial of n), for example if $n = 100$ then the number of operations is $\sim 10^{157}$, and thus intractable. For *number of operations*, we mean the number of floating-point operations (addition, subtraction, multiplication, division), or *flops*, involved in the algorithm, and thus the flops give an idea of the computational expense of the algorithm.

Obviously, there exist more efficient direct methods for solving linear systems, such as the *Gauss-Jordan* elimination or the *Gaussian elimination*; the latter will be treated more in detail in the next subsection.

3.1 Gaussian elimination

The Gaussian elimination is a very robust algorithm based on the manipulation of $\mathbf{A}$ to turn it into an upper triangular matrix, where the unknowns $x_1, ..., x_n$ are then obtained by back-substitution. Let's start with the coefficient matrix and known terms vector:

$$\mathbf{A} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{vmatrix}, \qquad \mathbf{B} = \begin{vmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{vmatrix}.$$

Forward elimination

First, we eliminate the first unknown x_1 from the second to the n-th equation by subtracting the first equation, multiplied by $a_{i,1}/a_{1,1}$, from each i-th ($i \geq 2$) equation, so that each i-th equation (from the second on) is replaced by:

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n - \frac{a_{i,1}}{a_{1,1}}(a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n) = b_i - \frac{a_{i,1}}{a_{1,1}}b_1,$$

where you can see that x_1 cancels out. We refer to the first equation as the *pivot equation*, and to $a_{1,1}$ as the *pivot element*. After sweeping all the $i = 2, ..., n$ rows, $\mathbf{A}$ and $\mathbf{B}$ become:

$$\mathbf{A}^{(1)} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2} - \frac{a_{2,1}}{a_{1,1}}a_{1,2} & a_{2,3} - \frac{a_{2,1}}{a_{1,1}}a_{1,3} & \cdots & a_{2,n} - \frac{a_{2,1}}{a_{1,1}}a_{1,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n,2} - \frac{a_{n,1}}{a_{1,1}}a_{1,2} & a_{n,3} - \frac{a_{n,1}}{a_{1,1}}a_{1,3} & \cdots & a_{n,n} - \frac{a_{n,1}}{a_{1,1}}a_{1,n} \end{vmatrix}, \qquad \mathbf{B}^{(1)} = \begin{vmatrix} b_1 \\ b_2 - \frac{a_{2,1}}{a_{1,1}}b_1 \\ \vdots \\ b_n - \frac{a_{n,1}}{a_{1,1}}b_1 \end{vmatrix},$$

and the first column of $\mathbf{A}$ under the first row now contains only zeros. We call $\mathbf{A}^{(1)}$ and $\mathbf{B}^{(1)}$ the new coefficient matrix and known terms vector obtained after this first set of operations.

Now, we eliminate the second unknown x_2 from the third to the n-th equation by subtracting the second equation, multiplied by $a_{i,2}^{(1)}/a_{2,2}^{(1)}$, from each i-th ($i \geq 3$) equation, so that each i-th equation (from the third on) is replaced by:

$$a_{i,2}^{(1)}x_2 + a_{i,3}^{(1)}x_3 + \cdots + a_{i,n}^{(1)}x_n - \frac{a_{i,2}^{(1)}}{a_{2,2}^{(1)}}\left(a_{2,2}^{(1)}x_2 + a_{2,3}^{(1)}x_3 + \cdots + a_{2,n}^{(1)}x_n\right) = b_i^{(1)} - \frac{a_{i,2}^{(1)}}{a_{2,2}^{(1)}}b_2^{(1)},$$

where you can see that x_2 cancels out. Thus, the pivot equation is now the second, and the pivot element is $a_{2,2}$. After sweeping all the $i = 3, ..., n$ rows, $\mathbf{A}$ becomes:

$$\mathbf{A}^{(2)} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2}^{(1)} & a_{2,3}^{(1)} & \cdots & a_{2,n}^{(1)} \\ 0 & 0 & a_{3,3}^{(2)} & \cdots & a_{3,n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n,3}^{(2)} & \cdots & a_{n,n}^{(2)} \end{vmatrix},$$

where now the second column, under the second row, contains only zeros. The same procedure is repeated $N-1$ times, when $\mathbf{A}^{(N-1)}$ has become an *upper triangular* matrix:

$$\mathbf{A}^{(N-1)} = \begin{vmatrix} a_{1,1}^{(N-1)} & a_{1,2}^{(N-1)} & a_{1,3}^{(N-1)} & a_{1,4}^{(N-1)} & a_{1,5}^{(N-1)} & \cdots & a_{1,n}^{(N-1)} \\ 0 & a_{2,2}^{(N-1)} & a_{2,3}^{(N-1)} & a_{2,4}^{(N-1)} & a_{2,5}^{(N-1)} & \cdots & a_{2,n}^{(N-1)} \\ 0 & 0 & a_{3,3}^{(N-1)} & a_{3,4}^{(N-1)} & a_{3,5}^{(N-1)} & \cdots & a_{3,n}^{(N-1)} \\ 0 & 0 & 0 & a_{4,4}^{(N-1)} & a_{4,5}^{(N-1)} & \cdots & a_{4,n}^{(N-1)} \\ 0 & 0 & 0 & 0 & a_{5,5}^{(N-1)} & \cdots & a_{5,n}^{(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & a_{n,n}^{(N-1)} \end{vmatrix}.$$

Note that, for simplicity in the notation, all the superscipts have been turned into $N-1$ to indicate that they refer to the values at the end of the last forward step.

