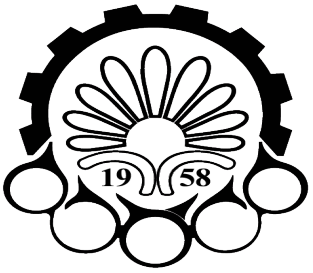

Amirkabir University of Technology
(Tehran Polytechnic)

Special Topics (Numerical Methods for Continuum Mechanics)

Lecture #2

Lecture Notes: Finite Difference Method

Mostafa Abbaszadeh

Spring 2026

Last updated: February 2026

Contents

Chapter 2

Finite Difference method (FDM)

The finite difference approximations for derivatives are one of the simplest and of the oldest methods to solve differential equations. It was already known by L. Euler (1707-1783) ca. 1768, in one dimension of space and was probably extended to dimension two by C. Runge (1856-1927) ca. 1908. The advent of finite difference techniques in numerical applications began in the early 1950s and their development was stimulated by the emergence of computers that offered a convenient framework for dealing with complex problems of science and technology. Theoretical results have been obtained during the last five decades regarding the accuracy, stability and convergence of the finite difference method for partial differential equations.

For the sake of simplicity, we shall consider the one-dimensional case only. The main concept behind any finite difference scheme is related to the definition of the derivative of a smooth function u at a point $x \in \mathbb{R}$

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h} \tag{2.1}$$

and to the fact that when h tends to 0 (without vanishing), the quotient on the right-hand side provides a "good" approximation of the derivative. In other words, h should be sufficiently small to get a good approximation. It remains to indicate what exactly is a good approximation, in what sense.

Actually, the approximation is good when the error commited in this approximation (i.e. when replacing the derivative by the differential quotient) tends towards zero when h tends to zero. If the function u is sufficiently smooth in the neighborhood of x, it is possible to quantify this error using a Taylor expansion.

Taylor series Suppose the function u is C^2 continuous in the neighborhood of x. For any $h > 0$ we have:

$$u(x+h) = u(x) + hu' + \frac{h^2}{2} u''(x+h_1) \tag{2.2}$$

where h_1 is a number between 0 and h (i.e. $x + h_1$ is point of $(x, x + h)$). For the treatment of problems, it is convenient to retain only the first two terms of the previous expression:

$$u(x + h) = u(x) + hu'(x) + \mathcal{O}(h^2) \tag{2.3}$$

where the term $\mathcal{O}(h^2)$ indicates that the error of the approximation is proportional to h^2. From the equation (2.2), we deduce that there exists a constant $C > 0$, such that for $h > 0$ sufficiently small we have

$$\left|\frac{u(x + h) - u(x)}{h} - u'(x)\right| \leq Ch, \qquad C = \sup_{y\in[x,x+h_0]} \frac{|u''(y)|}{2}, \tag{2.4}$$

This approximation is known as the *forward difference approximation* of u' . More generally, we define an approximation at order p of the derivative.

The approximation of the derivative u' at point x is of order $p(p > 0)$ if there exists a constant $C > 0$, independent of h, such that the error between the derivative and its approximation is bounded by Ch^p (i.e. is exactly $\mathcal{O}(h^p)$).

Likewise, we can define the first order *backward difference approximation* of u' at point x as:

$$u(x - h) = u(x) - hu'(x) = \mathcal{O}(h^2). \tag{2.5}$$

In order to improve the accuracy of the approximation, we define a consistant approximation, called the central difference approximation, by taking the points $x - h$ and $x + h$ into account. Suppose that the function u is three times differentiable in the vicinity of x :

$$u(x + h) = u(x) + hu' + \frac{h^2}{2}u'' + \frac{h^3}{6}u^{(3)}(\xi^+) \tag{2.6a}$$

$$u(x - h) = u(x) - hu' + \frac{h^2}{2}u'' - \frac{h^3}{6}u^{(3)}(\xi^+) \tag{2.6b}$$

where $\xi^+ \in (x, x + h)$ and $\xi^- \in (x - h, x)$. By subtracting these two expressions we obtain

$$\frac{u(x + h) - u(x - h)}{2h} = u'(x) + \frac{h^2}{6}u^{(3)}(\xi) \tag{2.7}$$

where $\xi \in (x - h, x + h)$. Hence, for every $h \in (0, h_0)$, we have the following bound on the approximation error:

$$\left|\frac{u(x + h) - u(x - h)}{h} - u'(x)\right| \leq Ch^2, \qquad C = \sup_{y\in[x-h_0,x+h_0]} \frac{|u^{(3)}(y)|}{6}, \tag{2.8}$$

Lemma 2.1. *Suppose u is a C^4 continuous function on an interval $[x - h_0, x + h_0]$, $h_0 > 0$. Then, there exists a constant $C > 0$ such that for every $h \in [0, h_0]$ we have:*

$$\left|\frac{u(x + h) - 2u(x) + u(x - h)}{h^2} - u''(x)\right| \leq Ch^2 \tag{2.9}$$

The differential quotient $\frac{u(x+h)-2u(x)+u(x-h)}{h^2}$ is a consistant second-order approximation of the second derivative u'' of u at point x.

Proof: practice!

