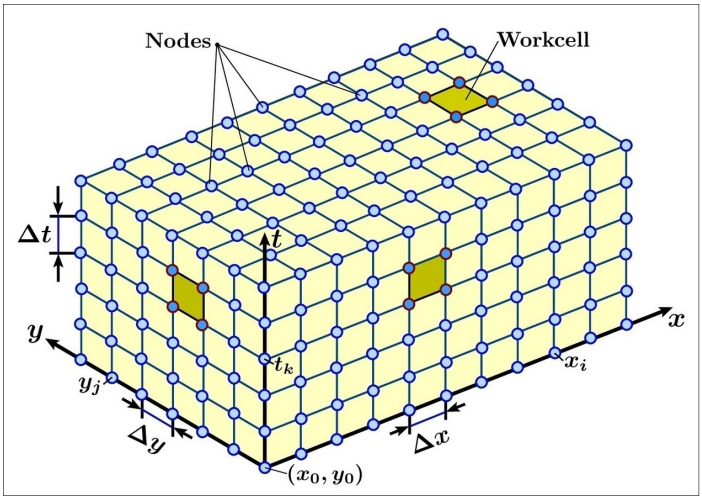
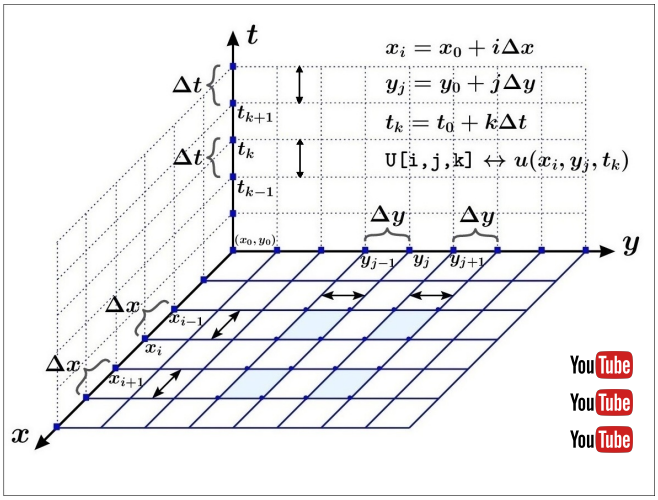
Project: Using the Finite Difference Method for Solving Differential Equations[1]

Daniel Rabayev (danielr@technion.ac.il), Samy Zafrany (samyz@technion.ac.il)

17.11.2026.d





Contents

1 General Introduction **2**
- **1.1** Formal Definition of a Difference Grid 5
- **1.2** Discrete Partial Derivatives 6
- **1.3** Solving Discrete Systems 8

2 The Heat Equation (1D/2D) **12**

3 Additional Examples **22**

4 The Wave Equation in the Plane (2D) **26**
- **4.1** Solution Using the Finite Difference Method 27

5 The Black-Scholes-Merton Equation **34**
- **5.1** Solution Using the Finite Difference Method 35
- **5.2** Boundary Conditions 37

6 Appendix: Accuracy, Stability, and Convergence **40**
- **6.1** Accuracy and Stability 41
- **6.2** Exceptional Discrete Solutions 42

7 Appendix: The Theoretical Basis of the BSM Equation **43**
- **7.1** Stochastic Processes 43
- **7.2** Statistical Background 45
- **7.3** Itô's Lemma (Kiyoshi Itô 1915-2008) 46
- **7.4** The Black-Scholes-Merton Strategy 48

8 Bibliography **51**

[1] Translated from Hebrew by ChatGPT model 5.6 Sol.

1. General Introduction

Unlike the customary approach in branches of classical mathematics, which are based on **continuous** models of space, time, mass, heat, and other physical quantities, for representing and solving scientific and engineering problems, the finite difference method is based on **discrete models** in order to solve a broader range of problems that cannot be solved by classical methods. The simplest discrete model is a **grid** such as those shown in Figures 1, 3, 4. It is a one-dimensional or multidimensional array of discrete values distributed uniformly over the problem domain. The finite difference method disregards the infinite continuum of space and time in which the problem was previously formulated, and instead focuses on a small sample of discrete values from the problem domain.

