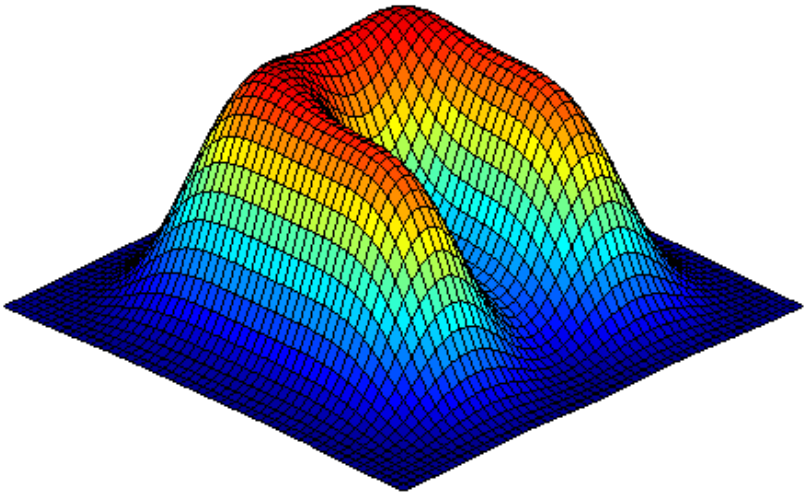
Manchester Metropolitan University

6G6Z3004 Numerical Methods for Partial Differential Equations II

Finite-Difference Methods

Dr Jon Shiach

2019 – 2020



Contents

Part II - Finite Difference Methods

In part I of this unit you were introduced to methods used find the analytical solution to a PDE. Unfortunately, for most practical applications attempting to find the analytical solution is either very difficult or not possible so we need to use a numerical approach instead. There are several different numerical methods that can be used to solve a PDE and in this unit we will be focusing on two of the most common of these: finite-difference methods (taught in term 1) and finite-volume methods (taught in term 2).

Term 1 Teaching Schedule – Dr Jon Shiach

Week	Date (w/c)	Material
5	21/10/2019	**Chapter 1 Partial Differential Equations:** Definitions and terminology, classifying PDEs, use of PDEs in modelling, Eulerian and Lagrangian methods. **Chapter 2 Finite-Difference Approximations** The Taylor series; the truncation error; first and second-order finite-difference approximations; method of undetermined coefficients; the finite-difference toolkit.
6	28/10/2019	**Chapter 3 Elliptic PDEs:** Laplace's and Poisson's equations; discretising a 2D domain; Jacobi, Gauss-Seidel and SOR methods; the L^2 error.
7	04/11/2019	**Chapter 4 Hyperbolic PDEs:** The advection equation; the first-order upwind and forward-time central-space schemes
8	11/11/2019	**Chapter 4 Hyperbolic PDEs continued:** Convergence, consistency and stability; von Neumann stability analysis; Lax-Friedrichs, Leapfrog, Lax-Wendroff and Crank-Nicolson schemes
9	18/11/2019	**Chapter 5 Parabolic PDEs:** The heat diffusion equation; grid refinement study
10	25/11/2019	**Chapter 6 Solving Multidimensional PDEs:** Differential marching operator; dimensional splitting; term splitting **Coursework assignment handed out**
11	02/12/2019	**Chapter 7 Systems of PDEs:** The Shallow Water Equations; linearising a system of PDEs; the Jacobian matrix
12	09/12/2019	**Consolidation of Finite-Difference Methods**

Chapter 1

Partial Differential Equations

This chapter will introduce the concept of partial differential equations, the different types and the general methodology used to solve partial differential equations using numerical methods.

1.1 Definitions

1.1.1 Partial derivatives

A partial derivative of a function of several variables is the derivative of that function with respect to one of those variables, the other variables being treated as constants. The partial derivative is denoted by the symbol ∂ instead of the 'd' that is used for the derivative of a single variable function. The same rules for derivatives of single variable functions apply to partial derivatives.

For example, let $U(x, y) = x^2 + xy$ be a two-variable differential function then U can be differentiated with respect to x and y, e.g.,

$$\frac{\partial U}{\partial x} = 2x + y,$$
$$\frac{\partial U}{\partial y} = x.$$

The partial derivatives are commonly denoted using subscripts, i.e.,

$$\frac{\partial U}{\partial x} = U_x = \partial_x U.$$

Higher order partial derivatives can be represented by repeating the variable in the subscript, i.e.,

$$\frac{\partial^2 U}{\partial x^2} = U_{xx} = \partial_{xx} U.$$

Mixed partial derivatives can be formed by differentiating a derivative with respect to a different variable, i.e.,

$$\frac{\partial}{\partial x}\left(\frac{\partial U}{\partial y}\right) = \frac{\partial}{\partial y}\left(\frac{\partial U}{\partial x}\right) = \frac{\partial^2 U}{\partial x \partial y} = U_{xy} = U_{yx} = \partial_{xy} U = \partial_{yx} U.$$

1.1.2 Partial differential equations

A Partial Differential Equation (PDE) is a differential equation that is expressed in terms partial derivatives of a function with two or more variables. For example,

$$aU_x + bU_y + cU_{xx} + dU_{yy} + eU_{xy} = f,$$

here U is an unknown function of the independent variables x and y and a, b, c, d, e and f are either known functions of x and y or known constants.

1.1.3 Order of a PDE

The *order* of a PDE is the order of its highest derivative. For example,

$$aU_x + bU_y = 0,$$

is a first-order PDE whereas

$$aU_{xx} + bU_y = 0,$$

is a second-order PDE.

1.1.4 Linear PDEs

A PDE that can be expressed as a linear combination of first-order partial derivatives is said to be a *linear* PDE. For example,

$$aU_x + bU_y + cU_z = f,$$

is a linear PDE.

1.1.5 Domain

The *domain* is defined as the set of all values of the independent variables for which there exists a solution to a PDE. In most cases the domain can be split into the *spatial domain* which is the subset of the domain for the spatial variables, x, y, z etc., and the *time domain* which is the subset of the domain for the difference values of the time variable t. Often the spatial domain is simply referred to as the domain even if time is also an independent variable of the PDE.

1.1.6 Initial conditions

The *initial conditions* is a function U of the known solution at the initial point in time. For PDEs that do not have time as an independent variable the initial conditions be any function. If $U(t, x)$ is a function of two variables where x is spatial position and t is time then the initial conditions at $t = 0$ is the function $U(0, x)$.

1.1.7 Boundary conditions

The *boundary conditions* determine the value of U at the boundary of the spatial domain. If $U(t, x)$ is a function of two variables where $a \leq x \leq b$ is the spatial position and t is time then the boundary conditions are described by the functions $U(t, a)$ and $U(t, b)$.

1.1.8 Solution of a PDE

The solution of a PDE is the function U that satisfies:

- the PDE itself;
- any boundary conditions;
- any initial conditions.

Most PDEs do not have exact analytical solutions so a numerical approach is required. Numerical methods for solving PDEs calculate an *approximate* solution to the PDE at discrete values of the independent variables using a numerical method that is usually implemented using a computer program.

Since a numerical solution is only an approximation of the exact solution U, it is common to denote the numerical solution using the lowercase character u.

Example 1

Show that the following PDE

$$U_t = kU_{xx}, \tag{1.1}$$

with the following initial and boundary conditions

$$\begin{aligned} U(0,x) &= 2\sin\left(\frac{\pi x}{L}\right), \\ U(t,0) &= 0, \\ U(t,L) &= 0, \end{aligned}$$

has the solution

$$U(t,x) = 2\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right). \tag{1.2}$$

Solution: We need to show that the function in equation (1.2) satisfies the PDE in equation (1.1), the initial conditions and the boundary conditions. The PDE is expressed in terms of the derivatives U_t and U_{xx}, evaluating these gives

$$\begin{aligned} U_t &= -\frac{2k\pi^2}{L^2}\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right), \\ U_{xx} &= -\frac{2\pi^2}{L^2}\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right). \end{aligned}$$

Substituting into equation (1.1) gives

$$-\frac{2k\pi^2}{L^2}\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right) = -\frac{2k\pi^2}{L^2}\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right),$$

since $LHS = RHS$ then the function in equation (1.2) satisfies the PDE in equation (1.2). Now consider the initial conditions

$$\begin{aligned} U(t=0,x) &= 2\sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{k\pi^2(0)}{L^2}\right) \\ &= 2\sin\left(\frac{\pi x}{L}\right)\exp(0) \\ &= 2\sin\left(\frac{\pi x}{L}\right), \end{aligned}$$

therefore equation (1.2) satisfies the initial conditions. Finally consider the boundary conditions

$$\begin{aligned} U(t,x=0) &= 2\sin\left(\frac{\pi.0}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right) = 2\sin(0)\exp\left(-\frac{k\pi^2 t}{L^2}\right) = 0, \\ U(t,x=L) &= 2\sin\left(\frac{\pi.L}{L}\right)\exp\left(-\frac{k\pi^2 t}{L^2}\right) = 2\sin(\pi)\exp\left(-\frac{k\pi^2 t}{L^2}\right) = 0, \end{aligned}$$

which are both satisfied. Since equation (1.2) satisfies the PDE, the initial conditions and the boundary conditions then we can say that this is a solution to the PDE in equation (1.1).

1.2 Classifying PDEs

Linear second-order PDEs can be classified as one of three types: *parabolic*, *hyperbolic* and *elliptic*. Consider the second-order linear PDE

$$aU_{xx} + bU_{xy} + cU_{yy} + dU_x + eU_y + f = 0. \tag{1.3}$$

A PDE of this form can be classified by the following

$$b^2 - 4ac \begin{cases} < 0, & \text{elliptic,} \\ = 0, & \text{parabolic,} \\ > 0, & \text{hyperbolic.} \end{cases}$$

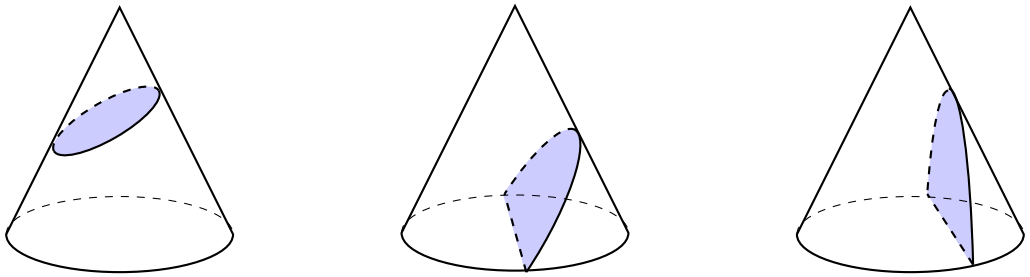
The terminology used to classify PDEs comes from that used to classify conic sections (Hoffman, 2001) (figure 1.1). Conic sections are described by the equation

$$ax^2 + bxy + cy^2 + dx + ey + f = 0, \tag{1.4}$$

and the type of curve represented by equation (1.4) is classified as follows

$$b^2 - 4ac \begin{cases} < 0, & \text{ellipse,} \\ = 0, & \text{parabola,} \\ > 0, & \text{hyperbola.} \end{cases}$$

It should be obvious that there are similarities between the types of PDEs and the different conic sections. However, there is no significance between the names of the types of PDEs and the behaviour of their solutions, they are simply names given to the different types and do not share any properties with the different conic sections.



(a) Ellipse (b) Parabola (c) Hyperbola

Figure 1.1: Conic sections

Example 2

For each of the PDEs given below, classify them as either an elliptic, parabolic or hyperbolic PDE.

(i) $U_{xx} + U_{yy} = 0$;
(ii) $U_t = kU_{xx}$;
(iii) $U_{tt} = c^2 U_{xx}$.

Solution:

(i) Comparing the PDE to equation (1.3) we have $a = 1$, $b = 0$ and $c = 1$ therefore

$$b^2 - 4ac = 0 - 4 = -4 < 0,$$

so this is an elliptic PDE.

(ii) Here we have $a = -k$, $b = 0$ and $c = 0$ therefore

$$b^2 - 4ac = 0 + 0 = 0,$$

so this is a parabolic PDE.

(iii) Letting $t = y$ we have $a = -c^2$, $b = 0$ and $c = 1$ therefore

$$b^2 - 4ac = 0 + c^2 > 0$$

for all $c \in \mathbb{R}$ so this is a hyperbolic PDE.

1.2.1 Elliptic PDEs

Elliptic PDEs are associated with steady state behaviour where the behaviour of the solution does not depend upon a time variable. A steady state problem is defined by the elliptic PDE and the boundary conditions that are applied to every point on the boundary. Any initial conditions defined in the domain will not affect the solution. Solutions to elliptic PDEs are smooth with no discontinuities. Examples of elliptic PDEs are Laplace's equation

$$U_{xx} + U_{yy} = 0,$$

and Poisson's equation

$$U_{xx} + U_{yy} = f(x, y),$$

which can be used to model magnetic, electrical and gravitational fields.

1.2.2 Parabolic PDEs

Parabolic PDEs are associated with problems that include the diffusion of some quantity across the domain, i.e., a propagation problem. Like elliptic PDEs, the solutions are smooth with no discontinuities but unlike elliptic PDEs, parabolic PDEs allow for transport to be modelled with the inclusion of the time variable. Therefore the solutions are not steady state problems and will evolve over time and require initial conditions that satisfy the PDE at the lower bound of the time domain.

An example of a parabolic PDE is the heat diffusion equation

$$U_t = \alpha U_{xx},$$

where α is some heat diffusion coefficient that determines the conductivity of a material. Since t is an independent variable the solution to this equation will evolve over time.

1.2.3 Hyperbolic PDEs

Hyperbolic PDEs are associated with propagation problems where some quantity or change in quantity travels across the domain. However, unlike parabolic PDEs where the solution at a point in the domain depends upon the solution at all other points in the domain, in the hyperbolic case the solution only depends upon points in the neighbourhood region. Examples of hyperbolic PDEs include the wave equation

$$U_{tt} = c^2 U_{xx},$$

where c is the speed of sound and is used in the modelling of acoustic waves and the advection equation

$$U_t + vU_x = 0,$$

where v is the advection velocity which is used to transport of a pollutant along a channel.

1.3 Domain of dependence

The differences between the three types of PDEs defined in section 1.2 can be illustrated by considering what is known as the *domain of dependence*. Let $U(t, x)$ be the solution of a PDE at a given point in space x and time$t$.

Definition 1: Domain of dependence

The domain of dependence is defined by the set of all points in the domain that influences the value of $U(t, x)$.

Definition 2: Range of influence

The *range of influence* is defined as the set of all points in the domain for which $U(t, x)$ is a member of the domain of dependence, i.e., the set of all points that the value of $U(t, x)$ influences.

Consider an elliptic PDE where the spatial domain is defined in the xy-plane with $a \leq x \leq b$ and $c \leq y \leq d$ (figure 1.2). As mentioned in section 1.2 elliptic PDEs have a steady state solution and do not change over time. As such, for elliptic PDEs the domain of dependence is the same as the range of influence and includes all points in the domain.

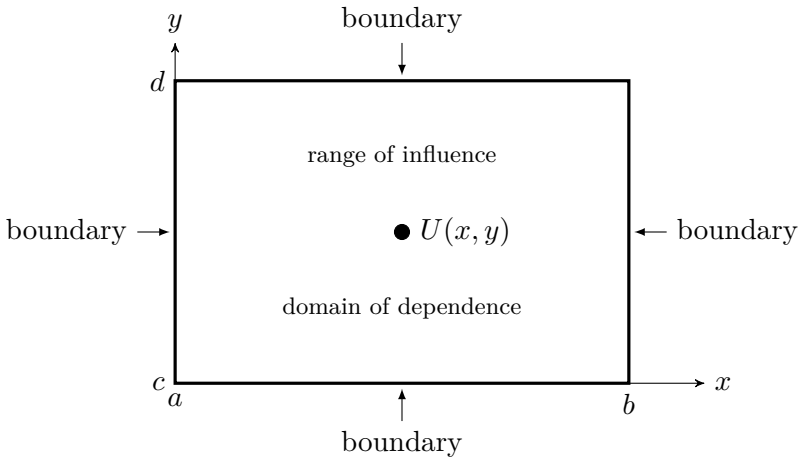


Figure 1.2: Domain of dependence and range of influence for an elliptic PDE.

The solution to a parabolic PDE must satisfy the PDE, the boundary conditions and, because parabolic PDEs include a time variable, the initial conditions for the PDE. Since parabolic PDEs model propagation, the solution $U(t, x)$ is dependent upon the solution at previous time values. Also, the solution across the spatial domain depends upon all other points at the same time level. Therefore the domain of dependence for a parabolic PDEs is bounded by the spatial boundaries $x = a$ and $x = b$ and the initial conditions at $t = t_{\min}$ (figure 1.3). The range of influence is the set of all points at later time levels up until $t = t_{\max}$.

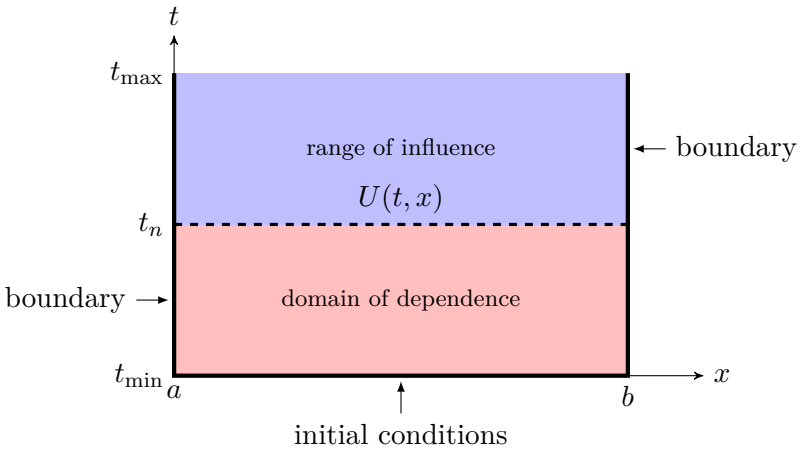


Figure 1.3: Domain of dependence and range of influence for a parabolic PDE.

Hyperbolic PDEs model propagation of a quantity across the spatial domain but unlike elliptic and parabolic PDEs, the solution of $U(t,x)$ is not influenced by all points in the spatial domain at once. For example, if a disturbance occurs at a particular point, the effect of the disturbance will propagate away from this point causing wave like structures. Only those points within the region of the propagating waves after a certain amount of time will be affected by the disturbance. Therefore the domain of dependence of a hyperbolic PDE is dependent upon the speed of propagation. Let c be the fastest propagation speed permitted by the hyperbolic PDE, then the domain of dependence is bounded by two lines called *characteristics* that extend back through time from $U(t,x)$ to the initial conditions at $t = t_{\min}$. The range of influence is also bounded by the characteristics and extends forward in time until $t = t_{\max}$ (figure 1.4).

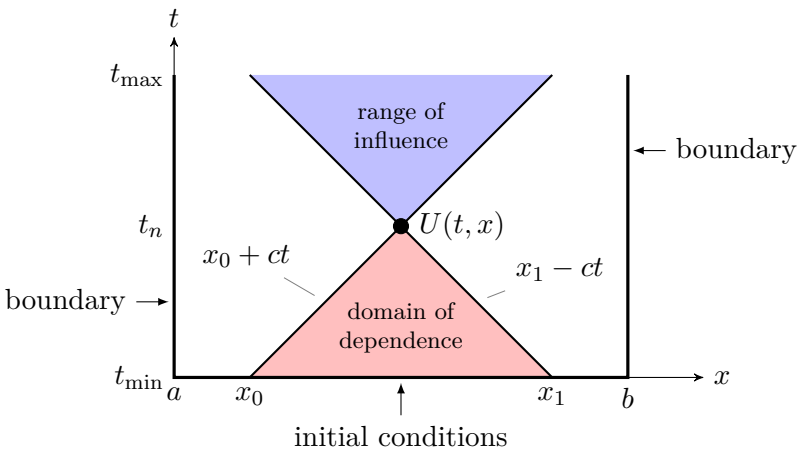


Figure 1.4: Domain of dependence and range of influence for a hyperbolic PDE.

1.4 Using PDEs to model real world phenomena

The methodology for using PDEs to model real world phenomena will depend on the problem or scenario that is being investigated, but the for most cases the following steps are used:

1. **Build the model** – an investigation will usually be prompted by a brief that outlines the purpose of the study and what questions need answering. In almost all cases, assumptions that simplify the development of the model are applied, for example, if modelling heat transfer of an object we could assume that the material has uniform heat conduction properties throughout. These assumptions should simplify the model but still retain the behaviour of the phenomena that is being modelled. The model can then be created using PDEs, or systems of PDEs where

the dependent variables provide information of the state of the system. In almost all instances, the PDEs that are used in modelling have already been derived previously and are then applied to the specific case.

Mathematical modelling usually begins with a very simplified approach that provides a basic model. This is then improved by including additional terms or equations that enable the model to include other properties of the system. For example, the modelling of water waves may begin with a model that can approximate the depth and velocity of the water over a flat bed surface, this can then be improved by including terms that model bed topography, bed friction, wave breaking etc.

2. **Derive the numerical scheme** – the numerical method chosen is based upon the PDE being solved, the geometry of the physical domain, the accuracy required, the computational resources available and the behaviour of the phenomena being modelled. There are several different methods currently available to do this, examples include finite-difference methods, finite-volume methods, finite-element methods, spectral methods and particle based methods. The complete numerical model may use a single method or combination of two or more methods.

3. **Write the program** – a computer program is written to perform the calculations of the numerical scheme. This can be time consuming and frustrating step. Do not expect to write a working program straight away, even experienced computer programmers rarely write an error-free programs on the first attempt. The process of writing a program should be done in steps, once a new part of the program is written it should be tested to see whether it performs as required. Only when you are satisfied that the new part performs as required you should proceed to write the next step in the program.

4. **Verify the program** – just because a program is working and produces an output does not mean that the output is what the numerical scheme should produce. Minor errors called 'bugs' in the program will affect the numerical solution and render the model useless. To verify that the program is performing as expected, the numerical solution is obtained using pen and paper (know as performing a 'hand calculation') and compared to the results from the program. Since performing the calculations by hand can be a time consuming process, the program is tested using a small number of solution points. Care should be taken to ensure that every variable and array calculated in the program matches that of the hand calculation. It is important that you know what every line in the program does and understand the structure of the program. It can be difficult to work with a program written by someone else but by stepping through the program line-by-line will give you an understanding of what it does (this is when a well commented program is useful).

5. **Validate the model** – once the program is performing as expected the next step is to validate the model by comparing the output against real world data, either from controlled experiments or real life measurements. Remember that a mathematical model is only an approximation of a real world scenario and the assumptions and compromises made in developing the model may mean that the model solutions may not exactly replicate the real world data. What is important is that the behaviour of what is being modelled is replicated.

6. **Present the model** – the final step in the modelling process is to present the model, the results and any conclusions that have been arrived at during the investigation. This usually takes the form of a report or a paper published in a journal or conference proceedings. It is important that the model is explained in enough detail so that the reader can reproduce the results for themselves and apply the model to other cases. This step can be daunting when first encountered and takes practice to get good at it. In this unit you will be asked to produce reports on work in the computer labs as part of formative assessment in addition to your summative coursework assessment. Guidance material on how to write mathematical reports is provided on Moodle.

1.4.1 Eulerian and Lagrangian methods

Consider the solution U to a particular PDE with the spatial domain Ω. If we were to use a numerical method to solve U then it stands to reason that we will need to calculate our solution at a finite number of discrete points in Ω else it would take forever to calculate the solution. Numerical solutions methods are classified into two types based on the method used to *discretise* the domain: *Eulerian* methods and *Lagrangian* methods.

Eulerian methods calculate the solution at points in the domain that have a fixed position (figure 1.5). The values of the variables in the PDE such as pressure, density, temperature and velocity etc. are calculated based on these fixed points. Examples of Eulerian methods include finite-difference, finite-volume and finite-element methods.

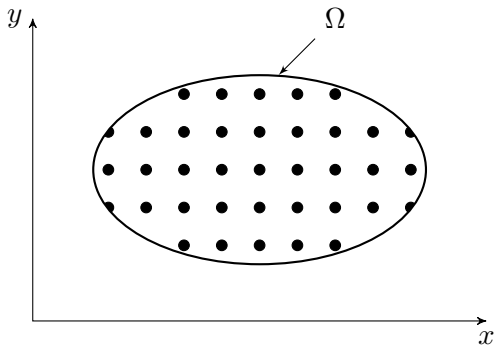


Figure 1.5: An Eulerian method calculates the solution at points in the domain whose positions are fixed.

Lagrangian methods calculate the solution at points that move around the domain based on the flow conditions of the solution (figure 1.6). The values of the variables such as pressure, density, temperature etc. are calculated at these points and the velocities are used to determine the position. Examples of Lagrangian methods, which are also known as *mesh free* methods include Smoothed Partial Hydrodynamics (SPH) and Moving Particle Semi-implicit (MPS) methods.

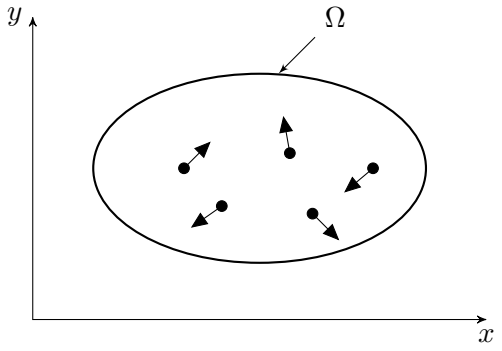


Figure 1.6: An Lagrangian method calculates the solution at points in the domain whose positions change depending on the flow conditions.

1.4.2 Exact solution versus numerical approximation

It should always be noted that when using a numerical method to calculate the solution of a PDE, we are not actually solving the PDE, rather we are calculating an approximation of the solution. Of course we aim to make our numerical approximation as close a possible to the exact solution. There are three factors that determine the accuracy of a numerical approximations

- The accuracy of the numerical method. There are many different methods that we can employ to solve a PDE with varying degrees of accuracy. In general, more accurate methods require

more complicated derivations and more computational resources. In practice we often reach a compromise between the accuracy of our chosen method and the effort required to derive an implement a more accurate method.

- The number of discrete points where we are calculating the solution. The more points we use to calculate a solution the more accurate our solution will be. We are limited to the number of points by the computational resources available.
- Computational rounding errors. There is a limit to the accuracy of a floating point operation (i.e., operations involving decimals) performed by a computer. Computational rounding errors can compound to affect the accuracy of a numerical solution.

Chapter 2

Finite-Difference Approximations

Finite-difference methods are one of the simplest and most common methods used to solve PDEs. Approximations of partial derivatives that use the difference of the function value at two or more points are derived from the well known Taylor series. The derivatives in the PDE are replaced by these difference approximations to form an algebraic system, the solution of which gives an approximation to the solution of the PDE. This chapter will introduce finite-differences, show how to derive finite-difference approximations and how we can use them to solve PDEs.

2.1 The Taylor series

The *Taylor series*, named after English mathematician Brook Taylor (1685–1731), is an infinite series expansion of a function written in terms of the derivatives of the function about a single point. Let $f(x)$ be a differentiable function and h be some small positive quantity then the Taylor series is written as

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} f_{(n)}(x),$$

where $f_{(n)}(x)$ denotes the n^{th} derivative of $f(x)$ with respect to x. Writing out the first few terms gives

Forward Taylor Series

$$f(x+h) = f(x) + hf_x(x) + \frac{h^2}{2!} f_{xx}(x) + \frac{h^3}{3!} f_{xxx}(x) + \frac{h^4}{4!} f_{(4)}(x) + \cdots \tag{2.1}$$

The quantity h is known as the *step length* since we are approximating the function f a small step along the number line from x. As h is a positive quantity then $x+h$ is said to be forward of x so equation (2.1) is colloquially known as the *forward Taylor series*. We can also approximate f at $x-h$ simply by replacing h by $-h$ in equation (2.1) to give

Backwards Taylor Series

$$f(x-h) = f(x) - hf_x(x) + \frac{h^2}{2!} f_{xx}(x) - \frac{h^3}{3!} f_{xxx}(x) + \frac{h^4}{4!} f_{(4)}(x) + \cdots \tag{2.2}$$

Since $x-h$ is a backwards step along the number line from x equation (2.2) is known as the *backwards Taylor series*. Note how in the backwards Taylor series the odd order terms have negative coefficients whilst the even order terms have positive coefficients, i.e.,

$$f(x-h) = \sum_{n=0}^{\infty} (-1)^n \frac{h^n}{n!} f_{(n)}(x).$$

2.1.1 Truncating the Taylor series

The Taylor series will give an exact values of $f(x+h)$ if we sum over an infinite number of terms. This is not possible in practice so we use the Taylor series to give an approximation of $f(x+h)$ by only including the first few terms in the series summation. In doing so we are *truncating* the Taylor series so that we omit the higher order terms. For example, truncating equation (2.1) after the first-order term gives

$$f(x+h) \approx f(x) + hf_x(x). \tag{2.3}$$

This is known as the *first-order forward Taylor series* since the highest order of the derivative in the series is a first-order derivative.

2.1.2 Truncation error

Truncating the Taylor series is useful as it greatly reduces the amount of computations we have to do, however we pay the price of a reduced accuracy in our approximation. We need to be able to gauge how inaccurate our series approximations are as a result of truncation and to do so we use the *truncation error* which is the error that is attributed to the omission of the higher order terms when truncating the Taylor series, i.e.,

$$f(x+h) = \text{series approximation} + \text{truncation error}.$$

In most cases we cannot calculate an exact value of the truncation error (if we could do that there would be no need to truncate the Taylor series in the first place) but we can examine the behaviour of the truncation error. Consider the truncation error for the first-order forward Taylor series

$$f(x+h) = \underbrace{f(x) + hf_x(x)}_{1^{\text{st}}\text{ order approximation}} + \underbrace{\frac{h^2}{2!}f_{xx}(x) + \frac{h^3}{3!}f_{xxx}(x) + \cdots}_{\text{truncation error}}.$$

Notice that the values of the denominators are factorials so the first term omitted will have a much larger value than any of the other terms and is therefore the dominant component of the truncation error. For example, consider the first-order Taylor series approximation of $\cos(0.1)$ using equation (2.3) with $x = 0$ and $h = 0.1$

$$\cos(0 + 0.1) \approx \cos(0) - 0.1\sin(0) = 1.$$

Since $\cos(0.1) = 0.995004$ (correct to 6 decimal places) then our truncation error is $|0.995004 - 1| = 0.004996$. The absolute value of the lowest order omitted term is

$$\left|\frac{h^2}{2!}\frac{\partial^2}{\partial x^2}\cos(x)\right| = \left|-\frac{0.1^2}{2!}\cos(0)\right| = 0.005,$$

so the error attributed to the other omitted terms is just $|0.004996 - 0.005| = 4 \times 10^{-6}$.

The truncation error is dominated by lowest order term omitted we can use this to express the truncation error as a polynomial function of h such that

$$f(x+h) = \text{series expansion} + E(h) + \text{higher order terms}.$$

For an n^{th} order Taylor series expansion

$$E(h) = \left|\frac{h^{n+1}}{(n+1)!}f_{(n+1)}(x)\right|.$$

$E(h)$ only includes the largest term from those omitted in the truncation of the Taylor series the actual truncation error will always be less than $E(h)$. Therefore we can say that

> "*As $h \to 0$ the truncation error will tend to zero at least as fast as $E(h)$.*"

The truncation errors for the first-order Taylor series approximation of $\cos(x+h)$ using values of $h = 0.5, 0.25, 0.125, 0.0625$ and the function $E(h)$ for a first-order approximation have been plotted in figure 2.1. Note is that as the value of the step length h decreases, so does the truncation error. This is an important result that applies to all numerical methods, i.e.,

> "*The smaller the step length, the more accurate the approximation.*"

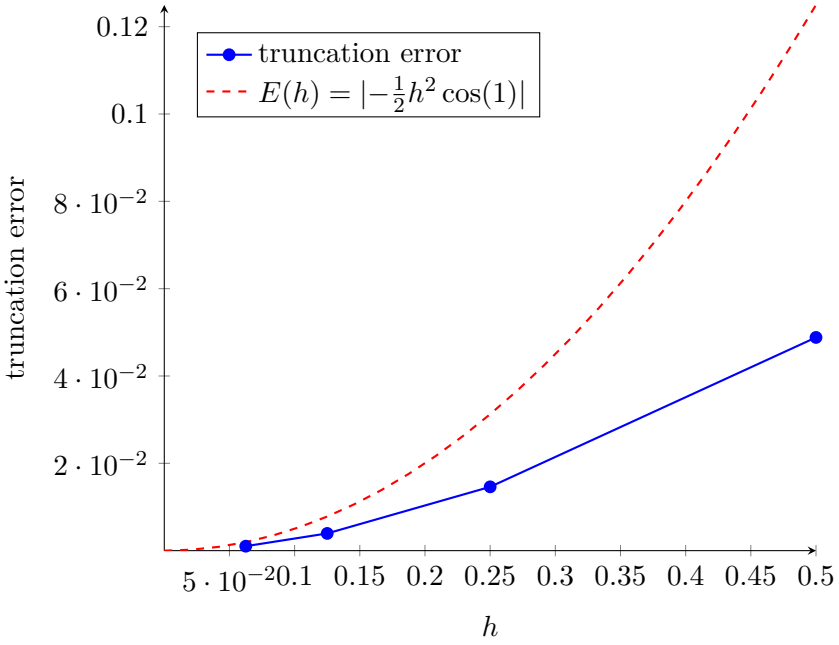


Figure 2.1: Plot of truncation errors for the first-order Taylor series expansion of $\cos(x+h)$ for $h = 0.5$, 0.25, 0.125 and 0.0625 compared with $E(h)$.

The truncation errors demonstrate quadratic behaviour as they tend to zero as h decreases. This is because the dominant term in the truncation error is a quadratic polynomial in terms of h. Finally the truncation errors are always less than the values of $E(h)$. In most practical cases it will not be possible to calculate the truncation error so we use $E(h)$ as a worst possible scenario to examine the behaviour of the truncation errors in numerical schemes.

2.2 "Big-oh" notation

We have seen in the previous section that instead of calculating a value for the truncation error, we examine the behaviour of the dominant term in the truncation error for different values of h. For example when comparing numerical methods, the method with a truncation error that tends to zero fastest as we decrease the step length will provide the more accurate approximation. Comparison of this convergence behaviour is done using "big-oh" notation.

Definition 3: $O(h^n)$

Let $f(h)$ be a function of h and define $f(h) = O(h^n)$ such that the following is satisfied

$$\lim_{h\to 0} \frac{f(h)}{h^n} = C, \tag{2.4}$$

where C is a non-zero positive constant.

Consider the case when h is small

$$\frac{f(h)}{h^n} \approx C$$
$$\therefore f(h) \approx Ch^n.$$

What this tells us is that since $C > 0$, the values of $f(h)$ will always be less than that of Ch^n and "as $h \to 0$, $f(h) \to 0$ at least as fast as $h^n \to 0$". If this is true then we can write $f(h) = O(h^n)$ and we say that "$f(h)$ is order h to the n". For example, consider the function $f(h) = 2h^3$, using equation (2.4)

$$\lim_{h\to 0} \frac{2h^3}{h^n} = C,$$

and since C is a scalar quantity n must be 3 in order to cancel out the h in the numerator so we can say that $2h^3 = O(h^3)$. Also consider $g(h) = \sin(h)$, using the series expansion of $\sin(h)$ we have

$$\lim_{h\to 0} \frac{\sin(h)}{h^n} = \lim_{h\to 0} \frac{h - \frac{h^3}{3!} + \frac{h^5}{5!} + \frac{h^7}{7!} + \cdots}{h^n} = C$$
$$\therefore h - \frac{h^3}{3!} + \frac{h^5}{5!} + \frac{h^7}{7!} + \cdots \approx Ch^n.$$

Since $h - \frac{h^3}{3!} + \frac{h^5}{5!} + \frac{h^7}{7!} + \cdots < h$ then $n = 1$ and we can say that $\sin(h) = O(h)$.

2.2.1 Expressing the truncation error as $O(h^{n+1})$

We use big-oh notation to denote the errors attributed to the omission of the higher order terms in the truncation of of the Taylor series. Let $E(h) = O(h^n)$ then the value of n will be one more than the order of the Taylor series expansion, e.g.,

$$f(x+h) = f(x) + hf_x(x) + \frac{h^2}{2!}f_{xx}(x) + \cdots + \frac{h^n}{n!}f_{(n)}(x) + O(h^{n+1}).$$

For example, a first-order Taylor series expansion, $n = 1$ and the truncation error is represented by $O(h^2)$, i.e.,

$$f(x+h) = f(x) + hf_x(x) + O(h^2),$$

and similarly for a second-order Taylor series

$$f(x+h) = f(x) + hf_x(x) + \frac{h^2}{2}f_{xx}(x) + O(h^3).$$

This means that for a first-order Taylor expansion, reducing the step length h by a factor of $\frac{1}{2}$ will result in the error being reduced by a factor of at least $h^2 = (\frac{1}{2})^2 = \frac{1}{4}$. For a second-order Taylor series expansion, reducing h by the same factor will result in the the error being reduced by a factor of at least $h^3 = (\frac{1}{2})^3 = \frac{1}{8}$.

2.2.2 Properties of $O(h^n)$

The following lists some useful properties of $O(h^n)$:

- Multiplying the truncation error by a scalar does not change the convergence behaviour of the truncation error, i.e.,

$$kO(h^n) = O(h^n).$$

- When summing series expansions of different orders, the truncation error of the lower order expansion dominates, i.e., if $m > n$

$$O(h^m) + O(h^n) = O(h^n).$$

- Dividing $O(h^n)$ by h^m where $m < n$ reduces the value of n by m, i.e.,

$$\frac{O(h^n)}{h^m} = O(h^{n-m}). \tag{2.5}$$

Proof. Assuming equation (2.5) is true, let $\left(\frac{f(h)}{h}\right)^m = O(h^n)$ and using equation (2.4)

$$\lim_{h\to 0} \frac{\frac{f(h)}{h^m}}{h^{n-m}} = \lim_{h\to 0} \frac{f(h)}{h^{n-m+m}} = \lim_{h\to 0} \frac{f(h)}{h^n} = C.$$

□

2.3 Deriving finite-difference approximations

Finite-differences approximate the derivatives of a function using the values of that function at equally spaced discrete points in the domain. For example consider the derivative of a function $f(x)$, we can approximate the value of the derivative at the point x by using the values of $f(x-h)$, $f(x)$ and $f(x+h)$.

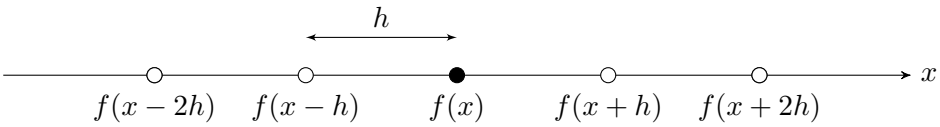


Figure 2.2: Neighbouring values are used to approximate the derivative of $f(x)$.

2.3.1 First-order accurate finite difference approximation of $f_x(x)$

The derivation of a finite-difference approximation is achieved by truncating the Taylor series and rearranging to make the desired derivative the subject of the equation. Consider the first-order forward Taylor series

$$f(x+h) = f(x) + hf_x(x) + O(h^2).$$

Rearranging to make $f_x(x)$ the subject gives

$$f_x(x) = \frac{f(x+h) - f(x)}{h} + \frac{O(h^2)}{h}.$$

Since $\frac{O(h^2)}{h} = O(h)$ by equation (2.5) then

First-order forward difference

$$f_x(x) = \frac{f(x+h) - f(x)}{h} + O(h). \tag{2.6}$$

Equation (2.6) is known as the first-order *forward* difference approximation of $f_x(x)$. Alternatively we can also use the first-order backwards Taylor series

$$f(x-h) = f(x) - hf_x(x) + O(h^2),$$

and doing similar gives the first-order *backward* difference approximation of $f_x(x)$

First-order backward difference

$$f_x(x) = \frac{f(x) - f(x-h)}{h} + O(h). \tag{2.7}$$

2.3.2 Second-order accurate finite-difference approximation of $f_x(x)$

Equations (2.6) and (2.7) are both first-order accurate approximations as denoted by $O(h)$. To construct a second-order accurate approximation we use the second-order forwards and backwards Taylor series

$$f(x+h) = f(x) + hf_x(x) + \frac{h^2}{2}f_{xx}(x) + O(h^3), \tag{2.8a}$$

$$f(x-h) = f(x) - hf_x(x) + \frac{h^2}{2}f_{xx}(x) + O(h^3). \tag{2.8b}$$

Since we wish to approximate $f_x(x)$ we need to eliminate the $f_{xx}(x)$ terms. This can be done by subtracting equation (2.8b) from (2.8a) to give

$$f(x+h) - f(x-h) = 2hf_x(x) + O(h^3),$$

which can be rearranged to give the second-order *central* difference approximation of $f_x(x)$.

Second-order central difference

$$f_x(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2). \tag{2.9}$$

2.3.3 Second-order accurate finite-difference approximation of $f_{xx}(x)$

To derive a finite-difference approximation of a higher order derivative we need to truncate the Taylor series to an order greater than the order of the derivative. For example, to derive a finite-difference approximation of $f_{xx}(x)$ we require the third-order forward and backward Taylor series

$$f(x+h) = f(x) + hf_x(x) + \frac{h^2}{2}f_{xx}(x) + \frac{h^3}{6}f_{xxx}(x) + O(h^4),$$
$$f(x-h) = f(x) - hf_x(x) + \frac{h^2}{2}f_{xx}(x) - \frac{h^3}{6}f_{xxx}(x) + O(h^4).$$

Here we want to approximation $f_{xx}(x)$ so we need to eliminate the $f_x(x)$ and $f_{xxx}(x)$ terms. Since these have different signs in the forward and backward Taylor series expansions we can do this by summing the two, i.e.,

$$f(x-h) + f(x+h) = 2f(x) + h^2 f_{xx}(x) + O(h^4)$$

Rearranging to make $f_{xx}(x)$ the subject we have

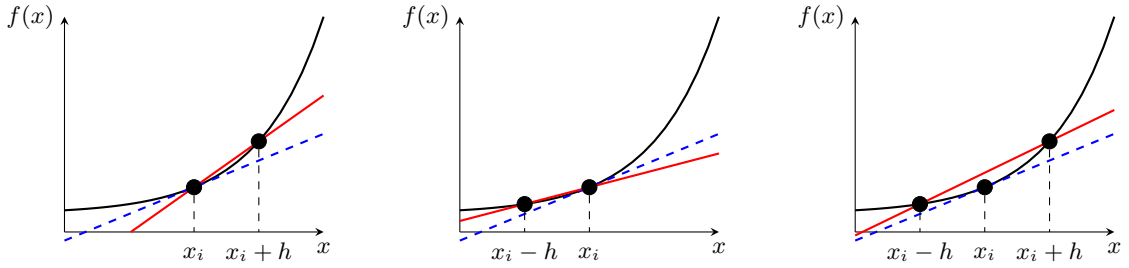
Second-order symmetric difference

$$f_{xx}(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2). \tag{2.10}$$

This is the second-order *symmetric* difference approximation of $f_{xx}(x)$. Note that when summing the forward and backwards Taylor series expansion the odd order terms cancel out. This means that using an odd order Taylor series expansion will result in an approximation that has an order of accuracy

one less than the order of expansion. For this reason, apart from the first-order approximations, the order of finite-difference approximations increase as even numbers.