Backward substitution

Now, from our original linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$, we have a new system $\mathbf{A}^{(N-1)} \cdot \mathbf{x} = \mathbf{B}^{(N-1)}$, which is much easier to solve because, starting from the n-th row:

$$a_{n,n}^{(N-1)}x_n = b_n^{(N-1)} \Rightarrow x_n = \frac{b_n^{(N-1)}}{a_{n,n}^{(N-1)}},$$

for the line n-1:

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

and so on, e.g. for the i-th line:

$$x_i = \frac{b_i^{(N-1)} - \sum_{k=i+1}^{n} a_{i,k}^{(N-1)}x_k}{a_{i,i}^{(N-1)}}.$$

Example. Consider the following system with 3 equations:

$$\begin{vmatrix} 2 & 1 & -1 \\ 1 & 3 & 2 \\ 1 & -1 & 4 \end{vmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} 1 \\ 13 \\ 11 \end{vmatrix}.$$

From the second and third rows, subtract the first row multiplied by $a_{i,1}/a_{1,1}$:

$$\begin{vmatrix} 2 & 1 & -1 \\ 0 & 2.5 & 2.5 \\ 0 & -1.5 & 4.5 \end{vmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} 1 \\ 12.5 \\ 10.5 \end{vmatrix}.$$

From the third line, substract the second multiplied by $a_{3,2}/a_{2,2}$:

$$\begin{vmatrix} 2 & 1 & -1 \\ 0 & 2.5 & 2.5 \\ 0 & 0 & 6 \end{vmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} 1 \\ 12.5 \\ 18 \end{vmatrix},$$

which, by back-substitution starting from the third line, yields $x_3 = 3$, $x_2 = 2$ and $x_1 = 1$.

The **main characteristics** of the Gaussian elimination are:

- The method is very reliable, it always calculates a solution (if a solution exists).
- It can be demonstrated (Chapra and Canale [2], Sec. 9.2.1) that the number of operations is proportional to n^3.
- Owing to the large number of operations done on the coefficients of the matrix **A**, direct methods suffer from *round-off errors*. Round-off errors originate from the fact that computers retain only a fixed number of significant figures during a calculation. The discrepancy introduced by this omission of significant figures is called round-off error. If the number of calculations is small, round-off errors can be ignored. However, if the number of equations is very large, the number of multiplication and division operations dramatically increases, and in some situations (such as in ill-conditioned matrices, see below) can cause a serious deterioration in accuracy. As a general rule of thumb, round-off errors become important when the number of equations exceeds 100 equations.
- Ill-conditioned systems pose particular difficulties. Remember that in ill-conditioned systems, small changes in coefficients result in large changes in the solution (Section 2). Therefore, round-off errors may introduce errors in **A** while this is being manipulated during the forward ellimination process, and if **A** is ill-conditioned these errors may result in large changes in the solution. Round-off errors are impactful when a pivot element is very close to zero, owing to the division by the pivot element during the process of elimination. The remedies to avoid this are *pivoting*, that is, to switch the rows so that the largest element is the pivot element, and *scaling*,

that is, normalise each row by scaling all the coefficients in a given row by dividing them by the largest coefficient in the row, i.e. the largest coefficient becomes equal to 1.

- The matrix of the coefficients **A** needs to be entirely stored, also including the zero elements, thus requiring a space in the memory proportional to n^2.
- Given the number of operations required to reduce **A** to an upper triangular matrix, the algorithm for the Gaussian elimination is more complex to program than those for iterative methods.

Further reading: Chapra and Canale [2], Ch. 9.

3.2 The Tri-Diagonal Matrix Algorithm (TDMA)

In the case where **A** is a tridiagonal matrix, which we have seen to happen when discretising the 1D diffusion equation in the previous lecture, the Gaussian elimination method turns into a convenient and efficient algorithm, with the number of operations being proportional to n instead of n^3.

Let's the denote the generic discretisation equation for the i-th node as:

$$a_i T_i = b_i T_{i+1} + c_i T_{i-1} + d_i \tag{3}$$

where $i = 1, ..., n$. The notation is chosen in agreement with that in Patankar [4], Ch. 4.2.7, remembering our previous notation: $a_i \equiv a_{i,P} \equiv a_{i,i}$, $b_i \equiv -a_{i,E} \equiv -a_{i,i+1}$, $c_i \equiv -a_{i,W} \equiv -a_{i,i-1}$ and $d_i \equiv b_i$. Since the leftmost boundary control volume has no west neighbour, we have $c_1 = 0$; since the rightmost boundary control volume has no east neighbour, we have $b_n = 0$.