For example, in the study of ordinary differential equations, the variable x will be restricted to a **one-dimensional grid** of discrete values x_i, $i = 0, 1, 2, \ldots, n$ (see Figure 1). The **grid** (one-dimensional) is formed by partitioning a given interval $[a, b]$ on the x-axis into n equal subintervals of length

$$\Delta x = \frac{b - a}{n}$$

The partition points

$$x_i = a + i\Delta x, \quad i = 0, 1, 2, \ldots, n$$

are called **nodes** or **partition points**, and the natural number n is called the **number of subdivisions** along the x-axis.

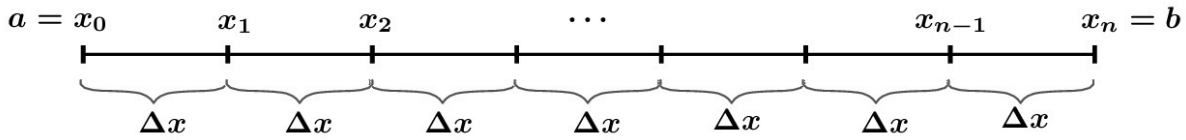


Figure 1: A one-dimensional grid: a uniform partition of a finite interval into n subintervals and $n + 1$ nodes

Observe that partitioning an interval into n subintervals creates $n + 1$ nodes. The first node is denoted by x_0 and the last by x_n. The study of a function $y = f(x)$ will be carried out by studying the values $y_i = f(x_i)$

$$y_i = f(x_i), \quad i = 0, 1, 2, \ldots, n$$

Unlike the grid values x_i, the differences $\Delta y = y_{i+1} - y_i$ are not necessarily uniform (unless the function $y = f(x)$ is linear). A limit-based concept such as the first derivative can be redefined using a quotient of discrete **differences** (hence the name "finite difference method")

$$\frac{dy}{dx} = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = \frac{y_{i+1} - y_i}{\Delta x}$$

Assuming that the partition of the domain of f is sufficiently fine, the expression $\frac{f(x_{i+1})-f(x_i)}{x_{i+1}-x_i}$ may provide a sufficiently accurate approximation of the "true" derivative for practical purposes.

For example, an ordinary differential equation such as

$$\begin{cases} y' = 4y + 3x, \quad 0 < x < 2 \\ y(0) = -2 \end{cases} \tag{1}$$

will be translated into the ***difference equation***

$$\begin{cases} \dfrac{y_{i+1} - y_i}{\Delta x} = 4y_i + 3x_i, \quad i = 0, 1, 2, \ldots, n-1 \\ y_0 = -2 \end{cases}$$

which can easily be solved using a simple ***recurrence formula***

$$\begin{cases} y_{i+1} = y_i + (4y_i + 3x_i)\,\Delta x, \quad i = 0, 1, 2, \ldots, n-1 \\ y_0 = -2 \end{cases}$$

If, for example, we partition the interval $[0, 2]$ into $n = 100$ parts, then $\Delta x = \frac{2}{100} = 0.02$, and therefore

$$\begin{cases} y_{i+1} = y_i + 0.02(4y_i + 3x_i), \quad i = 0, 1, 2, \ldots, 99 \\ y_0 = -2 \end{cases}$$

which in fact enables us to compute all the values y_i recursively!

$$\begin{aligned} y_0 &= -2 \\ y_1 &= y_0 + 0.02(4y_0 + 3x_0) = -2 + 0.02\,[4 \cdot (-2) + 3 \cdot 0.0] = -2.16 \\ y_2 &= y_1 + 0.02(4y_1 + 3x_1) = -2.08 + 0.02\,[4 \cdot (-2.08) + 3 \cdot 0.02] = -2.3316 \\ y_3 &= y_2 + 0.02(4y_2 + 3x_2) = -2.2452 + 0.02\,[4 \cdot (-2.2452) + 3 \cdot 0.04] = -2.515728 \\ &\vdots \quad \vdots \quad \vdots \qquad \vdots \qquad \vdots \\ y_{100} &= y_{99} + 0.02(4y_{99} + 3x_{99}) = -3988.7547771185805 \end{aligned}$$