The forward, backward and central difference formulae have been used to approximate the slope of the tangent to a function $f(x)$ in figure 2.3. Note that the forward and backward differences give a poor approximation of the tangent whereas the central difference is much more accurate. This is because the central difference is second-order accurate whereas the forward and backward differences are only first-order accurate.



(a) forward difference (b) backward difference (c) central difference

Figure 2.3: Difference approximations of the tangent slope of $f(x)$ at $x = x_i$ (dashed line).

Example 3

Use first and second-order accurate finite-differences to approximate the first and second derivative of $f(x) = \cos(x)$ at $x = \frac{\pi}{4}$ using a step length of $h = 0.1$. Calculate the absolute error between your approximations and the exact value.

Solution: Using the first-order forward difference from equation (2.6) we have

$$f_x\left(\frac{\pi}{4}\right) \approx \frac{\cos\left(\frac{\pi}{4}+0.1\right) - \cos\left(\frac{\pi}{4}\right)}{0.1} = -0.741255.$$

Using the first-order backward difference from equation (2.7) we have

$$f_x\left(\frac{\pi}{4}\right) \approx \frac{\cos(\left(\frac{\pi}{4}\right) - \cos\left(\frac{\pi}{4}-0.1\right)}{0.1} = -0.670603.$$

Using the second-order central difference formula from equation (2.9) we have

$$f_x\left(\frac{\pi}{4}\right) \approx \frac{\cos\left(\frac{\pi}{4}+0.1\right) - \cos\left(\frac{\pi}{4}-0.1\right)}{2(0.1)} = -0.705929.$$

The actual value is $f_x(\frac{\pi}{4}) = -\sin(\frac{\pi}{4}) = -0.707107$ (correct to 6 decimal places) so the error for the first-order forward approximation is 0.034148, the error for the first-order backward approximation is 0.036504 and the error for the second-order approximation is 0.001178.

The second-order approximation of $f_{xx}(x)$ is (a first-order approximation does not exist)

$$f_{xx}(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2},$$

so we have

$$f_{xx}\left(\frac{\pi}{4}\right) \approx \frac{\cos\left(\frac{\pi}{4}-0.1\right) - 2\cos\left(\frac{\pi}{4}\right) + \cos\left(\frac{\pi}{4}+0.1\right)}{0.1^2} = -0.706518,$$

which when compared to the actual value of $f_{xx}(\frac{\pi}{4}) = -\cos(\frac{\pi}{4}) = -0.707107$ (correct to 6 decimal places) gives an error of 0.000589.

2.3.4 Estimating the order of an approximation

Given a known truncation error it is possible to estimate the order of an approximation. We have seen earlier that for $E(h) = O(h^n)$

$$E(h) \approx Ch^n.$$

Taking logarithms of both sides gives

$$\log|E(h)| \approx n\log(h) + \log(C),$$

which is a linear function. Therefore we can approximate n in $O(h^n)$ by calculating the gradient of $\log|E(h)|$ for a series approximation for the different values of h, i.e., if $h_{\max}$ and $h_{\min}$ are the largest and smallest values of h for which the series approximation has been calculated then

$$n \approx \frac{\log|E(h_{\max})| - \log|E(h_{\min})|}{\log(h_{\max}) - \log(h_{\min})}.$$

For example consider the forward, backward and central difference approximations of the first derivative of $f(x) = \cos(x)$ at $x = \frac{\pi}{4}$ using step lengths $h = 0.1$, 0.05, 0.25 and 0.0125 shown in table 2.1.

Table 2.1: Finite-difference approximations of the first derivative of $f(x) = \cos(x)$ at $x = \frac{\pi}{4}$ using step lengths $h = 0.1$, 0.05, 0.25 and 0.0125.

h	forward	backward	central
0.1000	−0.741255	−0.670603	−0.705929
0.0500	−0.724486	−0.689138	−0.706812
0.0250	−0.715872	−0.698195	−0.707033
0.0125	−0.711508	−0.702669	−0.707088

The absolute errors between the finite-difference approximations and the exact value of $-\sin(\frac{\pi}{4}) = -0.707107$ are given in table 2.2.

Table 2.2: Truncation errors for the Taylor series approximations of $\cos(1+h)$ for $h = 0.5$, 0.25, 0.125 and 0.0625.

h	forward	backward	central
0.1000	0.034148	0.036504	0.001178
0.0500	0.017379	0.017969	0.000295
0.0250	0.008765	0.008912	0.000074
0.0125	0.004401	0.004438	0.000018

To estimate the order of the forward difference approximation we have

$$n \approx \frac{\log(0.034148) - \log(0.004401)}{\log(0.1) - \log(0.0125)} = 0.985305 \approx 1,$$

so we can say that the truncation error for the forward difference finite-difference approximation is $O(h)$. Doing similar for the backward and central difference approximations we get $n \approx 1.0134$ and $n \approx 1.9998$ which suggests truncation errors of $O(h)$ and $O(h^2)$ respectively.

A plot of the errors for the forward, backward and central difference approximations using a loglog scale is shown in figure 2.4. By using a loglog scale we can easily compare the behaviour of the errors as h decreases. Note that the forward and backward approximations have very similar errors

where as the central difference approximation is significantly more accurate. Furthermore the line representing errors in the central difference approximation has a gradient of 2 and is steeper than those of the forward and backward differences which have gradients of 1, this shows that as the step length decreases, the central difference approximation will converge to the exact value at a faster rate than the forward and backward difference approximations.

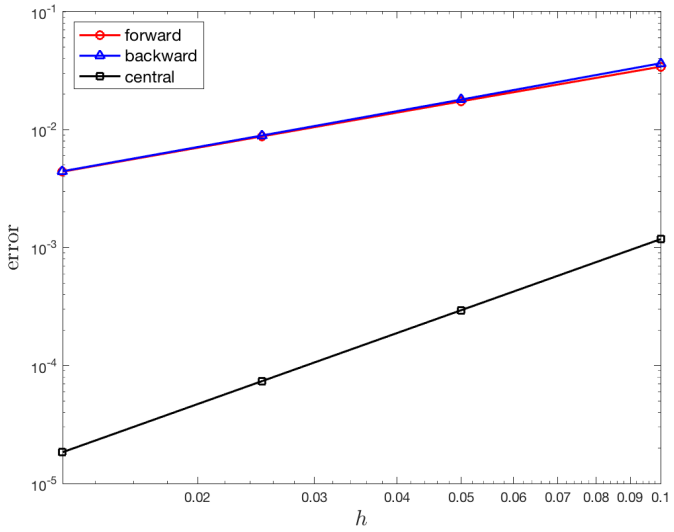


Figure 2.4: Plot of the error for the forward, backward and central difference approximations.

2.3.5 Finite-difference approximations of mixed derivatives

Finite-difference approximations of mixed derivatives where partial derivatives a function with respect to two or more independent variables are combined can be derived by applying finite-difference approximations of the function with respect to a single independent variable. For example consider f_{xy} where x and y are independent variables of f, we can write this as

$$f_{xy} = \frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right).$$

Let Δx and Δy be the step lengths for the x and y variables respectively and using a first-order forwards approximation for $f_x(x,y)$ and $f_y(x,y)$

$$f_x(x,y) = \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x} + O(\Delta x),$$
$$f_y(x,y) = \frac{f(x, y+\Delta y) - f(x,y)}{2\Delta y} + O(\Delta y),$$

then

$$\begin{aligned} f_{xy}(x,y) &= \frac{\partial}{\partial x}(f_y(x,y)) \\ &= \frac{f_y(x+\Delta x, y) - f_y(x,y)}{\Delta x} + O(\Delta x) \\ &= \frac{\dfrac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y)}{\Delta y} - \dfrac{f(x, y+\Delta y) - f(x,y)}{\Delta y}}{\Delta x} + O(\Delta x, \Delta y) \\ &= \frac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y) - f(x, y+\Delta y) + f(x,y)}{\Delta x \Delta y} + O(\Delta x, \Delta y) \end{aligned}$$

2.3.6 Deriving finite-difference formulae using the method of undetermined coefficients

Perhaps the best method for deriving finite-difference approximations of higher order derivatives is the method of *undetermined coefficients*. The approximation of a derivative is written as a sum of $f(x)$, $f(x-h)$, $f(x+h)$ etc. each multiplied by some coefficient. The value of these coefficients are determined by replacing $f(x+h)$, $f(x-h)$ etc. with the corresponding Taylor series expansion and then ensuring that the values coefficients match with the derivative that is being approximated.

For example, let us assume that we want a finite-difference approximation that uses the nodes $f(x)$, $f(x+h)$ and $f(x-h)$, i.e.,

$$f_{(n)}(x) = c_1 f(x-h) + c_2 f(x) + c_3 f(x+h), \tag{2.11}$$

where c_i are the undetermined coefficients. Substituting in the second-order forwards and backwards Taylor series in place of $f(x+h)$ and $f(x-h)$ and factorising gives

$$\begin{aligned} f_{(n)}(x) &= c_1 \left[f(x) - h f_x(x) + \frac{h^2}{2} f_{xx}(x) \right] + c_2 f(x) + c_3 \left[f(x) + h f_x(x) + \frac{h^2}{2} f_{xx}(x) \right] + O(h^3) \\ &= (c_1 + c_2 + c_3) f(x) + (-c_1 + c_3) h f_x(x) + \frac{1}{2}(c_1 + c_3) h^2 f_{xx}(x) + O(h^3). \end{aligned}$$

If we require a finite-difference approximation of $f_x(x)$ then we need the right-hand side to equal $f_x(x)$ so the coefficients of $f(x)$ and $f_{xx}(x)$ should be zero and the coefficient of $f_x(x)$ should be 1. This leads to the following linear system of equations

$$\begin{aligned} c_1 + c_2 + c_3 &= 0, \\ -c_1 + c_3 &= \frac{1}{h}, \\ c_1 + c_3 &= 0. \end{aligned}$$

The solution to this linear system is $c_1 = -\frac{1}{2h}$, $c_2 = 0$ and $c_3 = \frac{1}{2h}$. Substituting back into equation (2.11) gives the following finite-difference approximation

$$f_x(x) = -\frac{1}{2h} f(x-h) + \frac{1}{2h} f(x+h) + \frac{O(h^3)}{2h} = \frac{f(x+h) - f(x-h)}{2h} + O(h^2),$$

which is the same as equation (2.9). Alternatively if we required a finite-difference approximation of $f_{xx}(x)$ then the coefficients of $f(x)$ and $f_x(x)$ need to be zero and the coefficient of $f_{xx}(x)$ should be 1, i.e.,

$$\begin{aligned} c_1 + c_2 + c_3 &= 0, \\ -c_1 + c_3 &= 0, \\ c_1 + c_3 &= \frac{2}{h^2}. \end{aligned}$$

Solving gives $c_1 = \frac{1}{h^2}$, $c_2 = -\frac{2}{h^2}$ and $c_3 = \frac{1}{h^2}$ so the second-order finite-difference approximation of $f_{xx}(x)$ is

$$f_x(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2),$$

which is the same as equation (2.10).

The method of undetermined coefficients also allows us to derive skewed or one-sided finite-difference approximations which use more of the values of the neighbouring nodes to one side of $f(x)$ than the other. These can be useful for approximating derivatives close to the boundaries of the domain.

Example 4

Use the method of undetermined coefficients to derive a one-sided second-order finite-difference approximation of $f_x(x)$.

Solution: A one-sided finite-difference approximation means that the other nodes used in the approximation are only on one side of the $f(x)$ node, e.g.,

$$f_{(n)}(x) = c_1 f(x) + c_2 f(x+h) + c_3 f(x+2h).$$

Here we need the Taylor expansion of $f(x+2h)$ which is easily obtained by substituting $2h$ in place of h in the second-order forward Taylor series

$$\begin{aligned} f(x+2h) &= f(x) + 2hf_x(x) + \frac{(2h)^2}{2} f_{xx}(x) + O(h^3) \\ &= f(x) + 2hf_x(x) + 2h^2 f_{xx}(x) + O(h^3). \end{aligned}$$

Substituting in the second-order Taylor series for $f(x+h)$ and $f(x+2h)$ and factorising gives

$$\begin{aligned} f_{(n)}(x) &= c_1 f(x) + c_2 \left[f(x) + hf_x(x) + \frac{h^2}{2} f_{xx}(x) \right] + c_3 \left[f(x) + 2hf_x(x) + 2h^2 f_{xx}(x) \right] \\ &\quad + O(h^3) \\ &= (c_1 + c_2 + c_3) f(x) + (c_2 + 2c_3) h f_x(x) + \left(\frac{1}{2} c_2 + 2c_3 \right) h^2 f_{xx}(x) + O(h^3) \end{aligned}$$

For an approximation of $f_x(x)$ we need

$$\begin{aligned} c_1 + c_2 + c_3 &= 0, \\ c_2 + 2c_3 &= \frac{1}{h}, \\ \frac{1}{2} c_2 + 2c_3 &= 0, \end{aligned}$$

which can be written in matrix form as

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & \frac{1}{2} & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{h} \\ 0 \end{pmatrix}.$$

The inverse of the coefficient matrix is

$$\begin{pmatrix} 1 & -\frac{3}{2} & 1 \\ 0 & 2 & -2 \\ 0 & -\frac{1}{2} & 1 \end{pmatrix}$$

so the solution to this linear system is

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & -\frac{3}{2} & 1 \\ 0 & 2 & -2 \\ 0 & -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 0 \\ \frac{1}{h} \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{3}{2h} \\ \frac{2}{h} \\ -\frac{1}{2}h \end{pmatrix},$$

and the one-sided finite-difference approximation of $f_x(x)$ is

$$\begin{aligned} f_x(x) &= -\frac{3}{2h} f(x) + \frac{2}{h} f(x+h) - \frac{1}{2h} f(x+2h) + \frac{O(h^3)}{h} \\ &= \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} + O(h^2). \end{aligned}$$

2.4 The finite-difference toolkit

Finite-difference methods solve PDEs by replacing the partial derivatives with finite-difference approximations. The type and order of the finite-difference approximations used will depend upon the accuracy required and the computational resources available. We can build what is known as the *finite-difference toolkit* of common derivatives for convenience (see table 2.4) and we can pick and choose the approximations in order to suit our needs. Sometimes a non-symmetric finite-difference approximation is required, this usually happens near to the boundaries where the number of nodes each side of $f(x_i)$ may not be the same. In these cases we need to derive a finite-difference approximation using the method of undetermined coefficients (section 2.3.6).

We will consider PDEs written in terms of the function $f(t, x)$ where t usually denotes time and x denotes spatial positioning. If one of the independent variables is fixed, the partial derivative of $f(t, x)$ can be approximated using finite-difference approximations. For example, consider the approximation of $f_x(t, x)$ using the second-order central difference approximation in x with a step length of $h = \Delta x$ results in

$$\frac{\partial}{\partial x} f(t, x) = \frac{f(t, x + \Delta x) - f(t, x - \Delta x)}{2\Delta x},$$

or alternatively, $f_t(t, x)$ with step length $h = \Delta t$ is

$$\frac{\partial}{\partial t} f(t, x) = \frac{f(t + \Delta t, x) - f(t - \Delta t, x)}{2\Delta t}.$$

For convenience, a subscript/superscript notation is employed. Let the subscript i denote the spatial positioning and the superscript n denote the time level, i.e.,

$$f_i^n = f(t_n, x_i)$$

then we can write

$$\frac{\partial}{\partial x} f(t_n, x_i) = \frac{f_{i+1}^n - f_{i-1}^n}{2\Delta x},$$
$$\frac{\partial}{\partial t} f(t_n, x_i) = \frac{f_i^{n+1} - f_i^{n-1}}{2\Delta t}.$$

Finite-Difference Toolkit

Derivative	Order of accuracy	Finite-difference approximation
f_x	1^{st}	$\frac{f_{i+1}-f_i}{h}$ (forward) or $\frac{f_i-f_{i-1}}{h}$ (backward)
	2^{nd}	$\frac{f_{i+1}-f_{i-1}}{2h}$ (central)
	4^{th}	$\frac{f_{i-2}-8f_{i-1}+8f_{i+1}-f_{i+2}}{12h}$
f_{xx}	2^{nd}	$\frac{f_{i-1}-2f_i+f_{i+1}}{h^2}$ (symmetric)
	4^{th}	$\frac{-f_{i-2}+16f_{i-1}-30f_i+16f_{i+1}-f_{i+2}}{12h^2}$
f_{xxx}	2^{nd}	$\frac{-f_{i-2}+2f_{i-1}-2f_{i+1}+f_{i+2}}{2h^3}$
	4^{th}	$\frac{f_{i-3}-8f_{i-2}+13f_{i-1}-13f_{i+1}+8f_{i+2}-f_{i+3}}{8h^3}$
f_{xxxx}	2^{nd}	$\frac{f_{i-2}-4f_{i-1}+6f_i-4f_{i+1}+f_{i+2}}{h^4}$
	4^{th}	$\frac{-f_{i-3}+12f_{i-2}-39f_{i-1}+56f_i-39f_{i+1}+12f_{i+2}-f_{i+3}}{6h^4}$

2.5 Example of a finite-difference scheme

Now that we can approximate partial derivatives using our finite-difference toolkit the solution of a simple PDE is going to be used as an example. The PDE in question is known as the advection equation that models pure advection (the transport of a substance in space) is defined by

$$U_t + vU_x = 0. \tag{2.12}$$

The independent variables are t that denotes time and x that denotes space. The function $U = U(t, x)$ is the concentration of some substance and v is the velocity that the substance travels along the domain which is defined by $a \leq x \leq b$. In order to solve equation (2.12), we need the initial conditions that define the values of U at the start of the solution ($t = 0$), i.e.,

$$U(0, x) = f(x). \tag{2.13}$$

The solution of the advection equation is the function $U(t, x)$ which satisfies:

- equation (2.12) for all values of $a \leq x \leq b$ at time t;
- the initial conditions in equation (2.13)
- any conditions that are applied at the boundaries $x = a$ and $x = b$.

An exact solution is defined at an infinite number of values for the independent variables t and x. We will construct a Finite-Difference Scheme (FDS) to approximate U at a finite set of discrete points in t and x.

2.5.1 Spatial discretisation

The FDS solves a PDE at discrete points along the domain called *finite-difference nodes*. For simplicity it is common to locate the finite-difference nodes at equally spaced intervals across the domain creating what is known as a *uniform grid*. This process is called *discretising the domain*.

Consider the diagram in figure 2.5. Here, the x domain is the range $a \leq x \leq b$ and is discretised using a uniform grid of N nodes. The node at the left-hand boundary is given the label x_0 and we increment the subscript for each successive node until we get to node at the right-hand boundary that is given the label x_{N-1}. The first node in the finite-difference grid is located at $x_0 = a$, the second node is located at $x_1 = a + \Delta x$ where Δx is the *spatial step length* between nodes, therefore the i^{th} node in the grid is located at $x_i = a + i\Delta x$.

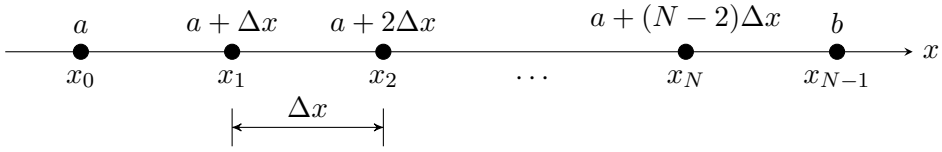


Figure 2.5: The x domain discretised using a uniform grid of finite-difference nodes

Using a uniform finite-difference grid means that the step length Δx and the number of nodes N are related. Since we require $x_{N+1} = b$ then

$$a + (N-1)\Delta x = b$$

which gives

$$\Delta x = \frac{b-a}{N-1}. \tag{2.14}$$

If however the spatial step length Δx is defined then it is easy to see that

$$N = 1 + \frac{b-a}{\Delta x}.$$

2.5.2 Deriving a finite-difference scheme

The next step is to derive a finite-difference scheme for the PDE. We do this by replacing the partial derivatives in the PDE with appropriate finite-difference approximations from our toolkit. The choice of a different finite-difference approximations will result in a different FDS and these depend on the order accuracy required. For example, if we want a first-order FDS then we can replace U_t and U_x in equation (2.12) with a forward and backward differences respectively, i.e.,

$$\begin{aligned} U_t + vU_x &= 0 \\ \frac{U_i^{n+1} - U_i^n}{\Delta t} + v\frac{U_i^n - U_{i-1}^n}{\Delta x} &= 0. \end{aligned}$$

We now rearrange this equation so that U_i^{n+1} is the subject

$$U_i^{n+1} = U_i^n - \frac{v\Delta t}{\Delta x}(U_i^n - U_{i-1}^n). \tag{2.15}$$

This is a FDS to solve the advection equation equation (2.12). Since the finite-difference approximations for derivatives in space and time are $O(\Delta x)$ and $O(\Delta t)$ respectively we can say that this FDS has formal accuracy of first-order in space and time. A FDS of the form of equation (2.15) is called a *time marching scheme* since the solution at the next time level, $n+1$, is calculated using values from

the current time level, n. Since $U(t,x)$ is only known exactly at the initial time level, equation (2.15) is rewritten replacing U_i^n with u_i^n to reflect that it is an approximation of the exact solution

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + O(\Delta t, \Delta x). \tag{2.16}$$

It is of interest to know which nodes are required to calculate the value of u_i^{n+1} for a given FDS. In equation (2.16) the calculation of the value of the node u_i^{n+1} is dependent upon two nodes at the current time level u_i^n and the neighbouring node to the left u_{i-1}^n. This information can be represented in a diagram known as a *finite-difference stencil* shown in figure 2.6.

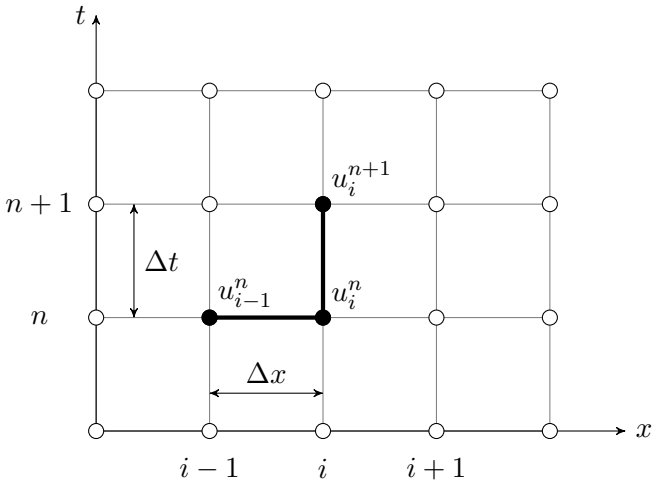


Figure 2.6: The stencil the first-order finite-difference scheme, equation (2.16), used to solve the advection equation.

2.5.3 Computational and boundary nodes

Looking at the finite-difference stencil in figure 2.6 for the FDS we will encounter a problem when trying to evaluate u_0^{n+1} since it requires the value of the node u_{-1}^n which doesn't exist. To counter this problem we only apply the FDS to the *computational nodes* that can be evaluated, i.e., $u_1^n, \cdots, u_{N-2}^n$. The *boundary node*, u_0^n, is removed from the calculation of the FDS and its value is determined using some boundary condition that is implemented at the boundary (boundary conditions are discussed in section 3.2 on page 32). The FDS used here only encounters a problem at the left-hand boundary, however if we were to use a forward or central discretisation U_x then we would also encounter this problem at the right-hand boundary. Although it may not always be necessary, it is common practice to use boundary nodes at either end of the spatial domain (figure 2.7). For this example we will assume the boundary conditions always have a value of zero, i.e.,

$$U(t,a) = U(t,b) = 0.$$

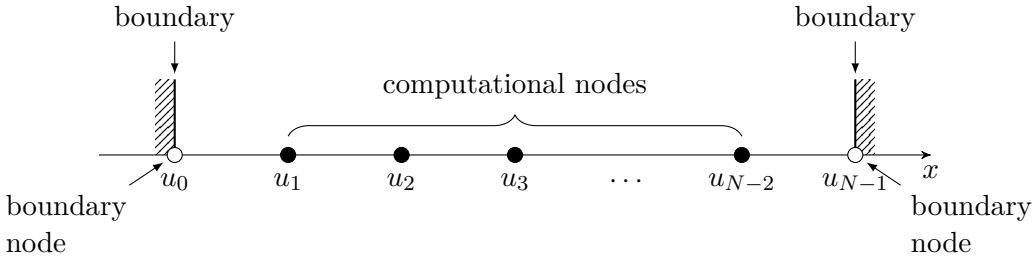


Figure 2.7: The boundary nodes are removed from the calculation of the FDS.

2.5.4 Hand calculation

To demonstrate how the FDS in equation (2.16) is used to calculate an approximation of the solution to the PDE in equation (2.12) we are going to calculate the values at the finite-difference nodes by hand*. This is a time consuming and labourious exercise and is not recommended practice for solving PDEs numerically. It is done here to help understand the process and can be useful for verifying and debugging programs.

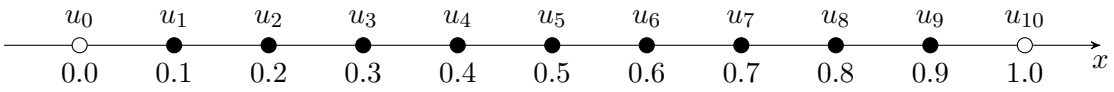
For this example we are going to use a spatial domain of $0 \leq x \leq 1$ which will be discretised using 11 nodes. The velocity is set at $v = 1$ and the time step is $\Delta t = 0.05$. First we calculate the spatial step using equation (2.14)

$$\Delta x = \frac{1-0}{11-1} = 0.1.$$

Using $x_i = a + i\Delta x$ then the co-ordinates of the finite-difference nodes are

$$\mathbf{x} = (0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0),$$

i.e.,



The initial conditions are given by the following exponential function

$$U(0, x) = \exp[-100(x - 0.4)^2], \tag{2.17}$$

and the boundary conditions are

$$U(t, 0) = 0,$$
$$U(t, 1) = 0.$$

Calculating the initial values of the first few nodes gives

$$\begin{aligned} u_0^0 &= 0, \qquad \text{(boundary node)}, \\ u_1^0 &= \exp(-100(0.1 - 0.4)^2) = 0.0001, \\ u_2^0 &= \exp(-100(0.2 - 0.4)^2) = 0.0183, \\ u_3^0 &= \exp(-100(0.3 - 0.4)^2) = 0.3679, \\ &\vdots \end{aligned}$$

For this FDS, Δt, Δx and v are constant so we can pre-calculate $\frac{v\Delta t}{\Delta x}$ in order to save time later on. It is good practice to pre-calculate any values that you know will not change over the course of applying a numerical scheme

$$\frac{v\Delta t}{\Delta x} = \frac{1 \times 0.05}{0.1} = 0.5,$$

so the FDS is

$$u_i^{n+1} = u_i^n - 0.5(u_i^n - u_{i-1}^n).$$

Calculating the solution at the first time step for the first few nodes

$$\begin{aligned} u_1^1 &= u_1^0 - 0.5(u_1^0 - u_0^0) = 0.0001 - 0.5(0.0001 - 0) = 0.0001, \\ u_2^1 &= u_2^0 - 0.5(u_2^0 - u_1^0) = 0.0183 - 0.5(0.0183 - 0.0001) = 0.0092, \\ u_3^1 &= u_3^0 - 0.5(u_3^0 - u_2^0) = 0.3679 - 0.5(0.3679 - 0.0183) = 0.1931, \end{aligned}$$

The values of the first three iterations of the FDS are given in table 2.3.

*When we say we are going to perform a hand calculation it means that we are going to perform each calculation using pen and paper and the help of a calculator.

Table 2.3: Values of the first three iterations of the FDS used to solve the advection equation

i	0	1	2	3	4	5	6	7	8	9	10
x_i	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
u_i^0	0.0000	0.0001	0.0183	0.3679	1.0000	0.3679	0.0183	0.0001	0.0000	0.0000	0.0000
u_i^1	0.0000	0.0001	0.0092	0.1931	0.6839	0.6839	0.1931	0.0092	0.0001	0.0000	0.0000
u_i^2	0.0000	0.0000	0.0046	0.1012	0.4385	0.6839	0.4385	0.1012	0.0046	0.0000	0.0000
u_i^3	0.0000	0.0000	0.0023	0.0529	0.2698	0.5612	0.5612	0.2698	0.0529	0.0023	0.0000

2.5.5 Solution procedure

The solution procedure for solving the advection equation FDS given in equation (2.15) is summarised by the flow chart shown in figure 2.8. Flow charts are useful for understanding the procedures and processes involved when using numerical methods. Here we can see that following the initialisation of the variables and arrays, a loop is required to iterate the FDS through the number of time steps. Within this loop the FDS is used to calculate $\mathbf{u}^{n+1}$ (this requires another loop to loop through all of the computational nodes to calculated the updated solution), $\mathbf{u}^n$ and t are updated ready for the next time step and the current solution is outputted.

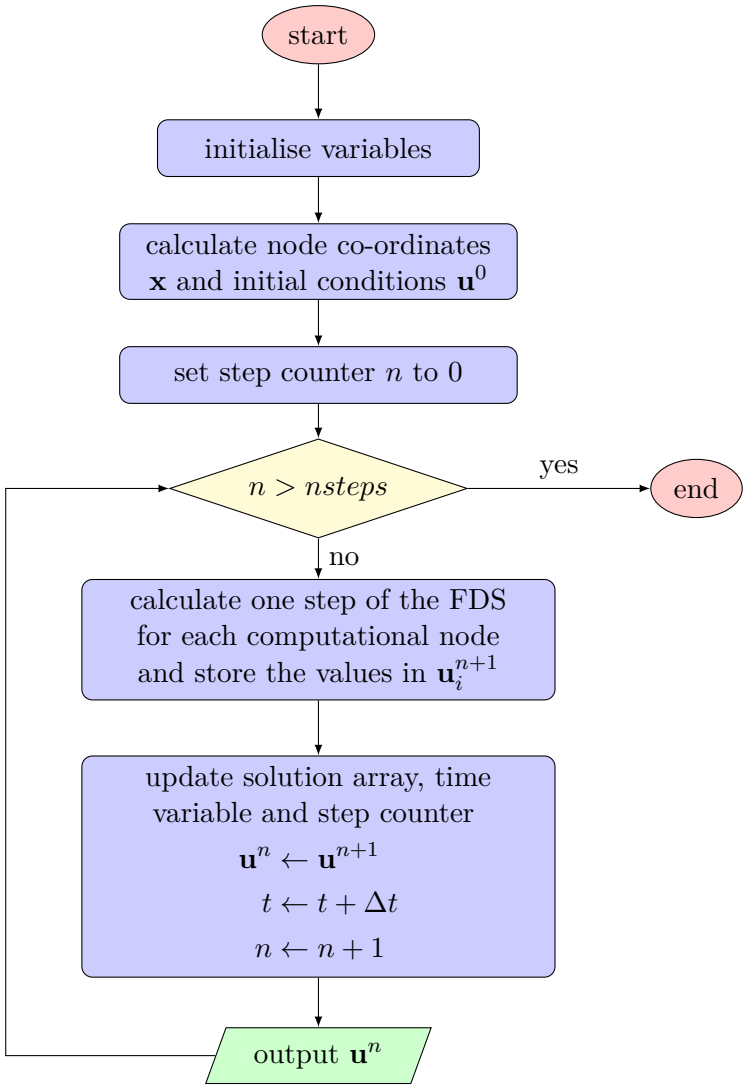


Figure 2.8: Flow chart for the FDS used to solve the advection equation.

2.5.6 MATLAB code

It should be obvious to the reader that calculating the solution to a PDE using a FDS is not usually done by hand. There are far too many calculations required which means there are more opportunities to make mistakes and it soon becomes quite tedious. Performing the calculations using a computer program saves time and effort and changes made to the problem can be implemented quickly and with the minimum of fuss. We can use the flow chart in 2.8 to help in writing a MATLAB program to perform the calculations. The MATLAB code shown in listing 2.1 was used to calculate the values in table 2.3.

Listing 2.1: MATLAB program to calculate the solution of the advection equation using the first-order FDS in equation (2.16).

```
% fds_example.m by Jon Shiach
%
% This program solves the linear advection equation using a first-order FDS

% Clear workspaces (should always be done at the start of an m-file)
clear
clc

% Define variables
N = 11;                 % number of nodes
xmin = 0;               % min value of x
xmax = 1;               % max value of x
dt = 0.05;              % time step
v = 1;                  % velocity
t = 0;                  % initial value of t

% Calculate node positions
dx = (xmax - xmin) / (N - 1);
x = xmin : dx : xmax;

% Calculate initial conditions
u = exp(-100 * (x - 0.4).^2);

% Output column headings and initial conditions to the command window
hline = repmat('-', 1, 9 * (N + 1));
fprintf('\n%s', hline)
fprintf('\n    t    ')
fprintf('|   u_%1i   ', [0:N-1])
fprintf('\n%s', hline)
fprintf('\n %6.2f ', t)
fprintf('|%7.4f ', u)

% pre-calculate C = v * dt / dx
C = v * dt / dx;

% Perform 3 iterations of the FDS
for n = 1 : 3

    % Calculate boundary conditions
    u(1) = 0;
    u(N) = 0;

    % Calculate new values of u
    unew = u;
    for i = 2 : N - 1
        unew(i) = u(i) - C * (u(i) - u(i-1));
    end

    % Update t and u
    t = t + dt;
    u = unew;

    % Output current solution
    fprintf('\n %6.2f ',t)
    fprintf('|%7.4f ',u)
end
fprintf('\n%s\n\n', hline)
```

The MATLAB program shown in listing 2.1 is explained below:

- Lines 6 and 7 – since this is an m-file it has access to any previously defined variables or arrays which might cause errors in the program, therefore it is always a good idea to clear the memory using `clear` and the command window using `clc`. Note that this step isn't necessary if you have used a function instead of an m-file.
- Lines 10 to 15 – defines all of the variables used in the program at once in the beginning. It is good practice to use comments to explain what each variable represents.
- Lines 18 and 19 – calculates the spatial step and uses an implicit for loop to generate an array of `x` values going from `xmin` to `xmax` in steps of `dx`
- Line 22 – calculates the initial values of u_i^0 using equation (2.17) and stores them in the array `u`.
- Lines 25 to 31 – outputs the column headings using formatted output. This is purely cosmetic and not really necessary of the successful calculation of the solution, however sometimes it is useful to be able to output the solution in a readable format.
- Line 34 – The value of $\frac{v\Delta t}{\Delta x}$ is pre-calculated since it is constant throughout. This should be done for all values that do not change over the iterations since computational effort is not wasted calculating the values when they are already known.
- Line 37 – a `for` loop is used to calculate 3 iterations of the FDS. The variable `n` is the iteration counter.
- Lines 40 and 41 – the values of the boundary nodes are calculated using the boundary conditions for the current problem.
- Lines 44 to 47 – a `for` loop is used to loop through the nodes and calculate the value of u_i^{n+1} using the FDS in equation (2.16). Since array indices in MATLAB begin at 1 rather that 0, the first computational node u_1 is the second node in `u` therefore we add 1 to the array indices, i.e., `i = 2 : N - 1` in the code represents the computational nodes $i = 1, \cdots, N-2$ in the finite-difference discretisation.
- Lines 50 and 51 – `u` is updated by setting it equal to `unew` ready for the next iteration. `t` is also updated by incrementing by `dt`, although not used in this program it is useful to keep track of the t value for output purposes.

Chapter 3

Elliptic Partial Differential Equations

Elliptic PDEs are used to model problems where the independent variables are restricted to spatial co-ordinates only. Since time is not a variable the solution of an elliptic PDE is constant and referred to as *steady state solutions*. Examples of elliptic equations are Laplace's equation and Poisson's equation that are used to model electrical and gravitational fields. Laplace's equation is the simplest elliptic PDE and was derived by French mathematician Pierre-Simon Laplace (1749–1827). The class of analytical solutions to Laplace's equation are called *harmonic functions* which have applications in many fields of science such as magnetism, astronomy and fluid dynamics. Laplace's equation is

Laplace's equation

$$U_{xx} + U_{yy} = 0, \tag{3.1}$$

where x and y are the spatial co-ordinates. Poisson's equation named after French mathematician Siméon Denis Poisson (1781–1840) is an extension of Laplace's equation

Poisson's equation

$$U_{xx} + U_{yy} = f(x, y), \tag{3.2}$$

where $f(x, y)$ is some function of x and y. This chapter will focus on the solution of Laplace's equation using finite-difference methods and introduces the reader to the discretisation of a two-dimensional spatial domain, calculation of boundary conditions and solving linear systems.

3.1 Discretising a two-dimensional domain

Since Laplace's equation is written in terms of x and y we need to discretise both spatial dimensions to produce a two-dimensional grid. If the spatial domain is defined by $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$. N_X and N_Y nodes are used to discretise the domain in the x and y directions respectively (the subscripts X and Y have been written in uppercase to avoid confusing it with first-order partial derivatives), then the spatial steps are

$$\Delta x = \frac{x_{\max} - x_{\min}}{N_X - 1},$$
$$\Delta y = \frac{y_{\max} - y_{\min}}{N_Y - 1}.$$

The position of the nodes in the finite-difference grid are the co-ordinates (x_j, y_i) where

$$x_i = x_{\min} + i\Delta x,$$
$$y_j = y_{\min} + j\Delta y.$$

To identify individual nodes in a two-dimensional finite-difference grid a subscript notation is employed where the index i denotes the position of the node in the x direction and j denotes the position of the node in the y direction, i.e., $u_{i,j}$ (figure 3.1)*.

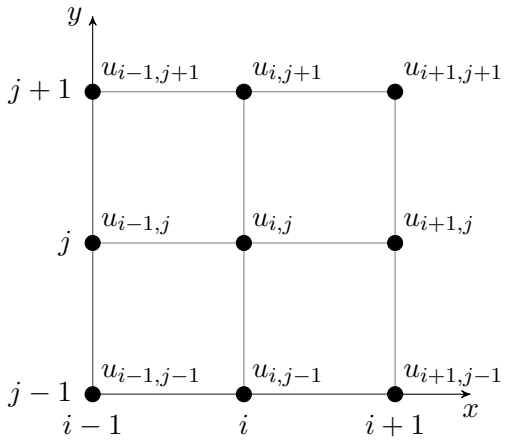


Figure 3.1: Node numbering convention for a two-dimensional finite-difference grid.

3.2 Boundary conditions

We saw in section 2.5.3 that it may not be possible to calculate the values of the nodes on the boundaries using a particular FDS. When this is the case, these boundary nodes are omitted from the calculation of the FDS and assume some value that depends on the behaviour of the system at the boundary. Determining the value of the boundary nodes is done by invoking *boundary conditions*. These notes will discuss two types of boundary conditions: Dirichlet and Neumann boundary conditions.

3.2.1 Dirichlet boundary conditions

Dirichlet boundary conditions are named after German mathematician Peter Gustav Lejeune Dirichlet (1805–1859) and specify the value of U at the boundary of the solution domain. This is useful in cases where the points on the boundary are not influenced by the behaviour of the points within the domain. Dirichlet boundary conditions are applied to the boundary nodes that are positioned on the boundaries, i.e., u_0 and u_{N-1}, and the FDS are applied to the computational nodes u_1 to u_{N-2} (figure 3.2).

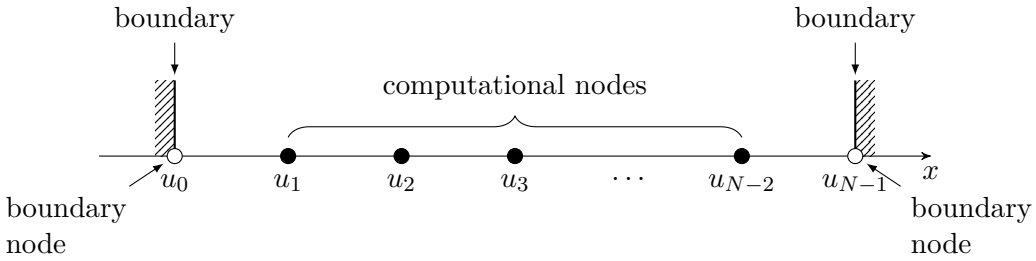


Figure 3.2: When using Dirichlet boundary conditions the nodes on the boundaries are excluded from the computation and determined by some known scalar or function.

The are three main uses for Dirichlet boundary conditions:

*Be careful to avoid confusing the node indexing convention used for finite-differences grids with the indexing used for matrices.

- Values on the boundaries are constant, e.g.,

$$u_0 = \alpha,$$
$$u_{N-1} = \beta,$$

  where u_0 and u_N denotes the boundary nodes and α and β are constants. This is used in cases where the solution on the boundary is known and constant. For example, modelling heat diffusion across a domain where the boundary temperature is always zero.

- Values on the boundaries adhere to some function, e.g.,

$$u_0 = f(t, x, y),$$
$$u_{N-1} = g(t, x, y),$$

  where $f(t, x, y)$ and $g(t, x, y)$ are some functions of the independent variables. These are used in cases where the solution at the boundary is not constant but known to behave in a certain way. For example, modelling wave propagation where the boundary condition is used to generate waves passing into the domain.

- Boundaries are used to model periodic behaviour, e.g.„

$$u_0 = u_{N-2},$$
$$u_{N-1} = u_1,$$

  where u_1 and u_{N-2} are the values of the first and last computational nodes (figure 3.3). This is used where we want to model an infinite domain. For example, modelling wave propagation where waves passing out through the right-hand boundary will pass in from the left boundary.