During the forward elimination procedure, we can write:

$$\mathbf{CV_1}) \quad a_1 T_1 = b_1 T_2 + d_1 \Rightarrow T_1 = \frac{b_1 T_2 + d_1}{a_1} = P_1 T_2 + Q_1, \quad \text{with } P_1 = \frac{b_1}{a_1},\ Q_1 = \frac{d_1}{a_1}, \tag{4}$$

where P_1 and Q_1 only include known coefficients. Then:

$$\mathbf{CV_2}) \quad a_2 T_2 = b_2 T_3 + c_2 \overbrace{(P_1 T_2 + Q_1)}^{T_1} + d_2 \Rightarrow T_2 = \frac{b_2 T_3 + c_2 Q_1 + d_2}{a_2 - c_2 P_1} = P_2 T_3 + Q_2,$$
$$\text{with } P_2 = \frac{b_2}{a_2 - c_2 P_1},\ Q_2 = \frac{c_2 Q_1 + d_2}{a_2 - c_2 P_1}, \tag{5}$$

where P_2 and Q_2 only include known or previously calculated coefficients. Then:

$$\mathbf{CV_i}) \quad a_i T_i = b_i T_{i+1} + c_i T_{i-1} + d_i \Rightarrow T_i = P_i T_{i+1} + Q_i, \quad \text{with } P_i = \frac{b_i}{a_i - c_i P_{i-1}},\ Q_i = \frac{c_i Q_{i-1} + d_i}{a_i - c_i P_{i-1}}, \tag{6}$$

where, in general, P_i multiplies the forward unknown (T_{i+1}) and Q_i is a known term. Note that both P_i and Q_i only include known or previously calculated coefficients. Finally, since there is no forward term for the rightmost control volume:

$$\mathbf{CV_n}) \quad T_n = Q_n, \quad \text{with } Q_n = \frac{c_n Q_{n-1} + d_n}{a_n - c_n P_{n-1}}, \tag{7}$$

where Q_n only includes known or previously calculated coefficients, and this can be used to evaluate T_n and start the back substitution procedure:

$$\begin{aligned} T_n &= Q_n \Rightarrow T_n \\ T_{n-1} &= P_{n-1}T_n + Q_{n-1} \Rightarrow T_{n-1} \\ T_i &= P_i T_{i+1} + Q_i \Rightarrow T_i \\ T_1 &= P_1 T_2 + Q_1 \Rightarrow T_1 \end{aligned}$$

In summary, an outline of the algorithm is:

1. Calculate P_1 and Q_1;
2. Calculate all the other P_i and Q_i in ascending order ($i = 2, ..., n-1$);
3. Calculate Q_n and set $T_n = Q_n$;
4. Calculate all the other T_i in descending order ($i = n-1, ..., 1$).

In worked example 2, see Section 6.2, we will see how to implement the TDMA algorithm in Matlab to solve the linear system of Worked example 2 of the previous lecture, thus replacing Matlab's backslash operator.

The TDMA algorithm can be easily extended to solve the penta- and epta-diagonal matrices arising from 2D and 3D diffusion problems with structured meshes, although the algorithm becomes partially iterative. **Further reading**: Patankar [4], Sec. 4.2.7; Versteeg and Malalasekera [5], from Sec. 7.2 to Sec. 7.5.

4 Iterative methods

Iterative methods for the solution of the linear system of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ start with an initial guess for the solution and use the algebraic equations to systematically improve it, until the solution is sufficiently close to the correct solution of the algebraic equations.

Be $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ the system that we want to solve, with $\mathbf{x}$ being the exact solution. After m iterations, we have an approximate solution $\mathbf{x}^{(m)}$ which does not satisfy these equations exactly. Instead, there is a nonzero residual $\mathbf{R}^{(m)}$ such that:

$$\mathbf{A} \cdot \mathbf{x}^{(m)} = \mathbf{B} - \mathbf{R}^{(m)}, \tag{8}$$

and an error vector:

$$\mathbf{e}^{(m)} = \mathbf{x} - \mathbf{x}^{(m)}. \tag{9}$$

The iterative procedure is successful if both $\mathbf{R}$ and $\mathbf{e}$ go to zero as the number of iterations increases. In practice, however, the iterative procedure is arrested when the residuals fall below a certain (small) tolerance, because the exact solution of the system would require an infinite number of iterations. Therefore, an important difference with the direct methods is that iterative methods always find an

approximate solution $\mathbf{x}$ that solves the system $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ within a certain tolerance, but not exactly (as direct methods, at least in principle, do).

The **main characteristics** of iterative methods are:

- The number of operations is proportional to $n \times m$, but the number of iterations m is not kwown beforehand.
- Convergence of the method is not guaranteed, as we will see in the next section.
- There is no need to store the entire matrix $\mathbf{A}$ if this is sparse and contains many zeros, only the nonzero elements can be stored, thus occupying a minimal memory storage of order n.
- For non-linear problems with a large and sparse matrix (typical CFD case), they are preferable to direct methods because they are less computationally expensive and require less storage.
- Round-off errors are insignificant, because $\mathbf{A}$ and $\mathbf{B}$ are not manipulated during the solution procedure and the final iteration may be regarded as an initial guess.
- They are much easier to implement than direct methods.