2.1 FDM for a one-dimensional problem

We consider a bounded domain $D = (0,1) \subset \mathbb{R}$ and $u : D \to \mathbb{R}$ solving the non-homogeneous Dirichlet problem:

$$\begin{cases} -u'' + c(x)u(x) = f(x) \qquad x \in (0,1), \\ \\ u(0) = \alpha, \qquad u(1) = \beta, \end{cases} \tag{2.10}$$

where c and f are two given functions, defined D, $c \geq 0$. Now, the problem is to find $u_h \in \mathbb{R}^N$, such that $u_i \approx u(x_i)$, for all $i \in \{1, \ldots, N\}$, where u is the solution of problem (2.10). Introducing the approximation of the second order derivative by a differential quotient, we consider the following discrete problem:

$$\begin{cases} \dfrac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + c(x_j)u_j = f(x_j) \qquad j \in \{1, \ldots, N\}, \\ \\ u_0 = \alpha, \qquad u_{N+1} = \beta, \end{cases} \tag{2.11}$$

$$\begin{cases} j = 1 \to \dfrac{u_2 - 2u_1 + u_0}{h^2} + c(x_1)u_1 = f(x_1), \\ \\ j = 2 \to \dfrac{u_3 - 2u_2 + u_1}{h^2} + c(x_2)u_2 = f(x_2), \\ \\ \vdots \\ j = N-1 \to \dfrac{u_N - 2u_{N-1} + u_{N-2}}{h^2} + c(x_{N-1})u_{N-1} = f(x_{N-1}), \\ \\ j = N \to \dfrac{u_{N+1} - 2u_N + u_{N-1}}{h^2} + c(x_N)u_N = f(x_N), \end{cases}$$

The problem has been discretized by a FDM based on a three-points centered scheme for the second-order derivative. The problem (2.11) can be written in the matrix form as:

$$A_h u_h = b_h, \tag{2.12}$$

where A_h is the tridiagonal matrix defined as:

$$A_h = A_h^{(0)} + \begin{pmatrix} c(x_1) & 0 & \dots & \dots & 0 \\ 0 & c(x_2) & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & c(x_{N-1}) & 0 \\ 0 & \dots & \dots & 0 & c(x_N) \end{pmatrix}$$

with

$$A_h^{(0)} = \frac{1}{h^2} \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix}$$

and

$$b_h = \begin{pmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ f(x_N) + \frac{\beta}{h^2} \end{pmatrix}$$

We can summarize the concept of finite differences for problem (6.3) in the following table:

Theory (continuous)	Finite differences (discrete)
domain $D = [0,1]$	$I_N = \{0, \frac{1}{N+1}, \dots, 1\}$
unknown $u : [0,1] \to \mathbb{R}, \quad u \in C^2(D)$	$u_h = (u_1, \dots, u_N) \in \mathbb{R}^N$
conditions $u(0) = \alpha, \quad u(1) = \beta$	$u_0 = \alpha, \quad u_{N+1} = \beta$
equation $-u'' + cu = f$	$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + c(x_j)u_j = f(x_j)$

2.2 FDM for time-dependent problems

We consider a time-dependent, first-order boundary value problem posed in a bounded domain $D = (0,1)$

$$\begin{cases} \dfrac{\partial u}{\partial t}(t,x) - \nu \dfrac{\partial^2 u}{\partial x^2}(t,x) = f(t,x) & \forall t \in (0,T) \forall x \in (0,1), \\ u(0,x) = u_0(x) & \forall x \in (0,1), \\ u(t,0) = u(t,1) = 0 & \forall t \in (0,T) \end{cases} \tag{2.13}$$

where $f(t,x)$ is a given source term and $\nu \geq 0$. This equation is the well-know heat equation that describes the distribution of heat in a given domain over time. It is the prototypical parabolic partial differential equation. The function $u(\cdot,\cdot)$, solution to this equation, describes the temperature at a given location x in time. The analysis of this equation has been pioneered by the French physicist J. Fourier (1768-1830) who invented influential methods for solving partial differential equations.

2.2.1 An explicit scheme

To discretize the domain $[0,T] \times D$, we introduce equidistributed grid points corresponding to a spatial step size $h = 1/(N+1)$ and to a time step $\delta = 1/(M+1)$, where M, N are positive integers, and we define the nodes of a regular grid:

$$(t_n, x_j) = (n\delta, jh), \qquad n \in \{0,\ldots,M+1\}, \quad j \in \{0,\ldots,N+1\}. \tag{2.14}$$

We denote as u_j^n the value of an approximate solution at point (t_n, x_j) and $u(t,x)$ is the exact solution of the problem. The initial data must also be discretized as:

$$u_j^0 = u_0(x_j), \qquad \forall j \in \{0,\ldots,N+1\}. \tag{2.15}$$

Finally, the homogeneous Dirichlet boundary conditions are discretized as:

$$u_0^n = u_{N+1}^n = 0, \qquad \forall n \in \{0,\ldots,M+1\}. \tag{2.16}$$

The problem is then to find, at each time step, a vector $u_h \in \mathbb{R}^N$, such that its components are the values $(u_j^n)_{1\leq j\leq N}$.

Introducing the approximation of the second-order space derivative given by formula (2.9), and considering a forward difference approximation for the time derivative

$$\frac{\partial u}{\partial t}(t_n, x_J) \approx \frac{u_j^{n+1} - u_j^n}{\delta}. \tag{2.17}$$

Therefore for the time-dependent problem, we have

$$\frac{u_j^{n+1} - u_j^n}{\delta} - \nu \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2} = f(t_n, x_j) \tag{2.18}$$

This scheme is called *explicit* because the values $(u_j^{n+1})_{1\leq j\leq N}$ at time t_{n+1} are computed using the values on the previous time level t_n. Indeed, this system can be written in a vector form as

$$\frac{U^{n+1} - U^n}{\delta} + A_h U^n = C^m, \qquad \forall n \in \{0,\ldots,M\} \tag{2.19}$$

where the matrix A_h and the vector C^n are defined as

$$A_h = \frac{\nu}{h^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix}$$

and

$$C^n = \begin{pmatrix} f(t_1, x_1) \\ \vdots \\ \vdots \\ \vdots \\ f(t_j, x_N) \end{pmatrix}$$

and with the initial data:

$$U^0 = \begin{pmatrix} u_1^0 \\ \vdots \\ u_N^0 \end{pmatrix}$$

2.2.2 Implicit schemes

Changing the approximation of the time derivative for a backward difference would have resulted in the following implicit scheme:

$$\frac{u_j^n - u_j^{n-1}}{\delta} - \nu \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2} = f(t_n, x_j). \tag{2.20}$$

It requires solving a system of linear equations to compute the values $(u_j^{n+1})_{1 \le j \le N}$ at time t_n using the values of the previous time level t_{n-1}. The scheme can be written in vector form as:

$$\frac{U^n - U^{n-1}}{\delta} + A_h U^n = C^m, \qquad \forall n \in \{1, \dots, M+1\} \tag{2.21}$$

where the matrix A_h and the vectors C^n and U^0 are identical to the corresponding terms in the explicit scheme (2.18). However, computing the values u_j^n with respect to the values u_j^{n-1} j requires finding the inverse of the tridiagonal matrix A_h.