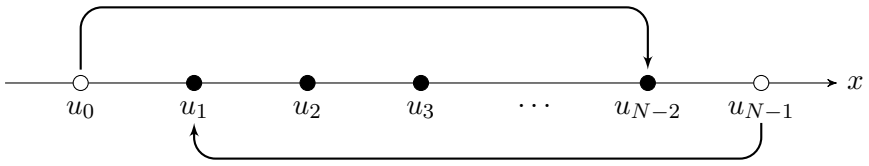


Figure 3.3: Periodic boundary conditions

3.2.2 Neumann boundary conditions

Neumann boundary conditions are named after German mathematician Karl Gottfried Neumann (1832–1925) and specifies the derivative of the solution across the boundary. Boundary conditions of this type are used where the points on the boundaries are influenced by the behaviour in the domain. When using Neumann boundary conditions, additional nodes called *ghost nodes* are added to each end of the domain so that we can calculated finite-difference approximations for the nodes on the boundary (u_{-1} and u_{N+1} in figure 3.4) and the FDS is applied to the computational nodes u_0 to u_N.

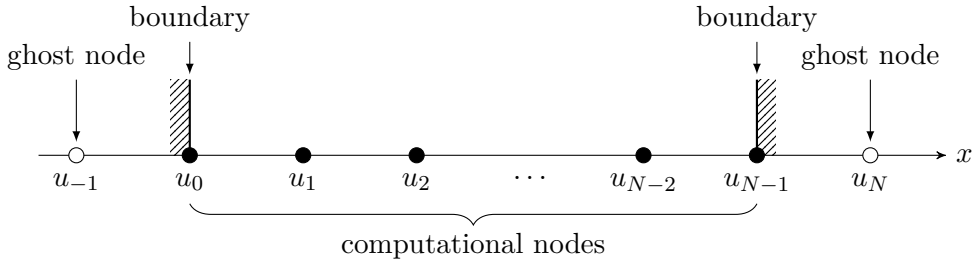


Figure 3.4: Neumann boundary conditions use the derivative across the boundary to determine the values of the ghost nodes.

To specify a Neumann boundary condition we need to approximate the derivative across the boundary using a finite-difference. Consider the Neumann boundary condition

$$U_x = f(U),$$

where $f(U)$ is a function of U. Approximating U_x using a central difference we have

$$\begin{aligned}\frac{u_1 - u_{-1}}{2\Delta x} &= f(U)\\ u_{-1} &= u_1 - 2\Delta x f(U),\end{aligned}$$

so the value of the ghost node u_{-1} is an expression involving the second computational node u_1 and $f(U)$. There are two main uses for Neumann boundary conditions:

- Transmissive boundary conditions: the derivative across the boundary is zero, i.e., $f(U) = 0$ so

$$\begin{aligned}u_{-1} &= u_1,\\ u_N &= u_{N-2}.\end{aligned}$$

  This has the effect that any flow will simply pass out of the boundary as if it wasn't there. For example, consider the modelling of water waves down a channel, by specifying a transmissive boundary at one end the waves will simply pass through as if the channel had an infinite length.

- Reflective (solid) boundary conditions: no flow through a reflective boundary. This used where one of the dependent variables of the PDE is velocity and the velocity of the ghost node is set to equal that of the velocity of a computational node with the sign reversed, e.g.,

$$\begin{aligned}v_{-1} &= -v_1,\\ v_N &= -v_{N-2},\end{aligned}$$

  where v_i is the velocity at node i.

3.3 Deriving a FDS to solve Laplace's equation

We will now proceed to derive a FDS to solve Laplace's equation. In equation (3.1) there are two second-order partial derivatives that need approximating. Looking at the finite-difference toolkit (page 23) we can use the second-order symmetric finite-difference approximations

$$\begin{aligned}U_{xx} &\approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2},\\ U_{yy} &\approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}.\end{aligned}$$

The FDS is derived by replacing the derivatives in equation (3.1) by the finite-difference approximations

$$\begin{aligned}U_{xx} + U_{yy} &= 0\\ \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} &= 0.\end{aligned}$$

We want an expression that gives the solution for the node $u_{i,j}$, therefore rearranging to make $u_{i,j}$ the subject results in

Second-order finite-difference scheme to solve Laplace's equation

$$u_{i,j} = \frac{\Delta y^2(u_{i-1,j} + u_{i+1,j}) + \Delta x^2(u_{i,j-1} + u_{i,j+1})}{2(\Delta x^2 + \Delta y^2)}. \tag{3.3}$$

This is a second-order accurate FDS that approximates the solution to Laplace's equation. The stencil for this FDS is shown in figure 3.5.

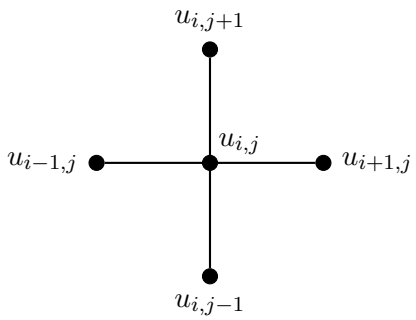


Figure 3.5: The finite-difference stencil for the second-order FDS used to solve Laplace's equation

3.4 Solving Laplace's equation

In order to demonstrate the solution of Laplace's equation using finite-differences we are going to use a simple example with a coarse grid shown in figure 3.6. The domain is $0 \leq x \leq 4$ and $0 \leq y \leq 3$ has been discretised using 5 nodes in the x direction and 4 nodes in the y direction so that $\Delta x = \Delta y = 1$. Dirichlet boundary conditions are implemented at all four boundaries so that the values of the boundary nodes are constant. We need to calculate the solution to Laplace's equation using the FDS in equation (3.3) for the six computational nodes $u_{1,1}$, $u_{2,1}$, $u_{3,1}$, $u_{1,2}$, $u_{2,2}$ and $u_{3,2}$.

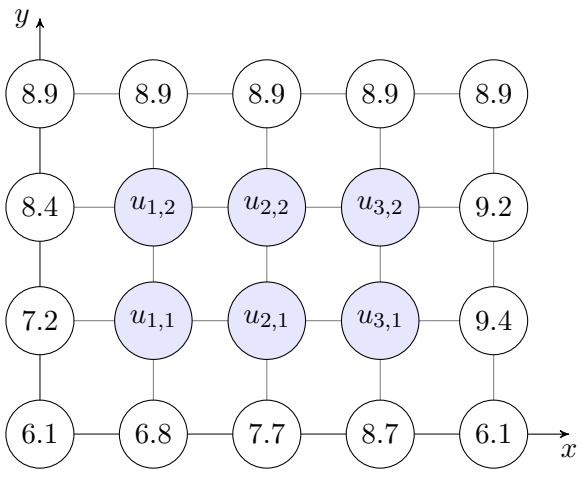


Figure 3.6: Finite-difference grid used to solve Laplace's equation.

Using equation (3.3) and considering the computational nodes going from left-to-right along each row

and up the rows from top-to-bottom we have

$$\begin{aligned}
u_{1,1} &= \frac{7.2 + u_{3,1} + 6.8 + u_{2,1}}{4}, \\
u_{2,1} &= \frac{u_{1,1} + u_{3,1} + 7.7 + u_{2,2}}{4}, \\
u_{3,1} &= \frac{u_{2,1} + 9.4 + 8.7 + u_{3,2}}{4}, \\
u_{1,2} &= \frac{8.4 + u_{2,2} + u_{1,1} + 8.9}{4}, \\
u_{2,2} &= \frac{u_{1,2} + u_{3,2} + u_{2,1} + 8.9}{4}, \\
u_{3,2} &= \frac{u_{2,2} + 9.2 + u_{3,1} + 8.9}{4}.
\end{aligned}$$

These equations can be written as the linear system by moving all of the unknown values to the left-hand side and the known values to the right-hand side

$$\begin{aligned}
4u_{1,1} - u_{2,1} - u_{1,2} &= 14, \\
-u_{1,1} + 4u_{2,1} - u_{3,1} - u_{2,2} &= 7.7, \\
-u_{2,1} + 4u_{3,1} - u_{3,2} &= 18.1, \\
-u_{1,1} + 4u_{1,2} - u_{2,2} &= 17.3, \\
-u_{2,1} - u_{1,2} + 4u_{2,2} - u_{3,2} &= 8.9, \\
-u_{3,1} - u_{2,2} + 4u_{3,2} &= 18.1,
\end{aligned}$$

or alternatively using matrix notation of the form $A\mathbf{u} = \mathbf{d}$ such that

$$\begin{pmatrix}
4 & -1 & 0 & -1 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 \\
-1 & 0 & 0 & 4 & -1 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & -1 & 0 & -1 & 4
\end{pmatrix}
\begin{pmatrix}
u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2}
\end{pmatrix}
=
\begin{pmatrix}
14 \\ 7.7 \\ 18.1 \\ 17.3 \\ 8.9 \\ 18.1
\end{pmatrix}.$$

This linear system can be solved using *direct* methods such matrix inversion or Gaussian elimination but these are inefficient for large systems. Instead we can use *indirect* methods which use an iterative process where an initial estimate of the solution at the nodes $\mathbf{u}$ is improved to form a better estimate. This estimate in turn is improved and the process continues until two successive estimates agree to a required accuracy.

3.5 The Jacobi method

The simplest indirect method is called the *Jacobi method* after German mathematician Carl Gustav Jacob Jacobi (1804 – 1851). If $u_{i,j}^k$ denotes the current estimate of $U_{i,j}$ and $u_{i,j}^{k+1}$ denotes the improved estimate then using equation (3.3) we have

Jacobi method for computing the solution to Laplace's equation

$$u_{i,j}^{k+1} = \frac{\Delta y^2(u_{i-1,j}^k + u_{i+1,j}^k) + \Delta x^2(u_{i,j-1}^k + u_{i,j+1}^k)}{2(\Delta x^2 + \Delta y^2)} \tag{3.4}$$

Note that all of the values on the right-hand side of equation (3.4) have superscript k meaning that they are all known values from the current iteration.

Example 5

Perform two iterations of the Jacobi method to solve Laplace's equation for the finite-difference grid shown in figure 3.7 with zero starting values for the computational nodes.

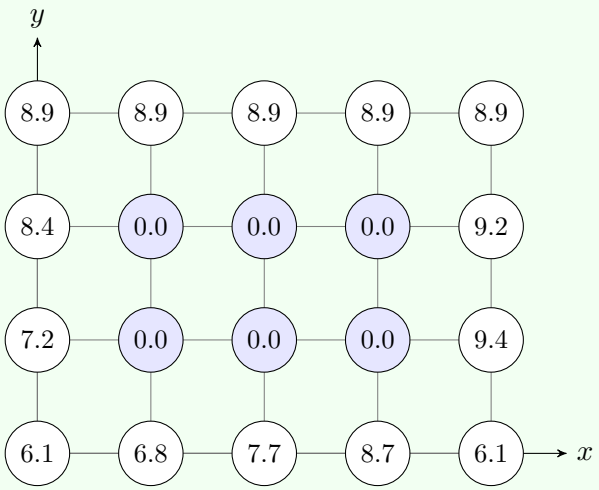


Figure 3.7: The values of the initial conditions and boundary nodes for solving Laplace's equation.

Solution: The values of the computational nodes after one iteration of the Jacobi method are

$$u^1_{1,1} = \frac{7.2 + u^0_{2,1} + 6.8 + u^0_{1,2}}{4} = \frac{7.2 + 0 + 6.8 + 0}{4} = 3.5,$$
$$u^1_{2,1} = \frac{u^0_{1,1} + u^0_{3,1} + 7.7 + u^0_{2,2}}{4} = \frac{0 + 0 + 7.7 + 0}{4} = 1.925,$$
$$u^1_{3,1} = \frac{u^0_{2,1} + 9.4 + 8.7 + u^0_{2,3}}{4} = \frac{0 + 9.4 + 8.7 + 0}{4} = 4.525,$$
$$u^1_{1,2} = \frac{8.4 + u^0_{2,2} + u^0_{1,1} + 8.9}{4} = \frac{8.4 + 0 + 0 + 8.9}{4} = 4.325,$$
$$u^1_{2,2} = \frac{u^0_{1,2} + u^0_{3,2} + u^0_{2,1} + 8.9}{4} = \frac{0 + 0 + 0 + 8.9}{4} = 2.225,$$
$$u^1_{3,2} = \frac{u^0_{2,2} + 9.2 + u^0_{3,1} + 8.9}{4} = \frac{0 + 9.2 + 0 + 8.9}{4} = 4.525.$$

The second iteration of the Jacobi method uses the values from the first iteration

$$u^2_{1,1} = \frac{7.2 + u^1_{2,1} + 6.8 + u^1_{1,2}}{4} = \frac{7.2 + 1.925 + 6.8 + 4.325}{4} = 5.0625,$$
$$u^2_{2,1} = \frac{u^1_{1,1} + u^1_{3,1} + 7.7 + u^1_{2,2}}{4} = \frac{3.5 + 4.525 + 7.7 + 2.225}{4} = 4.4875,$$
$$u^2_{3,1} = \frac{u^1_{2,1} + 9.4 + 8.7 + u^1_{2,3}}{4} = \frac{1.925 + 9.4 + 8.7 + 4.525}{4} = 6.1375,$$
$$u^2_{1,2} = \frac{8.4 + u^1_{2,2} + u^1_{1,1} + 8.9}{4} = \frac{8.4 + 2.225 + 3.5 + 8.9}{4} = 5.7562,$$
$$u^2_{2,2} = \frac{u^1_{1,2} + u^1_{3,2} + u^1_{2,1} + 8.9}{4} = \frac{4.325 + 4.525 + 1.925 + 8.9}{4} = 4.9188,$$
$$u^2_{3,2} = \frac{u^1_{2,2} + 9.2 + u^1_{3,1} + 8.9}{4} = \frac{2.225 + 9.2 + 4.525 + 8.9}{4} = 6.2125.$$

Figures 3.8 and 3.9 contain the values of the computational nodes after the first and second iteration

of the Jacobi method respectively.

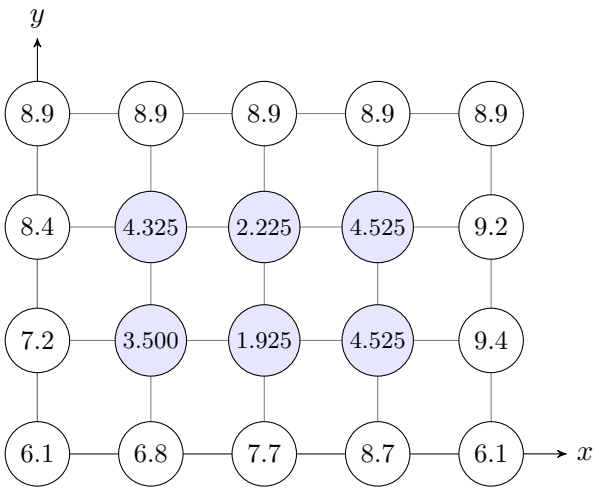


Figure 3.8: The values of the nodes for the first iteration of the Jacobi method.

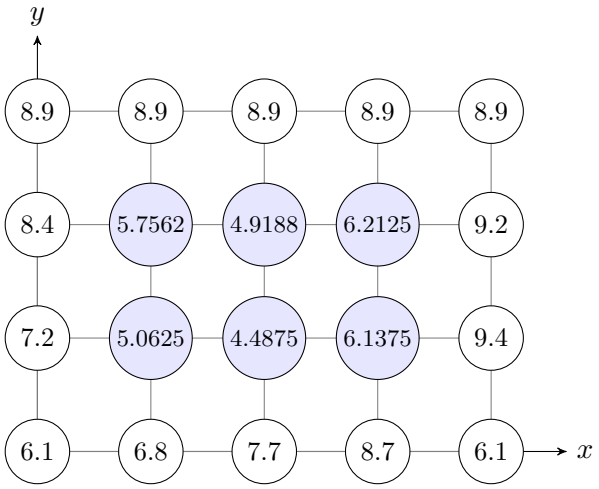


Figure 3.9: The values of the nodes for the second iteration of the Jacobi method.

3.6 Convergence criteria

Iterations for an indirect method continue until we are satisfied that the current estimate is accurate enough for our needs. This is done by calculating an error for the current estimate, err, and comparing it to a predetermined *tolerance* denoted by tol. The tolerance is some small number that we choose depending on the accuracy required. The smaller the value of tol, the more accurate the solution will be but will be will require more iterations[†]. If $err < tol$ then it is assumed that the last estimate has converged to the exact solution. The problem is since we do not know the exact solution we cannot calculate an exact value of err. However, assuming the sequence $u_0, u_1, u_2, \ldots$ converges to the exact solution, then $|u_i^{k+1} - u_i^k| \to 0$ as $k \to \infty$ so $|u_i^{k+1} - u_i^k|$ can be used as a measure of the accuracy of u_i^{k+1}.

Another issue is that the solution to a PDE is approximated at multiple points in the domain so the calculation of the error must take into account the solution at all of these points. To do this we can use the L^2 norm of the solution vector $\mathbf{u}$ which is defined as

[†]Note that this refers to the accuracy of the solution of the linear system resulting from the FDS. The accuracy of the FDS used to solve the PDE is dependent upon the order of the finite-difference approximations used to derive it. Just because the error between two estimates is small does not mean that the solution obtained is accurate for the PDE

Definition 4: The L^2 norm

$$\|\mathbf{u}\| = \sqrt{\sum_i u_i^2}. \tag{3.5}$$

This provides a single value that can be used for the error at each iteration. There are two main ways of calculating an error using the L^2 norm: the *absolute* error and the *relative* error. The absolute error is a measure of distance between two estimates of the solution

Definition 5: Absolute error

$$err_{abs} = \|\mathbf{u}^{k+1} - \mathbf{u}^k\|. \tag{3.6}$$

The relative error is the measure of the distance between two sets of solutions relative to the most recent estimation

Definition 6: Relative error

$$err_{rel} = \frac{\|\mathbf{u}^{k+1} - \mathbf{u}^k\|}{\|\mathbf{u}^{k+1}\|}. \tag{3.7}$$

The choice of whether to use the absolute or relative error will depend on the size of the numbers you are dealing with. For example, consider err calculated using the values $u^0 = 0.01$ and $u^1 = 0.001$,

$$\begin{aligned} err_{abs} &= |0.001 - 0.01| = 0.009, \\ err_{rel} &= \frac{|0.001 - 0.01|}{|0.001|} = 9. \end{aligned}$$

So the absolute error between these values is small suggesting convergence but the relative error is large. Doing the same of the values $u^0 = 100000$ and $u^1 = 100100$

$$\begin{aligned} err_{abs} &= |100100 - 100000| = 100, \\ err_{rel} &= \frac{|100100 - 100000|}{|100100|} = 9.99 \times 10^{-4}. \end{aligned}$$

In this case the absolute error is large whereas the relative error is small.

3.7 Solving a linear system using an indirect method

The general process of calculating the solution to a linear system using an indirect method is summarised in the flow chart shown in figure 3.10. Once the variables and arrays have been initialised a while loop is required to test for convergence. If the solution hasn't converged yet, the new solution $\mathbf{u}^{k+1}$ is calculated, then the error between the current solution and new solution and then the current solution is updated by setting it equal to the new solution.

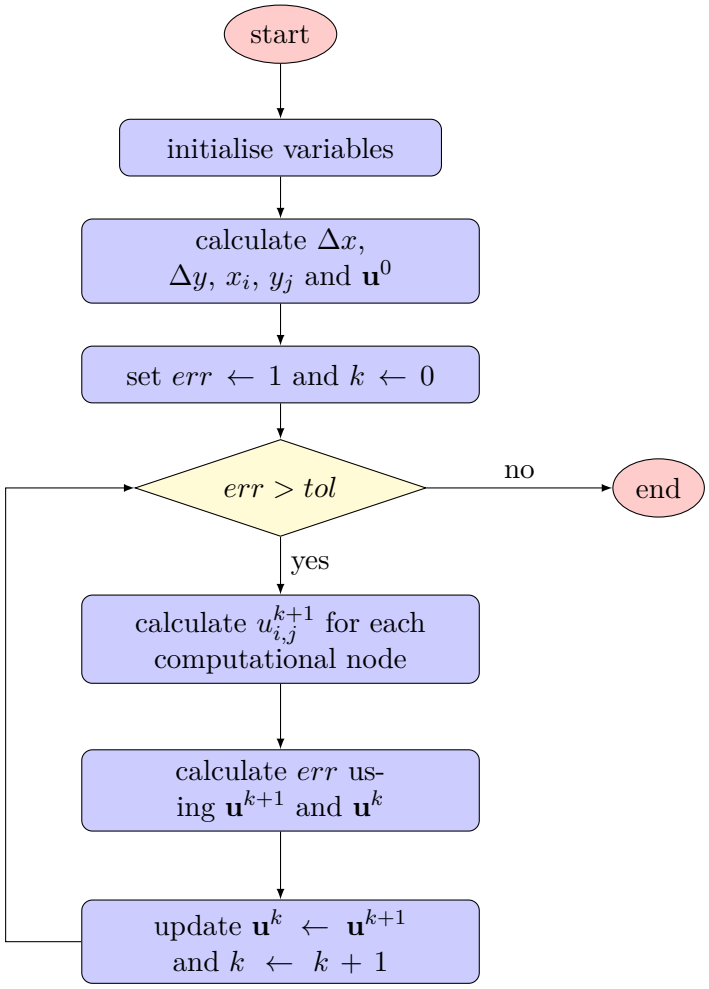


Figure 3.10: Flow chart for solving an elliptic PDE using an indirect method.

The MATLAB code for calculating the Jacobi iterations for the Laplace equation example is given in Listing 3.1.

Listing 3.1: MATLAB program to calculate the Jacobi iterations for the Laplace equation example.

```
% laplace.m by Jon Shiach
%
% This program calculates the solution to the Laplace equation using the
% Jacobi method

% define variables
Nx = 5;                 % no. of nodes in the x direction (inc ghost nodes)
Ny = 4;                 % no. of nodes in the y direction (inc ghost nodes)
lenx = 4;               % length of the x domain
leny = 3;               % length of the y domain
err = 1;                % error between successive iterations
tol = 1d-6;             % convergence tolerance
k = 0;                  % iteration counter

% Calculate x and y node co-ordinates
dx = lenx / (Nx - 1);
dy = leny / (Ny - 1);
x = 0 : dx : lenx;
y = 0 : dy : leny;
[x, y] = meshgrid(x, y);

```

```
22 % Initialise solution array
23 u = [ 8.9, 8.9, 8.9, 8.9, 8.9 ;
24     8.4, 0.0, 0.0, 0.0, 9.2 ;
25     7.2, 0.0, 0.0, 0.0, 9.4 ;
26     6.1, 6.8, 7.7, 8.7, 9.8 ];
27 u = flipud(u);          % invert u to be consistent with matrix indexing
28
29 % output column headings and initial values
30 hline = repmat('-', 1, 12 * ((Nx - 2) * (Ny - 2) + 2));
31 fprintf('%s\n     k    |', hline)
32 for j = 1 : Ny - 2
33     for i = 1 : Nx - 2
34         fprintf('   u(%1i,%1i)  |', i, j)
35     end
36 end
37 fprintf('  L2 error\n%s \n   %4i    ', hline, k);
38 fprintf('|%10.6f ', u(2:Ny-1, 2:Nx-1)')
39 fprintf('|%10.6f\n', err)
40
41 % perform iterations until convergence
42 while err > tol
43
44     % calculate improved estimate using the Jacobi method
45     unew = u;
46     for j = 2 : Ny - 1
47         for i = 2 : Nx - 1
48             unew(j, i) = (u(j+1, i) + u(j, i+1) + u(j-1, i) + u(j, i-1)) /
                   4;
49         end
50     end
51
52     % calculate absolute error
53     err = sqrt(sum(sum((unew - u).^2)));
54
55     % update u
56     u = unew;
57     k = k + 1;
58
59     % output current solution
60     fprintf('   %4i    ', k);
61     fprintf('|%10.6f ', u(2:Ny-1, 2:Nx-1)');
62     fprintf('|%10.6f \n', err);
63 end
64
65 fprintf('%s\n', hline)
66
67 % Plot solution
68 surf(x, y, u);
69 xlabel('$x$', 'fontsize', 16, 'interpreter', 'latex')
70 ylabel('$y$', 'fontsize', 16, 'interpreter', 'latex')
71 zlabel('$U(x,y)$', 'fontsize', 16, 'interpreter', 'latex')
```

The MATLAB code shown in listing 3.1 is explained below:

- Lines 7 to 13 – initialises the variables that define the problem. It is good practice to list all of the variables at the top of the program and use comments to explain what each variable represents.
- Lines 16 to 20 – calculates the spatial steps and the node co-ordinates. Note the use of the `meshgrid` command to produce two-dimensional arrays for plotting purposes.

- Lines 23 to 27 – defines the initial values for the finite-difference nodes. Note that this array is inverted using the `flipud` command so that it is consistent with matrix indexing (i.e., element `u(1,1)` corresponds to node $u_{1,1}$). This isn't strictly necessary and only done here so that we can directly compare the results to a hand calculation.

- Lines 30 to 39 – prints the column headings and initial values to the command window. This code includes `fprintf` commands that outputs the current iteration values. It isn't strictly necessary to do this but it can help in verifying the code, debugging and analysing the convergence behaviour of the method.

- Line 42 – a `while` loop is used to iterate the Jacobi method until $err > tol$.

- Lines 45 to 50 – the values of $\mathbf{u}^{k+1}$ are calculated using row-by-column order. Note that the subscript notation used in the FDS is not the same as standard matrix subscript notation, i.e., `u(j, i)` corresponds to the element $u_{i,j}$.

- Line 53 – calculates the absolute L^2 error.

- Lines 56 and 57 – the values of `u` and `k` are updated ready for the next iteration.

- Lines 60 to 62 – the iteration counter, current estimate and absolute L^2 error are outputted to the command window.

The Jacobi method took 33 iterations to converge to the solution shown in figure 3.11 using $tol = 10^{-6}$. The iteration values for the Jacobi method are shown in table 3.1.

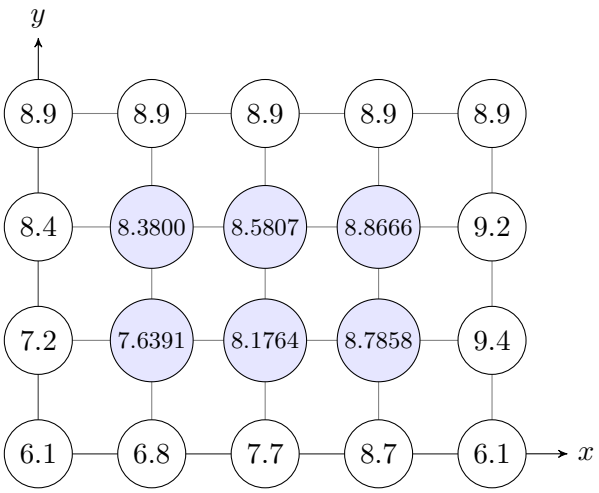


Figure 3.11: The final values of the nodes calculated using Jacobi method with $tol = 10^{-6}$.

Table 3.1: Iteration values for the Jacobi method applied to solve the Laplace equation example.

k	$u^k_{1,1}$	$u^k_{2,1}$	$u^k_{3,1}$	$u^k_{1,2}$	$u^k_{2,2}$	$u^k_{3,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
1	3.500000	1.925000	4.525000	4.325000	2.225000	4.525000	8.975696
2	5.062500	4.487500	6.137500	5.756250	4.918750	6.212500	4.874463
3	6.060937	5.954688	7.200000	6.820312	6.339063	7.289062	2.930346
4	6.693750	6.825000	7.835938	7.425000	7.241016	7.909766	1.768262
5	7.062500	7.367676	8.208691	7.808691	7.764941	8.294238	1.067219
6	7.294092	7.684033	8.440479	8.031860	8.092651	8.518408	0.644121
7	7.428973	7.881805	8.575610	8.171686	8.283575	8.658282	0.388761
8	7.513373	7.997040	8.660022	8.253137	8.402943	8.739796	0.234638
9	7.562544	8.069085	8.709209	8.304079	8.472493	8.790741	0.141617
10	7.593291	8.111062	8.739956	8.333759	8.515976	8.820426	0.085473
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
32	7.639088	8.176397	8.785755	8.379958	8.580744	8.866625	0.000001
33	7.639089	8.176397	8.785755	8.379958	8.580745	8.866625	0.000001

3.8 The Gauss-Seidel method

The Jacobi method calculates the values of the next estimate using the previous estimate for all computational nodes. This is inefficient since the values of nodes that have already been updated in the current iteration are not used. The *Gauss-Seidel method* named after German mathematicians Carl Friedrich Gauss (1777–1855) and Philipp Ludwig von Seidel (1821–1896) improves the rate of convergence of the Jacobi method by using values updated during the current iteration. Recall that the Jacobi method for solving Laplace's equation, equation (3.4), is

$$u^{k+1}_{i,j} = \frac{\Delta y^2(u^k_{i-1,j} + u^k_{i+1,j}) + \Delta x^2(u^k_{i,j-1} + u^k_{i,j+1})}{2(\Delta x^2 + \Delta y^2)}.$$

If the values of $u_{i,j}$ are calculated along each row from left to right before moving up to the next row, the nodes below and to the left of $u_{i,j}$ have already been updated in the current iteration. Using these updated values in equation (3.4) gives

Gauss-Seidel method for computing the solution to Laplace's equation

$$u^{k+1}_{i,j} = \frac{\Delta y^2(u^{k+1}_{i-1,j} + u^k_{i+1,j}) + \Delta x^2(u^{k+1}_{i,j-1} + u^k_{i,j+1})}{2(\Delta x^2 + \Delta y^2)}, \tag{3.8}$$

which is the Gauss-Seidel method used to solve Laplace's equation (3.3).

Example 6: Gauss-Seidel method

Perform two iterations of the Gauss-Seidel method to solve Laplace's equation for the finite-difference grid shown in figure 3.7 with zero starting values for the computational nodes. Calculate the absolute L^2 error after each iteration.

Solution: The first iteration of the Gauss-Seidel method for this problem is

$$
\begin{aligned}
u^1_{1,1} &= \frac{7.2 + u^0_{2,1} + 6.8 + u^0_{1,2}}{4} = \frac{7.2 + 0 + 6.8 + 0}{4} = 3.5,\\
u^1_{2,1} &= \frac{u^1_{1,1} + u^0_{3,1} + 7.7 + u^0_{1,2}}{4} = \frac{3.5 + 0 + 7.7 + 0}{4} = 2.8,\\
u^1_{3,1} &= \frac{u^1_{2,1} + 9.4 + 8.7 + u^0_{3,2}}{4} = \frac{2.8 + 9.4 + 8.7 + 0}{4} = 5.225,\\
u^1_{1,2} &= \frac{8.4 + u^0_{2,2} + u^1_{1,1} + 8.9}{4} = \frac{8.4 + 0 + 3.5 + 8.9}{4} = 5.2,\\
u^1_{2,2} &= \frac{u^1_{1,2} + u^0_{3,2} + u^1_{2,1} + 8.9}{4} = \frac{5.2 + 0 + 2.8 + 8.9}{4} = 4.225,\\
u^1_{3,2} &= \frac{u^1_{2,2} + 9.2 + u^1_{3,1} + 8.9}{4} = \frac{4.225 + 9.2 + 5.225 + 8.9}{4} = 6.8875.
\end{aligned}
$$

Calculating the absolute L^2 error using iterations 0 and 1 gives

$$
\begin{aligned}
err &= \sqrt{(3.5 - 0^2) + (2.8 - 0)^2 + (5.225 - 0)^2 + (5.2 - 0)^2 + (4.225 - 0)^2 + (6.8875 - 0)}\\
&= 11.820275.
\end{aligned}
$$

The second iteration of the Gauss-Siedel method for this problem is

$$
\begin{aligned}
u^2_{1,1} &= \frac{7.2 + u^1_{2,1} + 6.8 + u^1_{1,2}}{4} = \frac{7.2 + 2.8 + 6.8 + 5.2}{4} = 5.5,\\
u^2_{2,1} &= \frac{u^2_{1,1} + u^1_{3,1} + 7.7 + u^1_{1,2}}{4} = \frac{5.5 + 5.225 + 7.7 + 5.2}{4} = 5.6625,\\
u^2_{3,1} &= \frac{u^2_{2,1} + 9.4 + 8.7 + u^1_{3,2}}{4} = \frac{5.6625 + 9.4 + 8.7 + 6.8875}{4} = 7.6625,\\
u^2_{1,2} &= \frac{8.4 + u^1_{2,2} + u^2_{1,1} + 8.9}{4} = \frac{8.4 + 4.225 + 5.5 + 8.9}{4} = 6.75625,\\
u^2_{2,2} &= \frac{u^2_{1,2} + u^1_{3,2} + u^2_{2,1} + 8.9}{4} = \frac{6.75625 + 6.8875 + 5.6625 + 8.9}{4} = 7.051563,\\
u^2_{3,2} &= \frac{u^2_{2,2} + 9.2 + u^2_{3,1} + 8.9}{4} = \frac{7.051563 + 9.2 + 7.6625 + 8.9}{4} = 8.203516.
\end{aligned}
$$

Calculating the absolute L^2 error using iterations 1 and 2 gives

$$
\begin{aligned}
err &= \sqrt{\begin{aligned}&(5.5 - 3.5)^2 + (5.6625 - 2.8)^2 + (7.6625 - 5.225)^2\\ &\quad + (6.75625 - 5.2)^2 + (7.051563 - 4.225)^2 + (8.203516 - 6.8875)^2\end{aligned}}\\
&= 5.502598.
\end{aligned}
$$

The iteration values for the Gauss-Seidel method for this problem are shown in table 3.2. The Gauss-Seidel method converges to the solution significantly faster than the Jacobi method. For this example, 18 iterations were required for the Gauss-Seidel method to converge using $tol = 10^{-6}$ compared to 33 iterations for the Jacobi method.

Table 3.2: Iteration values for the Gauss-Seidel method applied to solve the Laplace equation example.

k	$u^k_{1,1}$	$u^k_{2,1}$	$u^k_{3,1}$	$u^k_{1,2}$	$u^k_{2,2}$	$u^k_{3,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
1	3.500000	2.800000	5.225000	5.200000	4.225000	6.887500	11.820275
2	5.500000	5.662500	7.662500	6.756250	7.051562	8.203516	5.502598
3	6.604687	7.254687	8.389551	7.739062	8.024316	8.628467	2.525223
4	7.248437	7.840576	8.642261	8.143188	8.378058	8.780080	1.064415
5	7.495941	8.054065	8.733536	8.293500	8.506911	8.835112	0.396726
6	7.586891	8.131835	8.766737	8.348451	8.553849	8.855146	0.145074
7	7.620071	8.160164	8.778828	8.368480	8.570948	8.862444	0.052882
8	7.632161	8.170484	8.783232	8.375777	8.577176	8.865102	0.019266
9	7.636565	8.174243	8.784836	8.378435	8.579445	8.866070	0.007018
10	7.638170	8.175613	8.785421	8.379404	8.580272	8.866423	0.002557
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
17	7.639088	8.176397	8.785755	8.379958	8.580745	8.866625	0.000002
18	7.639089	8.176397	8.785756	8.379958	8.580745	8.866625	0.000001

3.9 The Successive Over Relaxation (SOR) method

The convergence of the Gauss-Seidel method can be increased by extrapolating the solution using a weighted average between the current solution $u^k_{i,j}$ and the Gauss-Seidel solution for the next iteration, $\bar{u}^{k+1}_{i,j}$,

$$u^{k+1}_{i,j} = (1-\omega)u^k_{i,j} + \omega\bar{u}^k_{i,j}, \tag{3.9}$$

where ω is called the *relaxation parameter*. The value of the relaxation parameter is restricted to the range $0 < \omega < 2$. When $\omega = 1$ it is easy to see that equation (3.9) reduces to the Gauss-Seidel method. When $0 \leq \omega < 1$ the convergence of the Gauss-Seidel method is slowed down, this is called *under relaxation* and is used when the convergence behaviour of the indirect method oscillates about the solution. When $1 < \omega \leq 2$ the convergence of the Gauss-Seidel method increased, this is called *over relaxation* and is used to accelerate the convergence when the convergence behaviour is monotonic. The optimal value of ω will depend upon the problem being solved.

Writing out equation (3.9) in full gives

Point SOR method for computing the solution to Laplace's equation

$$u^k_{i,j} = (1-\omega)u^k_{i,j} + \omega\left(\frac{\Delta y^2(u^{k+1}_{i-1,j} + u^k_{i+1,j}) + \Delta x^2(u^{k+1}_{i,j-1} + u^k_{i,j+1})}{2(\Delta x^2 + \Delta y^2)}\right). \tag{3.10}$$

Equation (3.10) is known as the *point SOR method* since it updates each point in the finite-difference grid at a time. The choice of ω will determine the convergence rate of the point SOR method. There are methods that can be used to find the optimal value of ω for Laplace's and Poisson's equations (see Yang and Gobbert (2009) as an example) but these are outside the scope of this unit. A numerical investigation where solutions are calculated using different values of ω and choosing the one that uses the fewest iterations can be used to approximate an optimal ω value (but this may be different when using a different number of nodes).

For this example the optimal value was found to be $\omega = 1.12$ and the point SOR method required 10 iterations to converge to $tol = 10^{-6}$ (table 3.3).

Table 3.3: Iteration values for the point SOR method applied to solve the Laplace equation example.

k	$u^k_{1,1}$	$u^k_{2,1}$	$u^k_{3,1}$	$u^k_{1,2}$	$u^k_{2,2}$	$u^k_{3,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
1	3.920000	3.253600	5.979008	5.941600	5.066656	8.160786	13.757648
2	6.024256	6.545146	8.468180	7.236463	8.027872	8.707600	5.674735
3	7.055940	7.965140	8.720186	8.199092	8.492768	8.842715	2.074922
4	7.599272	8.147607	8.778868	8.365880	8.572604	8.865286	0.605516
5	7.631864	8.173621	8.785430	8.377345	8.579838	8.866441	0.044353
6	7.638447	8.176206	8.785689	8.379838	8.580715	8.866620	0.007557
7	7.639079	8.176391	8.785760	8.379962	8.580747	8.866627	0.000674
8	7.639089	8.176400	8.785756	8.379959	8.580747	8.866626	0.000015
9	7.639090	8.176398	8.785756	8.379959	8.580746	8.866625	0.000003
10	7.639089	8.176398	8.785756	8.379959	8.580745	8.866625	0.000001

3.10 Line SOR method

The point SOR method given in equation (3.10) can improved upon by taken advantage of the efficiency of solving tridiagonal systems. The idea is that each row of computational nodes is considered one at a time using the point SOR method which can be written as a tridiagonal linear system for the current row. The values of the points along the row are updated by solving the tridiagonal system using the very efficient Thomas algorithm.

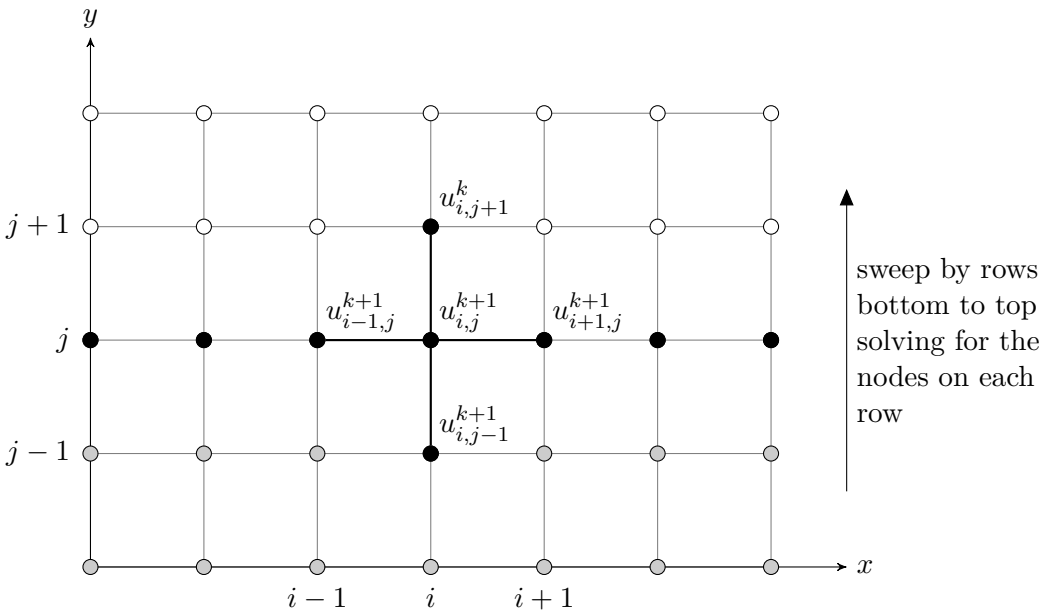


Figure 3.12: The line SOR method solves for each row of the finite-difference at a time.