The most popular iterative method is the *Gauss-Seidel* method, which is reviewed in detail in the next section.

Further reading: Ferziger and Peric [3], Sec. 5.3.

4.1 The Gauss-Seidel iterative method

In the Gauss-Seidel method, at each m-th iteration, the equations are solved sequentially from the 1-st to the n-th, using guess values for the non-i-th unknowns of each i-th equation:

$$a_{i,1}x_1 + \ldots + a_{i,i}x_i + \ldots + a_{i,n}x_n = b_i \Rightarrow x_i = \frac{b_i}{a_{i,i}} - \sum_{j=1, j\neq i}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^*, \tag{10}$$

where the asterisk denotes the guess values. What distinguishes the Gauss-Seidel method from other iterative methods is that the guess values are the most recently available values:

$$x_j^* = \begin{cases} x_j^{(m)}, & \text{if } j < i \\ x_j^{(m-1)}, & \text{if } j > i \end{cases}$$

so that at each m-th iteration, the i-th unknown can be obtained using the following formula:

$$x_i^{(m)} = \frac{b_i}{a_{i,i}} - \sum_{j=1}^{i-1} \frac{a_{i,j}}{a_{i,i}} x_j^{(m)} - \sum_{j=i+1}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^{(m-1)}. \tag{11}$$

In worked example 1, see Section 6.1, we will see how to implement the Gauss-Seidel algorithm in Matlab to solve the linear system of Worked example 2 of the previous lecture.

Example. Consider the system of equations:

$$x_1 = 0.4x_2 + 0.2$$
$$x_2 = x_1 + 1,$$

and use the Gauss-Seidel method with initial guess $(0, 0)$ to iteratively solve the system with an accuracy up to the 2nd decimal.

m	0	1	2	3	4	5	6	**7**
x_1	0	0.2	0.68	0.872	0.949	0.98	0.992	**0.997**
x_2	0	1.2	1.68	1.872	1.949	1.98	1.992	**1.997**

We can see that, while we use the previous iteration values for x_2 when we solve the first equation for x_1, when we solve the second equation for x_2 we actually use the newer values for x_1, and not those of the previous iteration, in line with Eq. (11). We can see that the system is solved within the required tolerance with 7 iterations, with the approximate solution being close to the correct solution, $x_1 = 1$ and $x_2 = 2$.

An important, negative aspect of iterative methods is that convergence to the correct solution is not guaranteed, as this depends on the form of the coefficient matrix. Let's simply rearrange the two equations above in the following form:

$$x_1 = x_2 - 1$$
$$x_2 = 2.5x_1 - 0.5.$$

The correct solution is still $x_1 = 1$ and $x_2 = 2$, however if we repeat the same iterative procedure as before, this diverges:

m	0	1	2	3	4	5	$\cdots$	∞
x_1	0	-1	-4	-11.5	-30.25	-77.13	$\cdots$	divergence
x_2	0	-3	-10.5	-29.25	-76.13	-193.32	$\cdots$	divergence

So, how can we make sure that the iterative solution will converge? There exists a sufficient condition for the convergence of the Gauss-Seidel method, called *Scarborough criterion*, which states that the method converges if:

$$S_i = \frac{\sum\limits_{j=1, j\neq i}^{n} |a_{i,j}|}{|a_{i,i}|} = \begin{cases} \leq 1, & \text{for all equations} \\ < 1, & \text{for at least one equation} \end{cases} \tag{12}$$

If we apply this criterion to the first set of equations (where the method was converging), for the first equation we obtain $S_1 = |a_{1,2}|/|a_{1,1}| = |-0.4|/|1| = 0.4$ and for the second equation $S_2 = |a_{2,1}|/|a_{2,2}| = |-1|/|1| = 1$, and thus the criterion is satisfied because $S_i \leq 1$ for each equation and $S_i < 1$ for at least one equation (the first). However, for the second set of equations (where the method diverges), we obtain $S_1 = |a_{1,2}|/|a_{1,1}| = |-1|/|1| = 1$ and $S_2 = |a_{2,1}|/|a_{2,2}| = |-2.5|/|1| = 2.5$, where now $S_2 > 1$ and the criterion is not satisfied. Therefore, divergence was expected. For a system

with a larger number of equations, the numerator of Eq. (12) represents the sum of the absolute values of the off-diagonal elements of the i-th row of the matrix, whereas the denominator is the diagonal element. In short, the criterion requires that the coefficients on the diagonal are larger than those off-diagonal, which is a property often referred to as *diagonal dominance*. Therefore, in order for the Gauss-Seidel method to converge, the coefficient matrix **A** must be diagonally dominant.

Further reading: Patankar [4], Sec. 4.4.3; Chapra and Canale [2], Sec. 11.2; Versteeg and Malalasekera [5], Sec. 7.6.