2.3 Two-dimensional Poisson equation with FD

In 2D case, on a rectangle $D = [a, b] \times [c, d]$ we have the 2-D Poisson equation

$$-u_{xx} - u_{yy} = f, \qquad (x, y) \in (a, b) \times (c, d) \tag{2.22a}$$

$$u(x, y)|_\Omega = u_0(x, y) \tag{2.22b}$$

u is electrical charge and f is fixed charge. Under the regularity condition, $f \in L^2(\Omega)$, the solution $u(x,y) \in C^2(D)$ exists and it is unique. An analytic solution is often unavailable, and the FD procedure is explained below.

- Step 1: Generate a grid. For example, a uniform Cartesian grid can be generated:

$$x_i = a + ih_x, \quad i = 0, 1, 2, \ldots, m, \quad h_x = \frac{b-a}{m} \tag{2.23}$$

$$y_i = c + jh_y, \quad j = 0, 1, 2, \ldots, n, \quad h_y = \frac{d-c}{n} \tag{2.24}$$

  In seeking an approximate solution U_{ij} at the grid points (x_i, y_j) where $u(x,y)$ is unknown, there are $(m-1)(n-1)$ unknowns.

- Step 2: Step 2: Represent the partial derivatives with FD formulas involving the function values at the grid points. For example, if we adopt the three-point central FD formula for second-order partial derivatives in the x- and y-directions respectively, then

$$\begin{aligned} &- \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h_x^2} - \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h_x^2} \\ &= f_{ij} + T_{ij}, \qquad i = 1, \ldots, m-1 \quad j = 1, \ldots, n-1, \end{aligned} \tag{2.25}$$

  where $f_{ij} = f(x_i, y_j)$. The local truncation error satisfies

$$T_{ij} \approx \frac{h_x^2}{12}\frac{\partial^4 u}{\partial x^4}(x_i, y_j) + \frac{h_y^2}{12}\frac{\partial^4 u}{\partial y^4}(x_i, y_j) + \mathcal{O}(h^4), \tag{2.26}$$

  where

$$h = \max\{h_x, h_y\} \tag{2.27}$$

  and the finite difference discretization is consistent if

$$\lim_{h \to 0} \|T\| = 0, \tag{2.28}$$

  so this FD discretization is consistent and second-order accurate. The approximate solution values U_{ij} obtained from solving the linear system of algebraic equations, i.e.,

$$\begin{aligned} &- \frac{U_{i-1,j} + U_{i+1,j}}{h_x^2} - \frac{U_{i,j-1} + U_{i,j+1}}{h_y^2} + \left(\frac{2}{h_x^2} + \frac{2}{h_y^2}\right) U_{ij} = f_{ij} \\ &i = 1, \ldots, m-1 \quad j = 1, \ldots, n-1, \end{aligned} \tag{2.29}$$

  For more simplicity we assume $m = n$, i.e., $h_x = h_y = h$ therefore we have

$$-\frac{U_{i-1,j} + U_{i,j-1} - 4U_{i,j} + U_{i+1,j} + U_{i,j+1}}{h^2} = f_{i,j} \tag{2.30}$$

  The FD equation at the grid point (x_i, y_j) involves five grid points in a five-point stencil, (x_{i-1}, y_j), (x_{i+1}, y_j),(x_i, y_{j-1}), (x_i, y_{j+1}) and (x_i, y_j). The grid points in the FD stencil are sometimes labeled east, north, west, south, and the center in the literature. The center

(x_i, y_j) is called the master grid point, where the FD equation is used to approximate the partial differential equation.

- Step 3: Solve the linear system of algebraic equations (5.9), to get the approximate values for the solution at all of the grid points.
- Step 4: Error analysis

The corresponding coefficient matrix is block tridiagonal,

$$A = \frac{1}{h^2}\begin{pmatrix} B & -I & 0 \\ -I & B & -I \\ 0 & -I & B \end{pmatrix}$$

where I is the 3×3 identity matrix and

$$B = \frac{1}{h^2}\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}$$

In general, for an $n+1$ by $n+1$ grid we obtain

$$A = \frac{1}{h^2}\begin{pmatrix} B & -I & & \\ -I & B & -I & \\ & \ddots & \ddots & \ddots \\ & & -I & B \end{pmatrix}$$

and

$$B = \frac{1}{h^2}\begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & 4 \end{pmatrix}$$

Since A is symmetric positive definite and weakly diagonally dominant, the coefficient matrix A is a non-singular, and hence the solution of the system of the FD equations is unique.

We review the definition of above terms:

Definition: A symmetric matrix is a square matrix that is equal to its transpose. Formally, matrix A is symmetric if

$$A = A^T$$

For example the following matrix is symmetric:

$$\begin{pmatrix} 1 & 7 & 3 \\ 7 & 4 & -5 \\ 3 & -5 & 6 \end{pmatrix}$$

Definition: A symmetric $n \times n$ real matrix A is said to be positive definite if the scalar $z^T Az$ is strictly positive for every non-zero vector z of n real numbers.