Consider the finite-difference grid in figure 3.12. If we want to update each row of nodes going from bottom to top then the nodes $u_{i,j}$, $u_{i-1,j}$ and $u_{i+1,j}$ will be updated for the $(k+1)^{\text{th}}$ iteration. The node $u_{i,j-1}$ has already been updated in the previous row and node $u_{i,j+1}$ is yet to be updated. Rewriting equation (3.10) for $\Delta x = \Delta y$ in this way gives

$$u^{k+1}_{i,j} = (1-\omega)u^k_{i,j} + \omega\left(\frac{u^{k+1}_{i-1,j} + u^{k+1}_{i+1,j} + u^{k+1}_{i,j-1} + u^k_{i,j+1}}{4}\right).$$

Rearranging so that the nodes in row j with the superscript $k+1$ are on the left-hand side gives

$$-\omega u_{i-1,j}^{k+1} + 4u_{i,j}^{k+1} - \omega u_{i+1,j}^{k+1} = 4(1-\omega)u_{i,j}^{k} + \omega u_{i,j-1}^{k+1} + \omega u_{i,j+1}^{k}.$$

Considering each computational node $i = 1, 2, \ldots, N-2$ in row j we have

$$\begin{aligned}
-\omega u_{0,j}^{k+1} + 4u_{1,j}^{k+1} - \omega u_{2,j}^{k+1} &= 4(1-\omega)u_{1,j}^{k} + \omega(u_{1,j-1}^{k+1} + u_{1,j+1}^{k}),\\
-\omega u_{1,j}^{k+1} + 4u_{2,j}^{k+1} - \omega u_{3,j}^{k+1} &= 4(1-\omega)u_{2,j}^{k} + \omega(u_{2,j-1}^{k+1} + u_{2,j+1}^{k}),\\
&\vdots\\
-\omega u_{N-4,j}^{k+1} + 4u_{N-3,j}^{k+1} - \omega u_{N-2,j}^{k+1} &= 4(1-\omega)u_{N-3,j}^{k} + \omega(u_{N-3,j-1}^{k+1} + u_{N-3,j+1}^{k}),\\
-\omega u_{N-3,j}^{k+1} + 4u_{N-2,j}^{k+1} - \omega u_{N-1,j}^{k+1} &= 4(1-\omega)u_{N-2,j}^{k} + \omega(u_{N-2,j-1}^{k+1} + u_{N-2,j+1}^{k}).
\end{aligned}$$

Since $u_{0,j}^{k+1}$ and $u_{N-1,j}^{k+1}$ are the boundary nodes their values will be known so moving these over to the right-hand side gives

$$\begin{aligned}
4u_{1,j}^{k+1} - \omega u_{2,j}^{k+1} &= 4(1-\omega)u_{1,j}^{k} + \omega(u_{1,j-1}^{k+1} + u_{1,j+1}^{k} + u_{0,j}^{k+1}),\\
-\omega u_{1,j}^{k+1} + 4u_{2,j}^{k+1} - \omega u_{3,j}^{k+1} &= 4(1-\omega)u_{2,j}^{k} + \omega(u_{2,j-1}^{k+1} + u_{2,j+1}^{k}),\\
&\vdots\\
-\omega u_{N-4,j}^{k+1} + 4u_{N-3,j}^{k+1} - \omega u_{N-2,j}^{k+1} &= 4(1-\omega)u_{N-3,j}^{k} + \omega(u_{N-3,j-1}^{k+1} + u_{N-3,j+1}^{k}),\\
-\omega u_{N-3,j}^{k+1} + 4u_{N-2,j}^{k+1} &= 4(1-\omega)u_{N-2,j}^{k} + \omega(u_{N-2,j-1}^{k+1} + u_{N-2,j+1}^{k} + u_{N-1,j}^{k+1}).
\end{aligned}$$

which can be written as the matrix equation

$$\begin{pmatrix}
4 & -\omega & & & \\
-\omega & 4 & -\omega & & \\
 & \ddots & \ddots & \ddots & \\
 & & -\omega & 4 & -\omega \\
 & & & -\omega & 4
\end{pmatrix}
\begin{pmatrix}
u_{1,j}^{k+1} \\ u_{2,j}^{k+1} \\ \vdots \\ u_{N-3,j}^{k+1} \\ u_{N-2,j}^{k+1}
\end{pmatrix}
=
\begin{pmatrix}
4(1-\omega)u_{1,j}^{k} + \omega(u_{1,j-1}^{k+1} + u_{1,j+1}^{k} + u_{0,j}^{k+1}) \\
4(1-\omega)u_{2,j}^{k} + \omega(u_{2,j-1}^{k+1} + u_{2,j+1}^{k}) \\
\vdots \\
4(1-\omega)u_{N-3,j}^{k} + \omega(u_{N-3,j-1}^{k+1} + u_{N-3,j+1}^{k}) \\
4(1-\omega)u_{N-2,j}^{k} + \omega(u_{N-2,j-1}^{k+1} + u_{N-2,j+1}^{k} + u_{N-1,j}^{k+1})
\end{pmatrix}.$$

This matrix equation can be solved using any linear solver but since it is a tridiagonal system, the Thomas algorithm is by far the most efficient for doing this.

3.10.1 Thomas algorithm

The *Thomas algorithm* is a well known highly efficient solution algorithm for solving tridiagonal systems. It works by using LU decomposition where the system is manipulated so that it is an upper triangular form which can then be solved using back substitution. Consider a linear system of N equations of the form

$$\begin{pmatrix}
b_1 & c_1 & & & \\
a_2 & b_2 & c_2 & & \\
 & \ddots & \ddots & \ddots & \\
 & & a_{N-1} & b_{N-1} & c_{N-1} \\
 & & & a_N & b_N
\end{pmatrix}
\begin{pmatrix}
x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N
\end{pmatrix}
=
\begin{pmatrix}
d_1 \\ d_2 \\ \vdots \\ d_{N-1} \\ d_N
\end{pmatrix},$$

where a_i, b_i and c_i are elements of the lower, main and upper diagonals respectively. The Thomas algorithm proceeds by performing a forward sweep modifying the upper diagonal coefficients c_i and

the right-hand side values d_i

$$c'_i = \begin{cases} \dfrac{c_i}{b_i}, & i = 1, \\ \dfrac{c_i}{b_i - a_i c'_{i-1}}, & i = 2, \ldots, N, \end{cases}$$

$$d'_i = \begin{cases} \dfrac{d_i}{b_i}, & i = 1, \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}, & i = 2, \ldots, N. \end{cases}$$

The solution is obtained using back substitution

$$\begin{aligned} x_N &= d'_N, \\ x_i &= d'_i - c'_i x_{i+1}, \qquad i = N-1, \ldots, 1. \end{aligned}$$

The derivation of the Thomas algorithm can be found in appendix A on page 117 and the MATLAB code is given in listing 3.2.

Listing 3.2: MATLAB function for the Thomas algorithm

```
function x = Tridiag(A, d)

% This function solves a tridiagonal linear system of the form Ax = d using
% the Thomas algorithm

% Calculate size of the system
N = length(d);

% Determine lower, main and upper diagonal elements
a = [0 ; diag(A, -1)];
b = diag(A);
c = [diag(A, 1) ; 0];

% Forward sweep
c(1) = c(1) / b(1);
d(1) = d(1) / b(1);
for i = 2 : N
    c(i) = c(i) / (b(i) - a(i) * c(i-1));
    d(i) = (d(i) - a(i) * d(i-1)) / (b(i) - a(i) * c(i-1));
end

% Backward sweep
x(N) = d(N);
for i = N - 1 : -1 : 1
    x(i) = d(i) - c(i) * x(i+1);
end

end
```

3.10.2 Optimal value of the relaxation parameter

The optimal value of the relaxation parameter is the value of ω for which the SOR method takes the fewest number of iterations to converge to a solution. For tridiagonal systems of the form $A\mathbf{u} = \mathbf{d}$, such as the one seen in the line SOR method, it can be shown that the optimal value of ω is

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho(T_J)^2}}. \tag{3.11}$$

T_J is the iteration matrix for the Jacobi method defined by

$$T_J = D^{-1}(L + U),$$

where $A = D - L - U$, i.e., D are the diagonal elements of A and L and U are the a lower and upper triangular elements of A respectively with their signed reversed, $\rho(T_J)$ the spectral radius of A which is defined as the largest absolute eigenvalue of T_J.

Since A that is used to calculate ω_{opt} contains ω we have an implicit relationship. In order to determine ω_{opt}, we first calculate equation (3.11) using some starting value of ω and then use this to calculate A and the next estimate of ω_{opt}. The iterations continue until two successive estimates agree to some tolerance. For this Laplace equation example, $\omega_{opt} = 1.036$ and the line SOR method took just 8 iterations to converge to the solution (table 3.4).

Table 3.4: Iteration values for the line SOR method applied to solve the Laplace equation example.

k	$u^k_{1,1}$	$u^k_{2,1}$	$u^k_{3,1}$	$u^k_{1,2}$	$u^k_{2,2}$	$u^k_{3,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
1	4.825480	4.739728	5.881230	7.647557	7.573769	8.125412	16.192856
2	7.472641	7.934598	8.619766	8.346390	8.531502	8.833439	5.159570
3	7.633276	8.167999	8.780027	8.379191	8.579668	8.865868	0.332282
4	7.639055	8.176354	8.785722	8.379981	8.580778	8.866648	0.011751
5	7.639100	8.176413	8.785766	8.379962	8.580750	8.866629	0.000095
6	7.639090	8.176399	8.785757	8.379959	8.580746	8.866625	0.000021
7	7.639089	8.176398	8.785756	8.379959	8.580745	8.866625	0.000002
8	7.639089	8.176398	8.785756	8.379959	8.580745	8.866625	0.000000

Table 3.5 contains an analysis of the rate of convergence for each indirect method discussed in this chapter. The CPU time was recorded by placing the `tic` and `toc` MATLAB commands at each end of the program and taking the average time over several runs. The normalised column was calculated by dividing the CPU time taken for each method by the time taken for the Jacobi method. The Jacobi method is the most computationally expensive with the Gauss-Seidel, point SOR and line SOR methods taking 1.7%, 1.3% and 1.3% of the time taken by the Jacobi method respectively. Usually it is expected that the line SOR method would offer more of a saving than the point SOR method but in this example there aren't enough nodes in each row of the finite-difference grid for the increased efficiency to take effect.

Table 3.5: Analysis of the rate of convergence of different indirect methods applied to the Laplace equation example.

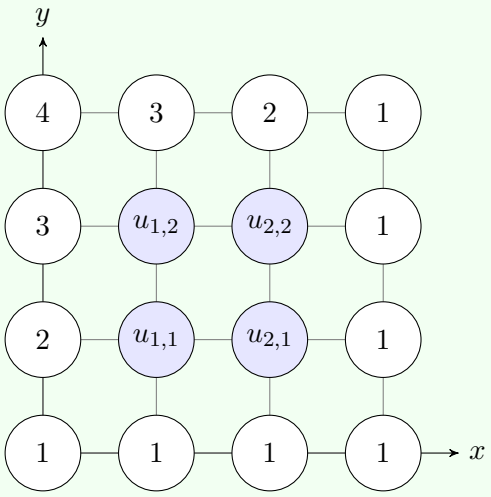
Method	Iterations	CPU time (s)	Normalised CPU time
Jacobi	33	3.16×10^{-5}	1.000
Gauss-Seidel	18	1.82×10^{-7}	0.017
SOR	10	1.37×10^{-7}	0.013
Line SOR	8	1.35×10^{-7}	0.013

Example 7

An electrical field over a two-dimensional domain is to be modelled using Poisson's equation

$$U_{xx} + U_{yy} = f(x, y),$$

where $f(x, y) = x + y$. The domain of $0 \leq x, y \leq 1$ is discretised using 4 nodes in the x and y directions and Dirichlet boundary conditions provide the values of the nodes on the boundary as shown in the diagram below.



Compute the first two iterations of the Jacobi, Gauss-Seidel and point SOR methods (with $\omega = 1.1$) for this problem.

Solution: Using the second-order symmetric differences from the finite-difference toolkit to approximate U_{xx} and U_{yy},

$$U_{xx} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2},$$
$$U_{yy} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2},$$

then Poisson's equation becomes

$$U_{xx} + U_{yy} = x + y$$
$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = x_i + y_j.$$

Since $\Delta x = \Delta y = \frac{1}{3}$ then multiplying both sides by $\Delta x^2 = (\frac{1}{3})^2 = \frac{1}{9}$ gives

$$-4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = \frac{x_i + y_j}{9},$$

and rearranging to make $u_{i,j}$ the subject gives

$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4} - \frac{x_i + y_j}{36}.$$

The Jacobi method for solving this finite-difference scheme is

$$u_{i,j}^{k+1} = \frac{u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k}}{4} - \frac{x_i + y_j}{36}.$$

The co-ordinates of the computational nodes are $x_1 = y_1 = \frac{1}{3}$ and $x_2 = y_2 = \frac{2}{3}$. Calculating the first iteration:

$$u_{1,1}^1 = \frac{2 + u_{2,1}^0 + 1 + u_{1,2}^0}{4} - \frac{\frac{1}{3} + \frac{1}{3}}{36} = \frac{2 + 0 + 1 + 0}{4} - \frac{2}{108} = 0.731481,$$
$$u_{2,1}^1 = \frac{u_{1,1}^0 + 1 + 1 + u_{2,2}^0}{4} - \frac{\frac{2}{3} + \frac{1}{3}}{36} = \frac{0 + 1 + 1 + 0}{4} - \frac{1}{36} = 0.472222,$$
$$u_{1,2}^1 = \frac{3 + u_{2,2}^0 + u_{1,1}^0 + 3}{4} - \frac{\frac{1}{3} + \frac{2}{3}}{36} = \frac{3 + 0 + 0 + 3}{4} - \frac{1}{36} = 1.472222,$$
$$u_{2,2}^1 = \frac{u_{1,2}^0 + 1 + u_{2,1}^0 + 2}{4} - \frac{\frac{2}{3} + \frac{2}{3}}{36} = \frac{0 + 1 + 0 + 2}{4} - \frac{4}{108} = 0.712963$$

The values of $u_{i,j}^k$ for the first two iterations are shown in the table below.

k	$u_{1,1}$	$u_{2,1}$	$u_{1,2}$	$u_{2,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	-
1	0.731481	0.472222	1.472222	0.712963	1.853055
2	1.217593	0.833333	1.833333	1.199074	0.856394

The Gauss-Seidel method for solving this finite-difference scheme is

$$u_{i,j}^{k+1} = \frac{u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k}}{4} - \frac{x_i + y_j}{36}.$$

Calculating the first iteration:

$$\begin{aligned}
u_{1,1}^1 &= \frac{2 + u_{2,1}^0 + 1 + u_{1,2}^0}{4} - \frac{\frac{1}{3} + \frac{1}{3}}{36} = \frac{2 + 0 + 1 + 0}{4} - \frac{2}{108} = 0.731481, \\
u_{2,1}^1 &= \frac{u_{1,1}^1 + 1 + 1 + u_{2,2}^0}{4} - \frac{\frac{2}{3} + \frac{1}{3}}{36} = \frac{0.731481 + 1 + 1 + 0}{4} - \frac{1}{36} = 0.655093, \\
u_{1,2}^1 &= \frac{3 + u_{2,2}^0 + u_{1,1}^1 + 3}{4} - \frac{\frac{1}{3} + \frac{2}{3}}{36} = \frac{3 + 0 + 0.731481 + 3}{4} - \frac{1}{36} = 1.655093, \\
u_{2,2}^1 &= \frac{u_{1,2}^1 + 1 + u_{2,1}^1 + 2}{4} - \frac{\frac{2}{3} + \frac{2}{3}}{36} = \frac{1.655093 + 1 + 0.655093 + 2}{4} - \frac{4}{108} = 1.290509.
\end{aligned}$$

The values of $u_{i,j}^k$ for the first two iterations are shown in the table below.

k	$u_{1,1}$	$u_{2,1}$	$u_{1,2}$	$u_{2,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	-
1	0.731481	0.655093	1.655093	1.290509	2.317101
2	1.309028	1.122106	2.122106	1.524016	0.907904

The SOR method for solving this finite-difference scheme is

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^k + \omega\left(\frac{u_{i,j+1}^k + u_{i+1,j}^k + u_{i,j-1}^{k+1} + u_{i-1,j}^{k+1}}{4} - \frac{x_i + y_j}{36}\right).$$

Calculating the first iteration (using $\omega = 1.1$):

$$\begin{aligned}
u^1_{1,1} &= -0.1u^0_{1,1} + 1.1\left(\frac{2 + u^0_{2,1} + 1 + u^0_{1,2}}{4} - \frac{\frac{1}{3} + \frac{1}{3}}{36}\right) \\
&= 0 + 1.1\left(\frac{2 + 0 + 1 + 0}{4} - \frac{1}{54}\right) = 0.804630, \\
u^1_{2,1} &= -0.1u^0_{2,1} + 1.1\left(\frac{u^1_{1,1} + 1 + 1 + u^0_{2,2}}{4} - \frac{\frac{2}{3} + \frac{1}{3}}{36}\right) \\
&= 0 + 1.1\left(\frac{0.804630 + 1 + 1 + 0}{4} - \frac{1}{36}\right) = 0.740718, \\
u^1_{1,2} &= -0.1u^0_{1,2} + 1.1\left(\frac{3 + u^0_{2,2} + u^1_{1,1} + 3}{4} - \frac{\frac{1}{3} + \frac{2}{3}}{36}\right) \\
&= 0 + 1.1\left(\frac{3 + 0 + 0.804630 + 3}{4} - \frac{1}{36}\right) = 1.840718, \\
u^1_{2,2} &= -0.1u^0_{2,2} + 1.1\left(\frac{u^1_{1,2} + 1 + u^1_{2,1} + 2}{4} - \frac{\frac{2}{3} + \frac{2}{3}}{36}\right) \\
&= 0 + 1.1\left(\frac{1.840718 + 1 + 0.740718 + 2}{4} - \frac{2}{54}\right) = 1.494154.
\end{aligned}$$

The values of $u^k_{i,j}$ for the first two iterations are shown in the table below.

k	$u_{1,1}$	$u_{2,1}$	$u_{1,2}$	$u_{2,2}$	L^2 error
0	0.000000	0.000000	0.000000	0.000000	-
1	0.804630	0.740718	1.840718	1.494154	2.610906
2	1.434061	1.250632	2.240632	1.594941	0.909003

Chapter 4

Hyperbolic Partial Differential Equations

Hyperbolic PDEs are used to model propagation problems where a quantity travels across a domain. Common applications are the modelling of water flow, weather forecasting, blast wave analysis and aerodynamics. Unlike elliptic equations where the solutions are steady state, hyperbolic PDEs contain a time derivative term so the solution will evolve over time from a set of initial conditions. Any perturbation in the dependent variables will propagate outward over time, however, unlike elliptic PDEs, not every point in the domain will feel this disturbance at once. For example, consider dropping a stone in a pool of water initially at rest. Waves caused by the disturbance will propagate out from where the stone was dropped and not all parts of the pool will be affected by the disturbance at once. As the waves propagate across the surface of the pool, the region that is affected by the disturbance expands until all of the pool has been affected. This chapter will discuss the use of FDS to solve the advection equation and introduces the concept of the stability of a numerical scheme.

4.1 The advection equation

The simplest example of a hyperbolic PDE is the one-dimensional *advection equation* that models pure advection along a one-dimensional channel. The one-dimensional advection equation is written as

The advection equation

$$U_t + vU_x = 0 \tag{4.1}$$

where t and x are independent variables denoting time and space respectively and v is the velocity. If the initial conditions are described by the function, $U(0, x)$, then the exact solution at time t is $U(t, x - vt)$.

Consider figure 4.1 where an initial profile is defined that consists of a triangular perturbation (dashed red line). If v is the advection velocity, then at time t the triangular profile has travelled distance vt in the x direction. Note that the triangular perturbation has not deformed through any other process.

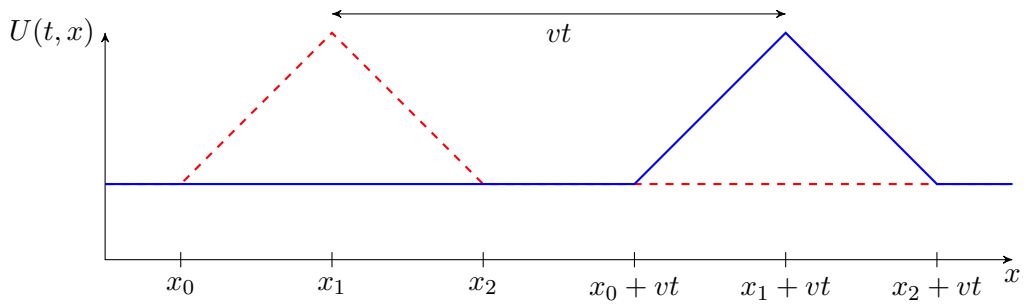


Figure 4.1: Pure advection of an initial profile. Note that the shape is unchanged is it propagates along the x domain.

The advection equation is useful for developing FDSs to solve hyperbolic PDEs since it is simple to calculate an exact solution for a problem and discontinuities which are challenging for a numerical scheme to resolve can be easily specified. Any numerical scheme that performs poorly when solving an advection problem will be unsuitable for other applications involving more complicated hyperbolic PDEs.

4.2 The First-Order Upwind (FOU) scheme

The derivation of a finite-difference scheme (FDS) for solving the advection equation involves approximating the partial derivatives using the finite-differences from the toolkit (table 2.4 on page 23). For example, using a first-order forward difference to approximate U_t and a first-order backward difference for U_x, i.e.,

$$U_t \approx \frac{u_i^{n+1} - u_i^n}{\Delta t},$$
$$U_x \approx \frac{u_i^n - u_{i-1}^n}{\Delta x},$$

results in

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + v\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0.$$

Rearranging to make u_i^{n+1} the subject gives

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_i^n - u_{i-1}^n).$$

Since both the time and space derivatives are approximated using first-order differences then we say that this FDS is first-order accurate in time and space and we write this using $O(\Delta t, \Delta x)$

The FOU scheme for the advection equation when $v > 0$

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + O(\Delta t, \Delta x). \tag{4.2}$$

Assuming that v in equation (4.1) is positive then the direction of flow is from left-to-right along the x domain. To capture this flow, the discretisation of U_x needs to use the neighbouring node in the upwind direction (the node to the left). Therefore equation (4.2) is known as the *first-order upwind scheme*. If v is negative, then flow travels from right-to-left, therefore the spatial discretisation uses the node to the right, and the FOU is

The FOU scheme for the advection equation when $v < 0$

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_{i+1}^n - u_i^n) + O(\Delta t, \Delta x). \tag{4.3}$$

Equations (4.2) and (4.3) are examples of *time marching schemes* where the solution of $u(t, x)$ is marched forward in time by one time step of Δt through a single application of the scheme. The stencil for the FOU scheme when used to solve the advection equation can be seen in figure 4.2.

(a) $v > 0$ (b) $v < 0$

Figure 4.2: Finite-difference stencils for the FOU scheme.

4.2.1 Solution procedure

The solution procedure for solving the advection equation is summarised in the flow chart in figure 4.3. Once the variables and arrays have been initialised, a loop is required to march the solution forward through time until $t = t_{\max}$. Within the time step loop, the boundary conditions are calculated prior to calculating $\mathbf{u}^{n+1}$ using the chosen FDS. The current solution $\mathbf{u}^n$ is updated by setting it equal to the updated solution and the time step is incremented by Δt. Finally the current solution is plotted to produce an animation.

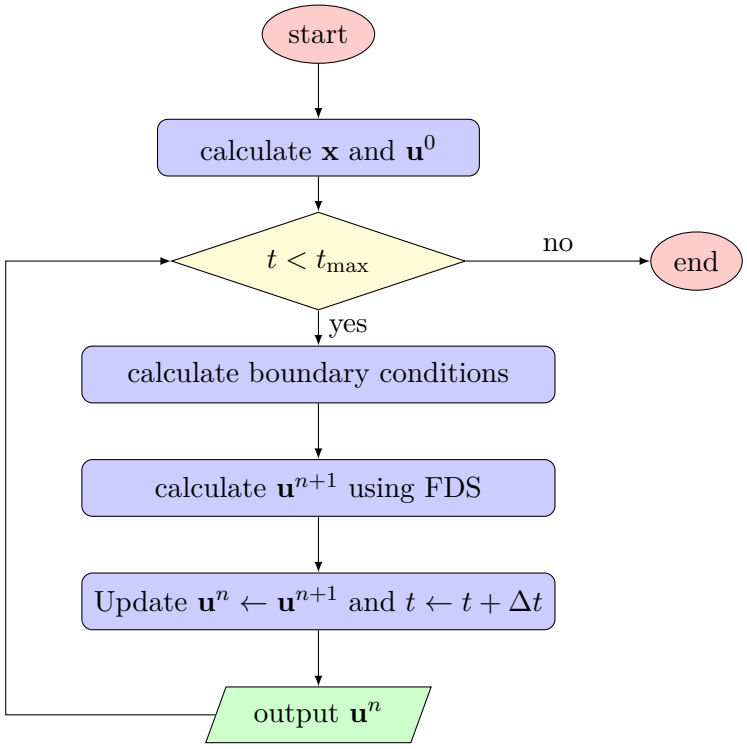


Figure 4.3: Flow chart for applying a time marching scheme to solve the advection equation.

Example 8

The concentration of a pollutant in a river is to be modelled using the advection equation. The stretch of river under study is of length 10m with an average flow velocity of $v = 5\text{ms}^{-1}$. At time $t = 0$s a pollutant is released into the river such that the concentration of pollutant in the

river, $U(t,x)$, can be described by

$$U(0,x) = \begin{cases} 1, & 2 \leq x \leq 4, \\ 0, & \text{elsewhere.} \end{cases}$$

The domain is discretised using a finite-difference grid of six nodes and first-order zero gradient boundary conditions are used to calculate the ghost nodes such that

$$u_{-1} = u_0, \qquad\qquad u_N = u_{N-1}.$$

Use the FOU scheme with $\Delta t = 0.25$ to solve the advection equation to model the concentration of the pollutant after $t = 1$s.

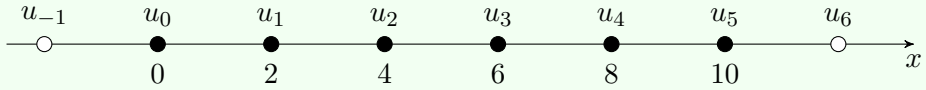
Solution: The first thing we need to do is discretise the spatial domain. Since $0\text{m} \leq x \leq 10\text{m}$ and we are using 6 computational nodes the spatial step is

$$\Delta x = \frac{10}{6-1} = 2\text{m},$$

so the co-ordinates of the nodes are

$$x_0 = 0, \qquad x_1 = 2, \qquad x_2 = 4, \qquad x_3 = 6 \qquad x_4 = 8, \qquad x_5 = 10.$$

The finite-difference grid for this problem can be represented using the diagram below. The computational nodes are represented by filled circles and the ghost nodes u_{-1} and u_6 are represented by unfilled circles. The finite-difference schemes will be applied to calculate the solutions for the computational nodes whereas the ghost nodes will be calculated using the zero gradient boundary condition given above.



The concentration of the pollutant at $t = 0$s is given by a case statement where the initial values of the nodes is 1 where the node co-ordinates are between 2 and 4 inclusive and 0 elsewhere, therefore

$$u_0^0 = 0, \qquad u_1^0 = 1, \qquad u_2^0 = 1, \qquad u_3^0 = 0, \qquad u_4^0 = 0, \qquad u_5^0 = 0,$$

and the ghost nodes are $u_{-1} = u_0 = 0$ and $u_6 = u_5 = 0$. We now need to compute the solution using the FOU scheme for each computational node. Since v, Δt and Δx are constant throughout we can precalculate $\frac{v\Delta t}{\Delta x}$

$$\frac{v\Delta t}{\Delta x} = \frac{5(0.25)}{2} = 0.625.$$

The first step of the FOU is

$$\begin{aligned}
u_0^1 &= u_0^0 - 0.625(u_0^0 - u_{-1}^0) = 0 - 0.625(0-0) = 0, \\
u_1^1 &= u_1^0 - 0.625(u_1^0 - u_0^0) = 1 - 0.625(1-0) = 0.375, \\
u_2^1 &= u_2^0 - 0.625(u_2^0 - u_1^0) = 1 - 0.625(1-1) = 1, \\
u_3^1 &= u_3^0 - 0.625(u_3^0 - u_2^0) = 0 - 0.625(0-1) = 0.625, \\
u_4^1 &= u_4^0 - 0.625(u_4^0 - u_3^0) = 0 - 0.625(0-0) = 0, \\
u_5^1 &= u_5^0 - 0.625(u_5^0 - u_4^0) = 0 - 0.625(0-0) = 0.
\end{aligned}$$

The values for the computational nodes for the first four time steps are shown in the table below.

n	t^n	u_0^n	u_1^n	u_2^n	u_3^n	u_4^n	u_5^n
0	0.00	0.000000	1.000000	1.000000	0.000000	0.000000	0.000000
1	0.25	0.000000	0.375000	1.000000	0.625000	0.000000	0.000000
2	0.50	0.000000	0.140625	0.609375	0.859375	0.390625	0.000000
3	0.75	0.000000	0.052734	0.316406	0.703125	0.683594	0.244141
4	1.00	0.000000	0.019775	0.151611	0.461426	0.695801	0.518799

4.3 Validating the time marching schemes

To validate a numerical scheme (test whether it provides suitable approximations) we can make use of *test cases*. These are often simplified scenarios where problems are defined using idealised values that can often mean we have a known exact solution. To validate the FDS used to solve the advection equation, a test case that involves the advection of a Gaussian curve profile along a channel is used. The spatial domain is defined as $0 \leq x \leq 1$ and the initial conditions are described by

$$U(0, x) = f(x) = \exp(-200(x - 0.25)^2),$$

which produces a Gaussian curve centred at $x = 0.25$. The exact solution for the advection equation is $U(t, x - vt)$ so for this test case

$$U(t, x) = f(x - vt) = \exp(-200(x - vt - 0.25)^2).$$

The FDS is calculated using 101 nodes so that $\Delta x = 0.01$ and a time step of $\Delta t = 0.01$ is used. Transmissive boundary conditions are employed at either end of the domain so that a quasi-infinite domain is modelled. This is achieved by using the following values for the ghost nodes

$$u_{-1} = u_1, \qquad u_N = u_{N-2},$$

The velocity is set at $v = 0.5$ and the FDS is marched forward in time until $t = 1$ when the centre of the Gaussian curve is at $x = 0.75$.

The computed solutions for the FOU scheme are plotted against the exact solution in figure 4.4 for increasing values of t. The numerical solutions show a marked reduction in the height and width of the Gaussian curve which is due to the FOU scheme only being first-order accurate in space. This phenomena is called *numerical dissipation* and is a common feature, although undesirable, of first-order methods.

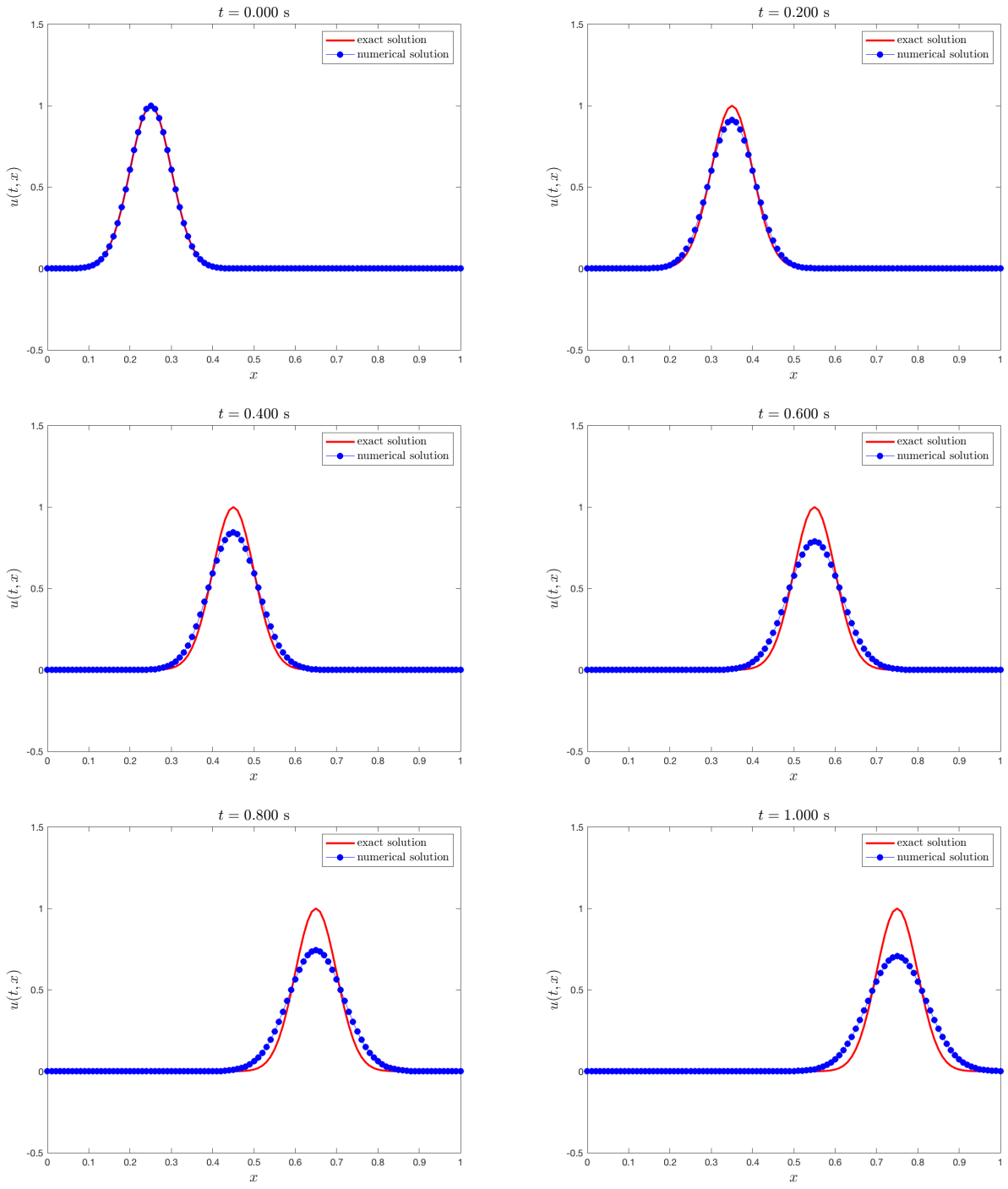


Figure 4.4: The solution of the advection equation using the FOU scheme.

The MATLAB code used to perform the calculations is given in listing 4.1.

Listing 4.1: MATLAB code for the FOU scheme used to solve the advection equation.

```
% advection.m by Jon Shiach
%
% This program solves the advection equation using the FOU scheme

% Clear workspaces
clear
clc

% Define variables
```

```
N = 101;                          % number of nodes
xmin = 0;                         % lower bound of x
xmax = 1;                         % upper bound of x
v = 0.5;                          % velocity
t = 0;                            % time variable
tmax = 1;                         % max value of t
dt = 0.01;                        % time step

% Discretise the domain (remember to include ghost nodes)
dx = (xmax - xmin) / (N - 1);
x = xmin - dx : dx : xmax + dx;

% Define initial conditions
u = f(x);                         % solution array u^n

% Time marching loop
while t < tmax

    % Ensure that t does not exceed tmax
    dt = min(dt, tmax - t);

    % Calculate boundary conditions
    u(1) = u(3);
    u(N+2) = u(N);

    % Calculate values of u^(n+1) using the FOU scheme
    unew = u;
    C = v * dt / dx;
    for i = 2 : N + 1
        unew(i) = u(i) - C * (u(i) - u(i-1));
    end

    % Update u and t
    u = unew;
    t = t + dt;

    % Calculate exact solution
    uexact = f(x - v * t);

    % Plot the numerical against the exact solution
    plot(x, uexact, 'r-', 'linewidth', 2)
    hold on
    plot(x, u, 'bo-', 'markerfacecolor','b')
    hold off
    axis([xmin, xmax, -0.5, 1.5])
    xlabel('$x$', 'fontsize', 16, 'interpreter', 'latex')
    ylabel('$u(t, x)$', 'fontsize', 16, 'interpreter', 'latex')
    title(sprintf('$t = %1.3f$ s', t), 'fontsize', 16, 'interpreter', 'latex
        ')
    shg
    pause(0.001)
end

% Add legend to plot
leg = legend('exact solution', 'numerical solution');
set(leg, 'fontsize', 12, 'interpreter', 'latex')

% -------------------------------------------------------------------------
function u = f(x)

% This function defines the initial profile.
```

```
70 u = exp(-200 * (x - 0.25).^2);
71
72 end
```

The MATLAB code shown in listing 4.1 is explained below:

- Lines 10 to 16 – initialises the variables used to define the problem.
- Lines 19 and 20 – calculates the spatial step Δx and the x co-ordinates of the finite-difference nodes. Note that the co-ordinates of the ghost nodes ($x_{-1} = x_{\min} - \Delta x$ and $x_{N+1} = x_{\max} + \Delta x$) have also been included in the `x()` array
- Line 23 – initialises the solution array `u()` using a Gaussian curve function `f(x)` that is defined at the bottom of the m-file.
- Line 26 – a `while` loop is used to loop through time until `t = tmax`.
- Line 29 – the value of `dt` is checked to ensure that `t` does not exceed `tmax`.
- Lines 32 and 33 – the values of the ghost nodes are calculated so that transmissive boundary conditions are employed.
- Lines 36 to 40 – a `for` loop is used to loop through the values of `u()` and calculate `unew()`. Since we have added two ghost nodes to the finite-difference grid and arrays in MATLAB start at 1, the index `2 : N + 1` correspond to nodes $0, \ldots, N-1$.
- Lines 43 and 44 – `u` are `t` are updated ready for the next iteration.
- Line 47 – the exact solution to the advection equation is calculated using the same function that was used to define the initial conditions.
- Lines 50 to 59 – the numerical and exact solutions are plotted for the current time step. This produces an animation which shows the evolution of the wave which will loop back around and finish where it started due to the use of the periodic boundary conditions. Plotting the solution at every time step is useful for seeing the evolution of the solution over time but can cause the program to slow down considerably.
- Lines 67 to 72 – a function `f(x)` is defined that calculates the initial profile given the array containing the node co-ordinates `x()`. This function is used twice in the main program, first it is used to define the initial values of `u()` and then again to calculate the exact solution $U(t, x) = f(x - vt)$ at each time step.

4.4 The Forward-Time Central-Space (FTCS) scheme

We have seen in figure 4.4 that the FOU schemes gives a relatively poor approximation of the exact solution due to a first-order finite-difference approximation being used to approximation U_x in equation (4.1). Using a second-order central difference approximation should improve the accuracy of our numerical solutions. Recall that that central difference approximation is

$$U_x = \frac{u_{i-1} + u_{i+1}}{2\Delta x} + O(\Delta x^2),$$

and substituting this along with a forward difference approximation for U_t into equation (4.1) gives

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + v\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0,$$

which is rearranged to give

The FTCS scheme for solving the advection equation

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{2\Delta x}(u_{i+1}^n - u_{i-1}^n) + O(\Delta t, \Delta x^2), \tag{4.4}$$

This FDS is known as the *Forward-Time Central Space (FTCS)* scheme. The FTCS scheme is first-order accurate in time and second-order in space. The stencil for the FTCS scheme is given in figure 4.5.

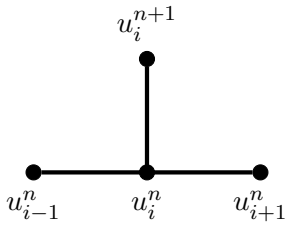


Figure 4.5: Finite-difference stencil for the FTCS scheme.

A plot of solution for the FTCS scheme when applied to solve the test problem given in section 4.3 at $t = 1$ is shown in figure 4.6. Here it is clear that something has gone wrong and we are seeing the solution oscilate behind the Gaussian curve. If the calculations were allowed to continue these oscilations would increase and the program would eventually crash. This is an example of an *unstable* scheme and it is well known that the FTCS scheme is unstable when applied to the advection equation. It was included here to demonstrate that not all FDS are applicable for certain PDEs. It is important that we can determine whether a FDS is stable for a particular PDE.

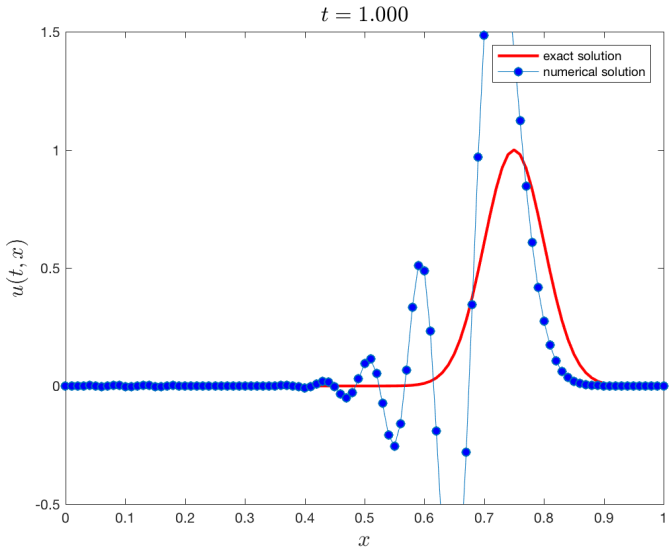


Figure 4.6: The numerical and exact solutions of the advection equation at $t = 0.1$ using the FTCS scheme.

4.5 Convergence, consistency and stability

We have seen in figure 4.6 that not all FDS are suitable for solving the advection equation. Once we have derived a FDS we need to determine whether it can be applied to solve the PDE. One way to do this is to apply the scheme to a test problem to see if it approximates the known solution. However, this is time consuming and may not give you the correct result, for example a suitable scheme may be deemed unsuitable due to errors in the code.

There are three important characteristics that a FDS must have in order for it to be suitable for solving a PDE. These are *consistency*, *stability* and *convergence*.

Definition 7: Consistency

A FDS is said to be *consistent* if as $\Delta t, \Delta x \to 0$ then the numerical scheme is equivalent to the PDE.

Definition 8: Stability

A FDS is said to be *stable* if the errors at each step of the time marching scheme caused by the finite-difference approximations and computational rounding remain bounded and do not increase.

Definition 9: Convergence

A FDS is said to be *convergent* if as $\Delta t, \Delta x \to 0$, $u(t_n, x_i) \to U(t, x)$ (the numerical solution approaches the exact solution of the PDE).

These three characteristics are related by the *Lax equivalence theorem* which is stated below

Theorem 1: Lax equivalence theorem

$$\text{consistency} + \text{stabilily} \Leftrightarrow \text{convergence}$$

i.e., a scheme is convergent if it is *consistent* and *stable* (Lax and Richtmyer, 1956).

Convergence is essential if we want to apply our FDS to solve a PDE, therefore we must check to see if a FDS is both consistent and stable.

4.5.1 Checking for consistency

To check that a scheme is consistent we replace the terms involving the individual nodes, u_i^n, with the corresponding Taylor series expansion for that node. Then we can write the resulting expression in the form of the PDE that is being solved plus the truncation error. If the truncation error tends to zero as $\Delta t, \Delta x \to 0$ then the scheme is consistent. Furthermore, the formal accuracy of the scheme can be determined by the truncation error.

Example 9

Show that the FOU scheme is consistent for the advection equation and has formal accuracy $O(\Delta t, \Delta x)$.