4.2 Solution relaxation

In iterative methods, the rate of change of the solution from one iteration to the following one can be controlled by introducing a *relaxation factor* that can either slow down (under-relaxation) or speed-up (over-relaxation) the convergence. Let's rewrite the generic iteration equation, Eq. (10), as follows:

$$x_i^{(m)} = \frac{b_i}{a_{i,i}} - \sum_{j=1, j\neq i}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^* = x_i^{(m-1)} + \left[\frac{b_i}{a_{i,i}} - \sum_{j=1}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^* \right], \tag{13}$$

where $x_i^{(m-1)}$ is the value at the previous iteration and the term between square brackets identifies the rate of change of x_i from the previous to the new iteration. We can control this rate of change by introducing the relaxation factor ω:

$$x_i^{(m)} = x_i^{(m-1)} + \omega \left[\frac{b_i}{a_{i,i}} - \sum_{j=1}^{n} \frac{a_{i,j}}{a_{i,i}} x_j^* \right], \tag{14}$$

where:

- $\omega > 1$ identifies **over-relaxation**. This speeds-up the otherwise slow convergence rate of iterative methods, however it makes the algorithm more at risk of divergence. ω should never be above 2.
- $\omega < 1$ identifies **under-relaxation**. This slows down changes thus stabilizing the iterative procedure, and it is sometimes necessary when solving a system of nonlinear equations, where the coefficients of the matrix depend on the values of **x**.

Note that the best value of ω which guarantees convergence with a minimum number of iterations is problem-dependent and it is not known beforehand.

Further reading: Patankar [4], Sec. 4.5; Chapra and Canale [2], Sec. 11.2.2; Versteeg and Malalasekera [5], Sec. 7.6.3.

4.3 Convergence criteria for iterative methods

In this section we see some criteria that we can use to decide when the iterative procedure for the solution is to be arrested. Bear in mind that we do not know the exact solution and therefore we can use it to estimate how far we are from it.

- **Absolute iteration error**:

$$|\delta^{(m)}| = |\mathbf{x}^{(m)} - \mathbf{x}^{(m-1)}| < tol. \tag{15}$$

It arrests the procedure when the rate of solution change is below a tolerance, but there is no indication that we are converging to the correct solution.

- **Scaled iteration error**:

$$\frac{|\delta^{(m)}|}{|\mathbf{x}^{(m-1)}|} < tol. \tag{16}$$

Rescales the iteration error to make it nondimensional, that is, independent of the scale of the problem.

- **Absolute residuals**:

$$|\mathbf{R}^{(m)}| = |\mathbf{B} - \mathbf{A} \cdot \mathbf{x}| < tol. \tag{17}$$

Indicates how far are we from the exact solution of the linear system.

- **Scaled residuals**:

$$\frac{|\mathbf{R}^{(m)}|}{|diag(\mathbf{A}) \cdot \mathbf{x}|} < tol. \tag{18}$$

Rescales the residual error to make it nondimensional.

- **Scaled residuals (alternative)**:

$$\frac{|\mathbf{R}^{(m)}|}{|\mathbf{R}^{(1)}|} < tol. \tag{19}$$

Rescales the residual error with the error after the 1st iteration, thus indicating how much has the solution improved since the beginning of the iterative process.

Note that we had already seen the scaled residuals when discussing the indicators of the accuracy of the numerical solution in Section 9 of Lecture 1. Although they are calculated in the same way, the concept is different. Those of Lecture 1 were used to judge the solution accuracy and did apply irrespective of the solution method (direct/iterative). Those defined in this section refer to the threshold applied to arrest the iterative solution. Of course, once the iterative solution is arrested, the final residual error becomes an indicator of the numerical accuracy of the solution.

5 Direct vs iterative methods

We are now ready to outline a short summary of the pros and cons of direct and iterative methods:

	Gaussian Elimination	Iterative Gauss-Seidel
Number of operations:	Order of n^3	Order of (m x n) where m =number of iterations
Programming effort:	More difficult to program.	Easier to program.
Memory requirement:	Requires large storage (n^2).	Requires minimal storage. Very suitable for very large sparse matrices.
Round-off errors:	Round-off errors accumulate in the elimination process. However, this is only an issue if the matrices are ill-conditioned.	Round-off errors are insignificant, since the final iteration may be regarded as an initial guess.
Reliability:	Robust, almost guaranteed to compute a solution (if there is one).	Less reliable, since it may not converge. Also, it requires diagonal dominance to converge.

6 Matlab tutorials: Implementation of the Gauss-Seidel and TDMA algorithms to solve a linear system

6.1 Worked example 1: Gauss-Seidel algorithm

Repeat the exercise of Worked example 2 of lecture 1 (finite-volume, 1D steady heat conduction, Dirichlet boundary conditions, nonzero source term) but, instead of using Matlab's backslash operator $T = A \setminus B$ to solve the linear system, implement the Gauss-Seidel iterative method. Suggestions:

- Implement the Gauss-Seidel method as a Matlab function; this will keep your code clean and will enable you to reuse the method to solve other exercises.
- As initial guess, it is reasonable to start with $T = T_a$ in all the nodes of the domain.
- As a convergence criterion, you can use the scaled residuals (Sec. 4.3), with a tolerance of $tol = 1e - 12$.
- Set a max number of iterations allowed, e.g. 10000, to avoid the solver to run forever in the case of divergence.