Definition: The $n \times n$ matrix A is non-singular (invertible) if there exists an $n \times n$ square matrix B such that $AB = BA = I_n$

2.4 Two-dimensional heat equation with FD

We now revisit the transient heat equation, this time with sources/sinks, as an example for two-dimensional FD problem. In 2D (x, z space), we can write

$$\frac{\partial T}{\partial t} = \kappa \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial z^2} \right) \frac{Q}{\rho c_p}, \tag{2.31}$$

where ρ is density, c_p is heat capacity, κ is the thermal conductivities and Q is radiogenic heat production. The simplest way to discretize (2.31) on a domain, e.g. a box with width L and height H, is to employ the explicit method like in 1-D

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \kappa \left(\frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{(\Delta x)^2} + \frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{(\Delta z)^2} \right) + \frac{Q_{i,j}^{n}}{\rho c_p} \tag{2.32}$$

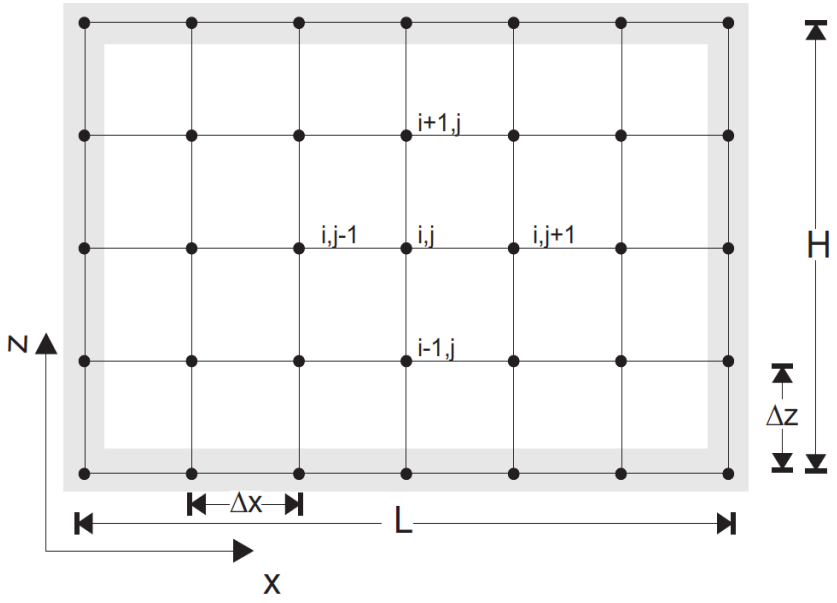

FIGURE 2.1: Finite difference discretization of the 2D heat problem.

where Δx and Δz indicates the node spacing in both spatial directions, and there are now two indices for space, i and j for z_i and x_j , respectively (see Figure 2.1). Rearranging gives

$$T_{i,j}^{n+1} = T_{i,j}^{n} + s_x \left(T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n} \right) + s_z \left(T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n} \right) + \frac{Q_{i,j}^{n} \Delta t}{\rho c_p} \tag{2.33}$$

where

$$s_x = \frac{\kappa \Delta t}{(\Delta x)^2} \qquad \text{and} \qquad s_z = \frac{\kappa \Delta t}{(\Delta z)^2} \tag{2.34}$$

Boundary conditions can be set the usual way. A constant (Dirichlet) temperature on the left-hand side of the domain (at $j = 1$), for example, is given by

$$T_{i,j-1} = T_{\text{left}} \qquad \text{for all } i. \tag{2.35}$$

A constant flux (Neumann BC) on the same boundary at $(i, j = 1)$ is set through fictitious boundary points

$$\begin{aligned} \frac{\partial T}{\partial x} &= c_1, \\ \frac{\partial T_{i,2} - Ti,0}{2\Delta x} &= c_1, \\ T_{i,0} &= T_{i,2} - 2\Delta x c_1, \end{aligned}$$

which can then be plugged into (2.33) so that for $j = 1$, for example,

$$\begin{aligned} T_{i,1}^{n+1} = T_{i,1}^n &+ s_x(2T_{i,2}^n - 2(T_{i,1}^n + \Delta x c_1)) \\ &+ s_z(T_{i+1,1}^n - 2T_{i,1}^n + T_{i-1,1}^n) + \frac{Q_{i,1}^n \Delta t}{\rho c_p} \end{aligned} \tag{2.36}$$

The implementation of this approach is straightforward as T can be represented as a matrix with MATLAB , to be initialized, for example, for n_z and n_x rows and columns, respectively, as

$$\text{T = zeros(nz, nx);} \tag{2.37}$$

and then accessed as $T(i, j)$ for $T_{i,j}$. The major disadvantage of fully explicit schemes is, of course, that they are only stable if

$$\frac{2\kappa \Delta t}{\min((\Delta x)^2, (\Delta z)^2)} \leq 1 \tag{2.38}$$

2.4.1 Fully implicit method

If we employ a fully implicit, unconditionally stable discretization scheme as for the 1D exercise (2.31) can be written as

$$\frac{T_{i,j}^{n+1} - T_{i,j}^n}{\Delta T} = \kappa \left(\frac{T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{(\Delta x)^2} + \frac{T_{i+1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i-1,j}^{n+1}}{(\Delta z)^2} \right) + \frac{Q_{i,j}^n}{\rho c_p} \tag{2.39}$$

Rearranging terms with $n + 1$ on the left and terms with n on the right hand side gives

$$-s_z T_{i+1,j}^{n+1} - s_x T_{i,j+1}^{n+1} + (1 + 2s_z + 2s_x) T_{i,j}^{n+1} - s_z T_{i-1,j}^{n+1} - s_x T_{i,j-1}^{n+1} = T_{i,j}^n + \frac{Q_{i,j}^n \Delta t}{\rho c_p} \tag{2.40}$$

As in the 1D case, we have to write these equations in a matrix A and a vector b (and use MATLAB x=A \ b to solve for T^{n+1}). From a practical point of view, this is a bit more complicated than in the 1D case, since we have to deal with "book-keeping" issues, i.e. the mapping of $T_{i,j}$ to the entries of a temperature vector $T(k)$ (as opposed to the more intuitive matrix T(i,j) we could use for the explicit scheme). If a 2D temperature field is to be solved

for with an equivalent vector T, the nodes have to be numbered continuously, for example as in Figure 2.2. The derivative versus x-direction is then e.g.