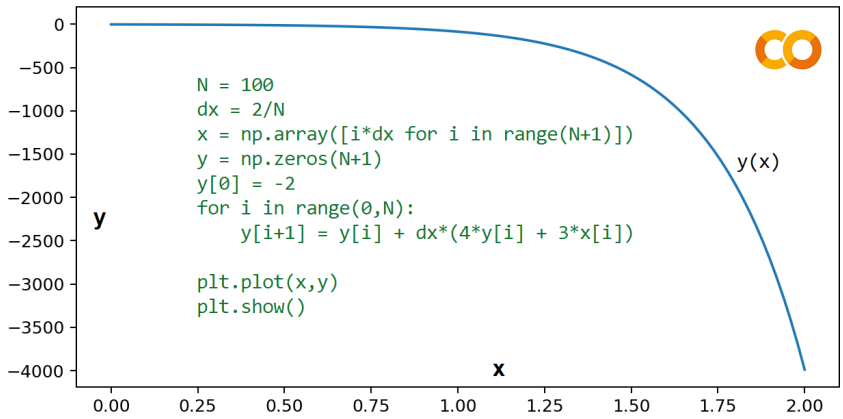


Figure 2: A discrete solution of equation (1) with $n = 100$ subdivisions

This is a list of **100** simple computations that can easily be performed by a computer and displayed

in a graph such as the one shown in Figure 2. The code can be accessed by clicking the link to the Colab notebook .

We have therefore obtained a ***discrete solution*** of our differential equation! The Swiss mathematician Leonhard Euler (1707-1783) was apparently the first mathematician to discover the finite difference method and to show how it can be used to solve ordinary differential equations. Further details are available at this Colab notebook link.

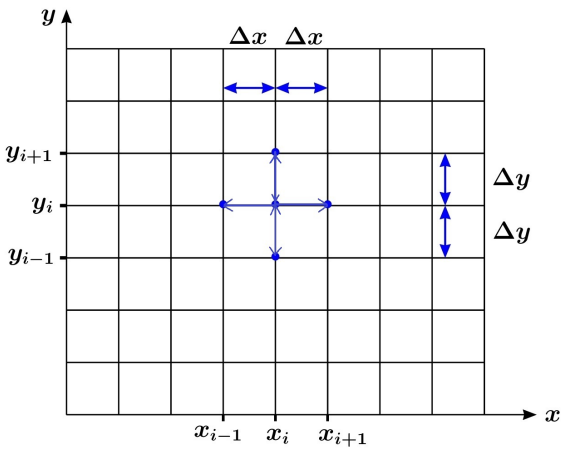


Figure 3: A typical two-dimensional grid for the finite difference method

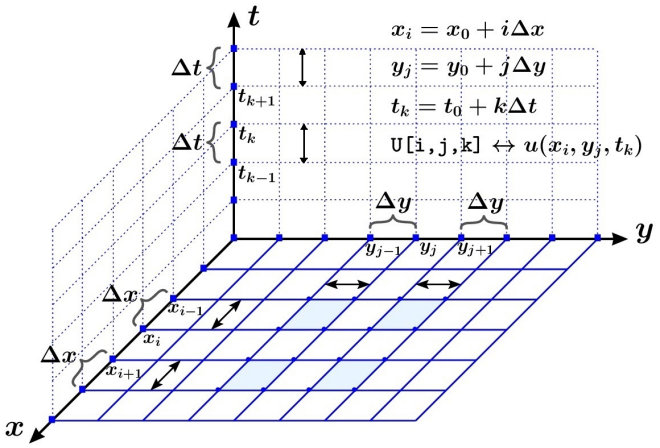


Figure 4: A three-dimensional grid for the finite difference method

Similarly, the study of partial differential equations for a function of two variables will be carried out over a two-dimensional grid such as the one shown in Figure 3, and the study of partial differential equations in three variables will be carried out over a three-dimensional grid such as the one shown in Figure 4.

The principal reason for developing the finite difference method was to enable mathematical problems to be solved by means of numerical algorithms and modern computing technologies that emerged at the end of the previous century and continue to develop at an accelerating pace in the age of artificial intelligence and ever-growing computing power. Numerical algorithms are based on results from fields such as combinatorics, discrete mathematics, and computability theory, which have also developed greatly over the past centuries. The aim is to obtain solutions that are as close as possible to the[2] "true" solutions. We relinquish absolute accuracy but gain the ability to solve a broader range of problems that cannot be solved by classical methods.