Solution

We can use the same Matlab code that we have developed in Worked example 2 of Lecture 1, take a look at the code snippets in the notes for L1. We will add another piece of code to solve the linear system once again, but now using the Gauss-Seidel method, and we will compare the solution error and residuals with those obtained using the backslash operator.

It is suggested to code the Gauss-Seidel method as a Matlab function, in order to keep your code tidy and to enable you to reuse the linear solver in any other exercise. In the main code, a Matlab function is called as follows (type `help function` on the command window for further info):

```
[output1,output2,...]=functionName(input1,input2,...);
```

where the `functionName` can be any name. Then, you need to define the function as another Matlab .m file, to be placed in the same folder as your main file, and make sure to name it `functionName.m`, otherwise Matlab will not be able to link the function call to the function algorithm.

Our Gauss-Seidel function will need to take the following steps:

1. Receive the initial guess, matrices **A** and **B**, max number of iterations and tolerance as input.
2. Start a loop where, while the residuals are above the tolerance and the number of iterations are below the max allowed number, at each iteration the equations are solved in sequence, from $i = 1$ to $i = n$, using the iteration equation Eq. (11).
3. At the end of each iteration, update the values of the residuals.
4. Once any of the criteria for arresting the procedure are met, return the solution to the main code, and possibly also return the history of the residuals and the number of iterations as outputs, so that you can track how the iterative procedure developed.

The matlab code that we can add to the main script that we wrote in Worked example 2 of Lecture 1 is as follows:

```
39      %%%%% Check of residuals
40 -    residual=sum(abs(B-A*T))/sum(abs(diag(A).*T)) %%% defined as |B-A*T|/|diag(A).T|
41
42      %%%%%%%%%%%%%% Gauss-Seidel
43 -    maxit=10000; % max number of iterations allowed
44 -    toll=1e-12; % convergence tolerance
45 -    T0=Ta*ones(n,1); % Initial guess
46 -    [T_gaussSeidel,nIter,residuals_gaussSeidel]=GaussSeidel(T0,A,B,maxit,toll); % GS as a function
47 -    hold on; plot(x0,T_gaussSeidel,'mv'); legend('Backslash','Exact','Gauss-Seidel')
48
49 -    error_gaussSeidel=mean(abs(T_gaussSeidel-Tteo'))
50 -    residuals_gaussSeidel(nIter)
51 -    figure('color','w','units','Centimeters','position',[5 5 7.5 7])
52 -    semilogy(residuals_gaussSeidel); grid on; xlabel('Iterations'); ylabel('Residuals')
```

Lines 1-40 are unchanged, see Worked example 2 of Lecture 1; we add the solution with the Gauss-Seidel algorithm in lines 42-52. Lines 43-45 set the max number of iterations, the tolerance on the residuals, and the initial guess vector for each nodes value of the temperature. Line 46 calls the function `GaussSeidel`, that will need to be defined, which takes as input the initial guess, matrices **A** and **B**, max number of iterations and tolerance, and yields as output the solution vector, the number of iterations to reach convergence, and the residuals; the residuals will be returned as a vector which contains the residuals at each iteration, so that you can track the convergence history and make sure that the tolerance is respected. We plot the solution in line 47, comparing it with the exact solution

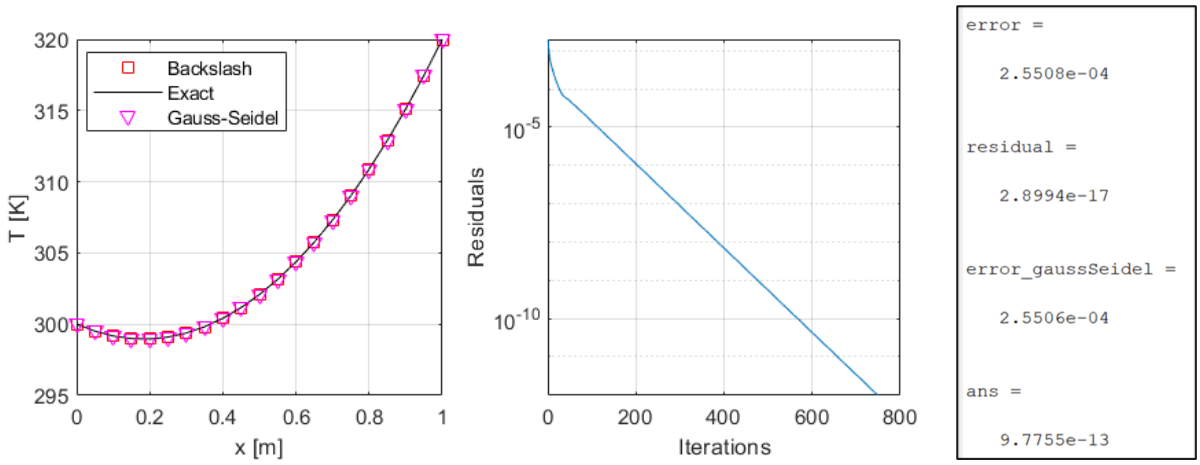
and that provided by the backslash operator. We output the solution error and the final residual value to the command window in lines 49-50. In lines 51-52, we plot the convergence history of the residuals.