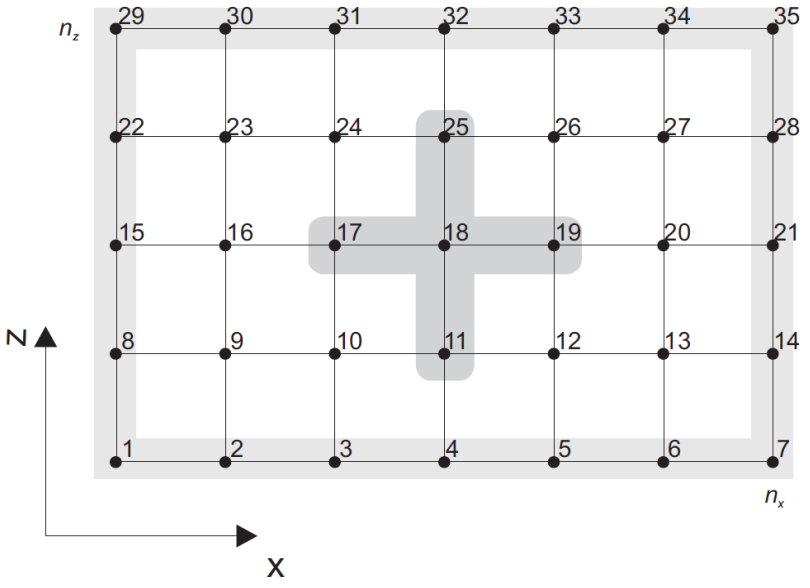

FIGURE 2.2: Numbering scheme for a 2D grid with $n_x = 7$ and $n_z = 5$.

$$\frac{\partial^2 T}{\partial x^2}|_{i=3,j=4} = \frac{1}{(\Delta x)^2}(T_{19} - 2T_{18} + T_{17}) \tag{2.41}$$

and the derivative versus z-direction is given by

$$\frac{\partial^2 T}{\partial z^2}|_{i=3,j=4} = \frac{1}{(\Delta z)^2}(T_{25} - 2T_{18} + T_{11}) \tag{2.42}$$

If n_x are the number of grid points in x-direction and n_z the number of points in z-direction, we can write (2.41) and (2.42) in a more general way as:

$$\frac{\partial^2 T}{\partial x^2}|_{i,j} = \frac{1}{(\Delta x)^2}(T_{(i-1)n_x+j+1} - 2T_{(i-1)n_x+j} + T_{(i-1)n_x+j-1}) \tag{2.43}$$

$$\frac{\partial^2 T}{\partial z^2}|_{i,j} = \frac{1}{(\Delta z)^2}(T_{i \cdot n_x+j} - 2T_{(i-1)n_x+j} + T_{(i-2)n_x+j}) \tag{2.44}$$

In matrix format this gives something like

$$A = \begin{pmatrix}
1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
0 & 1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\vdots & \vdots & & & & & & & & & & & \vdots & \vdots \\
0 & 0 & & -s_z & \dots & -s_x & (1+2s_x+2s_z) & -s_x & \dots & -s_z & 0 & & 0 & 0 \\
0 & 0 & & 0 & -s_z & \dots & -s_x & (1+2s_x+2s_z) & -s_x & \dots & -s_z & 0 & 0 & 0 \\
\vdots & \vdots & & & & & & & & & & & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1
\end{pmatrix}$$

Note that we now have five diagonals filled with non-zero entries as opposed to three diagonals in the 1D case. The solution vector x is given by

$$x = \begin{pmatrix} T_1^{n+1} = T_{1,1} \\ T_2^{n+1} = T_{1,2} \\ \vdots \\ T_{(i-1)n_x+j}^{n+1} = T_{i,j} \\ T_{(i-1)n_x+j+1}^{n+1} = T_{i,j+1} \\ \vdots \\ T_{n_x n_z-1}^{n+1} = T_{n_z,n_x-1} \\ T_{n_x n_z}^{n+1} = T_{n_z,n_x} \end{pmatrix},$$

and the load (right hand side) vector is given by ($Q = 0$ for simplicity)

$$b = \begin{pmatrix} T_{\text{bottom}} \\ T_{\text{bottom}} \\ \vdots \\ T_{(i-1)n_x+j}^{n} \\ T_{(i-1)n_x+j+1}^{n} \\ \vdots \\ T_{\text{top}} \\ T_{\text{top}} \end{pmatrix},$$