Solution: The FOU scheme is

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_i - u_{i-1}),$$

so we need to replace u_i with U and u_i^{n+1} and u_{i-1}^n with Taylor series expansions for $U(t+\Delta t, x)$ and $U(t, x - \Delta x)$ respectively

$$\begin{aligned} u_i^n &= U, \\ u_{i-1}^n &= U - \Delta x U_x + \frac{\Delta x^2}{2} U_{xx} + O(\Delta x^3), \\ u_i^{n+1} &= U + \Delta t U_t + \frac{\Delta t^2}{2} U_{tt} + O(\Delta t^3). \end{aligned}$$

Therefore the FOU scheme becomes

$$U + \Delta t U_t + \frac{\Delta t^2}{2} U_{tt} + O(\Delta t^3) = U - \frac{v\Delta t}{\Delta x}\left[U - U + \Delta x U_x - \frac{\Delta x^2}{2} U_{xx} + O(\Delta t^3)\right].$$

Simplifying

$$U_t + \frac{\Delta t}{2} U_{tt} + O(\Delta t^2) = -v U_x + \frac{v\Delta x}{2} U_{xx} + O(\Delta x^2)$$

$$U_t + v U_x + \frac{\Delta t}{2} U_{tt} + O(\Delta t^2) - \frac{v\Delta x}{2} U_{xx} + O(\Delta x^2) = 0.$$

For the FOU scheme to be consistent we need this equation to be equivalent to the advection equation plus the additional truncation errors. As it stands we have two terms involving second-order derivatives U_{tt} and U_{xx}, however, these can be omitted by increasing the truncation errors from $O(\Delta t^2)$ and $O(\Delta x^2)$ to $O(\Delta t)$ and $O(\Delta x)$ so we now have

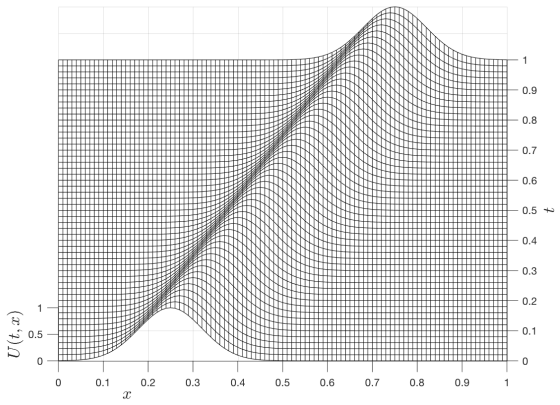
$$U_t + v U_x + O(\Delta t) + O(\Delta x) = 0.$$

We have shown that the FOU scheme is consistent for the advection equation and is first-order accurate in time and space.

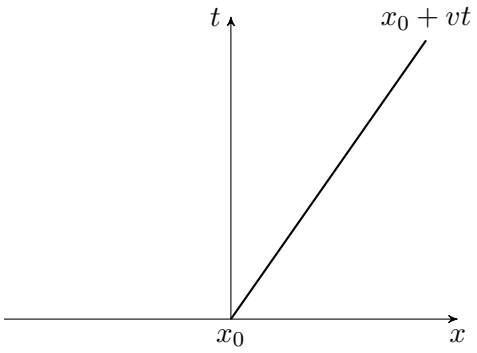
4.5.2 The CFL condition

The program used to solve the advection equation with the FOU scheme in listing 4.1 uses a small time step of $\Delta t = 0.01$. Since each step of the time marching scheme calculates the solution over a time bound of Δt, the larger the value of Δt, fewer time steps are required and therefore fewer computations are needed to calculate $u(t, x)$. Attempting to run the program with a larger value of the time step, $\Delta t = 0.1$ say, results in the solution becoming unstable. There exists a condition on the value of Δt which is known as the *CFL condition** (Courant et al., 1967) which must be satisfied in order for a numerical scheme to be stable.

Consider figure 4.7(a) that shows the advection of a Gaussian curve in space and time. We can define a curve in the t–x space by tracking the propagation of a single point on the curve over time (e.g., the peak of the curve). This curve is known as a *characteristic* curve which can the described in this case by $x_0 + vt$ (figure 4.7(b)).



(a) Gaussian curve.



(b) Characteristic curve.

Figure 4.7: The advection of a Gaussian curve in the xt domain and its corresponding characteristic curve.

*Named after German mathematicians Richard Courant (1888–1972), Kurt Otto Friedrichs (1901–1982) and Hans Lewy (1904–1988).

Definition 10: The CFL condition

When using a fixed grid to calculate the solution to a PDE, the characteristic of the PDE cannot travel more than one spatial step in a single time step. This can be written as

$$|v|\Delta t \leq \Delta x,$$

This can be rearranged to give an expression for the *Courant number*

$$C = \frac{|v|\Delta t}{\Delta x} \leq 1, \tag{4.5}$$

which according to the CFL condition must satisfy $C \leq 1$.

Definition 11: Numerical domain of dependence

The numerical domain of dependence is a set of all nodes that are required to compute the value of the node u_i^n for a given finite-difference scheme.

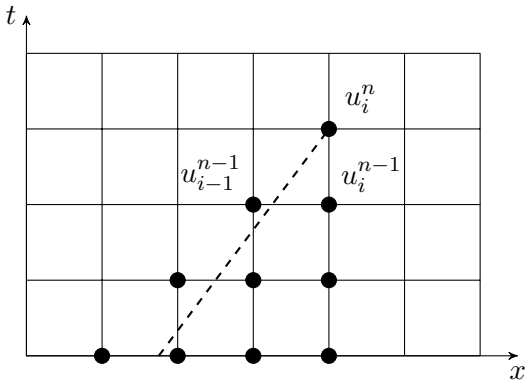


Figure 4.8: Numerical domain of dependence for the FOU scheme.

Consider figure 4.8 that shows the numerical domain of dependence for the FOU scheme used to solve the advection equation where node u_i^n has been calculated using the two nodes u_{i-1}^{n-1} and u_i^{n-1} which are each calculated using two nodes at the previous time step and so on. For the CFL condition to be satisfied, the characteristic represented by the dashed line must be contained within the numerical domain of dependence.

The CFL condition is useful for calculating a value of the time step. Rearranging equation (4.5) gives

$$\Delta t \leq \frac{\Delta x}{|v|}. \tag{4.6}$$

4.5.3 von Neumann stability analysis

The CFL condition is a necessary condition for stability but not a sufficient one. This means that the CFL condition must be satisfied for a numerical scheme to be stable but just because it is satisfied does not mean that the numerical scheme is stable. The purpose of stability analysis is to determine whether a numerical scheme is stable.

The stability of a numerical scheme can be analysed using *von Neumann stability analysis* named after Hungarian mathematician John von Neumann (1903–1957). This assumes that the PDE is linear and periodic boundary conditions are employed but its implementation is often applied to other non-linear PDEs with non-periodic boundary conditions.

Consider the FTCS scheme

$$u_j^{n+1} = u_j^n - \frac{C}{2}(u_{j+1}^n - u_{j-1}^n),$$

where $C = \frac{v\Delta t}{\Delta x}$ is the Courant number. Note that the subscripts are written in terms of j and not i as used elsewhere so to avoid confusion when using complex numbers. The error in the numerical approximation, $\varepsilon_j^n = U_j^n - u_j^n$, of the approximations also satisfy the FTCS scheme, i.e.,

$$\varepsilon_j^{n+1} = \varepsilon_j^n - \frac{C}{2}(\varepsilon_{j+1}^n - \varepsilon_{j-1}^n). \tag{4.7}$$

For linear PDEs with periodic boundary conditions, assuming that the errors grow or shrink exponentially over time then we can write the error at each node as a Fourier mode[†]

$$\varepsilon_i^n = e^{at}e^{ik_m x}, \tag{4.8}$$

where a is some constant, k_m is a wave number and $i = \sqrt{-1}$ is the imaginary number. In this equation the e^{at} term describes the growth of the error at a single node as a function of time. The growth of the error in a numerical scheme can be determined by substituting equation (4.8) into (4.7)

$$e^{a(t+\Delta t)}e^{ik_m x} = e^{at}e^{ik_m x} - \frac{C}{2}(e^{at}e^{ik_m(x+\Delta x)} - e^{at}e^{ik_m(x-\Delta x)})$$

Dividing throughout by $e^{at}e^{ik_m x}$ gives

$$e^{a\Delta t} = 1 - \frac{C}{2}(e^{ik_m\Delta x} - e^{-ik_m\Delta x}) \tag{4.9}$$

The term on the left-hand side is known as the *growth factor* which gives the growth in the error over one time step of Δt so we use $G = e^{a\Delta t}$ for convenience. For stability we require

$$|G| \leq 1, \tag{4.10}$$

i.e., the errors do not increase over a single step. Using the exponential trigonometric identity

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i},$$

we can combine equations (4.9) and (4.10) to give the following condition for stability

$$1 \geq |1 - iC\sin(k_m\Delta x)|.$$

The modulus of a complex number is defined by $|a + bi| = \sqrt{a^2 + b^2}$ so squaring both sides gives

$$1 \geq 1 + C^2\sin^2(k_m\Delta x).$$

Since $\sin^2(x)$ is bounded by the range $[0, 1]$ and C^2 is always positive then this inequality only holds when $C = 0 = \frac{v\Delta t}{\Delta x}$. This would imply that $\Delta t = 0$ which means there is no feasible value of the time step for which the FTCS scheme remains stable. We have shown that the FTCS scheme is *unconditionally unstable* when applied to the advection equation (note that this is not a general result and the FTCS scheme may be stable for other PDEs).

[†]I have skipped over some explanations here for brevity. Readers seeking further explanation are advised to read Hirsch (1994).

von Neumann stability analysis

In summary, the von Neumann stability analysis is carried out using the following steps:

1. Replace each instance of u_j^n in the FDS with its corresponding Fourier mode, e.g., $u_j^n = e^{at}e^{ik_m x}$.

2. Rearrange to obtain an expression for the growth factor $G = e^{a\Delta t}$.

3. The condition $G \leq 1$ to determine for what values of C is the scheme stable. This can be a tricky stage requiring use of trigonometric identities and/or properties of moduli. If equation (4.10) cannot be satisfied for $C > 0$ then the FDS is *unconditionally unstable*; else if it is satisfied for certain values of C then the FDS is *conditionally stable*; and if it is satisfied for all values of C then the FDS is *unconditionally stable*.

Note that it is not recommended in practice to use the largest value of C for which a scheme remains stable in a computer program. This is because small errors due to the limit of machine precision for floating point operations can cause the calculation of C to exceed the limit for stability. For example, if a particular scheme is stable for $C \leq 1$ then we could multiply the maximum value of Δt for which a FDS is stable by a safety factor of 0.9 to ensure that we always have a stable scheme.

Example 10

Use von-Neumann stability analysis to show that the FOU scheme is stable for $\Delta t \leq \dfrac{\Delta x}{|v|}$.

Solution: The FOU scheme is

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{\Delta x}(u_j^n - u_{j-1}^n),$$

substituting the Courant number $C = \frac{v\Delta t}{\Delta x}$

$$u_j^{n+1} = (1 - C)u_j^n + Cu_{j-1}^n.$$

Substituting in the following Fourier modes

$$\begin{aligned} u_j^n &= e^{at}e^{ik_m \Delta x}, \\ u_j^{n+1} &= e^{a(t+\Delta t)}e^{ik_m x}, \\ u_{j-1}^n &= e^{at}e^{ik_m (x-\Delta x)}, \end{aligned}$$

then we have

$$e^{a(t+\Delta t)}e^{ik_m x} = (1 - C)e^{at}e^{ik_m \Delta x} - Ce^{at}e^{ik_m (x-\Delta x)}.$$

Dividing throughout by $e^{at}e^{ik_m x}$ gives

$$G = (1 - C) + Ce^{-ik_m \Delta x}.$$

For stability we require $|G| \leq 1$ so

$$\begin{aligned} 1 &\geq |(1 - C) + Ce^{-ik_m \Delta x}| \\ &\geq |1 - C| + C|e^{-ik_m \Delta x}|. \end{aligned}$$

Since $|e^{i\theta}| = 1$ by Euler's identity then

$$1 \geq |1 - C| + C.$$

This is satisfied for $C \leq 1$ so the maximum allowable time step for the FOU scheme is

$$\Delta t \leq \frac{\Delta x}{|v|}.$$

4.6 The Lax-Friedrichs scheme

The *Lax-Friedrichs scheme* named after American mathematician Peter Lax (1926–1969) and Kurt Otto Friedrichs is a FDS that attempts to overcome the instabilities in the FTCS scheme by artificially dampening the oscillations that cause the scheme to become unstable. The u_i^n term in the forward difference for the time derivative in the FTCS scheme is replaced by an average of the two adjacent nodes, i.e.

$$\frac{u_i^{n+1} - \dfrac{u_{i-1}^n + u_{i+1}^n}{2}}{\Delta t} + v\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0$$

which results in the following FDS

The Lax-Friedrichs scheme

$$u_i^{n+1} = \frac{1}{2}(u_{i-1}^n + u_{i+1}^n) - \frac{v\Delta t}{2\Delta x}(u_{i+1}^n - u_{i-1}^n) + O(\Delta t, \Delta x). \tag{4.11}$$

The Lax-Friedrichs scheme is a first-order in space and time with the stencil given in figure 4.9 (note that even though a central difference was used to approximate U_x, the Lax-Friedrichs scheme is still only first-order accurate in space). Use of von Neumann stability analysis shows that the Lax-Friedrichs scheme is stable for $C \leq 1$.

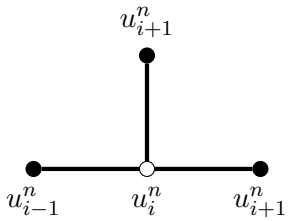


Figure 4.9: Finite-difference stencil for the Lax-Friedrichs scheme.

The Lax-Friedrichs solution to the test problem is shown in figure 4.10. Here the solution has dissipated even more than the FOU scheme due the averaging of the u_i^n term used to remove the oscillations in the FTCS scheme.

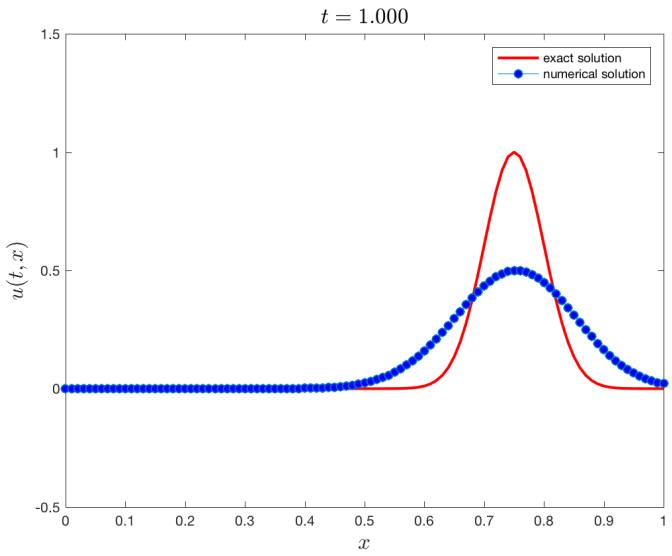


Figure 4.10: The numerical solution of the advection equation using the Lax-Friedrichs scheme compared to the exact solution.

4.7 The Leapfrog scheme

The *Leapfrog* scheme is a second-order scheme in time and space derived by approximating both the time and space derivatives using second-order accurate central differences i.e.,

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t} + v\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0,$$

which is rearranged to give

The Leapfrog scheme

$$u_i^{n+1} = u_i^{n-1} - \frac{v\Delta t}{\Delta x}(u_{i+1}^n - u_{i-1}^n) + O(\Delta t^2, \Delta x^2). \tag{4.12}$$

The stencil for the Leapfrog scheme is shown in figure 4.11. Note that it is the node at the previous time level u_i^{n-1} that is updated to calculate u_i^{n+1} and not u_i^n so we say that u_i^n has been 'leapfrogged' hence the name of the scheme. Using von Neumann stability analysis it can be shown that the Leapfrog scheme is stable when $C \leq 1$.

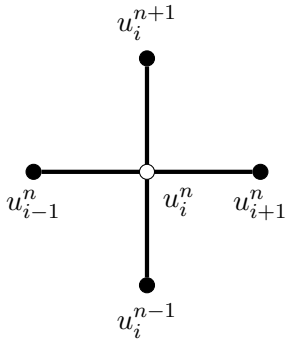


Figure 4.11: Finite-difference stencil for the Leapfrog scheme.

We have a problem with the Leapfrog scheme in that attempting to calculate the solution at the first time level, u_i^1, we need values from a time level prior to the initial conditions, u_i^{-1}, which do not exist. A solution to this problem is to solve the first time step using a first-order in time scheme which only requires the values at one time level u_i^0, and then continue with the Leapfrog scheme for all other time levels. The FTCS scheme can be used to calculate u_i^1, even though it is unconditionally unstable since we are only using it for one step.

The solution to the test problem using the Leapfrog scheme is shown in figure 4.12. Here the numerical solution quite closely replicates the exact solution and the dissipation that was evident for the first-order in time schemes is not present. However, the Leapfrog scheme is notoriously prone to oscillations in the solution for problems with rapidly varying profiles (the Gaussian curve used here has a relatively gentle change in the gradient). Also, the calculation of the Leapfrog scheme requires the values of nodes at two different time steps which requires a computer program to keep track of two solution arrays, $\mathbf{u}^{n-1}$ and $\mathbf{u}^n$, as opposed to just one.

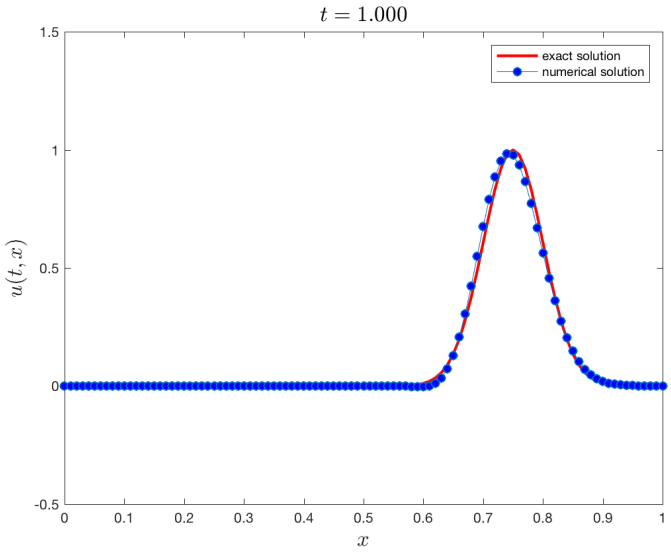


Figure 4.12: The numerical solution of the advection equation using the Leapfrog scheme compared to the exact solution.

4.8 The Lax-Wendroff scheme

The *Lax-Wendroff* scheme named after Peter Lax and American mathematician Burton Wendroff (1930–present) is an extension of the Lax-Friedrichs scheme but uses a second-order discretisation of the time derivative (Lax and Wendroff, 1960). This is done by calculating the Lax-Friedrichs scheme over a half time step for "dummy" nodes that are assumed to be located halfway between two finite-difference nodes. The solution over a full time step is then calculated using the Leapfrog scheme of a full time step using the half time step solution, therefore the full solution is second-order in time and space.

The Lax-Wendroff scheme can be written as a two-step process, the first step (called a *predictor* step) uses a Lax-Friedrichs scheme to solve over a half time step

$$u_{i+\frac{1}{2}}^{n+\frac{1}{2}} = \frac{1}{2}\left(u_i^n + u_{i+1}^n\right) - \frac{v\Delta t}{2\Delta x}(u_{i+1}^n - u_i^n), \tag{4.13}$$

where the $u_{i+\frac{1}{2}}^{n+\frac{1}{2}}$ refers to the dummy node placed a half spatial step from node u_i^n, i.e., $x + \frac{\Delta x}{2}$. The second step (often called a *corrector* step) uses the Leapfrog scheme to update the solution over a full time step (figure 4.13)

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}\left(u_{i+\frac{1}{2}}^{n+\frac{1}{2}} - u_{i-\frac{1}{2}}^{n+\frac{1}{2}}\right). \tag{4.14}$$

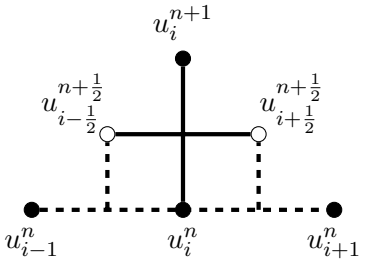


Figure 4.13: Finite-difference stencil for the two-step Lax-Wendroff scheme.

Equations (4.13) and (4.14) can be combined to give a single expression for the Lax-Wendroff scheme

$$
\begin{aligned}
u_i^{n+1} &= u_i^n - \frac{v\Delta t}{\Delta x}\left(u_{i+\frac{1}{2}}^{n+\frac{1}{2}} - u_{i-\frac{1}{2}}^{n+\frac{1}{2}}\right) \\
&= u_i^n - \frac{v\Delta t}{\Delta x}\left(\frac{1}{2}(u_i^n + u_{i+1}^n) - \frac{v\Delta t}{2\Delta x}(u_{i+1}^n - u_i^n) - \frac{1}{2}(u_{i-1}^n + u_i^n) + \frac{v\Delta t}{2\Delta x}(u_i^n - u_{i-1}^n)\right) \\
&= u_i^n - \frac{v\Delta t}{2\Delta x}(u_i^n + u_{i+1}^n - u_{i-1}^n - u_i^n) + \frac{v^2\Delta t^2}{2\Delta x^2}(u_{i+1}^n - u_i^n - u_i^n + u_{i-1}^n) \\
&= u_i^n - \frac{v\Delta t}{2\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right) + \frac{v^2\Delta t^2}{2\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right).
\end{aligned}
$$

Analysis of the consistency of the Lax-Wendroff scheme shows that it is second-order in time and space

The Lax-Wendroff scheme

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{2\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right) + \frac{v^2\Delta t^2}{2\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + O(\Delta t^2, \Delta x^2). \tag{4.15}$$

The stencil for the Lax-Wendroff scheme is shown in figure 4.14. The solution to the test problem using the Lax-Wendroff scheme is shown in figure 4.15. The numerical solution closely approximates the exact solution and there is no numerical dissipation of the curve.

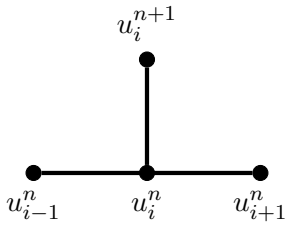


Figure 4.14: Finite-difference stencil for the single step Lax-Wendroff scheme.

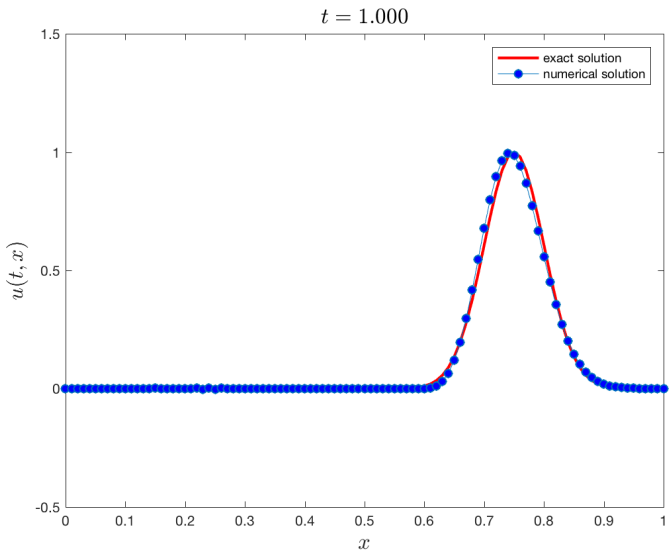


Figure 4.15: The numerical solution of the advection equation using the Lax-Wendroff scheme compared to the exact solution.

4.9 The Crank-Nicolson scheme

The *Crank-Nicolson scheme* named after English mathematicians John Crank (1916–2006) and Phyllis Nicolson (1917–1968) uses a combination of two first-order in time methods to produce a method that is second-order in time (Crank and Nicolson, 1947). The first-order methods used are the FTCS scheme and the *Backwards-Time, Central-Space* (BTCS) scheme. The BTCS scheme is the same as the FTCS scheme except the spatial differencing uses values from the $(n+1)$ time level

$$\frac{u_i^{n+1}-u_i}{\Delta t}+v\frac{u_{i+1}^{n}-u_{i-1}^{n}}{2\Delta x}=0, \qquad \text{(FTCS)}, \tag{4.16}$$

$$\frac{u_i^{n+1}-u_i}{\Delta t}+v\frac{u_{i+1}^{n+1}-u_{i-1}^{n+1}}{2\Delta x}=0, \qquad \text{(BTCS)}. \tag{4.17}$$

The Crank-Nicolson scheme uses the average of equations (4.16) and (4.17)

$$\frac{u_i^{n+1}-u_i^{n}}{\Delta t}=-\frac{v}{2}\left(\frac{u_{i+1}^{n}-u_{i-1}^{n}}{2\Delta x}+\frac{u_{i+1}^{n+1}-u_{i-1}^{n+1}}{2\Delta x}\right),$$

which is rearranged to give

The Crank-Nicolson scheme

$$u_i^{n+1}=u_i^{n}-\frac{v\Delta t}{4\Delta x}\left(u_{i+1}^{n}-u_{i-1}^{n}+u_{i+1}^{n+1}-u_{i-1}^{n+1}\right)+O(\Delta t^2,\Delta x^2). \tag{4.18}$$

Equation (4.18) is an implicit equation where the solution at the next time step u_i^{n+1} is dependent on values of nodes from the current time level, $u_{i\pm1}^{n}$ and from those at the next time level, $u_{i\pm1}^{n+1}$ which are unknown (see the stencil in figure 4.16). Let $r=\frac{v\Delta t}{4\Delta x}$ and moving all u^{n+1} terms to the left-hand side of equation (4.18) gives

$$-ru_{i-1}^{n+1}+u_i^{n+1}+ru_{i+1}^{n+1}=ru_{i-1}^{n}+u_i^{n}-ru_{i+1}^{n}. \tag{4.19}$$

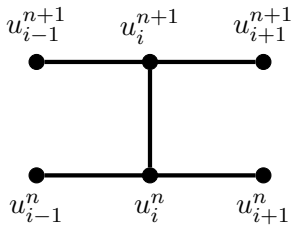


Figure 4.16: Finite-difference stencil for the Crank-Nicolson scheme.

The linear system in equation (4.19) can be solved to determine the values of the nodes within the domain, u_i^{n+1} where $i=1\ldots N-1$. The nodes on the boundaries, u_0^{n+1} and u_N^{n+1}, need to take into account the boundary condition that is applied at that boundary. For example, consider the calculation of equation (4.19) for the node on the left-hand boundary

$$-ru_{-1}^{n+1}+u_0^{n+1}+ru_1^{n+1}=ru_{-1}^{n}+u_0^{n}-ru_1^{n}. \tag{4.20}$$

If zero gradient boundary conditions are used then $u_{-1}^{n+1}=u_{-1}^{n}=u_1^{n}$ so

$$\begin{aligned}-ru_1^{n}+u_0^{n+1}+ru_1^{n+1}&=ru_1^{n}+u_0^{n}-ru_1^{n}\\ \therefore\ u_0^{n+1}+ru_1^{n+1}&=u_0^{n}+ru_1^{n}.\end{aligned}$$

Doing similar for the right-hand boundary gives

$$-ru_{N-2}^{n+1}+ru_{N-1}^{n+1}=-ru_{N-2}^{n}+u_{N-1}^{n}.$$

The solution of the advection equation using Crank-Nicolson scheme given in equation (4.19) with transmissive boundary conditions at each end can be written as the following tridiagonal matrix equation

$$\begin{pmatrix} 1 & r & & & \\ -r & 1 & r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1 & r \\ & & & -r & 1 \end{pmatrix} \begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ \vdots \\ u_{N-2}^{n+1} \\ u_{N-1}^{n+1} \end{pmatrix} = \begin{pmatrix} 1 & r & & & \\ r & 1 & -r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 1 & -r \\ & & & -r & 1 \end{pmatrix} \begin{pmatrix} u_0^{n} \\ u_1^{n} \\ \vdots \\ u_{N-2}^{n} \\ u_{N-1}^{n} \end{pmatrix}.$$

This linear system can be solved using the Thomas algorithm discussed in section 3.10.1 on page 47 to calculate the values of $\mathbf{u}^{n+1}$. Performing a von Neumann stability analysis shows that the Crank-Nicolson scheme is *unconditionally stable* for all values of Δt. This means if we wanted to we could use a single step to update the solution to any value of t. However, since the approximation of the time derivative is still second-order, choosing a high value for Δt will increase errors even though the scheme remains stable.

The Crank-Nicolson scheme was used to model the propagation of the Gaussian curve is shown in figure 4.17.

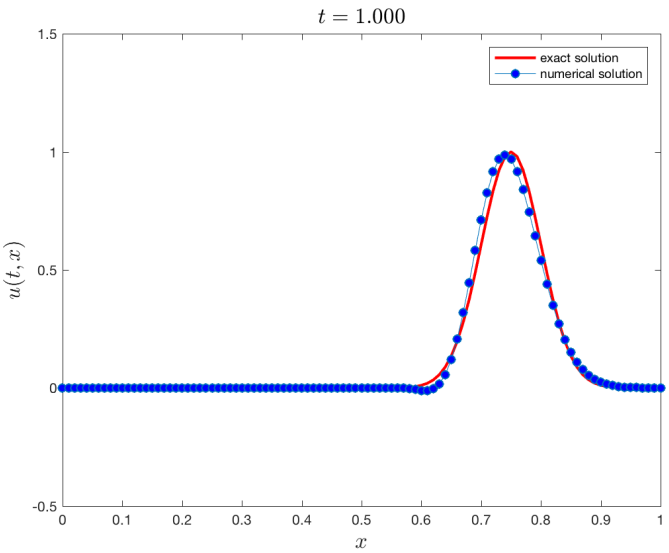


Figure 4.17: The numerical solution of the advection equation using the Crank-Nicolson scheme compared to the exact solution.

4.10 Analysis of finite-difference schemes used to solve the advection equation

A number of finite-difference schemes have been presented in this chapter and applied to solve the advection equation for a Gaussian curve initial conditions. The advection equation is a very simple example of a hyperbolic PDE but has little use when modelling physical phenomena. However, it is useful to know how FDS perform when applied to the advection equation since if a FDS gives a poor approximation of the exact solution, then it is unlikely to perform well for more complicated hyperbolic PDEs.

One of the features of hyperbolic PDEs is that the solutions can admit discontinuities, therefore in order for a numerical scheme to be applicable it needs to be able to handle these discontinuities in addition to very step gradients. The Gaussian curve test used to examine the FDS is an easy test for

a numerical scheme since there is no abrupt changes in the gradient. A tougher test of a numerical scheme is the propagation of a 'top hat' function defined using the following case statement

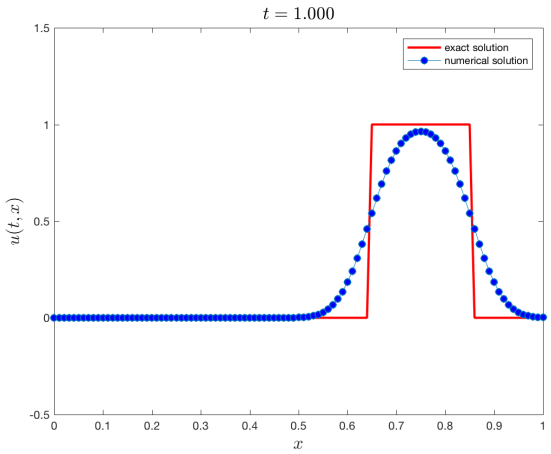
$$U(t,x) = \begin{cases} 1, & |x - vt - 0.25| \leq 0.1, \\ 0, & \text{otherwise.} \end{cases} \tag{4.21}$$

The FDS presented in this chapter with that exception of the FTCS scheme have all been applied to solve this problem and the solutions are shown in figure 4.18.

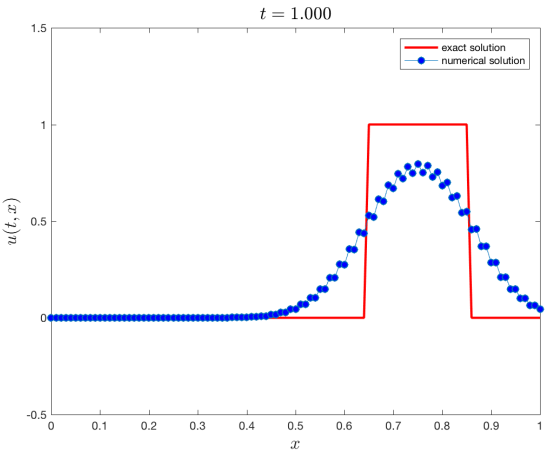
The first-order in time schemes both exhibit numerical dissipation but it is more pronounced in the Lax-Friedrichs scheme (figures 4.18(a) and 4.18(b)). This is a common feature of first-order methods. The second-order in time schemes all exhibit over/undershoots where the profile makes an abrupt change in the gradient. This is known as *Gibbs phenomenon* and is characteristic of second-order methods. The Leapfrog and Crank-Nicolson schemes exhibit severe oscillations in the upwind region behind the tophat (figures 4.18(c) and 4.18(e)) whereas the Lax-Wendroff scheme (figure 4.18(d)) does not have these oscillations and is clearly the best performing of all of the methods seen here. A summary of the FDS described in this chapters is given in table 4.1.

Table 4.1: Summary of the finite-difference schemes presented in this chapter.

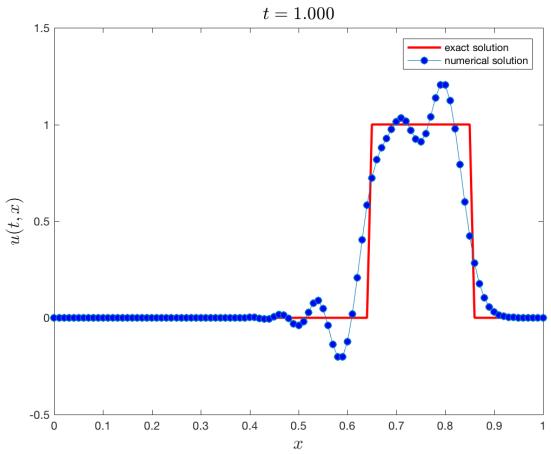
Scheme	$O(\Delta t^n)$	$O(\Delta x^n)$	Stability	Comments
FOU	1^{st}	1^{st}	$C \leq 1$	Dissipative
FTCS	1^{st}	2^{nd}	unconditionally unstable	Cannot be used to solve advection equation
Lax-Friedrichs	1^{st}	1^{st}	$C \leq 1$	Dissipative
Leapfrog	2^{nd}	2^{nd}	$C \leq 1$	Oscillations in upwind region when applied to discontinuities
Lax-Wendroff	2^{nd}	2^{nd}	$C \leq 1$	Over/undershoots at discontinuities
Crank-Nicolson	2^{nd}	2^{nd}	unconditionally stable	Implicit method. Oscillations in upwind region when applied to discontinuities



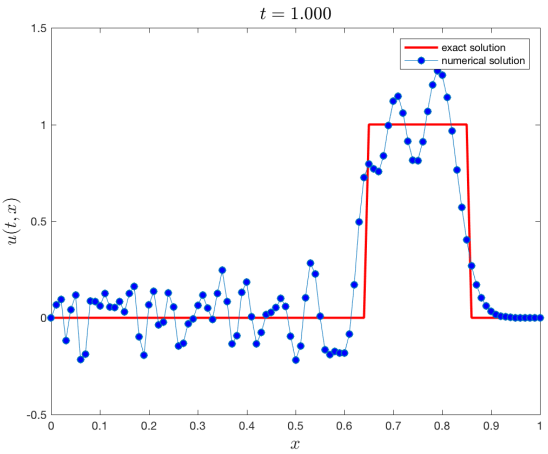
(a) FOU



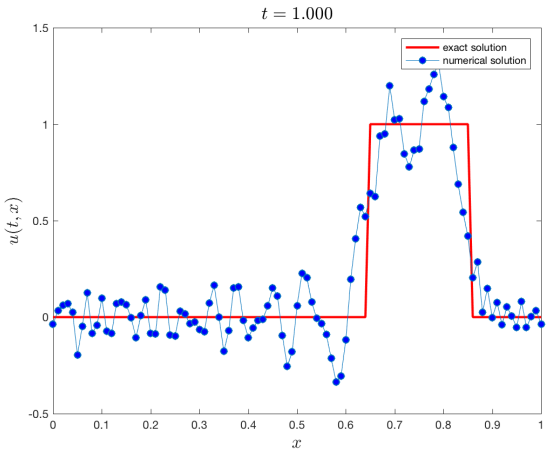
(b) Lax-Friedrichs



(c) Leapfrog



(d) Lax-Wendroff



(e) Crank-Nicolson

Figure 4.18: The solutions of the various FDS used to model the propagation of a 'top hat' profile

Chapter 5

Parabolic Partial Differential Equations

Parabolic PDEs combine second-order spatial derivatives seen in elliptic PDEs with first-order derivatives in time and space that model propagation in hyperbolic PDEs. Parabolic PDEs are most commonly used to model heat diffusion and acoustic problems. This chapter uses the heat diffusion equation to demonstrate the solution of one and two-dimensional parabolic PDEs in addition to performing a grid refinement study.

5.1 Heat diffusion equation

The *heat diffusion equation* is the simplest parabolic PDE and models the diffusion of heat across the domain. The one-dimensional form of the heat diffusion equation is

The heat diffusion equation

$$U_t - \alpha U_{xx} = 0, \tag{5.1}$$

where α is the *thermal diffusivity* of the material defined by

$$\alpha = \frac{k}{\rho c_p}, \tag{5.2}$$

and k is the thermal conductivity of the material, ρ is the density and c_p is the specific heat capacity.

5.1.1 One-dimension test problem

The use of equation (5.1) can be demonstrated by the modelling of the heat diffusion along a metal bar. Consider figure 5.1 that shows a bar of length L that is perfectly insulated along its length so that no heat is transferred in the radial direction. The temperature of at each end of the bar set at a constant 0°C.

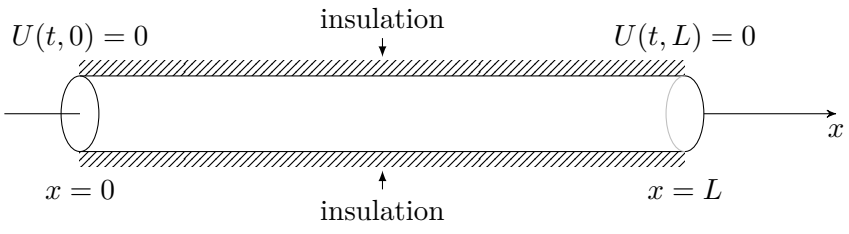


Figure 5.1: Heat diffusion along a metal bar.

The initial temperature profile across the length of the bar is given by the function

$$U(0,x) = \sin\left(\frac{\pi x}{L}\right). \tag{5.3}$$

It can be shown that the exact solution to this problem with initial conditions given by equation (5.3) is

$$U(x,t) = \sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{\alpha\pi^2 t}{L^2}\right). \tag{5.4}$$

5.1.2 Solving the one-dimensional heat diffusion equation

Equation (5.1) can be solved by applying finite-difference approximations from the finite-difference toolkit (table 2.4 on page 23) and rearranged to give expressions for the updated approximation u_i^n similar to what was done for the advection equation in chapter 4. For example, using a forward difference in time and a symmetric difference in space gives

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \alpha\frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} = 0,$$

therefore

The FTCS for the heat diffusion equation

$$u_i^{n+1} = u_i^n + \frac{\alpha\Delta t}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) + O(\Delta t, \Delta x^2). \tag{5.5}$$

This is the FTCS scheme for the heat diffusion equation. In order to determine the value for Δt for which equation (5.5) is stable we need to conduct a von Neumann stability analysis. Let $r = \frac{\alpha\Delta t}{\Delta x^2}$ then equation (5.5) can be written in compact form as

$$u_i^{n+1} = (1-2r)u_i^n + r(u_{i-1}^n + u_{i+1}^n).$$

Substituting in the Fourier modes and rearranging gives (see section 4.5.3)

$$G = (1-2r) + 2r\cos(k_m\Delta x).$$

For stability we require $|G| \leq 1$

$$1 \geq |1 + 2r(\cos(k_m\Delta x) - 1)|,$$

which is satisfied when $r \leq \frac{1}{2}$ so the FTCS scheme is stable when

$$\Delta t \leq \frac{\Delta x^2}{2\alpha}. \tag{5.6}$$

Note that unlike the advection equation where the FTCS scheme was unconditionally unstable, the FTCS scheme is conditionally stable for the heat diffusion equation.

The test problem has been solved using the FTCS scheme. The length of the bar was set at $L = 1$ and thermal diffusivity $\alpha = 1$ was used. The domain was discretised using 11 nodes so that $N = 10$ and $\Delta x = 0.1$. Dirichlet boundary conditions $U(t,0) = U(t,1) = 0$ were employed at both ends of the domain. The FTCS scheme was iterated until $t = 0.05$ using equation (5.6) to calculate Δt. The solution is plotted against the initial profile and exact solution in figure 5.2.

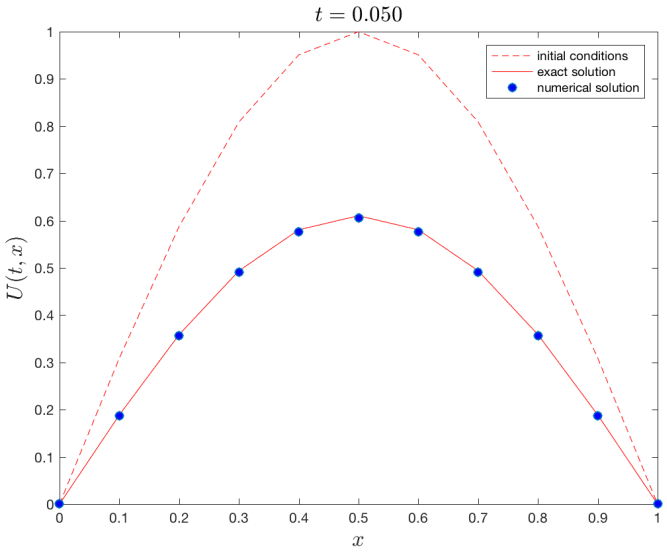


Figure 5.2: The solutions to the one-dimensional heat diffusion equation using the FTCS scheme with $N = 10$ at $t = 0.05$.