The function is defined as another file named `GaussSeidel.m`, which will look like:

```
1   function [x,m,res]=GaussSeidel(x,A,B,maxit,toll)
2   res=sum(abs(B-A*x))/sum(abs(diag(A).*x));
3   m=0;
4
5   while (res>toll & m<maxit)
6       m=m+1;
7       for i=1:numel(x)
8           x(i)=B(i)/A(i,i)-A(i,1:i-1)/A(i,i)*x(1:i-1)-A(i,i+1:end)/A(i,i)*x(i+1:end);
9           %x(i)=x(i)+(B(i)/A(i,i)-A(i,:)/A(i,i)*x);  %%% This is in more compact form
10      end
11      res(m)=sum(abs(B-A*x))/sum(abs(diag(A).*x));
12  end
```

and that we will save in the same folder. In line 1, the input and output variables are defined. Note that the names of the variables do not need to match the names used in the main Matlab script, as the variables defined within the function will "exist" only in the function. However, the order of the calls is important, and the order of the multiple inputs and outputs must be the same as that in the main file. In short, Matlab does not care about the names of the input/output variables, but cares about the order in which they are called. In line 2, we evaluate a starting value of the residuals, based on the initial guess of temperature. In line 3, we initialize the number of iterations to zero. Then, line 5 starts the loop of the iterations, which will run `while` the redisuals are above the tolerance and the number of iterations are below the limit value that we set. When any of these two conditions is no longer satisfied, the loop terminates. In line 6, we update the iteration number. The `for` cycle in lines 7-10 is the core of the algorithm. Line 8 implements exactly the same equation as that in Eq. (11): `A(i,1:i-1)/A(i,i)*x(1:i-1)` performs the sum from $j = 1$ to $j = i - 1$, whereas `A(i,i+1:end)/A(i,i)*x(i+1:end)` performs the sum from $j = i + 1$ to $j = n$. Note that, automatically, the guess values of `x(1:i-1)` will be the most up-to-date values (iteration m), whereas the guess values `x(i+1:end)` will be those of the previous iteration $(m - 1)$, because the `for` loop has not yet updated them. Line 9 does the same job as line 8, but writes the iteration equation in a more compact form. Finally, in line 11 the scaled residuals are calculated and placed in a vector that contains the residuals for each iteration.

The results are displayed in Fig. 1. You can see that there seem to be no difference between the two numerical solutions and the exact one, in fact the command window output of the `error_gaussSeidel` is the same as that obtained when using the backslash operator. The residual history shows that residuals convergence to the set tolerance in about 750 iterations, and this is confirmed by the last command window output that reports the residuals after the final iteration. As expected, the reported value of `9.7755e-13` is below the set tolerance of `toll=1e-12`.



(a) (b) (c)

Figure 1: Worked example 1: (a) Comparison of theoretical and numerical solutions, (b) convergence history of the residuals and (c) output on the command window

6.2 Worked example 2: TDMA algorithm

Repeat the previous exercise, now solving the linear system with the TDMA algorithm, to be implemented as a function in Matlab.

Solution

The implementation of the TDMA algorithm can be done by following the pseudo-code outlined at the end of Sec. 3.2. We will need to write a function that takes **A** and **B** as inputs, and uses these to perform the steps outlined in Sec. 3.2. Let's recall here the generic discretisation equation for the i-th node used to develop the TDMA in Sec. 3.2:

$$a_i T_i = b_i T_{i+1} + c_i T_{i-1} + d_i, \tag{20}$$

where $i = 1, ..., n$. The same equation can be written using our usual double-index notation for **A**, as done in Sec. 4.5 of Lecture 1:

$$a_{i,i-1} T_{i-1} + a_{i,i} T_i + a_{i,i+1} T_{i+1} = b_i. \tag{21}$$

Therefore, we need to be careful when "translating" the coefficients from the nomenclature in Eq. (20) to that used in the Matlab code, which is more similar to that in Eq. (21). Specifically, the "translation" from (20) to (21) is as follows:

$$a_i \to a_{i,i}, \quad b_i \to -a_{i,i+1}, \quad c_i \to -a_{i,i-1}, \quad d_i \to b_i. \tag{22}$$

Starting from the matlab main code developed in the previous worked example, we add:

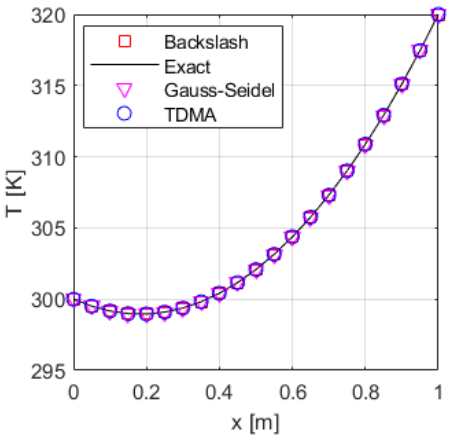
```
54    %%%%%%%%%%%%%% TDMA
55 -  [T_TDMA,residuals_TDMA]=TDMA(A,B); % TDMA as a function
56
57 -  figure(1);hold on; plot(x0,T_TDMA,'bo');
58 -  legend('Backslash','Exact','Gauss-Seidel','TDMA')
59 -  error_TDMA=mean(abs(T_TDMA-Tteo'))
60 -  residuals_TDMA
```

where the TDMA algorithm is called with a function. Note that, because the TDMA algorithm is a direct method, there is no need to pass as input the initial guess, max iteration number, or residual tolerance, because no iterations will be performed; therefore, we only need to pass **A** and **B**. The rest of the main code should be straightforward.