2.5 MATLAB codes

2.5.1 Implicit Euler FDM: 1D heat equation

```
%% Implicit backward Euler method for the heat equation U_t = beta*U_xx (1D)
x1=0;
a = 1.; % length of bar in x-direction
x2=a;

T = 12000.; % length of time for solution

n = 2;   % no of grid points Xn
m = 100.;

dx = a/(n+1.); % grid spacing in x direction

dt = T/m;

t=0.; % initial time = 0

beta = 1.e-5; % thermal diffusivity constant

s = beta*dt/(dx^2);  % gain parameter

if s > 0.5
```

```
   fprintf('gain parameter > 0.5 - implicit backward Euler method is still stable');
end

% build the A matrix to march finite difference solution forward in time

% check first with n=3 to test that you are getting the right A matrix and b vector:
%s=1  % checkpoint values comment out later!!!
%n=3  % checkpoint values comment out later!!!
% because we have a Neumann boundary condition at x=a we are solving for U_1 to U_(n+1)

Adiag = (1+2*s)*ones(n+1,1);

Asubs = -s*ones(n+1,1);

Asuper = -s*ones(n+1,1);

A = spdiags([Asubs,Adiag,Asuper],[-1 0 1],n+1,n+1);
% spdiags: Extract and create sparse band and diagonal matrices
% specify boundary conditions through vector b and by changing any rows in A matrix needed - for Neumann boundary conditions

% for Neumann boundary conditions at x = a: (approximating du/dx with leapfrog in spatial derivative)
A(n+1,n) = -2.*s;

b= zeros(n+1,1);

% for Dirichlet boundary conditions at x=0: b[1] = s*g1(t_k)
% U(x=0, t) = 1.
b(1) = s*1.;
% for Neumann boundary conditions at x = a: b[n] = s*dx*g2(t_k)
% U_x(x=a,t) = 2.
b(n+1) =  4.*s*dx;
% b % check b is right too!

% set up mesh in x-direction:
x = linspace(x1+dx,x2, n+1)';

% Set up vector U^0 at time tk=0: U(x,0) = f(x)
% using U(x,0) = 2*x + sin(2*pi*x) + 1

U_tk = 2.*x + sin(2*pi*x) + 1.;

figure(1)
plot (x,U_tk,'-')
title ('Initial condition for Temperature distribution')
xlabel ('x')
ylabel ('U')

% store solution for each time in matrix U((n+1) x m):

U=zeros(n+1,m);
```

```
tvec=zeros(m,1);

U(:,1) = U_tk;
tvec(1) = t;

% now march solution forward in time using U_tk+1 = inverse(A)*(U_tk + b):
for k = 1:m
    t = t+dt;
    % if boundary conditions vary with time you need to update b here
    % with implicit method we solve a matrix equation at each step:
    c = U_tk + b;
    U_tk_1 = A\c;
    % this is very time consuming later we will discuss faster ways to solve
    this problem using iterative methods

    U(:,k) = U_tk_1;
     % for next time step:
    U_tk = U_tk_1;
    tvec(k) = t;

end

figure(2)
mesh (tvec,x,U)
title ('Variation of Temperature distribution with time using Backward Euler
    method')
xlabel ('t')
ylabel ('x')
zlabel ('U')
```

2.5.2 Explicit Euler FDM: 1D heat equation

```
%% Implicit backward Euler method for the heat equation U_t = beta*U_xx (1D)
x1=0;
a = 1.; % length of bar in x-direction
x2=a;

T = 12000.; % length of time for solution

n = 2;   % no of grid points Xn
m = 100.;

dx = a/(n+1.); % grid spacing in x direction

dt = T/m;

t=0.; % initial time = 0

beta = 1.e-5; % thermal diffusivity constant

s = beta*dt/(dx^2);  % gain parameter
```

```
20
21 if s > 0.5
22    fprintf('gain parameter > 0.5 - implicit backward Euler method is still
      stable');
23 end
24
25 % build the A matrix to march finite difference solution forward in time
26
27 % check first with n=3 to test that you are getting the right A matrix and b
      vector:
28 %s=1  % checkpoint values comment out later!!!
29 %n=3  % checkpoint values comment out later!!!
30 % because we have a Neumann boundary condition at x=a we are solving for U_1 to
      U_(n+1)
31
32 Adiag = (1+2*s)*ones(n+1,1);
33
34 Asubs = -s*ones(n+1,1);
35
36 Asuper = -s*ones(n+1,1);
37
38 A = spdiags([Asubs,Adiag,Asuper],[-1 0 1],n+1,n+1);
39 % spdiags: Extract and create sparse band and diagonal matrices
40 % specify boundary conditions through vector b and by changing any rows in A
      matrix needed - for Neumann boundary conditions
41
42 % for Neumann boundary conditions at x = a: (approximating du/dx with leapfrog
      in spatial derivative)
43 A(n+1,n) = -2.*s;
44
45 b= zeros(n+1,1);
46
47 % for Dirichlet boundary conditions at x=0: b[1] = s*g1(t_k)
48 % U(x=0, t) = 1.
49 b(1) = s*1.;
50 % for Neumann boundary conditions at x = a: b[n] = s*dx*g2(t_k)
51 % U_x(x=a,t) = 2.
52 b(n+1) =  4.*s*dx;
53 % b % check b is right too!
54
55 % set up mesh in x-direction:
56 x = linspace(x1+dx,x2, n+1)';
57
58 % Set up vector U^0 at time tk=0: U(x,0) = f(x)
59 % using U(x,0) = 2*x + sin(2*pi*x) + 1
60
61 U_tk = 2.*x + sin(2*pi*x) + 1.;
62
63 figure(1)
64 plot (x,U_tk,'-')
65 title ('Initial condition for Temperature distribution')
66 xlabel ('x')
67 ylabel ('U')
68
69 % store solution for each time in matrix U((n+1) x m):
```

```

U=zeros(n+1,m);
tvec=zeros(m,1);

U(:,1) = U_tk;
tvec(1) = t;

% now march solution forward in time using U_tk+1 = inverse(A)*(U_tk + b):
for k = 1:m
    t = t+dt;
    % if boundary conditions vary with time you need to update b here
    % with implicit method we solve a matrix equation at each step:
    c = U_tk + b;
    U_tk_1 = A\c;
    % this is very time consuming later we will discuss faster ways to solve
    this problem using iterative methods

    U(:,k) = U_tk_1;
     % for next time step:
    U_tk = U_tk_1;
    tvec(k) = t;

end

figure(2)
mesh (tvec,x,U)
title ('Variation of Temperature distribution with time using Backward Euler
    method')
xlabel ('t')
ylabel ('x')
zlabel ('U')
```