The temperature profile along the bar remains centred as the scheme is marched forward in time. This is because the initial profile was symmetrical and the boundary conditions are the same at either end of the domain. Dirichlet boundary conditions ensure the temperature at the boundary nodes remains zero and this has a cooling effect on the nodes in the domain. The FTCS scheme gives very good agreement with the exact solution where the absolute L^2 error between the exact and computed solutions is 0.0095.

5.2 Two dimensional heat diffusion equation

The one-dimensional heat diffusion equation equation (5.1) can be extended into two spatial dimensions by the inclusions of double derivative terms for the y variable, i.e.,

Two-dimensional heat diffusion equation

$$U_t - \alpha(U_{xx} + U_{yy}) = 0. \tag{5.7}$$

The process of solving equation (5.7) is similar to the one-dimensional case. If a FTCS scheme is employed then using a forward difference in time and symmetric differences in both spatial directions then

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \alpha\left(\frac{u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n}{\Delta x^2} + \frac{u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n}{\Delta y^2}\right) = 0$$

which gives

FTCS scheme for the two-dimensional heat diffusion equation

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\alpha\Delta t}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{\alpha\Delta t}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n). \tag{5.8}$$

Performing a von Neumann stability analysis on equation (5.8) gives

$$G = 1 - \frac{2\alpha\Delta t}{\Delta x^2}[1 - \cos(k_m\Delta x)] - \frac{2\alpha\Delta t}{\Delta y^2}[1 - \cos(\ell_m\Delta y)]$$

where ℓ_m is the wave number in the y direction. For stability we require $|G| \leq 1$ and G is at a maximum when $k_m \Delta x = \ell_m \Delta y = \pi$ which gives $1 - \cos(\pi) = 2$ so

$$\left| 1 - 4 \left(\frac{\alpha \Delta t}{\Delta x^2} + \frac{\alpha \Delta t}{\Delta y^2} \right) \right| \leq 1.$$

This is satisfied when the term in brackets is less than or equal to $\frac{1}{2}$, therefore

$$\frac{\alpha \Delta t}{\Delta x^2} + \frac{\alpha \Delta t}{\Delta y^2} \leq \frac{1}{2}$$

$$\Delta t \leq \frac{\Delta x^2 \Delta y^2}{2\alpha(\Delta x^2 + \Delta y^2)}.$$

Note that if $\Delta x = \Delta y$ then

$$\Delta t \leq \frac{\Delta x^2}{4\alpha}.$$

Comparing this expression to equation (5.6) we see that the two-dimensional scheme requires a time step half of that of the one-dimensional scheme.

5.2.1 Modelling heat diffusion across a metal sheet

The FTCS scheme for the two-dimensional heat diffusion equation is applied to model the temperature across a metal sheet. The sheet is a unit square that is discretised using 51 nodes for both the x and y domains giving $\Delta x = \Delta y = 0.02$. The initial temperature across the domain is zero with the exception of a C-shaped region which has a temperature of 1 (figure 5.3).

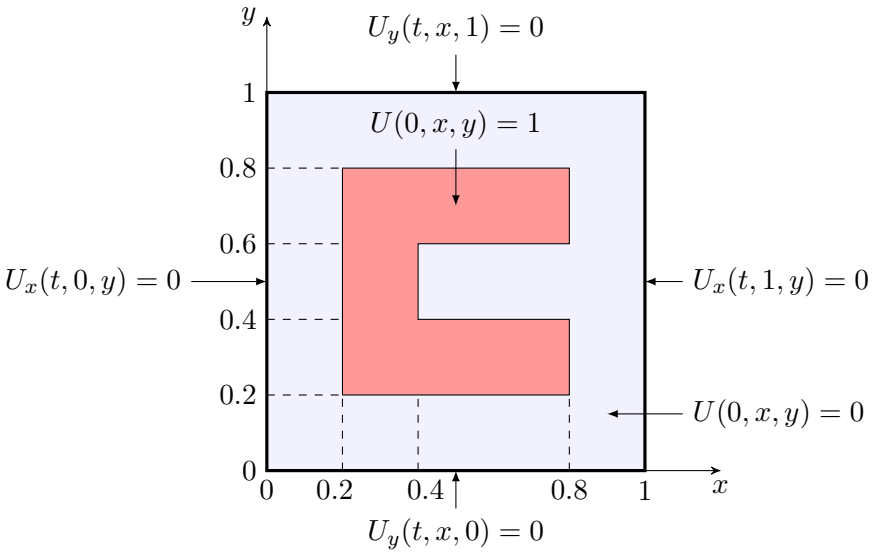


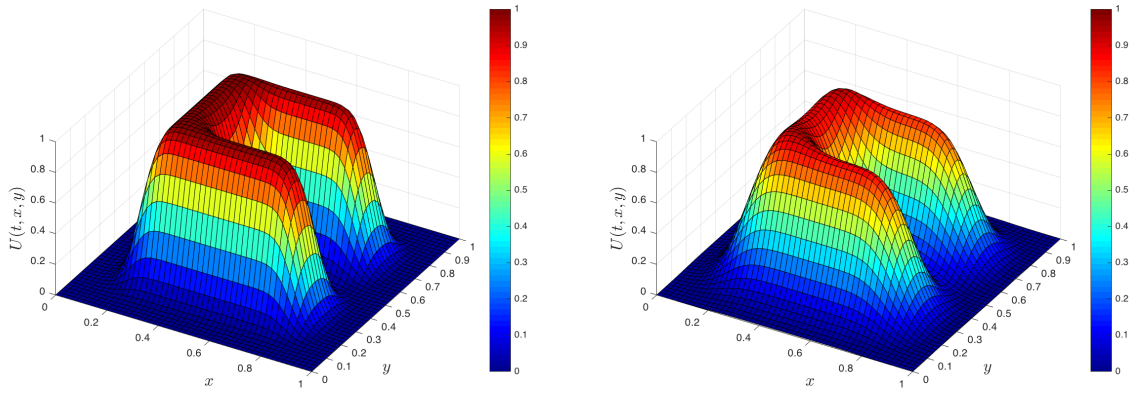
Figure 5.3: Modelling heat diffusion across a metal sheet: initial conditions.

Zero gradient Neumann boundary conditions using ghost nodes are implemented at all four boundaries, i.e.,

$$u^n_{-1,j} = u^n_{1,j}, \qquad u^n_{N_X,j} = u^n_{N_X-2,j}, \qquad u^n_{i,-1} = u_{i,1}, \qquad u^n_{i,N_Y} = u^n_{i,N_Y-2}.$$

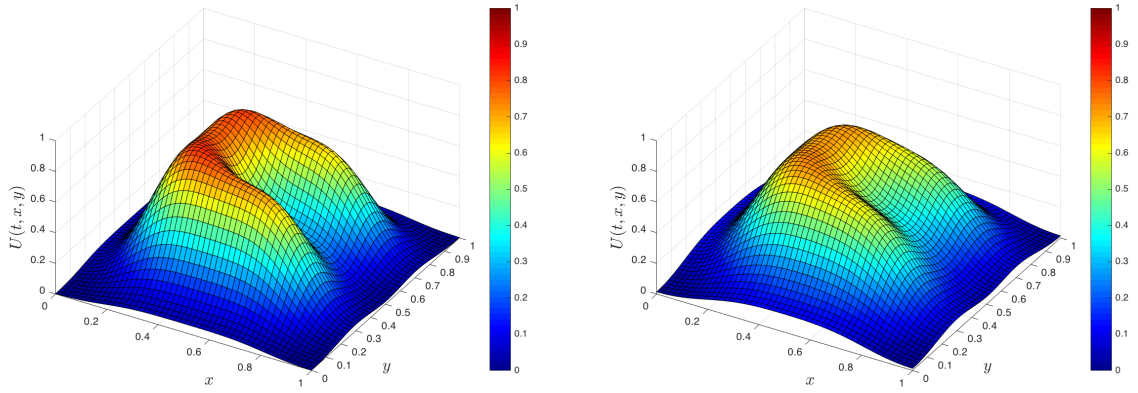
This means that the temperature gradient across the boundaries is zero so that the metal sheet is perfectly insulated (i.e., the temperature outside of the domain has no effect on the temperature within the domain).

The solutions for this problem at times $t = 0.00$, 0.25, 0.75, 1.00 and 1.50 have been plotted and shown in figure 5.4. As the solution is marched forward in time the area surrounding the C-shape begins to heat up whilst C-shape itself begins to cool. This diffusion continues until the initial shape is no longer recognisable.



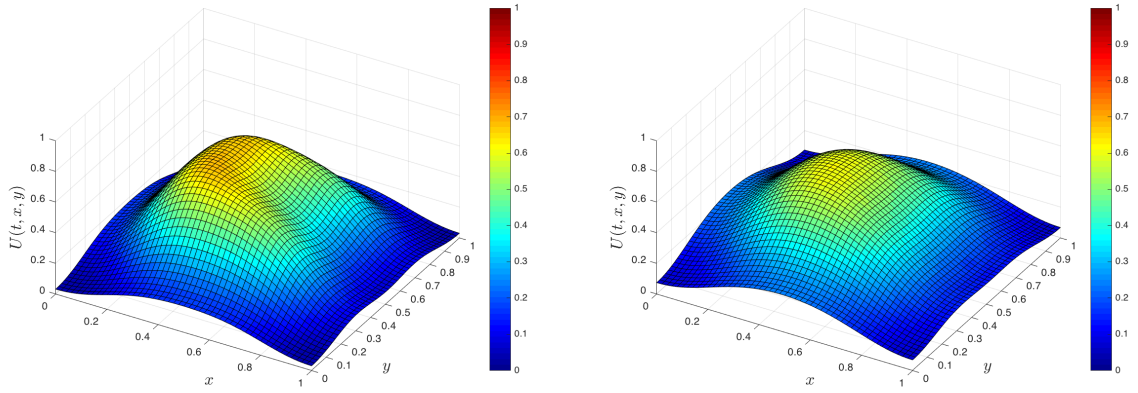
(a) $t = 0.1$

(b) $t = 0.25$



(c) $t = 0.50$

(d) $t = 0.75$



(e) $t = 1.00$

(f) $t = 1.50$

Figure 5.4: Solutions to the two-dimension heat diffusion problem.

5.3 Grid refinement study

It was discussed previously in section 4.5 that a convergent scheme is one where the error between the numerical approximations to a PDE and the exact solution of a PDE converge to zero as $\Delta t, \Delta x \to 0$. So choosing a smaller value of Δx will give a better approximation to the exact solution, but this improvement in the accuracy comes at an additional computational cost since we are calculating the solution using more grid points and we will need to use a smaller value of Δt for stability. In practice there is a compromise between accuracy and computational speed and we need to ask at what point is it no longer worthwhile reducing Δx to gain a more accurate result? This is the purpose behind a *grid refinement study.*

A grid refinement study is performed using the following steps:

1. The PDE is solved using a finite-difference grid with the fewest number of nodes that can be reasonably expected to capture the solution.
2. The solution is recalculated using a finer finite-difference grid with more nodes than before.
3. Steps 1 and 2 are repeated increasing the number of nodes used in the finite-difference grid. Metrics used to analyse the performance of the model are compared and a decision is made to the optimum number of nodes.

The metrics used to analyse the performance of the numerical model will depend upon the phenomenon begin modelled, but in most cases will involve the accuracy of the numerical solution and the time taken to compute this solution.

5.3.1 Grid convergence index

In the majority of cases there are no known exact solutions for a PDE so we cannot compute the error between the numerical solution and the exact solution. What we can do is calculate the difference between the solutions using different sized grids and see whether the solution is converging at a rate expected for the order of the method. To do this we calculate the *Grid Convergence Index (GCI)* which is a measure of the convergence to an estimate of the exact solution between two different sized grids.

Let E be the error between the exact solution f_{exact} and the numerical solution using grid spacing h, $f(h)$ such that

$$E = f(h) - f_{exact} = Ch^p, \tag{5.9}$$

where p is the *order of convergence* of the numerical method and C is some constant. In practice factors such as boundary conditions, grid spacings and sink and source terms mean that the actual order of convergence is less than that of the theoretical order of convergence. So we need to calculate the *observed order of convergence* for the numerical method.

The observed order of convergence can be calculated using equation (5.9) for three different grid sizes. Lets say we have a fine, medium and coarse grid with grid spacing h_1, h_2 and h_3 respectively where r is the *grid refinement ratio* such that

$$r = \frac{h_2}{h_1} = \frac{h_3}{h_2},$$

writing out equation (5.9) for each of these grids and solving for p gives

$$p = \frac{\ln\left(\dfrac{f_3 - f_2}{f_2 - f_1}\right)}{\ln(r)},$$

where f_i is the numerical solution using grid spacing h_i. Applying Richardson extrapolation using f_1 and f_2 gives

$$f_{exact} \approx f_1 + \frac{f_1 - f_2}{r^p - 1}. \tag{5.10}$$

The second term on the right-hand side of equation (5.10) is the fractional error for f_1. Calculating the percentage error between f_1 and f_{exact} gives

$$100\left(\frac{f_{exact}-f_1}{f_1}\right) \approx 100\left(\frac{f_1-f_2}{r^p-1}\frac{1}{f_1}\right) = \frac{\varepsilon_{21}}{r^p-1},$$

where ε_{21} is the percentage relative error between the grids using h_1 and h_2, i.e.,

$$\varepsilon_{21} = 100\left(\frac{f_2-f_1}{f_1}\right).$$

The GCI gives an estimate of the discretisation error in the fine grid solution (f_1) relative to the extrapolated solution f_{exact}. Roache (1994) defines the GCI index calculated using the fine and medium grids as

$$GCI_{21} = \frac{1.25|\varepsilon_{21}|}{r^p-1}. \tag{5.11}$$

and similar for the medium and coarse grids GCI_{32}. A solution is said to be in the *asymptotic range of convergence* if

$$r^p GCI_{21} \approx GCI_{32}. \tag{5.12}$$

If equation (5.12) is satisfied then the solution using the h_2 grid has achieved grid convergence and no further refinement of the grid will provide more accurate results.

To perform a grid refinement study using the GCI we do the following (Roache, 1994):

Grid refinement study

1. Calculate the solution of the PDE using three different finite-difference grids with grid spacings h_1 (fine grid), h_2 (medium grid) and h_3 (coarse grid) where $h_1 < h_2 < h_3$ and $r = \dfrac{h_2}{h_1} = \dfrac{h_3}{h_2}$.

2. For a sample of points in the grids calculate the observed order of convergence using

$$p = \frac{\left|\ln\left|\dfrac{f_3-f_2}{f_2-f_1}\right|\right|}{\ln(r)}, \tag{5.13}$$

   where f_1, f_2 and f_3 are the solutions at the points for grid spacings h_1, h_2 and h_3 respectively. The use of the moduli in equation (5.13) is so we can deal with oscillatory convergence.

3. Calculate the grid convergence index for the fine and medium grid solutions and the medium and coarse grid solutions.

$$GCI_{21} = \frac{1.25\left|100\left(\dfrac{f_2-f_1}{f_1}\right)\right|}{r^p-1}, \qquad GCI_{32} = \frac{1.25\left|100\left(\dfrac{f_3-f_2}{f_2}\right)\right|}{r^p-1}.$$

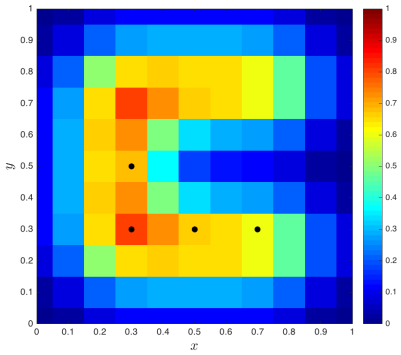
4. If $r^p GCI_{21} \approx GCI_{32}$ then the medium grid solution has reached asymptotic convergence. If not, refine the grid further and repeat steps 1–3.

The FDS described previously to solve the heat diffusion equation has been applied using square finite-difference grids with $N_3 = 10$, $N_2 = 20$ and $N_1 = 40$ nodes such that $r = 2$ and $h_3 = 0.1$, $h_2 = 0.05$ and $h_1 = 0.025$. The GCI was calculated for a sample of four points from the domain located at $(0.3, 0.3)$, $(0.5, 0.3)$, $(0.7, 0.3)$ and $(0.3, 0.5)$ and these are shown in the table below. Note that for the point at (0.7,0.3), $GCI_{32} \ll r^p GCI_{21}$ which indicates that the h_2 grid has not reached asymptotic convergence.

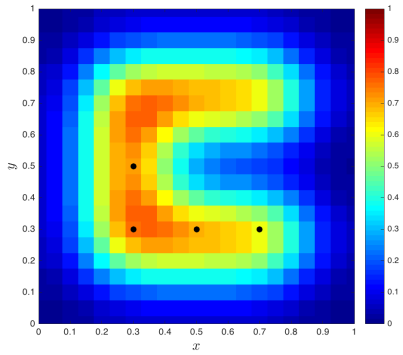
(x, y)	f_{exact}	f_1	f_2	f_3	p	GCI_{21}	GCI_{32}	$r^p GCI_{21}$
(0.3,0.3)	0.6210	0.7221	0.7564	0.8022	0.42	17.50	22.37	23.43
(0.5,0.3)	0.7073	0.7045	0.6966	0.6658	1.95	0.49	1.92	1.90
(0.7,0.3)	0.5893	0.5932	0.6050	0.6079	2.00	0.83	0.20	3.30
(0.3,0.5)	0.7315	0.7279	0.7178	0.6797	1.91	0.63	2.40	2.36

The finite-difference grid was refined further so that the fine, medium and coarse grids use $N_1 = 80$, $N_2 = 40$ and $N_1 = 20$. The GCI for the same sample of points are shown in the table below. Here, all of the points sampled satisfy equation (5.12) (or close enough) which suggests that the discretisation using $N = 40$ nodes is sufficient in this case.

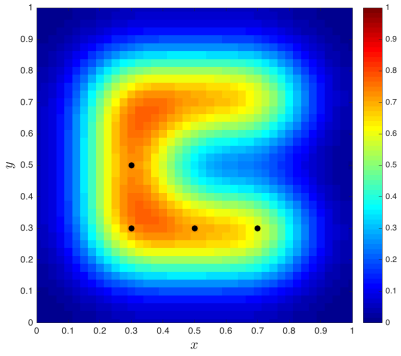
(x, y)	f_{exact}	f_1	f_2	f_3	p	GCI_{21}	GCI_{32}	$r^p GCI_{21}$
(0.3,0.3)	0.6792	0.7030	0.7221	0.7564	0.85	4.24	7.43	7.63
(0.5,0.3)	0.7070	0.7064	0.7045	0.6966	2.05	0.11	0.45	0.45
(0.7,0.3)	0.5675	0.5852	0.5932	0.6050	0.54	3.76	5.41	5.48
(0.3,0.5)	0.7310	0.7302	0.7279	0.7178	2.09	0.12	0.53	0.53



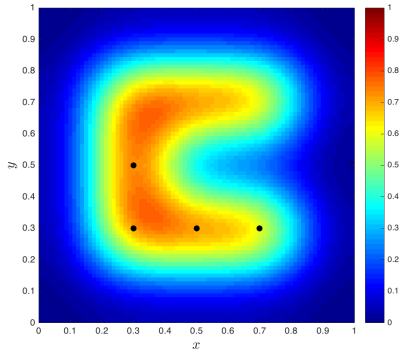
(a) $N = 10$



(b) $N = 20$



(c) $N = 40$



(d) $N = 80$

Figure 5.5: Solutions to the heat diffusional example at $t = 0.5$ using four different grid sizes.

Example 11

A numerical solver was used to calculate the solution to a particular PDE using five different grid sizes. The values of the spatial step used and solutions at two points in the finite-difference grid are shown in the table below.

h	point 1	point 2
0.1520	0.5549	0.5432
0.0760	0.5533	0.5577
0.0380	0.5462	0.5668
0.0190	0.5513	0.5719
0.0095	0.5490	0.5693

Use the grid convergence index to determine the optimum grid spacing that provides the best compromise between accuracy and computational time.

Solution: Considering the first three grids we have $h_1 = 0.038$, $h_2 = 0.076$, $h_3 = 0.152$ and

$$\begin{aligned} &\text{point 1:} && f_3 = 0.5549, && f_2 = 0.5533, && f_1 = 0.5462, \\ &\text{point 2:} && f_3 = 0.5432, && f_2 = 0.5577, && f_1 = 0.5668. \end{aligned}$$

Calculate the ratio of successive step lengths:

$$r = \frac{h_2}{h_1} = \frac{0.038}{0.076} = 2.$$

Calculate the observed order of convergence:

$$\text{point 1:} \qquad p = \frac{\left|\ln\left|\dfrac{0.5549 - 0.5533}{0.5533 - 0.05462}\right|\right|}{\ln(2)} = \frac{|\ln(0.2254)|}{\ln(2)} = 2.1497,$$

$$\text{point 2:} \qquad p = \frac{\left|\ln\left|\dfrac{0.5432 - 0.5577}{0.5577 - 0.5668}\right|\right|}{\ln(2)} = \frac{|\ln(1.5934)|}{\ln(2)} = 0.6721.$$

Calculate GCIs:

$$\text{point 1 :} \qquad GCI_{21} = \frac{1.25\left|100\left(\dfrac{0.5533 - 0.5462}{0.5462}\right)\right|}{2^{2.1497} - 1} = 0.4727,$$

$$GCI_{32} = \frac{1.25\left|100\left(\dfrac{0.5549 - 0.5533}{0.5533}\right)\right|}{2^{2.1497} - 1} = 0.1052,$$

$$\text{point 2 :} \qquad GCI_{21} = \frac{1.25\left|100\left(\dfrac{0.5577 - 0.5668}{0.5668}\right)\right|}{2^{0.6721} - 1} = 3.3820,$$

$$GCI_{32} = \frac{1.25\left|100\left(\dfrac{0.5432 - 0.5577}{0.5577}\right)\right|}{2^{0.6721} - 1} = 5.4768.$$

Checking for asymptotic convergence:

$$\begin{aligned} &\text{point 1:} && r^p GCI_{21} = 2^{2.1497}(0.4727) = 2.0976 \neq 0.1052 = GCI_{32}, \\ &\text{point 2:} && r^p GCI_{21} = 2^{0.6721}(3.3820) = 5.3889 \approx 5.4768 = GCI_{32}, \end{aligned}$$

so for the solution using $h = h_2 = 0.076$ point 2 has reached asymptotic convergence point 1 has not. Therefore we need to use a finer grid, considering the next three grids where $h_1 = 0.019$, $h_2 = 0.038$, $h_3 = 0.076$ and

$$\begin{array}{lllll} \text{point 1}: & p = 0.4473, & GCI_{21} = 2.9487, & GCI_{32} = 4.1434, \\ \text{point 2}: & p = 0.8354, & GCI_{21} = 1.4212, & GCI_{32} = 2.5588. \end{array}$$

Repeating the calculations for these values gives

$$\begin{array}{lllll} \text{point 1}: & p = 0.4473, & GC1_{21} = 2.9487, & GCI_{32} = 4.1434, \\ \text{point 2}: & p = 0.8354, & GCI_{21} = 1.4212, & GCI_{32} = 2.5588. \end{array}$$

Checking for asymptotic convergence:

$$\begin{array}{ll} \text{point 1:} & r^p GCI_{21} = 2^{0.4473}(2.9487) = 4.1051 \approx 4.1434 = GCI_{32}, \\ \text{point 2:} & r^p GCI_{21} = 2^{0.8354}(1.4212) = 2.5359 \approx 2.5588 = GCI_{32}. \end{array}$$

So for the solution using $h = h_2 = 0.038$ both sample points have reached asymptotic convergence so this is the optimum grid spacing.

Chapter 6

Solving Multidimensional Partial Differential Equations

For most real world practical applications we require methods that can model systems over two or three-dimensional spatial domains. To simplify our numerical models we can split the problem up so that we solve for multiple models each with single spatial dimension. This method is known as *dimensional splitting* and can also be used for PDEs including multiple terms.

6.1 The two-dimensional advection equation

Consider the two-dimensional form of the advection equation seen in equation (4.1) on page 53

Two-dimensional advection equation

$$U_t + vU_x + wU_y = 0, \tag{6.1}$$

where v and w are the velocities in the x and y directions respectively. Equation (6.1) models pure advection so the shape of the initial profile is unchanged as it propagates though the spatial domain. If the initial conditions are defined by

$$U(0, x, y) = f(x, y),$$

then the exact solution is

$$U(t, x, y) = f(x - vt, y - wt).$$

6.1.1 Deriving an unsplit FDS for the two-dimensional advection equation

The derivation of an unsplit FDS to solve the two-dimensional advection equation proceeds in the same way as seen previously, i.e., we approximate the partial derivatives using finite-differences from the finite-difference toolkit (table 2.4 on page 23). Recall that when $v > 0$ the FOU scheme uses a backwards difference to discretise the spatial derivative, assuming that $v, w > 0$ then

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + v\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + w\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = 0,$$

which is rearranged to give

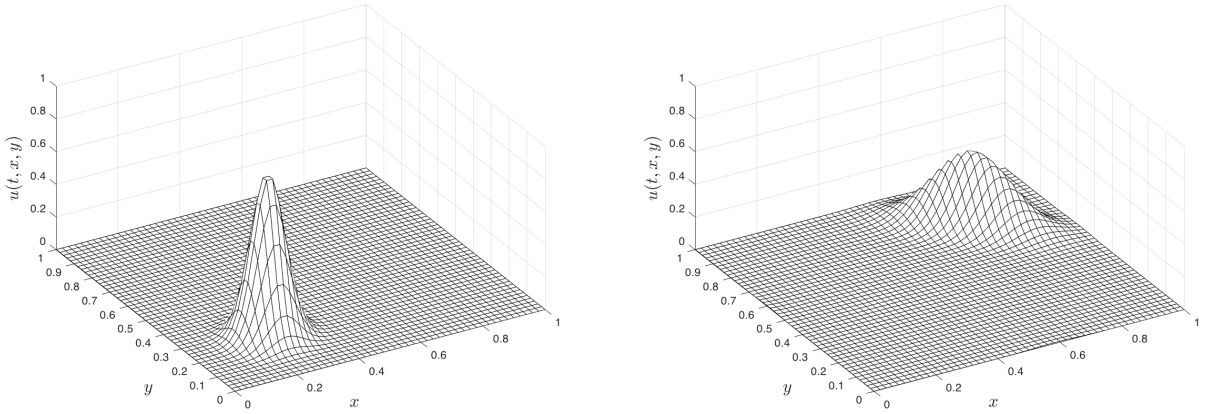
Unsplit FOU scheme for the two-dimensional advection equation

$$u_{i,j}^{n+1} = u_{i,j}^n - \frac{v\Delta t}{\Delta x}(u_{i,j}^n - u_{i-1,j}^n) - \frac{w\Delta t}{\Delta y}(u_{i,j}^n - u_{i,j-1}^n) + O(\Delta t, \Delta x, \Delta y). \tag{6.2}$$

Equation (6.2) is the FOU scheme for the two-dimensional advection equation which is first-order in time and space. To test this scheme the advection of a two-dimensional Gaussian curve is used. The domain is defined by the range $0 \leq x, y \leq 1$ which is discretised using a finite-difference grid with spatial steps $\Delta x = \Delta y = 0.02$. Dirichlet boundary conditions are used at all four boundaries where the boundary nodes remain a constant zero. The initial profile is defined by

$$U(0, x, y) = \exp[-200((x - 0.25)^2 + (y - 0.25)^2)]$$

and the velocities are $v = w = 1$. The initial profile the numerical solution at $t = 0.5$ is shown in figure 6.1. The numerical scheme exhibits dissipation perpendicular to the main direction of flow given by the vector $\mathbf{v} = v\mathbf{i} + w\mathbf{j}$.



(a) $t = 0.0$ (b) $t = 0.5$

Figure 6.1: Solutions of the two-dimensional advection equation using the FOU scheme.

Performing a von Neumann stability analysis on the FDS in equation (6.2) shows that it is stable when

$$\frac{v\Delta t}{\Delta x} + \frac{w\Delta t}{\Delta y} \leq 1,$$

so the maximum allowable time step is

$$\Delta t = \frac{\Delta x \Delta y}{v\Delta y + w\Delta x}. \tag{6.3}$$

This is a very restrictive constraint on the value of Δt and the analysis requires difficult algebra. In general, unsplit schemes are unwieldy, have restrictive stability constraints, are computationally expensive and are rarely used for parabolic and hyperbolic PDEs.

6.2 Operator splitting

Instead of deriving a single scheme to solve a multi-dimensional PDE which can be complicated, we can derive simpler schemes to solve PDEs that are related to the original PDE that we want to solve. This method is called *operator splitting* since we are splitting a complicated problem up into several simpler problems. The schemes used for the related PDEs are applied in sequence so that the completed sequence is equivalent to the unsplit scheme. The manner of splitting can be done by spatial dimensions, components of the PDE or a combination of both.

6.2.1 Differential marching operator

In order to ensure that the sequence of split schemes is equivalent to the unsplit scheme, we need a way of comparing the two types. This can be done by considering the *differential marching operator* for the schemes. The idea is to write the solution of the PDE at the next time step in the form of some operator that is applied to the current solution. This is done by expressing the time derivatives in the Taylor series using spatial derivatives. For example, consider the second-order Taylor series expansion for one step forward in time

$$U(t+\Delta t) = U + \Delta t U_t + \frac{\Delta t^2}{2} U_{tt} + O(\Delta t^3). \tag{6.4}$$

We want to replace the $U_t(t,x)$ and $U_{tt}(t,x)$ terms with spatial derivatives which will depend on the PDE begin solved. Let ∂_t and ∂_{tt} denote be two *differential operators* for partial derivatives for the t variable such that $\partial_t U = U_t$ and $\partial_{tt} U = U_{tt}$, equation (6.4) can then be written as

$$U(t+\Delta t) = U + \Delta t \partial_t U + \frac{\Delta t^2}{2} \partial_{tt} U + O(\Delta t^3) \tag{6.5}$$

We require expressions for $\partial_t U$ and $\partial_{tt} U$ which will depend upon the PDE being solved. For example, consider the one-dimensional advection equation equation (4.1) written using differential operator notation

$$\partial_t U + v \partial_x U = 0.$$

Therefore

$$\partial_t U = -v \partial_x U,$$

and

$$\partial_{tt} U = \partial_t(\partial_t U) = \partial_t(-v\partial_x U) = -v\partial_x(\partial_t U) = -v\partial_x(-v\partial_x U) = v^2 \partial_{xx} U.$$

Substituting $\partial_t U$ and $\partial_{tt} U$ into equation (6.5) gives

$$U(t+\Delta t) = U - v\Delta t \partial_x U + \frac{v^2 \Delta t^2}{2} \partial_{xx} U + O(\Delta t^3),$$

factorising out the U terms results in

$$U(t+\Delta t) = \left[1 - v\Delta t \partial_x + \frac{v^2 \Delta t^2}{2} \partial_{xx}\right] U + O(\Delta t^3).$$

Let $\mathcal{L}_X(\Delta t)$ be the term in the square brackets then

$$U(t+\Delta t) = \mathcal{L}_X(\Delta t) U + O(\Delta t^3), \tag{6.6}$$

where

Differential marching operator for the one-dimensional advection equation

$$\mathcal{L}_X(\Delta t) = 1 - v\Delta t \partial_x + \frac{v^2 \Delta t^2}{2} \partial_{xx} + O(\Delta t^3). \tag{6.7}$$

$\mathcal{L}_X(\Delta t)$ is the *differential marching operator* for the advection equation (an uppercase X has been used in the subscript so as to avoid confusing it with a partial derivative). The different finite-difference schemes presented in chapter 4 can be derived by replacing ∂_x and ∂_{xx} by different finite-difference approximations from the toolkit, i.e.,

$$u_i^{n+1} = \mathcal{L}_X(\Delta x) u_i^n.$$

6.2.2 Differential marching operator for the two-dimension advection equation

Following a similar process seen previously we can derivate the differential marching operator for the two-dimensional advection equation. Using differential operator notation, equation (6.1) can be written as

$$\partial_t U + v\partial_x U + w\partial_y U = 0,$$

therefore

$$\begin{aligned}
\partial_t U &= -v\partial_x U - w\Delta_y U, \\
\partial_{tt} U &= \partial_t(\partial_t U) = \partial_t(-v\partial_x U - w\partial_y U) = -v\partial_x(\partial_t U) - w\partial_y(\partial_t U) \\
&= -v\partial_x(-v\partial_x U - w\partial_y U) - w\partial_y(-v\partial_x U - w\partial_y U) \\
&= v^2\partial_{xx} U + 2vw\partial_{xy} U + w^2\partial_{yy} U.
\end{aligned}$$

Substituting $\partial_t U$ and $\partial_{tt} U$ into equation (6.5) gives

$$\begin{aligned}
U(t+\Delta t) &= U - \Delta t[v\partial_x U + w\partial_y U] \\
&\quad + \frac{\Delta t^2}{2}[v^2\partial_{xx} U + 2vw\partial_{xy} U + w^2\partial_{yy} U] + O(\Delta t^3) \\
&= \left[1 - \Delta t(v\partial_x + w\partial_y) + \frac{\Delta t^2}{2}(v^2\partial_{xx} + 2vw\partial_{xy} + w^2\partial_{yy})\right] U + O(\Delta t^3).
\end{aligned}$$

So the second-order differential marching operator for the two-dimensional advection equation is

Second-order differential marching operator for the two-dimensional advection equation

$$\mathcal{L}_{XY}(\Delta t) = 1 - \Delta t(v\partial_x + w\partial_y) + \frac{\Delta t^2}{2}(v^2\partial_{xx} + 2vw\partial_{xy} + w^2\partial_{yy}), \tag{6.8}$$

and the FDS can be written as

$$u_{i,j}^{n+1} = \mathcal{L}_{XY}(\Delta t) u_{i,j}^n. \tag{6.9}$$

Note that equation (6.8) assumes that the spatial steps Δx and Δy are constant throughout the domain.

6.3 Dimensional splitting

Dimensional splitting splits a PDE with an n-dimensional spatial domain into a system of n PDEs each having a single spatial domain. Each of these one-dimensional PDEs can be solved using our one-dimensional schemes and then combined in sequence to provide the solution to the n-dimensional PDE. For example, we can split the two-dimensional advection equation equation (6.1) into two one-dimensional PDEs

$$U_t + vU_x = 0, \tag{6.10a}$$
$$U_t + wU_y = 0. \tag{6.10b}$$

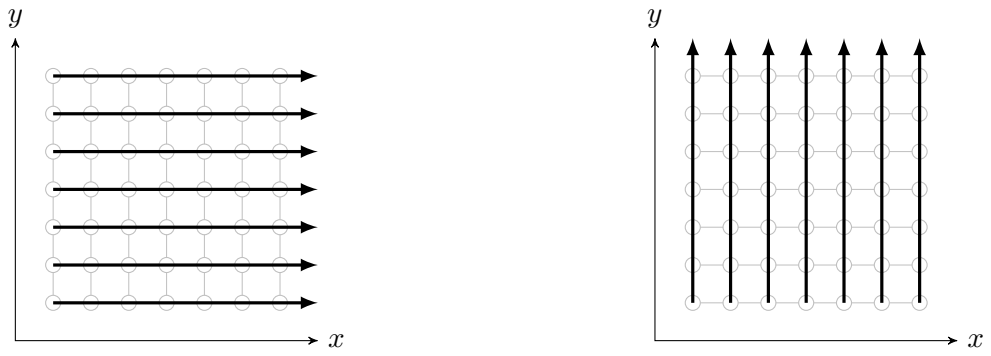
The differential marching operator for equation (6.10a) was seen in equation (6.7). Doing similar for equation (6.10b) we have

$$\mathcal{L}_Y(\Delta t) = 1 - w\Delta t\partial_y + \frac{w^2\Delta t^2}{2}\partial_{yy} + O(\Delta t^3). \tag{6.11}$$

Using dimensional splitting we solve a series of one-dimensional problems in one direction before solving another series of one-dimensional problems in the other direction(s). We can write this as

$$u_{i,j}^{n+1} = \mathcal{L}_Y(\Delta t)\mathcal{L}_X(\Delta t) u_{i,j}^n.$$

In this case we have chosen to solve in the x direction first (since $\mathcal{L}_X(\Delta t)$ is next to $u^n_{i,j}$) and then we use those solutions to solve in the y direction. This is shown pictorally in figure 6.2.



(a) x sweep (b) y sweep

Figure 6.2: Dimensional splitting using the sequence $\mathcal{L}_Y(\Delta t)\mathcal{L}_X(\Delta t)$.

In order for the split scheme to be equivalent to an unsplit scheme we need to show that $\mathcal{L}_Y(\Delta t)\mathcal{L}_X(\Delta t) = \mathcal{L}_{XY}(\Delta t)$

$$\begin{aligned}\mathcal{L}_Y(\Delta t)\mathcal{L}_X(\Delta t) &= \left(1 - w\Delta t\partial_y + \frac{w^2\Delta t}{2}\partial_{yy}\right)\left(1 - v\Delta t\partial_x + \frac{v^2\Delta t}{2}\partial_{xx}\right) + O(\Delta t^3)\\ &= 1 - \Delta t(v\partial_x + w\partial_y) + \frac{\Delta t^2}{2}(v^2\partial_{xx} + 2vw\partial_{xy} + w^2\partial_{yy}) + O(\Delta t^3)\\ &= \mathcal{L}_{XY}(\Delta t).\end{aligned}$$

So the split scheme is equivalent to the unsplit scheme. Note that if we applied the one-dimensional scheme in the y direction first we would still get the same result so the order in which the individual dimensional operators are applied doesn't matter. The choice of which one-dimensional scheme is used in the x and y directions will depend on the PDE being solved and the problem that it is applied to. For example, using the one-dimension FOU for both dimensions gives

$$u^*_{i,j} = u^n_{i,j} - \frac{v\Delta t}{\Delta x}(u^n_{i,j} - u^n_{i-1,j}), \tag{6.12a}$$

$$u^{n+1}_{i,j} = u^*_{i,j} - \frac{w\Delta t}{\Delta y}(u^*_{i,j} - u^*_{i,j-1}). \tag{6.12b}$$

The application of equation (6.12a) creates new values of the dependent variable at each node denoted by $u^*_{i,j}$. equation (6.12b) then uses these new values to update the values of the dependent variable for one full time step. The choice of the one-dimensional schemes do not have to be the same for each dimension although accuracy considerations may require that they be of the same order.

For stability, the maximum allowable time step for the split scheme is the smallest of the time steps for the individual one-dimensional schemes, e.g.,

$$\Delta t \leq \min\left(C_X\frac{\Delta x}{|v|}, C_Y\frac{\Delta y}{|w|}\right).$$

where C_X and C_Y are the Courant numbers that ensure the stability of the schemes used in the x and y directions respectively. This stability criterion is much less restrictive than that of the unsplit scheme equation (6.3).

The split FOU scheme has been applied to the same test problem described in section 6.1.1 and the solution is shown in figure 6.3. Here the numerical scheme exhibits the dissipation that is characteristic of first-order schemes, however, the anti-dissipation seen in the direction of flow using the unsplit scheme is no longer present.

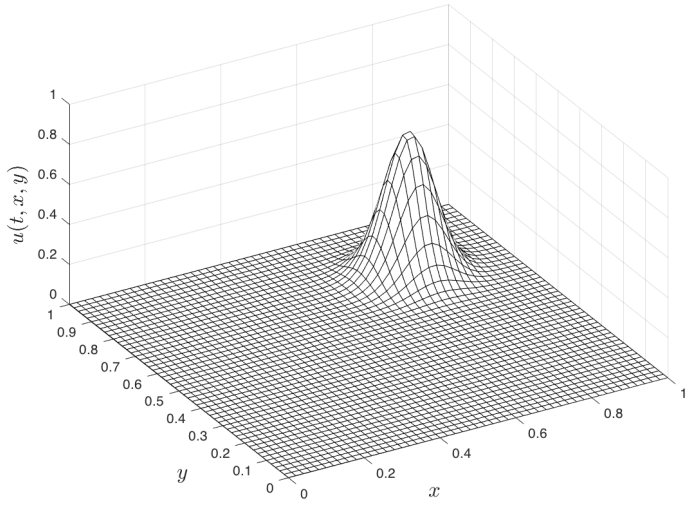


Figure 6.3: Solution of the two-dimensional advection equation using the split FOU scheme.

6.4 Term splitting

PDEs may contain several terms that model different physical processes. *Term splitting* can be used to split up the PDE and treats each term separately. For example, consider the one-dimensional advection-diffusion equation

The advection-diffusion equation

$$U_t + vU_x - \alpha U_{xx} = 0. \tag{6.13}$$

This PDE combines the advection equation equation (4.1) with the diffusion equation equation (5.1) to model transport and diffusion at the same time. Equation (6.13) is a parabolic PDE since it contains a U_{xx} term. Assuming $v > 0$ and using first-order differences for U_t and U_x and second-order difference for U_{xx} results in

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + v\frac{u_i^n - u_{i-1}}{\Delta x} - \alpha\frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} = 0,$$

which gives the following unsplit scheme

$$u_i^{n+1} = u_i^n - \frac{v\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \frac{\alpha\Delta t}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) + O(\Delta t, \Delta x). \tag{6.14}$$

To perform a von Neumann stability analysis on this scheme we can write it in the form

$$u_i^{n+1} = (C + D)u_{i-1}^n + (1 - C - 2D)u_i^n + Du_{i+1}^n,$$

where $C = \frac{v\Delta t}{\Delta x}$ and $D = \frac{\alpha\Delta t}{\Delta x^2}$. Substituting the Fourier nodes and applying the condition $|G| \leq 1$ results in (this bit is difficult)

$$1 - C - 2D \geq 0,$$

so the maximum allowable time step is

$$\Delta t \leq \frac{\Delta x^2}{v\Delta x + 2\alpha}. \tag{6.15}$$

Note that when $\alpha = 0$ we have the stability constraint for the heat diffusion equation equation (5.6) and when $v = 0$ we have the stability constraint for the FOU scheme when applied to the advection equation equation (4.6).