We will need to create a new Matlab file called `TDMA.m` where we will write the function, to be placed in the same folder as the main file. The function may look like this:

```
1    function [x,res]=TDMA(A,B)
2
3 -  n=length(B);
4    %%% Set P1 and Q1
5 -  P(1)=-A(1,2)/A(1,1);
6 -  Q(1)=B(1)/A(1,1);
7
8    %%% Calculate Pi, Qi from i=2 to i=n-1
9 -  for i=2:n-1
10 -     P(i)=-A(i,i+1)/(A(i,i)+A(i,i-1)*P(i-1));
11 -     Q(i)=(B(i)-A(i,i-1)*Q(i-1))/(A(i,i)+A(i,i-1)*P(i-1));
12 -  end
13
14   %%% Calculate Qn and set Tn=Qn
15 -  i=n;
16 -  Q(i)=(B(i)-A(i,i-1)*Q(i-1))/(A(i,i)+A(i,i-1)*P(i-1));
17 -  x(n)=Q(n);
18
19   %%% Back substitution
20 -  for i=n-1:-1:1
21 -     x(i)=P(i)*x(i+1)+Q(i);
22 -  end
23
24 -  x=x';
25 -  res=sum(abs(B-A*x))/sum(abs(diag(A).*x));
```

In line 3 we set the number of equations, that was not directly passed but can be extracted as the length of the known terms vector. The rest of the function follows exactly the pseudo-code outlined at the end of Sec. 3.2. In lines 5 and 6 we calculate P_1 and Q_1 according to Eq. (4), take a look at the "translations" in Eq. (22) if necessary. Then, the `for` loop in lines 9-12 evaluates the P_i and Q_i



(a)

```
error_TDMA =

    2.5508e-04


residuals_TDMA =

    6.8765e-17
```

(b)

Figure 2: Worked example 2: (a) Comparison of theoretical and numerical solutions and (b) output to the command window.

in ascending order from $i = 2$ to $i = n - 1$ according to Eq. (6), while we put $i = n$ outside the loop because $P_n = 0$ and line 10 would not work. We evaluate Q_n in line 16 and set $T_n = Q_n$ in line 17. We perform the back substitutions in lines 20-22, now in descending order from $i = n - 1$ to $i = 1$. In line 24 we transpose the solution vector to turn it into a column vector, which is required owing to the formula used for the residuals in line 25.

The output of the algorithm is shown in Fig. 2. It can be seen that the TDMA performs well, giving the same deviation with the exact solution as the backslash operator or Gauss-Seidel algorithm. The residuals are of the same order of magnitude as those observed when using the backslash operator.

6.3 Suggested exercises

1. Repeat Worked example 1 and add over-relaxation to the Gauss-Seidel method (Sec. 4.2) with $\omega = 1.1, 1.2, ..., 2$, and look at the number of iterations necessary for convergence.

2. Repeat Worked example 1 and test different values of the number of control volumes n, for example $n = 10, 20, 50, 100, 500, 1000, 5000$, and see how the number of iterations in Gauss-Seidel method changes with n. Note that you will need to increase the max number of iterations allowed when n is large. You can also keep track of the time that it takes to solve the linear system each time, using Matlab's `tic-toc` function. In the main Matlab code of Worked example 1, simply add `tic` before line 46 and `time=toc` after line 46. This will write in the variable `time` the time necessary to perform the calculations between the `tic` and `toc` commands, measured in seconds.

3. Compare the time it takes (using Matlab's `tic-toc` function) to solve the linear system using (i) backslash, (ii) Gauss-Seidel and (iii) TDMA, when varying the number of grid nodes from a small value ($n = 10$) to a large value, e.g. $n = 5000$ or above, according to the available power of your computer.

References

[1] R. L. Burden and J. Douglas Faires. *Numerical Analysis, 9th edition.* Brooks/Cole, Cengage Learning, Boston, USA, 2010. NUsearch; Download(may not work).

[2] S. C. Chapra and R. P. Canale. *Numerical Methods for Engineers, 7th edition.* McGraw-Hill Education, New York, USA, 2015. NUsearch; Download(may not work).

[3] J. H. Ferziger and M. Peric. *Computational Methods fo Fluid Dynamics.* Springer, Berlin, Germany, 2002. NUsearch; Download(may not work).

[4] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Hemisphere Publishing, New York, 1980. NUsearch; Download(may not work).

[5] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics – The Finite Volume Method, 2nd edition.* Pearson Education Limited, Harlow, England, 2007. NUsearch; Download(may not work).