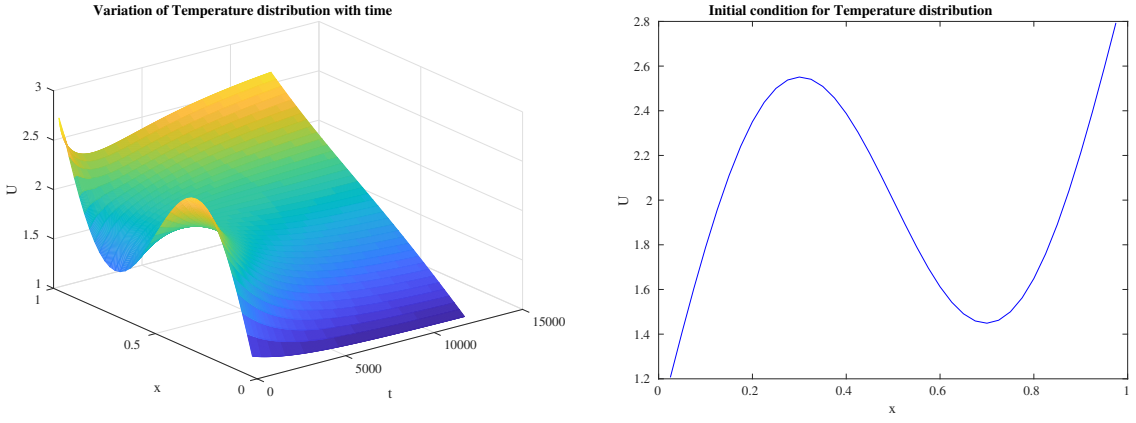

FIGURE 2.3: The FD solution of 1D heat equation for different intervals (left) and the initial condition (right).

2.5.3 2D Poisson equation with FDM

```

% Solve the Poisson's equation on unit rectangle.
%
```

```
% O(h^2) convergence rate is observed by using J=5,10,20,40,

clear
xl=0; xr=1;
yl=0; yr=1;
% x domain
% y domain
J=50;

% number of points in both x- and y-directions
h = (xr-xl)/J;

% mesh size
% build up the coefficient matrix
nr = (J-1)^2;
% order of the matrix
matA = zeros(nr,nr);
% can set J=3,4,5 etc to check the coefficient matrix

for i=1:nr
    matA(i,i)=4;
    if i+1 <= nr & mod(i,J-1) ~= 0
        matA(i,i+1)=-1;
    end
    if i+J-1 <= nr
        matA(i,i+J-1)=-1;
    end
    if i-1 >= 1 & mod(i-1,J-1) ~= 0
        matA(i,i-1)=-1;
    end
    if i-(J-1) >= 1
        matA(i,i-(J-1))=-1;
    end
end

% build up the right-hand side vector

for j=1:J-1
    y(j) = j*h;
    for i=1:J-1
        x(i) = i*h;
        [fij]=feval(@srcF,x(i),y(j)); % evaluate f(xi,yj)
        vecF((J-1)*(j-1)+i)=h^2*fij;
        [uij]=feval(@exU,x(i),y(j)); % evaluate exact solution
        ExU(i,j)=uij;
    end
end

% solve the system
vecU = matA\vecF'; % vecU is of order (J-1)^2

for j=1:J-1
    for i=1:J-1
        U2d(i,j)=vecU((J-1)*(j-1)+i); % change into 2-D array
    end
```

```
end

% display max error so that we can check convergence rate
disp('Max error ='), max(max(U2d-ExU)),

figure(1);
surf(x,y,U2d);
title('Numerical solution');
figure(2);
surf(x,y,ExU);
title('Exact solution');
```

```
function uu=srcF(x,y)
% The RHS function f for the governing PDE

uu=2*pi^2*sin(pi*x)*sin(pi*y);

end
```

```
% The exact solution of the governing PDE
function uu=exU(x,y)
% The exact solution of the governing PDE

uu=sin(pi*x)*sin(pi*y);

end
```

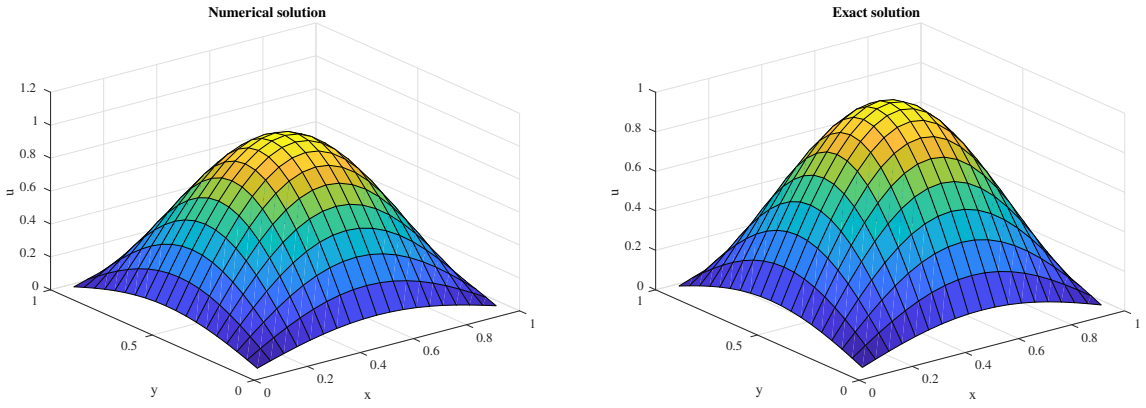

FIGURE 2.4: The FD solution of 2D Poisson equation with exact solution (left) and approximated solution (right).