6.4.1 Differential marching operator for the advection-diffusion equation

To establish whether a term splitting is valid for the advection-diffusion equation we need to compare the differential marching operators for the unsplit equation against the combined differential marching operators for the advection and diffusion equations. Following the steps described in section 6.2.1 for equation (6.13)

$$\begin{aligned}\partial_t U &= \alpha\partial_{xx}U - v\partial_x U,\\ \partial_{tt}U &= v^2\partial_{xx}U - 2v\alpha\partial_{xxx}U + \alpha^2\partial_{xxxx}U.\end{aligned}$$

Substituting these into the second-order Taylor series gives

$$\begin{aligned}U(t+\Delta t) &= U + \Delta t\partial_t U + \frac{\Delta t^2}{2}\partial_{tt}U + O(\Delta t^3)\\ &= U + \Delta t(\alpha\partial_{xx} - v\partial_x)U + \frac{\Delta t^2}{2}(v^2\partial_{xx} - 2v\alpha\partial_{xxx} + \alpha^2\partial_{xxxx})U + O(\Delta t^3)\\ &= \left[1 + \Delta t(\alpha\partial_{xx} - v\partial_x) + \frac{\Delta t^2}{2}(v^2\partial_{xx} - 2v\alpha\partial_{xxx} + \alpha^2\partial_{xxxx})\right]U + O(\Delta t^3),\end{aligned}$$

therefore the differential marching operator for the advection-diffusion equation is

$$\mathcal{L}_{AD}(\Delta t) = 1 + \Delta t(\alpha\partial_{xx} - v\partial_x) + \frac{\Delta t^2}{2}(v^2\partial_{xx} - 2v\alpha\partial_{xxx} + \alpha^2\partial_{xxxx}) + O(\Delta t^3). \tag{6.16}$$

The differential marching operators for the advection equation and the diffusion equation are

$$\begin{aligned}\mathcal{L}_A(\Delta t) &= 1 - v\Delta t\partial_x + \frac{v^2\Delta t^2}{2}\partial_{xx} + O(\Delta t^3),\\ \mathcal{L}_D(\Delta t) &= 1 + \alpha\Delta t\partial_{xx} + \frac{\alpha^2\Delta t^2}{2}\partial_{xxxx} + O(\Delta t^3).\end{aligned}$$

Combining these two and truncating terms higher than second-order gives

$$\begin{aligned}\mathcal{L}_A(\Delta t)\mathcal{L}_D(\Delta t) &= \left(1 - v\Delta t\partial_x + \frac{v^2\Delta t^2}{2}\partial_{xx}\right)\left(1 + \alpha\Delta t\partial_{xx} + \frac{\alpha^2\Delta t^2}{2}\partial_{xxxx}\right) + O(\Delta t^3)\\ &= 1 + \alpha\Delta t\partial_{xx} - v\Delta t\partial_x - v\alpha\Delta t^2\partial_{xxx} + \frac{v^2\Delta t^2}{2}\partial_{xx} + \frac{\alpha^2\Delta t^2}{2}\partial_{xxxx} + O(\Delta t^3)\\ &= 1 + \Delta t(\alpha\partial_{xx} - v\partial_x) + \frac{\Delta t^2}{2}(v^2\partial_{xx} - v\alpha\partial_{xxx} + \alpha^2\partial_{xxxx}) + O(\Delta t^3)\\ &= \mathcal{L}_{AD}(\Delta t).\end{aligned}$$

Therefore the term split scheme is equivalent to the unsplit scheme. For stability we choose a value of the time step that satisfies both the advection scheme and the diffusion scheme. If we choose the FOU scheme to solve the advection term and the FTCS scheme to solve the diffusion term so to be consistent with equation (6.14), then we have seen previously in equations (4.6) and (5.6) that the time steps for the separate schemes are

$$\begin{aligned}\Delta t_A &= \frac{\Delta x}{v},\\ \Delta t_D &= \frac{\Delta x^2}{2\alpha}.\end{aligned}$$

Comparing these time steps with equation (6.15) we can see that

$$\begin{aligned}\Delta t_A &\geq \Delta t_{AD},\\ \Delta t_D &\geq \Delta t_{AD},\end{aligned}$$

so the split scheme will have a higher maximum allowable time step than the unsplit scheme. When using the split scheme we choose

$$\Delta t = \min(\Delta t_A, \Delta t_D).$$

6.4.2 Solving the advection-diffusion equation using a split scheme

Splitting equation (6.13) by the advection and diffusion terms gives

$$U_t + vU_x = 0, \tag{6.17a}$$
$$U_t - \alpha U_{xx} = 0. \tag{6.17b}$$

Applying the FOU scheme for equation (6.17a) and the FTCS scheme for equation (6.17b) results in the following split scheme

$$u_i^* = u_i^n - \frac{v\Delta t}{\Delta x}(u_i^n - u_{i-1}^n), \tag{6.18a}$$
$$u_i^{n+1} = u_i^* + \frac{\alpha\Delta t}{\Delta x^2}(u_{i-1}^* - 2u_i^* + u_{i+1}^*). \tag{6.18b}$$

where u_i^* denotes the values of the dependent variable updated using the first scheme as before. Equations (6.18a) and (6.18b) have been applied model advection-diffusion of the same Gaussian curve used in section 4.3. All of variables remain the same and the diffusion coefficient of $\alpha = 0.01$ was used. The solution for times $t = 0.0$, 0.1, 0.2, 0.3, 0.4 and 0.5 is shown in figure 6.4.

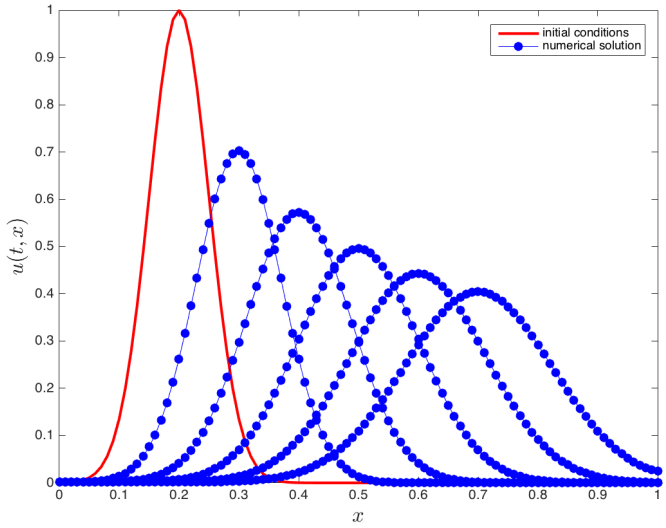


Figure 6.4: Solutions to the one-dimensional advection diffusion equation at times $t = 0$, 0.1, 0.2, 0.3, 0.4 and 0.5.

As the Gaussian curve travels down the channel the diffusion term causes it to dissipate. However, it was seen in figure 4.4 that the FOU scheme exhibits numerical dissipation when solving the advection equation so how much of this dissipation is due to the diffusion term and how much is because of the first-order scheme? The second-order Lax-Wendroff scheme has been used to solve the advection term in the split scheme and the numerical solutions are compared to the FOU scheme at time $t = 0.5$ in figure 6.5. This shows that at the peak of the curve, the first-order dissipation causes at 0.05 reduction in height.

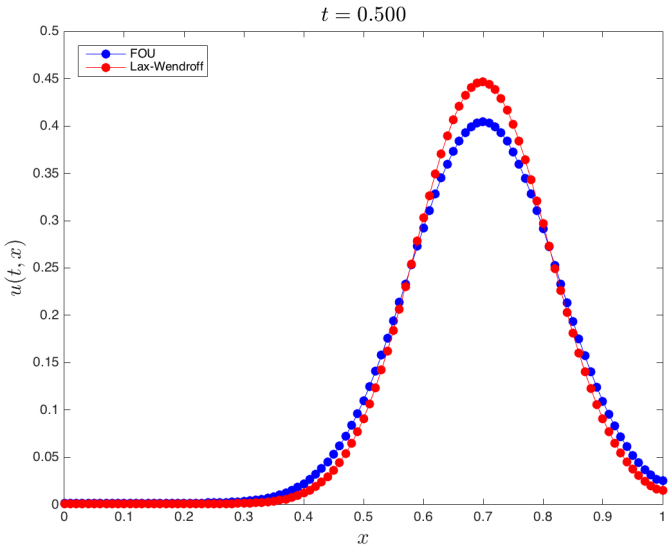


Figure 6.5: Comparison between the FOU and Lax-Wendroff schemes for solving the advection part of the split advection-diffusion equation.

The maximum time step for the unsplit scheme is $\Delta t = 0.0033$ and the split scheme is $\Delta t = 0.005$ which means that the unsplit scheme requires 50% more time steps than the split scheme.

Example 12

Consider the following two-dimensional PDE

$$U_t + \alpha U_x + \beta U_{yy} = 0.$$

(i) Derive the unsplit second-order differential marching $\mathcal{L}_{XY}(\Delta t)$ for this PDE.
(ii) Use dimensional splitting to split this 2D PDE into two 1D PDEs.
(iii) Derive the two second-order differential marching operators $\mathcal{L}_X(\Delta t)$ and $\mathcal{L}_Y(\Delta t)$ for each of the split PDEs.
(iv) Show that the split operator sequence $\mathcal{L}_X(\Delta t)\mathcal{L}_Y(\Delta t)$ is equivalent to the unsplit operator $\mathcal{L}_{XY}(\Delta t)$.

Solution:

(i) Using differential operator notation

$$\begin{aligned}
\partial_t U + \alpha\partial_x U + \beta\partial_{yy} U &= 0, \\
\therefore \partial_t U &= -\alpha\partial_x U - \beta\partial_{yy} U, \\
\partial_{tt} U &= -\partial_t(\alpha\partial_x U) - \partial_t(\beta\partial_{yy} U) \\
&= -\alpha\partial_x(\partial_t U) - \beta\partial_{yy}(\partial_t U) \\
&= -\alpha\partial_x(-\alpha\partial_x U - \beta\partial_{yy} U) - \beta\partial_{yy}(-\alpha\partial_x U - \beta\partial_{yy} U) \\
&= \alpha^2\partial_{xx} U + 2\alpha\beta\partial_{xyy} U + \beta^2\partial_{yyyy} U.
\end{aligned}$$

Substituting $\partial_t U$ and $\partial_{tt} U$ into the second-order Taylor series expansion

$$\begin{aligned} U(t+\Delta t) &= U + \Delta t \partial_t U + \frac{\Delta t^2}{2}\partial_{tt} U + O(\Delta t^3) \\ &= U + \Delta t(-\alpha\partial_x U - \beta\partial_{yy} U) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} U + 2\alpha\beta\partial_{xyy} U + \beta^2\partial_{yyyy} U) \\ &\quad + O(\Delta t^3) \\ &= \left[1 + \Delta t(-\alpha\partial_x - \beta\partial_{yy}) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} + 2\alpha\beta\partial_{xyy} + \beta^2\partial_{yyyy})\right] U + O(\Delta t^3) \end{aligned}$$

therefore

$$\mathcal{L}_{XY}(\Delta t) = 1 + \Delta t(-\alpha\partial_x - \beta\partial_{yy}) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} + 2\alpha\beta\partial_{xyy} + \beta^2\partial_{yyyy}) + O(\Delta t^3).$$

(ii) Using dimensional splitting the PDE can be written as the following system of two PDEs

$$U_t + \alpha U_x = 0, \tag{6.19}$$
$$U_t + \beta U_{yy} = 0. \tag{6.20}$$

(iii) Marching operator for equation (6.19):

$$\begin{aligned} \partial_t U &= -\alpha\partial_x U, \\ \partial_{tt} U &= -\alpha\partial_t(\partial_x U) = -\alpha\partial_x(\partial_t U) = -\alpha\partial_x(-\alpha\partial_x U) = \alpha^2\partial_{xx} U, \\ \therefore U(t+\Delta t) &= U + \Delta t(-\alpha\partial_x U) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} U) + O(\Delta t^3) \\ &= \left[1 - \alpha\Delta t\partial_x + \frac{\alpha^2\Delta t^2}{2}\partial_{xx}\right] U + O(\Delta t^3), \end{aligned}$$

so

$$\mathcal{L}_X(\Delta t) = 1 - \alpha\Delta t\partial_x + \frac{\alpha^2\Delta t^2}{2}\partial_{xx} + O(\Delta t^3).$$

Marching operator for equation (6.20):

$$\begin{aligned} \partial_t U &= -\beta\partial_{yy} U, \\ \partial_{tt} U &= -\beta\partial_t(\partial_{yy} U) = -\beta\partial_{yy}(\partial_t U) = -\beta\partial_{yy}(-\beta\partial_{yy} U) = \beta^2\partial_{yyyy} U, \\ \therefore U(t+\Delta t) &= U + \Delta t(-\beta\partial_{yy} U) + \frac{\Delta t^2}{2}(\beta^2\partial_{yyyy} U) + O(\Delta t^3) \\ &= \left[1 - \beta\Delta t\partial_{yy} + \frac{\beta^2\Delta t^2}{2}\partial_{yyyy}\right] U + O(\Delta t^3), \end{aligned}$$

so

$$\mathcal{L}_Y(\Delta t) = 1 - \beta\Delta t\partial_{yy} + \frac{\beta^2\Delta t^2}{2}\partial_{yyyy}.$$

(iv) Combining the split marching operators $\mathcal{L}_X(\Delta t)$ and $\mathcal{L}_Y(\Delta t)$

$$\begin{aligned} \mathcal{L}_X(\Delta t)\mathcal{L}_Y(\Delta t) &= \left(1 - \alpha\Delta t\partial_x + \frac{\alpha^2\Delta t^2}{2}\partial_{xx}\right)\left(1 - \beta\Delta t\partial_{yy} + \frac{\beta^2\Delta t^2}{2}\partial_{yyyy}\right) + O(\Delta t^3) \\ &= 1 - \beta\Delta t\partial_{yy} + \frac{\beta^2\Delta t^2}{2}\partial_{yyyy} - \alpha\Delta t\partial_x + \alpha\beta\Delta t^2\partial_{xyy} - \frac{\alpha\beta^2\Delta t^3}{2}\partial_{xyyyy} \\ &\quad + \frac{\alpha^2\Delta t^2}{2}\partial_{xx} - \frac{\alpha^2\beta\Delta t^3}{2}\partial_{xxyy} + \frac{\alpha^2\beta^2\Delta t^4}{4}\partial_{xxyyyy} + O(\Delta t^3) \\ &= 1 + \Delta t(-\alpha\partial_x - \beta\partial_y) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} + 2\alpha\beta\partial_{xyy} + \beta^2\partial_{yyyy}) \\ &\quad + \frac{\Delta t^3}{2}(-\alpha\beta^2\partial_{xyyyy} - \alpha^2\beta\partial_{xxyy}) + \frac{\Delta t^4}{4}\alpha^2\beta^2\partial_{xxyyyy} + O(\Delta t^3). \end{aligned}$$

Since the terms involving Δt^3 and Δt^4 are not $O(\Delta t^3)$ they can be omitted and we have

$$\mathcal{L}_X(\Delta t)\mathcal{L}_Y(\Delta t) = 1 + \Delta t(-\alpha\partial_x - \beta\partial_y) + \frac{\Delta t^2}{2}(\alpha^2\partial_{xx} + 2\alpha\beta\partial_{xyy} + \beta^2\partial_{yyyy}) + O(\Delta t^3)$$

which is the same as $\mathcal{L}_{XY}(\Delta t)$ from part (i).

6.5 Operator splitting for stiff problems

Cases may arise in split systems where the maximum allowable time step for one scheme may be much less than that of the other schemes. For example, if dimensional splitting was used to split up a PDE that models some kind of flow in a two-dimensional domain then the stability criteria are of the form

$$\Delta t_X = \frac{\Delta x}{|v|},$$
$$\Delta t_Y = \frac{\Delta y}{|w|}.$$

If $v \gg w$ ($\gg$ means 'much greater than') and if the spatial steps are roughly equal then it follows that $\Delta t_X \ll \Delta t_Y$. If the sequence $\mathcal{L}_Y(\Delta t)\mathcal{L}_X(\Delta t)$ is used then for stability both schemes will need to use the same value of Δt which in this cases will be the much more restrictive Δt_X. So even though the scheme for solving in the y direction is stable for a much larger value of Δt, it is constrained by the stability of the scheme in the x direction. This is an example of a *stiff* problem.

Fortunately, we can form a sequence where each scheme uses a different value of Δt whilst the whole sequence remains stable. Suppose that Δt_Y is some multiple of Δt_X, i.e., $\Delta t_Y = n\Delta t_X$ then the following sequence can be used,

$$\mathcal{L}_Y(\Delta t_Y)\underbrace{\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)\ldots\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)}_{n \text{ times}} = \mathcal{L}_Y(\Delta t_Y)\left[\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)\right]^n, \tag{6.21}$$

i.e., a single application of $\mathcal{L}_Y(\Delta t_Y)$ followed by n applications of $\mathcal{L}_X(\frac{\Delta t_Y}{n})$. To show that this is a valid sequence consider the first-order differential marching operator for the one-dimensional advection equation

$$\mathcal{L}(\Delta t) = 1 - v\Delta t\partial_x.$$

Applying this to equation (6.21) and multiplying out using the binomial theorem

$$\begin{aligned}
\mathcal{L}_Y(\Delta t_Y)\left[\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)\right]^n &= (1 - w\Delta t_Y\partial_y)\left[1 - v\left(\frac{\Delta t_Y}{n}\right)\partial_x\right]^n \\
&= (1 - w\Delta t_Y\partial_y)\left[1 - nv\left(\frac{\Delta t_Y}{n}\right)\partial_x + O(\Delta t^2)\right] \\
&= (1 - w\Delta t_Y\partial_y)[1 - v\Delta t_Y\partial_x + O(\Delta t^2)] \\
&= 1 - \Delta t_Y(v\partial_x + w\partial_y) + O(\Delta t^2) \\
&= \mathcal{L}_{XY}(\Delta t_Y),
\end{aligned}$$

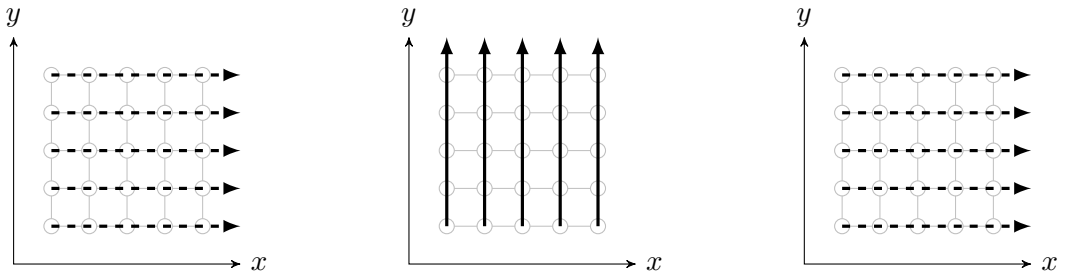
which is the same as the first-order form of equation (6.8(). Therefore the sequence in equation (6.21) is valid. If $\mathcal{L}_X(\Delta t_X)$ and $\mathcal{L}_Y(\Delta t_Y)$ are second-order and n is even then it can be shown that

$$\mathcal{L}_{XY}(\Delta t_Y) = \left[\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)\right]^{\frac{n}{2}}\mathcal{L}_Y(\Delta t_Y)\left[\mathcal{L}_X\left(\frac{\Delta t_Y}{n}\right)\right]^{\frac{n}{2}}. \tag{6.22}$$

An operator sequence of the form given in equation (6.22) is called a *symmetric operator sequence*. One particular example of a symmetric operator uses $n = 2$, i.e.,

$$\mathcal{L}_{XY}(\Delta t) = \mathcal{L}_X\left(\frac{\Delta t}{2}\right)\mathcal{L}_Y(\Delta t)\mathcal{L}_X\left(\frac{\Delta t}{2}\right), \tag{6.23}$$

which is known as *Strang splitting* after American mathematician Gilbert Strang (1934–present) (figure 6.6).



(a) x sweep using $\frac{\Delta t}{2}$ (b) y sweep using Δt (c) x sweep using $\frac{\Delta t}{2}$

Figure 6.6: Strang splitting using the sequence $\mathcal{L}_X(\frac{\Delta t}{2})\mathcal{L}_Y(\Delta t)\mathcal{L}_X(\frac{\Delta t}{2})$.

Chapter 7

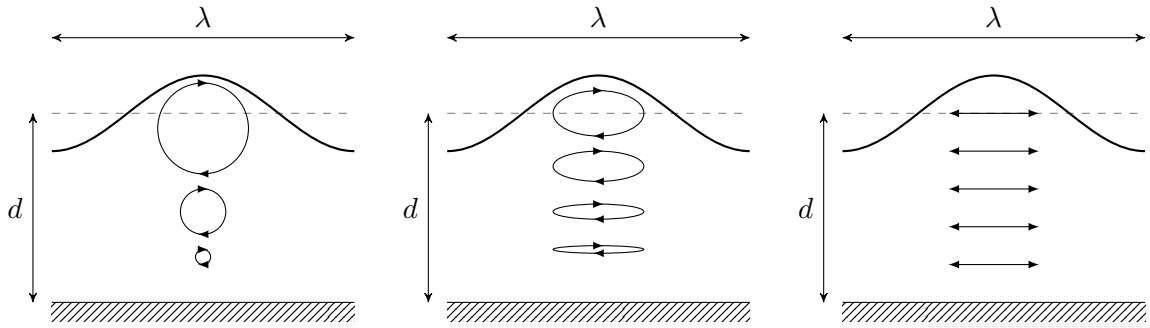
Solving Systems of Partial Differential Equations

The universe is governed by physical laws where variables are intrinsically linked. For example, the behaviour of a fluid depends upon the velocity, density, pressure, temperature, viscosity etc. Any change in one of these variables will have an effect on all of the others. Also in order to model physical phenomena we need to make sure that the relationships between the variables satisfy certain requirements. For example, at its most basic level, the modelling of fluid flow needs to ensure the conservation of mass and momentum in the system, i.e., mass and energy can't be added or taken away*. This means that instead of using a single PDE to model physical behaviour, a system of PDEs is required. We can use the numerical schemes and splitting methods developed for single PDEs on a system of PDEs with some minor changes. This chapter will use the shallow water equations to demonstrate the solution of a system of PDEs but the concepts are similar for other systems.

7.1 The Shallow Water Equations

The Shallow Water Equations (SWE) are a hyperbolic system of PDEs that model water flow in shallow regions. The derivation of the SWE is achieved by taking the Navier-Stokes system of PDEs and depth-averaging by assuming the flow in the vertical direction is negligible. This may seem like an odd assumption but it is valid due to the physics of water flow in shallow water. Consider figure 7.1 that shows the motion orbits of a neutrally buoyant particle in different water depths. The water depth d is defined as the distance between the bed surface and the mean water level. The wavelength λ for a regular wave is defined as the distance between the corresponding points on two successive waves (i.e., the troughs). The depth-to-wavelength ratio is used to classify water in three regions. When $\frac{d}{\lambda} > \frac{1}{2}$ the water is considered *deep* and the motion orbits are roughly circular with the radius decreasing quadratically as the distance under the surface increases. When $\frac{1}{20} \leq \frac{d}{\lambda} < \frac{1}{2}$ the water is considered to be in *intermediate depth* water and the motion orbits become elliptic with the minor axis decreasing as the depth under the water surface increases. When $\frac{d}{\lambda} < \frac{1}{20}$ the water is considered *shallow* and the motion of the particle follows a horizontal orbit with very little motion in the vertical direction. Therefore the assumption that vertical velocity is negligible in shallow water is valid.

*This only applies to the modelling of the behaviour of the fluid and not necessary the complete system being modelled. For example, when modelling water waves we may need to add mass or momentum into the system (using *sink* or *source* terms) to generate the waves. The conservation of mass and momentum in the governing equations ensures that the fluid behaves as required.



(a) Deep: $\frac{d}{\lambda} \geq \frac{1}{2}$ (b) Intermediate: $\frac{1}{20} \leq \frac{d}{\lambda} < \frac{1}{2}$ (c) Shallow: $\frac{d}{\lambda} < \frac{1}{20}$

Figure 7.1: The motion of a particle depends on the depth to wavelength ratio.

In one-dimension, the SWE are a system of two PDEs which are written as

The one-dimensional Shallow Water Equations (SWE)

$$h_t + (hu)_x = 0, \tag{7.1a}$$

$$(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x = -ghz_x. \tag{7.1b}$$

where h is the water depth, u is the horizontal velocity, $g = 9.81\text{ms}^{-2}$ is the acceleration due to gravity and z is the bed surface elevation (figure 7.2). Equation (7.1a) ensures that we have *conservation of mass*, h, and that no mass is added or removed from the system. Equation (7.1b) ensures that we have *conservation of momentum*, hu, which means that no energy is added or taken away. The term on the right-hand side of equation (7.1b) is the *bed source term* and ensures that change in velocity caused by changes in the bed topography is balanced by the change in momentum.

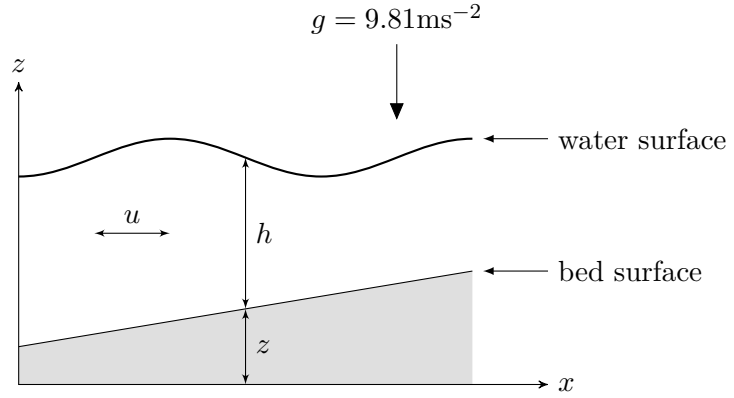


Figure 7.2: Definition of variables used in the shallow water equations.

It is more convenient to write the SWE in vector form as

$$U_t + F_x = S,$$

where U is the *vector of conserved variables*, F is the *flux vector* that models the changes in the conserved variables in the x direction and S is the vector of *source terms*, i.e.,

$$U = \begin{pmatrix} h \\ hu \end{pmatrix}, \qquad F = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{pmatrix}, \qquad S = \begin{pmatrix} 0 \\ -ghz_x \end{pmatrix}.$$

In addition to modelling shallow water flows, the SWE can be used to model tsunami propagation. A tsunami is generated by seismic activity on the ocean floor that causes the displacement of a large volume of water. This results in a *solitary* wave with a wavelength up to $\lambda = 200$km and a height of just 2m travelling at approximately 800kms^{-1}[†]. Since the average depths of the oceans are typically in region of 4km, the depth-to-wavelength ratio for a tsunami is 0.02, well within the range of applicability of the SWE. As the tsunami propagates into shallower water it begins to steepen and a bore wave forms which can be up to 5m in height. The damage that a tsunami can cause to coastal regions is catastrophic, one of the worst examples in recent years was the Indian ocean tsunami on Boxing day 2004 that claimed the lives of more than 280,000 people.

7.2 Estimating the wave speeds of the SWE

We saw in section 4.5.2 that in order for a scheme to remain stable, the CFL condition states that the time step cannot exceed the spatial step divided by the velocity. For linear PDEs such as the advection equation the velocity is constant, the SWE is a non-linear system meaning that the velocity is variable across the domain. We need to determine the fastest and slowest possible velocities permitted by the SWE to ensure that the calculation of the time step never violates the CFL condition. To do this we linearise the homogeneous form of the SWE (i.e., ignoring the source terms)

$$U_t + F_x = 0, \tag{7.2}$$

where

$$U = \begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = \begin{pmatrix} h \\ hu \end{pmatrix}, \qquad F = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{pmatrix}.$$

by writing it in the form

$$U_t + JU_x = 0, \tag{7.3}$$

where J is the *Jacobian matrix* defined by

Definition 12: The Jacobian matrix

$$J = \frac{\partial F}{\partial U} = \begin{pmatrix} \frac{\partial F_1}{\partial U_1} & \frac{\partial F_1}{\partial U_2} \\ \frac{\partial F_2}{\partial U_1} & \frac{\partial F_2}{\partial U_2} \end{pmatrix}. \tag{7.4}$$

To evaluate J we can write F using U_1 and U_2 such that

$$F = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} U_2 \\ U_2^2 U_1^{-1} + \frac{1}{2}gU_1^2 \end{pmatrix},$$

Evaluating the partial derivatives in the Jacobian gives

$$\begin{aligned}
\frac{\partial F_1}{\partial U_1} &= \frac{\partial U_2}{\partial U_1} = 0, \\
\frac{\partial F_1}{\partial U_2} &= \frac{\partial U_2}{\partial U_2} = 1, \\
\frac{\partial F_2}{\partial U_1} &= \frac{\partial}{\partial U_1}\left(U_2^2 U_1^{-1} + \frac{1}{2}gU_1^2\right) = -U_2^2 U_1^{-2} + gU_1 = -u^2 + gh, \\
\frac{\partial F_2}{\partial U_2} &= \frac{\partial}{\partial U_2}\left(U_2^2 U_1^{-1} + \frac{1}{2}gU_1^2\right) = 2U_2 U_1^{-1} = 2u.
\end{aligned}$$

[†]A feature of tsunamis is that if you were on a small boat out at sea a tsunami wave would be barely noticeable. It is only when the tsunami reaches shallower regions that it steepens to form a bore wave. Fisherman would sometimes return from being out at sea to find that their homes and communities had been destroyed by a tsunami when they didn't notice anything unusual when at sea.

so the Jacobian is

$$J = \begin{pmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{pmatrix},$$

and the SWE can be written as

$$\begin{pmatrix} h \\ hu \end{pmatrix}_t + \begin{pmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{pmatrix} \begin{pmatrix} h \\ hu \end{pmatrix}_x = 0. \tag{7.5}$$

J has two eigenvalues $\lambda_1 = u + \sqrt{gh}$ and $\lambda_2 = u - \sqrt{gh}$. These are real and distinct as long as $h \geq 0$ which classifies equation (7.2) as a system of hyperbolic PDEs. We want to split equation (7.5) into two separate PDEs that are analogous to the advection equation. To do this we diagonalise J to produce a matrix D where the eigenvalues of J form the main diagonal, i.e.,

$$D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}.$$

Let $P = \begin{pmatrix} \mathbf{p}_1 & \mathbf{p}_2 \end{pmatrix}$ where $\mathbf{p}_1$ and $\mathbf{p}_2$ are the column eigenvectors corresponding to λ_1 and λ_2 then

$$\begin{aligned} JP &= J \begin{pmatrix} \mathbf{p}_1 & \mathbf{p}_2 \end{pmatrix} \\ &= \begin{pmatrix} J\mathbf{p}_1 & J\mathbf{p}_2 \end{pmatrix} \\ &= \begin{pmatrix} \lambda_1\mathbf{p}_1 & \lambda_2\mathbf{p}_2 \end{pmatrix} && \text{(since } J\mathbf{p} = \lambda\mathbf{p}\text{)} \\ &= \begin{pmatrix} \mathbf{p}_1 & \mathbf{p}_2 \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \\ &= PD \\ P^{-1}JP &= D. \end{aligned}$$

Therefore $P^{-1}JP$ is the transformation that diagonalises J. Introducing a change in variable where $W = (W_1, W_2)^T$ such that $U = PW$ then equation (7.3) becomes

$$PW_t + JPW_x = 0,$$

premultiplying by P^{-1} and simplifying gives

$$\begin{aligned} P^{-1}PW_t + P^{-1}JPW_x &= 0 \\ W_t + DW_x &= 0. \end{aligned}$$

D is a diagonal matrix so we can write this as a system of two linear equations

$$\frac{\partial}{\partial t}(W_1) + \left(u + \sqrt{gh}\right) \frac{\partial}{\partial x}(W_1) = 0, \tag{7.6a}$$

$$\frac{\partial}{\partial t}(W_2) + \left(u - \sqrt{gh}\right) \frac{\partial}{\partial x}(W_2) = 0. \tag{7.6b}$$

Equations (7.6a) and (7.6b) are two linear advection equations with wave speeds $u + \sqrt{gh}$ and $u - \sqrt{gh}$ (the eigenvalues of J). These wave speeds are the fastest and slowest speeds permissible by the SWE. Since we are interested in the maximum absolute wave speed for stability purposes we can see that

$$\max\left(|u + \sqrt{gh}|, |u - \sqrt{gh}|\right) = |u| + \sqrt{gh}.$$

7.2.1 Calculating the time step for the SWE

Any explicit scheme to solve the SWE must take into account these wave speeds when determining the maximum allowable time step for stability. Recall that the maximum allowable time step for the advection equation is

$$\Delta t \leq \frac{\Delta x}{|v|}. \tag{7.7}$$

Since the wave speeds for the SWE are $|u| + \sqrt{gh}$ then the maximum allowable time step for the SWE is

$$\Delta t \leq \frac{\Delta x}{|u| + \sqrt{gh}}. \tag{7.8}$$

Equation (7.8) calculates a separate value of Δt for each point in the domain. We require a single value of Δt so that a numerical scheme is stable for all points in the domain, therefore we choose the smallest value of Δt, i.e.,

Maximum allowable time step for the SWE

$$\Delta t \leq \frac{\Delta x}{\max\limits_i(|u_i| + \sqrt{gh_i})}. \tag{7.9}$$

Since the values of h and u will change over time, equation (7.9) is calculated at the beginning of each time step (note that in practice the maximum allowable time step is multiplied by some safety, e.g., 0.9, factor to ensure stability).

7.3 Solving the shallow water equations using finite-difference schemes

The FDS developed for the advection equation in chapter 4 can be modified to solve the SWE. Recall that the Lax-Friedrichs scheme for solving the advection equation is

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - v\frac{\Delta t}{2\Delta x}(u_{i+1}^n - u_{i-1}^n) + O(\Delta t, \Delta x).$$

The velocity in the advection equation, v, is constant in space and time. The velocity in the SWE is variable in space and time and is governed by the flux vector F that is differentiated with respect to x. Therefore we can replace the $u_{i\pm1}^n$ terms in the spatial derivative with the corresponding flux terms and remove v giving the Lax-Friedrichs scheme for solving the SWE

The Lax-Friedrichs scheme for solving the one-dimensional SWE

$$U_i^{n+1} = \frac{1}{2}(U_{i+1}^n + U_{i-1}^n) - \frac{\Delta t}{2\Delta x}(F_{i+1}^n - F_{i-1}^n) + O(\Delta t, \Delta x). \tag{7.10}$$

7.3.1 Solution procedure

The solution procedure for solving the SWE is similar to that used for the advection equation with a couple of exceptions. Unlike the advection equation where the velocity, and therefore the maximum allowable time step Δt, is constant, the velocities in the SWE are variable so Δt is calculated at the beginning of each time step. Also the flux vector is calculated before the finite-difference scheme is applied to march the solution forward in time.

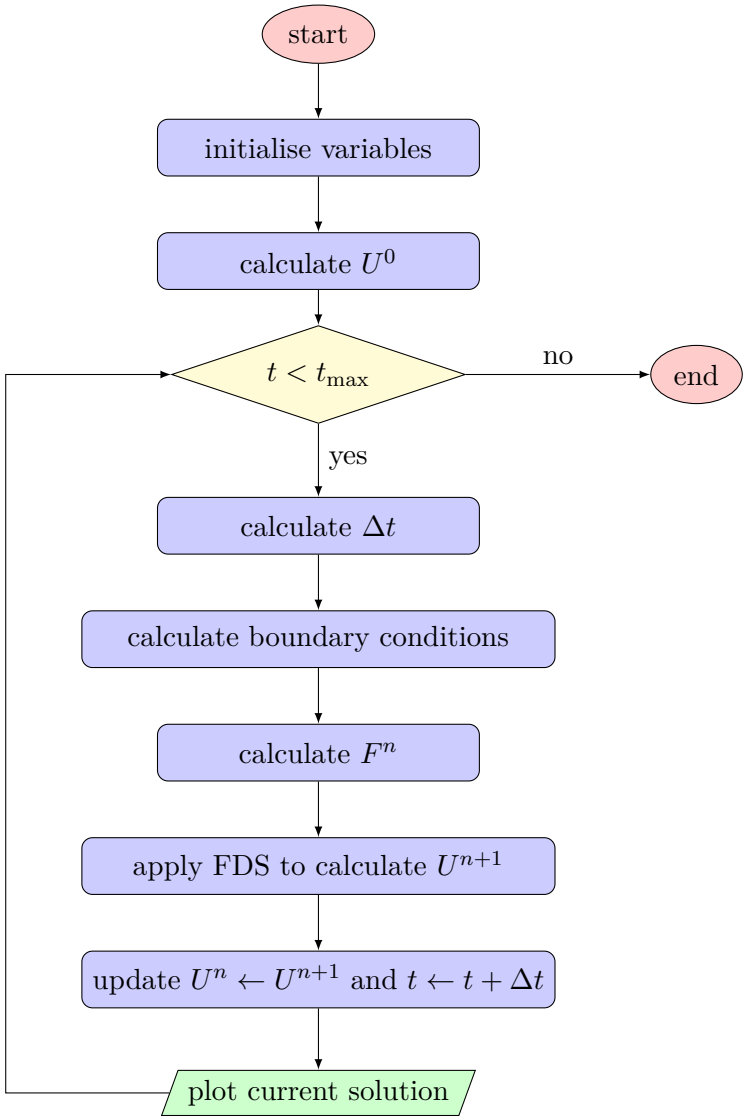


Figure 7.3: Flow chart for solving the SWE.

Example 13

The SWE are being used to model a one-dimensional dam break. The domain is 2m in length with a partition located at $x = 1$m separating two bodies of water with heights 5m and 2m for the upstream and downstream bodies respectively. First-order transient flow boundary conditions are employed at both ends of the domain that is discretised into a reduced grid with $\Delta x = 0.5$m for verification purposes. Acceleration due to gravity is assumed to be $g = 9.81\text{ms}^{-2}$.

Calculate two steps of the Lax-Friedrichs scheme for this problem.