2.5.4 2D heat equation with FDM

```

%%

clc
clear all

i_max = 15; j_max = 20;
```

```
8
9  N = (j_max-2)*(i_max-2); %Number of unknowns
10
11 fprintf('\n The number of unknowns is %g \n ',N)
12
13 Lx = 0.3; % System size in x direction(length)
14
15 Ly = 0.2; % System size in y direction(length)
16
17 dx = Lx/(i_max-1); % Grid spacing in x %h%
18
19 dy = Ly/(j_max-1); % Grid spacing in y
20
21 dt=1; %Time increment
22
23 Lt=20; %Final time
24
25 t_max = Lt/dt+1
26
27 T0=300;
28
29 beta = dx/dy; %Grid aspect ratio
30
31 fprintf('\n The grid aspect ratio is %g \n ',beta)
32
33 change_want = 1e-4; % Stop when the change is given fraction
34
35     x=linspace(0,Lx,i_max);
36     y=linspace(0,Ly,j_max);
37 %% Properties
38
39 alpha = 97.1e-6;
40
41 rho = 2702;
42
43 cp = 903;
44
45 %% ------- Initialization and Initial Condition------ %%.
46
47 sx=alpha*dt/(dx^2);
48
49 sy=alpha*dt/(dy^2);
50
51         T_old=zeros(j_max,i_max,t_max);
52 for j = 1:(j_max)
53     for i = 1:(i_max) T_old(j,i,1)=T0;
54     end
55 end
56
57 %% ------------ Source Term --------------%%
58
59 Q = zeros(j_max,i_max,t_max); % initialization
60
61 %% Other Properties %%
62
```

```
k = 200;

h = 200;

T_F = 310; %%%the solution is unstable(fluctuating) when T_F = T0, and the
    solution does not have zero gradient at the bottom when T_F~=T0

c1 = (-h/k);

c2 = h*T_F/k;

%% ---------------- MAIN LOOP --------------%%

T_new = T_old;
zeta = ((cos(pi/(i_max-1))+(beta^2*...
 cos(pi/(j_max-1))))/(1+beta^2))^2;
omega = 2/zeta*(1-sqrt(1-zeta));

 fprintf('\n the optimum omega  is %g \n ',omega) %%!!%% SOR only
max_iter = i_max*j_max; % Set max to avoid excessively long runs

for t=1:((t_max-1))

for iter=1:max_iter*20 %iterations

temp = 0;
 for j=1:j_max
 for i=1:i_max
%% SOR method %%

      if  j == 1
            if i==1
                T_new(j,i,t+1) = 1/(1+2*sx+2*sy+2*sy*dy*c1)*omega*(2*sy*T_old(j
    +1,i,t+1)-2*sy*dy*c2+ ...
                                +2*sx*T_old(j,i+1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
            elseif i==i_max
                T_new(j,i,t+1) = 1/(1+2*sx+2*sy+2*sy*dy*c1)*omega*(2*sy*T_old(j
    +1,i,t+1)-2*sy*dy*c2+ ...
                                +2*sx*T_old(j,i-1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
            else
              T_new(j,i,t+1) = 1/(1+2*sx+2*sy+2*sy*dy*c1)*omega*(2*sy*T_old(j+1,
    i,t+1)-2*sy*dy*c2+ ...
                                sx*T_old(j,i-1,t+1)+sx*T_old(j,i+1,t+1)+T_old(j,
    i,t)+dt*Q(j,i,t+1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
            end
       elseif j == j_max
           if i==1
                T_new(j,i,t+1) = 1/(1+2*sx+2*sy-2*sy*dy*c1)*omega*(2*sy*T_old(j
    -1,i,t+1)+2*sy*dy*c2+ ...
                                +2*sx*T_old(j,i+1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
            elseif i==i_max
```

```
108                 T_new(j,i,t+1) = 1/(1+2*sx+2*sy-2*sy*dy*c1)*omega*(2*sy*T_old(j
    -1,i,t+1)+2*sy*dy*c2+ ...
109                                 +2*sx*T_old(j,i-1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
110             else
111               T_new(j,i,t+1) = 1/(1+2*sx+2*sy-2*sy*dy*c1)*omega*(2*sy*T_old(j-1,
    i,t+1)+2*sy*dy*c2+ ...
112                                 sx*T_old(j,i-1,t+1)+sx*T_old(j,i+1,t+1)+T_old(j,
    i,t)+dt*Q(j,i,t+1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
113           end
114       else
115         if (i==1)
116           T_new(j,i,t+1) = 1/(1+2*sx+2*sy)*omega*(sy*T_old(j+1,i,t+1)+sy*T_old(j
    -1,i,t+1)+ ...
117                                 +2*sx*T_old(j,i+1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
118       elseif (i==i_max)
119           T_new(j,i,t+1) = 1/(1+2*sx+2*sy)*omega*(sy*T_old(j+1,i,t+1)+sy*T_old(j
    -1,i,t+1)+ ...
120                                 +2*sx*T_old(j,i-1,t+1)+T_old(j,i,t)+dt*Q(j,i,t
    +1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
121       else
122       T_new(j,i,t+1) = 1/(1+2*sx+2*sy)*omega*(sy*T_old(j+1,i,t+1)+sy*T_old(j-1,i
    ,t+1)+ ...
123                                 +sx*T_old(j,i-1,t+1)+sx*T_old(j,i+1,t+1)+T_old(
    j,i,t)+dt*Q(j,i,t+1)/rho/cp) +  (1-omega)*T_old(j,i,t+1);
124       end
125       end
126
    %-------------------------------------------------------------------------------%

127       temp = temp + abs((T_new(j,i,t+1)-T_old(j,i,t+1))/T_new(j,i,t+1)); %
    Relative Error
128       temp = temp + abs((T_new(j,i,t+1)-T_old(j,i,t+1))); % Relative Error
129       T_old(j,i,t+1) = T_new(j,i,t+1);
130    end
131 end
132    T_old = T_new;   %%!!%% Reset T_old
133 change(iter) = temp/(i_max-2)/(j_max-2);      % the average relative error of
    all interior points
134 if( change(iter) < change_want )
135       fprintf('\n The total number of iterations excuted is %g \n',iter)
136     disp('Desired accuracy achieved; breaking out of main loop');
137     break;
138 end
139 end
140
141     x=linspace(0,Lx,i_max);
142     y=linspace(0,Ly,j_max);
143      Z = flipud(T_new(:,:,t+1));
144     surf(x,y,Z)
145
146 end
```