Solution: Since $\Delta x = 0.5$m the co-ordinates of the finite-differences nodes are

$$x_0 = 0.0, \qquad x_1 = 0.5, \qquad x_2 = 1.0, \qquad x_3 = 1.5, \qquad x_4 = 2.0,$$

so the initial conditions are

$$\begin{aligned} &h_0 = 5, \quad h_1 = 5, \quad h_2 = 5, \quad h_3 = 2, \quad h_4 = 2, \\ &u_0 = 0, \quad u_1 = 0, \quad u_2 = 0, \quad u_3 = 0, \quad u_4 = 0. \end{aligned}$$

Since zero gradient boundary conditions are used at both boundaries, we have

$$\begin{aligned} h_{-1} &= h_1 = 5, & h_5 &= h_3 = 2, \\ u_{-1} &= u_1 = 0, & u_5 &= u_3 = 0. \end{aligned}$$

We form an array where each column contains the vector $U = (h, hu)^T$ for each node (including the ghost nodes)

$$U^0 = \begin{pmatrix} 5 & 5 & 5 & 5 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and do similar for the flux vector $F = (hu, hu^2 + \frac{1}{2}gh^2)^T$

$$F^0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 122.625 & 122.625 & 122.625 & 122.625 & 19.62 & 19.62 & 19.62 \end{pmatrix}$$

Calculate the maximum allowable time step (using a safety factor of 0.9)

$$\Delta t = 0.9 \left(\frac{\Delta x}{\max_i(|u_i| + \sqrt{g h_i})} \right) = 0.9 \left(\frac{0.5}{\sqrt{5g}} \right) = 0.0643.$$

Now perform one step of the Lax-Friedrichs scheme equation (7.10)

$$\begin{aligned} U_0^1 &= \frac{1}{2}\left[\begin{pmatrix}5\\0\end{pmatrix} + \begin{pmatrix}5\\0\end{pmatrix}\right] - 0.0643\left[\begin{pmatrix}0\\122.625\end{pmatrix} - \begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}5\\0\end{pmatrix}, \\ U_1^1 &= \frac{1}{2}\left[\begin{pmatrix}5\\0\end{pmatrix} + \begin{pmatrix}5\\0\end{pmatrix}\right] - 0.0643\left[\begin{pmatrix}0\\122.625\end{pmatrix} - \begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}5\\0\end{pmatrix}, \\ U_2^1 &= \frac{1}{2}\left[\begin{pmatrix}5\\0\end{pmatrix} + \begin{pmatrix}2\\0\end{pmatrix}\right] - 0.0643\left[\begin{pmatrix}0\\19.62\end{pmatrix} - \begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}3.5\\6.6184\end{pmatrix}, \\ U_3^1 &= \frac{1}{2}\left[\begin{pmatrix}5\\0\end{pmatrix} + \begin{pmatrix}2\\0\end{pmatrix}\right] - 0.0643\left[\begin{pmatrix}0\\19.62\end{pmatrix} - \begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}3.5\\6.6184\end{pmatrix}, \\ U_4^1 &= \frac{1}{2}\left[\begin{pmatrix}2\\0\end{pmatrix} + \begin{pmatrix}2\\0\end{pmatrix}\right] - 0.0643\left[\begin{pmatrix}0\\19.62\end{pmatrix} - \begin{pmatrix}0\\19.62\end{pmatrix}\right] = \begin{pmatrix}2\\0\end{pmatrix}. \end{aligned}$$

The conserved values can be extracted from $U = (U_1, U_2)^T$ using $h = U_1$ and $u = \dfrac{U_2}{U_1}$ therefore

$$\begin{aligned} &h_0 = 5, \quad h_1 = 5, \quad h_2 = 3.5, \quad h_3 = 3.5, \quad h_4 = 2, \\ &u_0 = 0, \quad u_1 = 0, \quad u_2 = 1.8910, \quad u_3 = 1.8910, \quad u_4 = 0. \end{aligned}$$

Moving onto the second step we first calculate the maximum allowable time step

$$\Delta t = 0.9\left(\frac{0.5}{1.8923+\sqrt{3.5g}}\right) = 0.0581,$$

and using the boundary conditions to calculate the values of the ghost nodes

$$\begin{aligned} h_{-1} &= h_1 = 5, & h_5 &= h_3 = 3.5,\\ u_{-1} &= u_1 = 0, & u_5 &= u_3 = 1.8910. \end{aligned}$$

The U^1 and F^1 arrays are

$$\begin{aligned} U^1 &= \begin{pmatrix} 5 & 5 & 5 & 3.5 & 3.5 & 2 & 3.5 \\ 0 & 0 & 0 & 6.6184 & 6.6184 & 0 & 6.6184 \end{pmatrix},\\ F^1 &= \begin{pmatrix} 0 & 0 & 0 & 6.6184 & 6.6184 & 0 & 6.6184 \\ 122.625 & 122.625 & 122.625 & 72.6014 & 72.6014 & 19.62 & 72.6014 \end{pmatrix}. \end{aligned}$$

Performing another step of the Lax-Friedrichs scheme

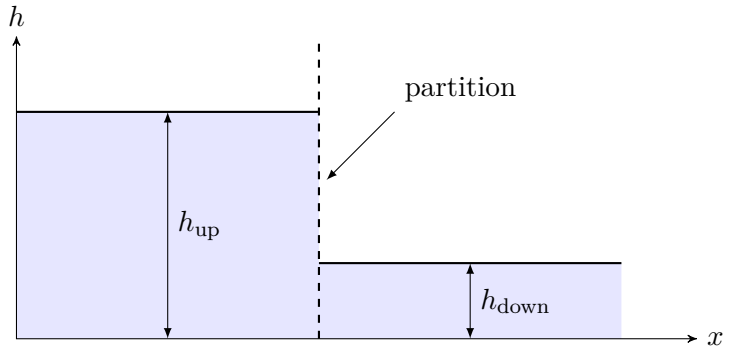
$$\begin{aligned} U_0^2 &= \frac{1}{2}\left[\begin{pmatrix}5\\0\end{pmatrix}+\begin{pmatrix}5\\0\end{pmatrix}\right] - 0.0581\left[\begin{pmatrix}0\\122.625\end{pmatrix}-\begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}5\\0\end{pmatrix},\\ U_1^2 &= \frac{1}{2}\left[\begin{pmatrix}3.5\\6.6184\end{pmatrix}+\begin{pmatrix}5\\0\end{pmatrix}\right] - 0.0581\left[\begin{pmatrix}0\\72.6014\end{pmatrix}-\begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}3.8657\\6.2136\end{pmatrix},\\ U_2^2 &= \frac{1}{2}\left[\begin{pmatrix}3.5\\6.6184\end{pmatrix}+\begin{pmatrix}5\\0\end{pmatrix}\right] - 0.0581\left[\begin{pmatrix}0\\72.6014\end{pmatrix}-\begin{pmatrix}0\\122.625\end{pmatrix}\right] = \begin{pmatrix}3.8657\\6.2136\end{pmatrix},\\ U_3^2 &= \frac{1}{2}\left[\begin{pmatrix}2\\0\end{pmatrix}+\begin{pmatrix}3.5\\6.6184\end{pmatrix}\right] - 0.0581\left[\begin{pmatrix}0\\19.62\end{pmatrix}-\begin{pmatrix}6\\72.6014\end{pmatrix}\right] = \begin{pmatrix}3.1343\\6.3853\end{pmatrix},\\ U_4^2 &= \frac{1}{2}\left[\begin{pmatrix}3.5\\6.6184\end{pmatrix}+\begin{pmatrix}3.5\\6.6184\end{pmatrix}\right] - 0.0581\left[\begin{pmatrix}6.6184\\72.6014\end{pmatrix}-\begin{pmatrix}6.6184\\72.6014\end{pmatrix}\right] = \begin{pmatrix}3.5\\6.6184\end{pmatrix}. \end{aligned}$$

The values of the conserved variables after two steps of the Lax-Friedrichs scheme are

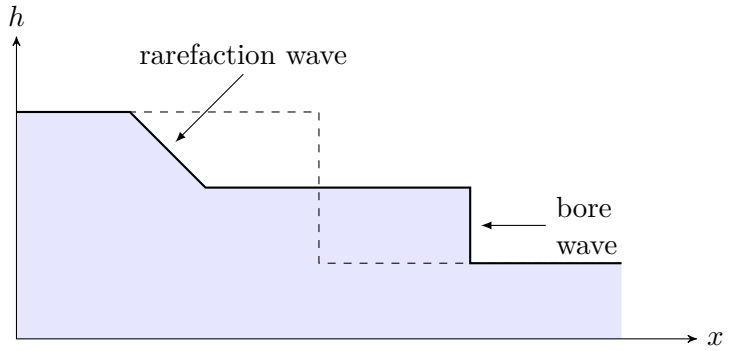
$$\begin{aligned} h_0 &= 5, & h_1 &= 3.8657, & h_2 &= 3.8657, & h_3 &= 3.1343, & h_4 &= 3.5,\\ u_0 &= 0, & u_1 &= 1.6073, & u_2 &= 1.6073, & h_3 &= 2.0373, & h_4 &= 1.8910. \end{aligned}$$

7.3.2 The one-dimensional dam break test

To test a numerical scheme for solving the SWE we can use the *one-dimensional dam break* test. This consists of a one-dimensional channel with two bodies or reservoirs of water separated by a partition or dam (figure 7.4(a)). At time $t = 0$ the partition is instantaneously removed and the two bodies of water are allowed to interact. The collapse of the upstream reservoir causes a *bore wave* to form the travels downstream (figure 7.4(b)). The loss of mass due to this bore wave causes an upstream travelling *rarefaction wave*, or depression wave, to form.



(a) initial conditions



(b) solution at time t

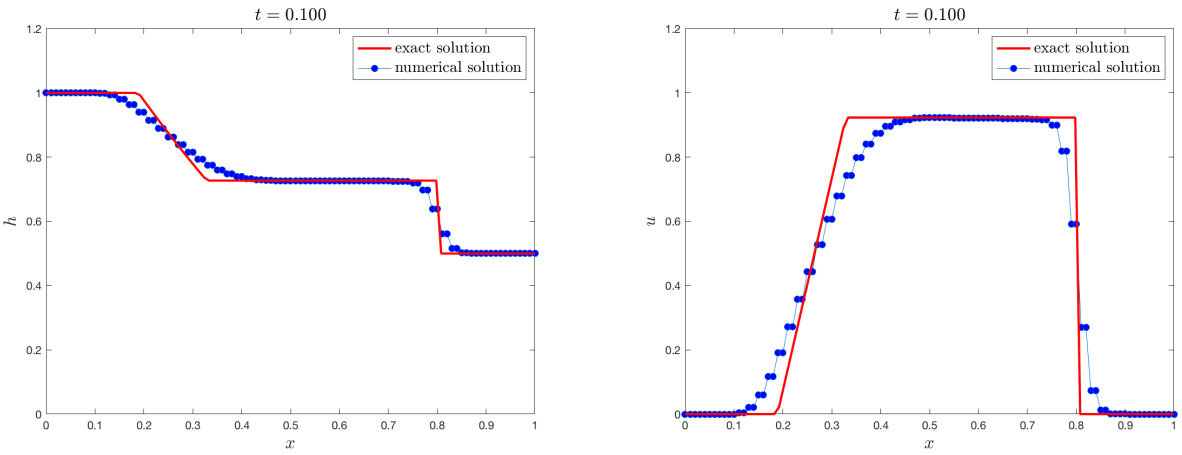
Figure 7.4: The one-dimensional dam break problem.

The dam break problem is a tough test for FDS since the initial conditions contain a discontinuity which is also present in the solutions in the form of a bore wave. The one-dimensional dam break problem is one of the few problems for the SWE where it is possible to calculate an analytical solution.

The Lax-Friedrichs scheme shown in equation (7.10) has been applied to the one-dimensional dam break problem. The domain $0 \leq x \leq 1$ is discretised using $N = 101$ nodes giving $\Delta x = 0.01$ with a partition located at $x = 0.5$ separating the reservoirs to the upstream (left) and downstream (right) sides of the partition with water depths $h_{\text{up}} = 1$ and $h_{\text{down}} = 0.5$ respectively. Transmissive boundary conditions using ghost nodes were employed at both ends of the domain so that waves will pass through the boundary without reflection, i.e.,

$$\begin{aligned} h_{-1} &= h_1, & h_N &= h_{N-2}, \\ u_{-1} &= u_1, & u_N &= u_{N-2}. \end{aligned}$$

The solutions for the water depth h and velocity u are shown in figure 7.5. Similar to the advection equation, the Lax-Friedrichs scheme exhibits numerical dissipation which means the numerical solution does not capture the discontinuity at the bore wave and smooths out the rarefaction wave. In order to capture these phenomena we require a higher order scheme.



(a) water height

(b) velocity

Figure 7.5: Solution to the one-dimensional dam break at time $t = 0.1$s using the Lax-Friedrichs scheme

7.3.3 MATLAB code

The MATLAB code for solving the one-dimensional SWE using the Lax-Friedrichs scheme is given in listing 7.1. Note that this program makes use of functions to calculate the time step, boundary conditions, flux vector and the Lax-Friedrichs scheme. It is good practice to structure a program in this way as it breaks down the program into manageable chunks and avoids repetition of code when performing the same computation multiple times.

Listing 7.1: MATLAB code for solving the SWE using the Lax-Friedrichs scheme

```
% oned_swe.m by Jon Shiach
%
% This program solves the SWE using the Lax-Friedrichs scheme

% Clear workspaces
clear
clc

% Define global parameters
global gravity
gravity = 9.81;                 % acceleration due to gravity

% Define variables
N = 101;                        % number of nodes
xmin = 0;                       % lower bound of x
xmax = 1;                       % upper bound of x
hL = 1;                         % water height to the left of the dam
hR = 0.5;                       % water height to the right of the dam
t = 0;                          % time variable
tmax = 0.1;                     % max value of t

% Define spatial array (including ghost nodes)
dx = (xmax - xmin) / (N - 1);
x = xmin - dx : dx : xmax + dx;

% Define initial conditions
U = zeros(2, length(x));
U(1, x <= 0.5) = hL;
U(1, x > 0.5) = hR;

```

```
% Time marching loop
while t < tmax

    % Calculate maximum allowable time step
    dt = calculate_dt(U, dx, t, tmax);

    % Calculate boundary conditions
    U = calculate_BC(U);

    % Calculate F vector
    F = calculate_F(U);

    % Calculate one step of the Lax-Friedrichs scheme
    U = Lax_Friedrichs(U, F, dt, dx);

    % Update t
    t = t + dt;

    % Plot current solution
    num_plot = plot(x, U(1, :), 'o-', 'markerfacecolor','blue');
    axis([xmin, xmax, 0, 1.2])
    xlabel('$x$', 'fontsize', 16, 'interpreter', 'latex')
    ylabel('$h$', 'fontsize', 16, 'interpreter', 'latex')
    title(sprintf('$t = %1.3f$', t), 'fontsize', 16, 'interpreter', 'latex')
    shg
    pause(0.001)
end

% Plot numerical solution against exact solution
load dambreak_exact
hold on
exact_plot = plot(xexact, hexact, 'r-', 'linewidth' , 2);
hold off
leg = legend([exact_plot, num_plot], 'exact solution', 'numerical solution')
    ;
set(leg, 'fontsize', 14, 'interpreter', 'latex')

% ------------------------------------------------------------------------
function dt = calculate_dt(U, dx, t, tmax)

% This function calculates the maximum allowable time step
global gravity
dt = 0.9 * dx / max(abs(U(2, :)./U(1, :)) + sqrt(gravity * U(1, :)));
dt = min(dt, tmax - t);

end

% ------------------------------------------------------------------------
function U = calculate_BC(U)

% This function calculates the values of the ghost nodes using transmissive
% boundary conditions
U(:, 1) = U(:, 3);
U(:, end) = U(:, end-2);

end

% ------------------------------------------------------------------------
function F = calculate_F(U)

% This function calculates the flux vector F from U.
```

```
global gravity
F(1, :) = U(2, :);
F(2, :) = U(2, :).^2 ./ U(1, :) + 0.5 * gravity * U(1, :).^2;

end

% -------------------------------------------------------------------------
function Unp1 = Lax_Friedrichs(U, F, dt, dx)

% This function calculates a single step of the Lax-Friedrichs scheme

N = size(U, 2) - 2;
Unp1 = U;
C = 0.5 * dt / dx;
for i = 2 : N + 1
    Unp1(:, i) = 0.5 * (U(:, i+1) + U(:, i-1)) - C * (F(:, i+1) - F(:, i-1))
        ;
end

end
```

7.3.4 The MacCormack scheme

The Lax-Wendroff scheme given in equation (4.15) can not be applied to solve the SWE in its current form since it can only handle flow in one direction (i.e., linear PDEs). Instead, we can write it as an equivalent two-step method which is known as the *MacCormack scheme* (MacCormack, 1969).

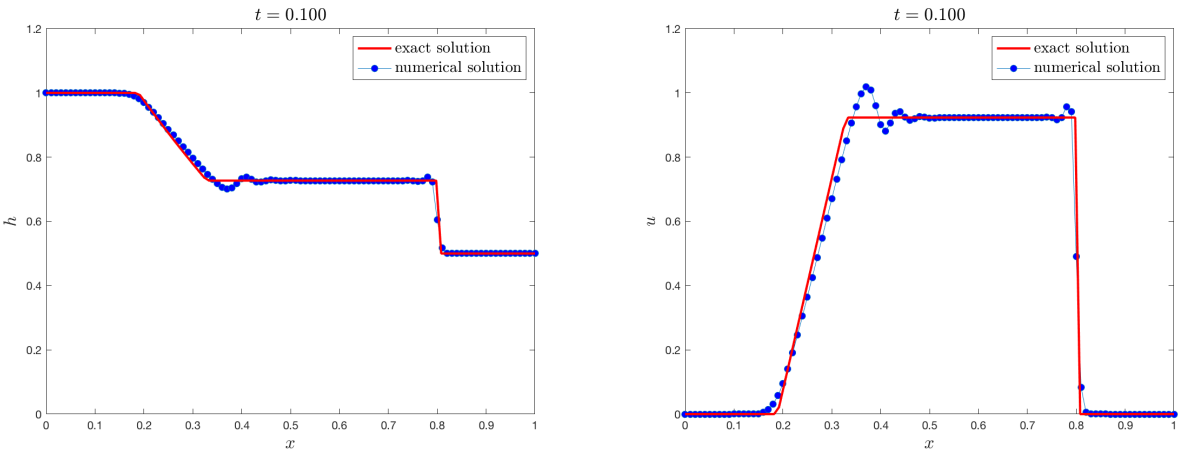
The MacCormack scheme for solving the one-dimensional SWE

$$U_i^{n+\frac{1}{2}} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1}^n - F_i^n\right), \tag{7.11a}$$

$$U_i^{n+1} = \frac{1}{2}(U_i^n + U_i^{n+\frac{1}{2}}) - \frac{\Delta t}{2\Delta x}(F_i^{n+\frac{1}{2}} - F_{i-1}^{n+\frac{1}{2}}) + O(\Delta t^2, \Delta x^2). \tag{7.11b}$$

The predictor step equation (7.11a) updates the solution over a partial time step and the corrector step equation (7.11b) uses the predictor values to update the solution over a full time step. The two-step MacCormack scheme is equivalent to the Lax-Wendroff scheme and has the same order and stability properties. Since the predictor step changes the values of the variables used in the corrector step, the boundary conditions and flux vector need to be calculated using the predictor values prior to the corrector step.

The MacCormack scheme was applied to the one-dimensional dam break problem and the solutions can be seen in figure 7.6. Unlike the Lax-Friedrichs scheme, the MacCormack scheme is able to capture the discontinuity at the bore wave and does not suffer from numerical dissipation. However, there are noticeable oscillations in the solution caused by the abrupt changes in the gradient of the surface. This is a common feature of second-order methods. There are methods that can be applied to the scheme in order to minimise these oscillations but this are outside the scope of this unit (interested readers are advised to search for 'artificial dissipation', 'TVD schemes' and 'MUSCL schemes').



(a) water height

(b) velocity

Figure 7.6: Solution to the one-dimensional dam break at time $t = 0.1$s using the MacCormack scheme

7.4 Two-dimensional shallow water equations

The two-dimensional SWE include a third equation that models flow in the y direction.

The two-dimensional Shallow Water Equations

$$h_t + (hu)_x + (hv)_y = 0,$$
$$(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghz_x,$$
$$(hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghz_y.$$

The variables h, u, g and z have been defined previously with the exception of v which is the horizontal flow velocity in the y direction. These can be written more succinctly in vector form as

$$U_t + F_x + G_y = S,$$

where U is the vector of conserved variables, F and G are flux vectors that model change in the conserved variables in the x and y directions respectively and S are the source terms that model bed topography

$$U = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \qquad F = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix}, \qquad G = \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix}, \qquad S = \begin{pmatrix} 0 \\ -ghz_x \\ -ghz_y \end{pmatrix}.$$

7.4.1 Solving the two-dimensional shallow water equations

Applying the second-order MacCormack scheme with dimensional splitting means we can use the following symmetric operator sequence

$$U_i^{n+1} = \mathcal{L}_X\left(\frac{\Delta t}{2}\right)\mathcal{L}_Y\left(\frac{\Delta t}{2}\right)\mathcal{L}_Y\left(\frac{\Delta t}{2}\right)\mathcal{L}_X\left(\frac{\Delta t}{2}\right)U_i^n.$$

The solution procedure for solving the two-dimensional SWE is summarised in the flow chart in figure 7.7.

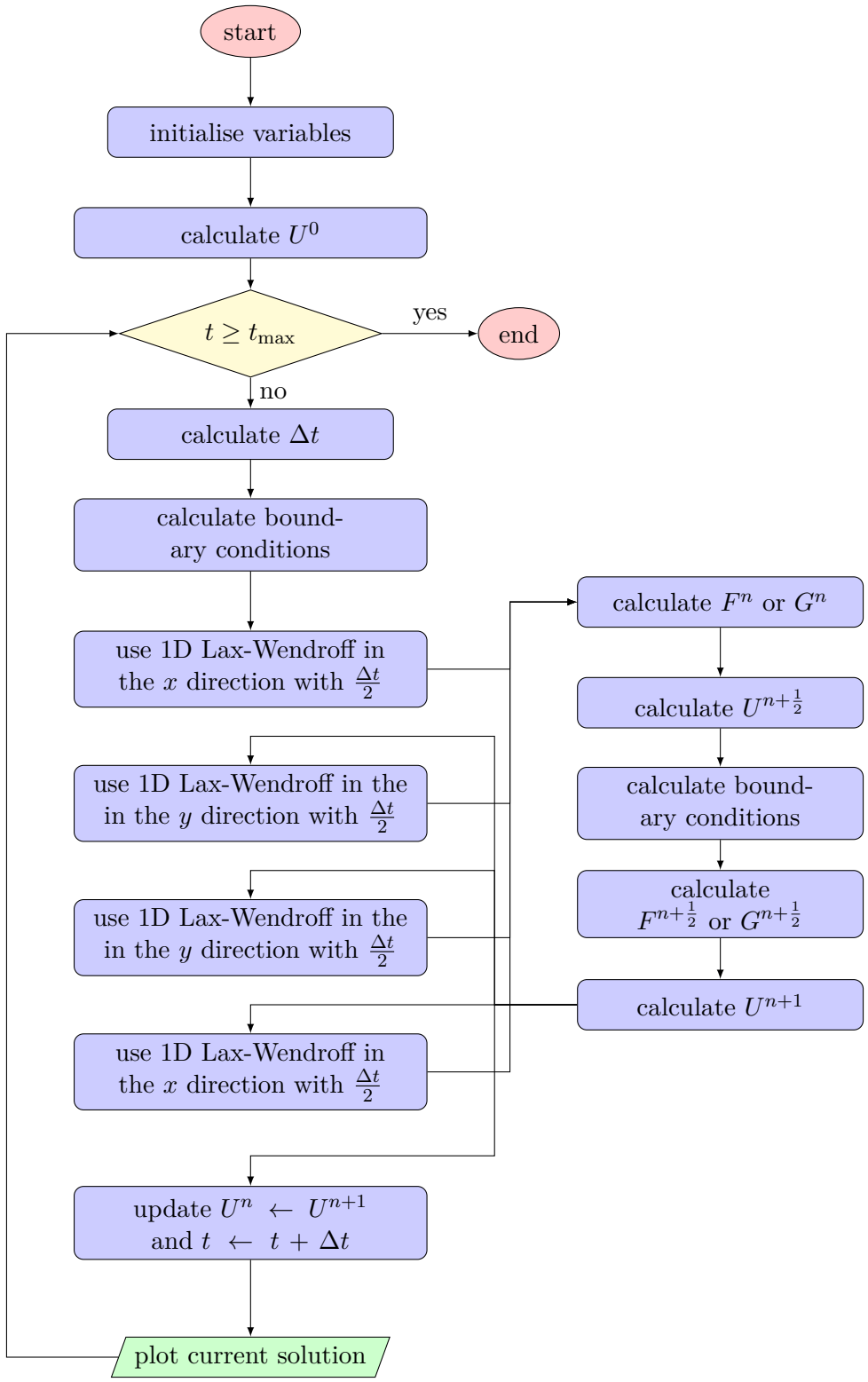


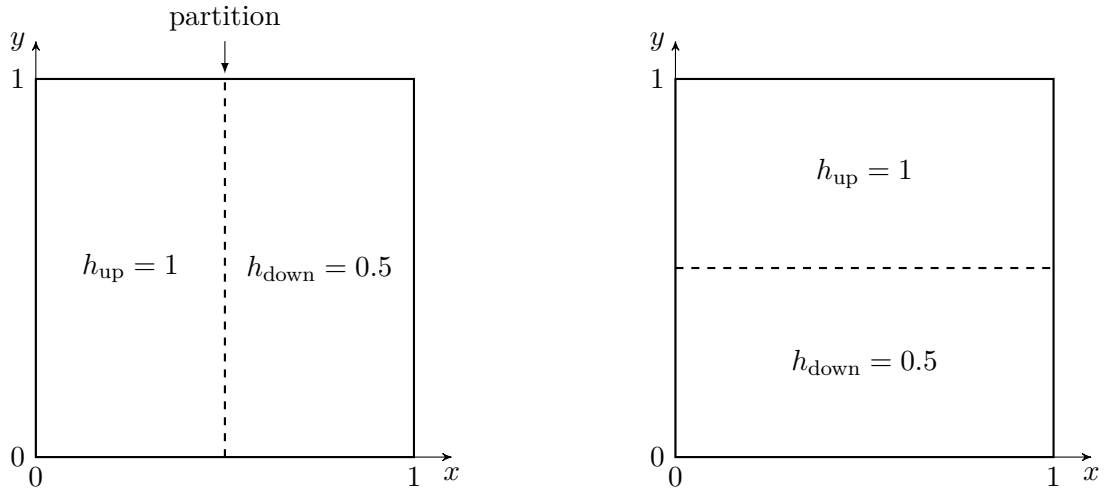
Figure 7.7: Flow chart for solving the two-dimensional SWE using the MacCormack scheme.

7.5 Validating the two-dimensional SWE solver

Adding a another spatial dimension to the SWE provides further complications when writing the computer program to perform the calculations. As always, it is a good idea to validate the program to test whether it produces expected results.

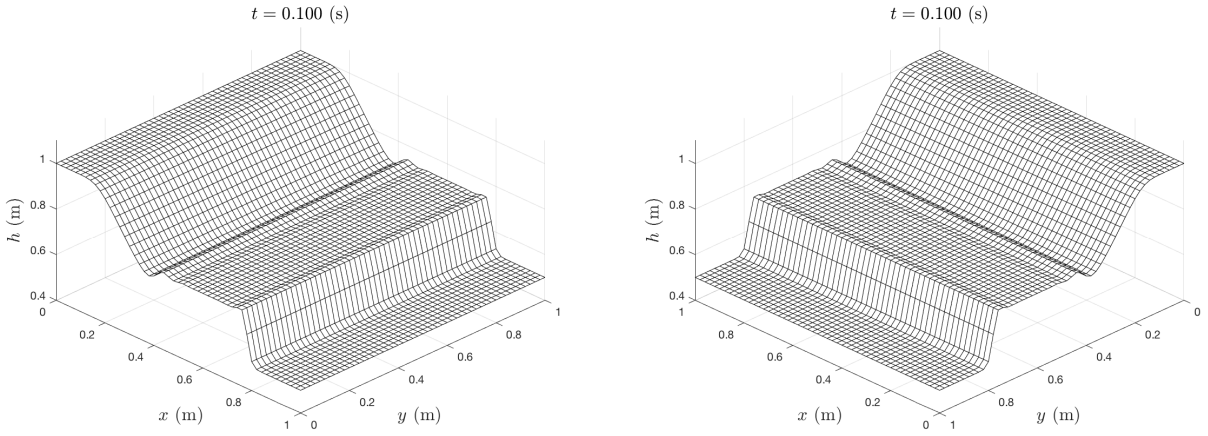
7.5.1 Pseudo one-dimensional dam break

The first test that is being used is a pseudo one-dimensional dam break problem where the one-dimensional dam break test described in section 7.3.2 is applied to a two-dimensional grid (figure 7.8). The domain is discretised using 50 nodes in both the x and y directions. The solution to this problem should be the same as that described for the one-dimensional case along the direction of flow and horizontal in the other direction. The numerical solutions for the pseudo one-dimensional dam break for both the x and y directions can be seen in figure 7.9. Taking a cross-section of the solutions shows that they are comparable to the one-dimensional solutions seen previously (figure 7.10).



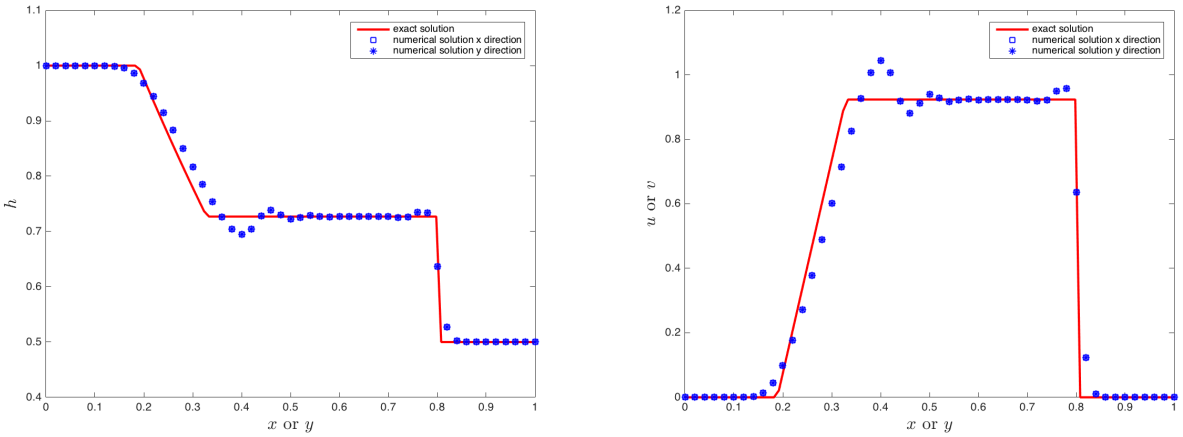
(a) x direction (b) y direction

Figure 7.8: Plan view of the pseudo one-dimensional dam break problem.



(a) x direction (b) y direction

Figure 7.9: Pseudo one-dimensional dam break test.



(a) water height

(b) velocity

Figure 7.10: Comparisons of the solutions for the water height and velocity for the pseudo one-dimensional dam break problem in the x and y directions.

7.5.2 Collapse of a cylindrical column

The pseudo one-dimensional dam break test does not test the numerical solutions for flow in both spatial dimensions at the same time. One test that will show up any flaws in the solver is the modelling of the collapse of a two-dimensional cylindrical column. The problem consists of a two-dimensional domain $0 \leq x, y \leq 1$ with a water depth of $h_{\text{down}} = 0.5$. The initial conditions consist of a perturbation in water depth in the form of a circular region centred at $(0.5, 0.5)$ with radius 0.2 in which the water height is $h_{\text{up}} = 1.0$ (figure 7.11). The velocities in the x and y directions are initialised to zero.

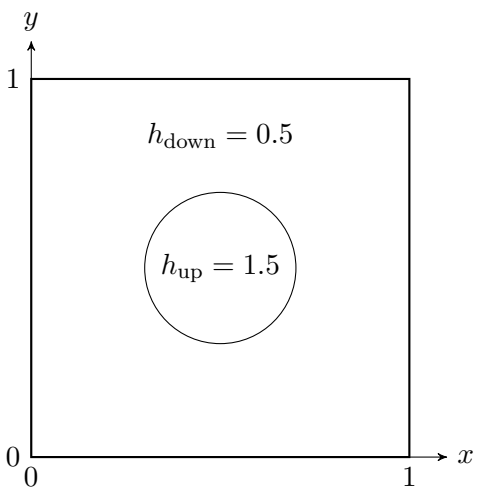


Figure 7.11: Initial conditions for the collapse of a cylindrical column problem.

The values of the initial conditions can be calculated using the following

$$h(x,y) = \begin{cases} 1, & \sqrt{(x-0.5)^2 + (y-0.5)^2} < 0.1, \\ 0.5, & \text{elsewhere}, \end{cases},$$
$$u(x,y) = 0,$$
$$v(x,y) = 0.$$

Solid wall boundary conditions are employed at all four boundaries and the values of the ghost nodes

are calculated using the following

$$
\begin{aligned}
h_{-1,j} &= h_{1,j}, & h_{N_X,j} &= h_{N_X-2,j}, & h_{i,-1} &= h_{i,1}, & h_{i,N_Y} &= h_{i,N_Y-2},\\
u_{-1,j} &= -u_{1,j}, & u_{N_X,j} &= -u_{N_X-2,j}, & u_{i,-1} &= u_{i,1}, & u_{i,N_Y} &= u_{i,N_Y-2},\\
v_{-1,j} &= v_{1,j}, & v_{N_X,j} &= v_{N_X-2,j}, & v_{i,-1} &= -v_{i,1}, & v_{i,N_Y} &= -v_{i,N_Y-2}.
\end{aligned}
$$

The collapse of the cylindrical column causes a circular bore wave to form propagating in the radial direction away from the centre. If there are any problems in resolving the spatial derivatives caused by the dimensional splitting it will be noticeable in the solution. A contour plot of the values of h at $t = 0.075$ is shown in figure 7.12. Here we can see the a circular bore wave is propagating at equal velocities away from the centre of the cylinder thus indicating that the two-dimensional solver can resolve the spatial dimensions equally.

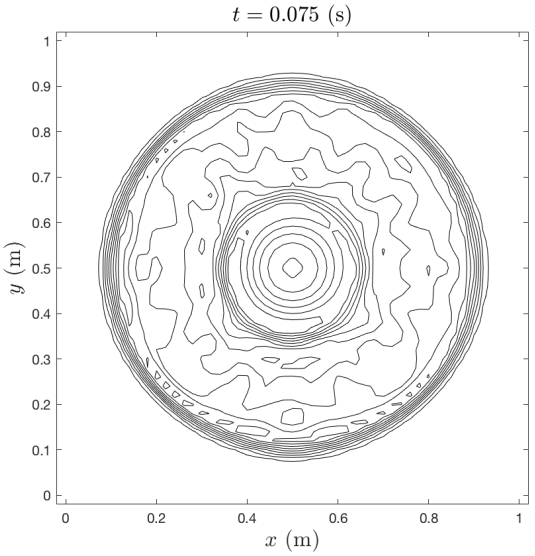


Figure 7.12: Contour plot of the collapse of a cylindrical column showing the bore waves travelling with equal velocities away from the centre of the cylinder.

The solutions at various times in the simulation are shown in figure 7.13. At time $t = 0.05$ the cylindrical column has collapsed in on itself and the outward travelling bore wave has started to form. At time $t = 0.10$ the bore wave has almost reached the boundaries and a depression has formed where the centre of the cylinder was. At time $t = 0.15$ the bore waves have reached the boundaries and are reflected back towards the interior of the domain. At time $t = 0.25$ and $t = 0.3$ the reflected waves have started to interact with each other causing the water surface to break up into lots of individual waves.

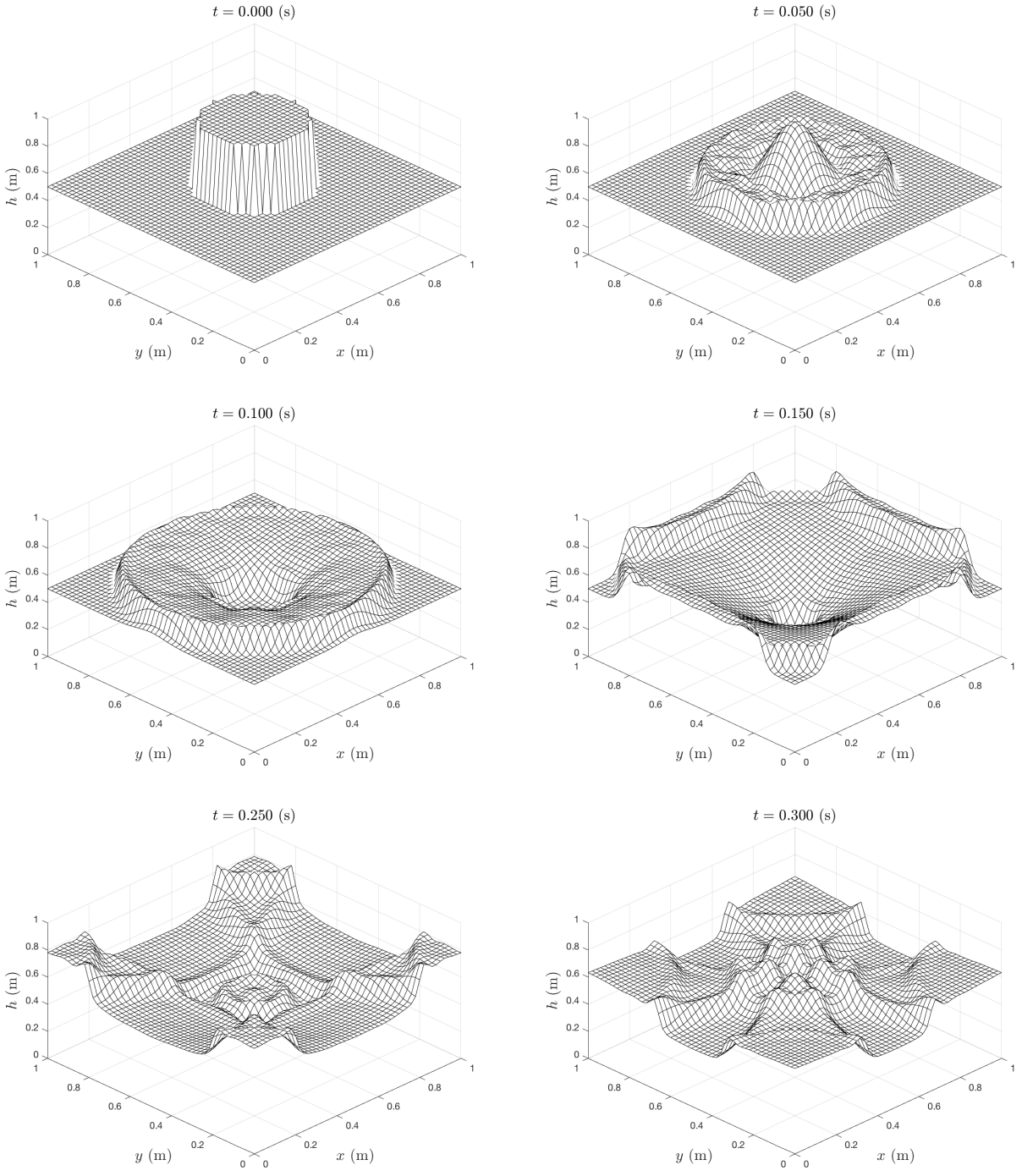


Figure 7.13: Surface plots of the solution to the collapse of a cylindrical column problem at $t = 0.00$, 0.05, 0.10, 0.15, 0.25 and 0.30.

References

Courant, R., Friedrichs, K., and Lewy, H. (1967). "On the partial difference equations of mathematical physics". In: *IBM Journal of Research and Development* 11.2. (English translation from German of the original paper published in 1927), pp. 215 –234. URL: `http://web.stanford.edu/class/cme324/classics/courant-friedrichs-lewy.pdf`.

Crank, J and Nicolson, P. (1947). "A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type." In: *Proc. Camb. Phil. Soc.* 1, pp. 50–67.

Hirsch, C. (1994). *Numerical Computation of Internal and External Flows*. Vol. 1: Fundamentals of Numerical Discretization. John Wiley & Sons. Chap. 8, pp. 283–288.

Hoffman, J.D. (2001). *Numerical Methods for Engineers and Scientists*. 2nd ed. CRC Press.

Lax, P.D. and Richtmyer, R.D. (1956). "Survey of the stability of linear finite-difference equations". In: *Comm. Pure Appl. Math.* 9, pp. 267–293.

Lax, P.D. and Wendroff, B. (1960). "Systems of conservation laws". In: *Comm. Pure Appl. Math.* 13.2, pp. 217–237.

MacCormack, R.W. (1969). "The effect of viscosity in hypervelocity impact cratering". In: pp. 69–354.

Roache, P.J. (1994). "Perspective: a method for uniform reporting of grid refinement studies". In: *J. Fluids Eng.* 116, pp. 405–412.

Yang, Shiming and Gobbert, Matthias K (2009). "The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions". In: *Applied Mathematics Letters* 22.3, pp. 325–331.

Appendix A

Derivation of the Thomas algorithm

Consider a tridiagonal linear system expressed in the form $A\mathbf{x} = \mathbf{d}$

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix} \tag{A.1}$$

This linear system is converted into upper triangular form $U\mathbf{x} = \mathbf{d}'$ where the main diagonal only contains 1's. Consider row 1 of equation (A.1)

$$b_1 x_1 + c_1 x_2 = d_1$$

Divide by b_1 so that the main diagonal is equal to 1

$$x_1 + \frac{c_1}{b_1} x_2 = \frac{d_1}{b_1}.$$

Let $c_1' = \dfrac{c_1}{b_1}$ and $d_1' = \dfrac{d_1}{b_1}$ then

$$x_1 + c_1' x_2 = d_1'. \tag{A.2}$$

Consider row 2 of equation (A.1)

$$a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2 \tag{A.3}$$

Eliminate x_1 using a_2 multiplied by equation (A.2)

$$\begin{aligned} a_2 x_1 + b_2 x_2 + c_2 x_3 - a_2(x_1 + c_1' x_2) &= d_2 - a_2 d_1' \\ (b_2 - a_2 c_1') x_2 + c_2 x_3 &= d_2 - a_2 d_1' \end{aligned}$$

Divide by $b_2 - a_2 c_1'$ so that the main diagonal element is equal to 1

$$x_2 + \frac{c_2}{b_2 - a_2 c_1'} x_3 = \frac{d_2 - a_2 d_1'}{b_2 - a_2 c_1'}.$$

Let $c_2' = \dfrac{c_2}{b_2 - a_2 c_1'}$ and $d_2' = \dfrac{d_2 - a_2 d_1'}{b_2 - a_2 c_1'}$ then

$$x_2 + c_2' x_3 = d_2'. \tag{A.4}$$

Consider row 3 of equation (A.1)

$$a_3x_2 + b_3x_3 + c_3x_4 = d_3$$

Eliminate x_1 using a_3 multiplied by equation (A.4)

$$\begin{aligned} a_3x_2 + b_3x_3 + c_3x_4 - a_3(x_2 + c'_2x_3) &= d_3 - a_3d'_2 \\ (b_3 - a_3c'_2)x_3 + c_3x_4 &= d_3 - a_3d'_2 \end{aligned}$$

Divide by $b_3 - a_3c'_2$ so that the main diagonal element is equal to 1

$$x_3 + \frac{c_3}{b_3 - a_3c'_2}x_4 = \frac{d_3 - a_3d'_2}{b_3 - a_3c'_2}.$$

Let $c'_3 = \dfrac{c_3}{b_3 - a_3c'_2}$ and $d'_3 = \dfrac{d_3 - a_3d'_2}{b_3 - a_3c'_2}$ then

$$x_3 + c'_3x_4 = d'_3. \tag{A.5}$$

Consider an i^{th} row of equation (A.1). Using a similar method shown for rows 2 and 3 above the i^{th} row can be written as

$$x_i + c'_ix_{i+1} = d'_i \tag{A.6}$$

where

$$c'_i = \frac{c_i}{b_i - a_ic'_{i-1}} \tag{A.7}$$

$$d'_i = \frac{d_i - a_id'_{i-1}}{b_i - a_ic'_{i-1}}. \tag{A.8}$$

Finally, consider the n^{th} row of equation (A.1)

$$a_nx_{n-1} + b_nx_n = d_n.$$

Eliminate x_{n-1} using a_n multiplied by equation (A.6

$$\begin{aligned} a_nx_{n-1} + b_nx_n - a_n(x_{n-1} + c'_{n-1}x_n) &= d_n - a_nd'_{n-1} \\ (b_n - a_nc'_{n-1})x_n &= d_n - a_nd'_{n-1} \end{aligned}$$

Divide by $b_n - a_nc'_{n-1}$ so that the main diagonal element is equal to 1

$$x_n = \frac{d_n - a_nd'_{n-1}}{b_n - a_nc'_{n-1}}.$$

The matrix equation can now be expressed in the form $U\mathbf{x} = \mathbf{d}'$

$$\begin{pmatrix} 1 & c'_1 & 0 & 0 & 0 & 0 \\ 0 & 1 & c'_2 & 0 & 0 & 0 \\ 0 & 0 & 1 & c'_3 & 0 & 0 \\ & & & \ddots & \ddots & \\ 0 & 0 & 0 & 0 & 1 & c'_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \\ \vdots \\ d'_{n-1} \\ d'_n \end{pmatrix}. \tag{A.9}$$

Once the system is written in upper triangular form the solution of $\mathbf{x}$ can be found by back-substitution.

Starting with row n of equation (A.9) the solution for x_n is simply d'_n

$$x_n = d'_n. \tag{A.10}$$

Consider a row i of equation (A.9) when $i = n-1, n-2, \ldots, 1$

$$x_i + c'_i x_{i+1} = d'_i,$$

therefore

$$x_i = d'_i - c'_i x_{i+1}. \tag{A.11}$$

The Thomas algorithm is written as follows

- Calculate modified coefficients c' and d' using

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & i = 1, \\ \dfrac{c_i}{b_i - a_i c'_{i-1}} & i = 2, 3, \ldots, n \end{cases}$$

$$d'_i = \begin{cases} \dfrac{d_i}{b_i} & i = 1, \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & i = 2, 3, \ldots, n \end{cases}$$

- Calculate solutions of x_i using

$$x_i = \begin{cases} d'_i - c'_i x_{i+1} & i = n-1, n-2, \ldots, 1, \\ d'_i & i = n. \end{cases}$$