In general, to study functions of n variables we need an n-dimensional difference grid. For example, to find numerical solutions of the two-dimensional heat equation $u_t = \alpha(u_{xx} + u_{yy})$, we must study

[2] The quotation marks are used because many physicists today believe that the physical structure of space, time, mass, energy, and so forth is discrete, and that the concepts of an infinite continuum in classical mathematics are in fact theoretical abstractions that do not reflect physical reality! Link to an article on the subject.

functions of three variables $\boldsymbol{u(x,y,t)}$, and therefore need a three-dimensional difference grid shown schematically in Figure 4.

1.1 Formal Definition of a Difference Grid

For every pair of integers $\boldsymbol{m}$, $\boldsymbol{n}$, define the ***range set*** $\boldsymbol{[m:n]}$ by

$$\boldsymbol{[m:n] = \{\, i \in \mathbb{Z} :\ m \le i < n \,\}}$$

where $\mathbb{Z}$ is the set of integers. The notation $\boldsymbol{[m:n]}$ is borrowed from the world of programming, and is intended to ease the transition to the programming section. We emphasize that the range set $\boldsymbol{[m:n]}$ does not include the terminal index $\boldsymbol{n}$. That is, it is the sequence of integers beginning with $\boldsymbol{m}$ and ending with $\boldsymbol{n-1}$. For convenience, we shall focus only on two-dimensional and three-dimensional grids.

We begin by describing a three-dimensional grid. Let $\boldsymbol{N_x}$, $\boldsymbol{N_y}$, $\boldsymbol{N_t}$ be the ***numbers of subdivisions*** in the $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{t}$ directions, respectively. Formally, a three-dimensional difference grid is defined by a three-dimensional array $\mathbf{U}$

$$\mathbf{U}\colon \Big([0:N_x+1]\times[0:N_y+1]\times[0:N_t+1]\Big) \longrightarrow \mathbb{R}$$

The Cartesian product $\boldsymbol{[0:N_x+1]\times[0:N_y+1]\times[0:N_t+1]}$ is called the ***grid domain***. It is the finite set of all ***keys*** $\boldsymbol{(i,j,k)}$ for the nodes $\boldsymbol{(x_i,y_j,t_k)}$ as shown in Figure 4. Each ***key*** $\boldsymbol{(i,j,k)}$ in the grid domain corresponds to a ***node*** $\boldsymbol{(x_i,y_j,t_k)}$ in the discrete domain of the function $\boldsymbol{u(x,y,t)}$, and therefore the value of the grid $\mathbf{U}$ at the key $\boldsymbol{(i,j,k)}$ is defined by

$$\mathbf{U}[i,j,k] = u(x_i,y_j,t_k)$$

where $\boldsymbol{u(x,y,t)}$ is the target function of the problem to be solved (which is usually unknown to us except on the boundary of the domain). It is important to distinguish between a ***key*** (a vector of indices) $\boldsymbol{(i,j,k)}$ and its corresponding ***node*** $\boldsymbol{(x_i,y_j,t_k)}$! The key $\boldsymbol{(i,j,k)}$ is a triple of **integers** that specifies the exact position of the node $\boldsymbol{(x_i,y_j,t_k)}$ in the grid. In other words, the key $\boldsymbol{(i,j,k)}$ is the ***address*** of the node $\boldsymbol{(x_i,y_j,t_k)}$ in the grid. As is customary in mathematical programming environments, we extend the use of range expressions as follows. The expression $\boldsymbol{[m:]}$ will denote the range of all indices from $\boldsymbol{m}$ through the final index on the corresponding axis. The expression $\boldsymbol{[:n]}$ will denote the range $\boldsymbol{[0:n]}$, and the expression $[:]$ will denote the full range from $\mathbf{0}$ through the final index on the corresponding axis.

When working with program code, these notations will enable us to initialize the grid over large sets of indices. For example, in the Python or Matlab programming languages, the expression $\mathbf{U}[:,:,0] = 3$

initializes the values of $\mathbf{U}$ at all keys $(i, j, 0)$ to 3, and the expression $\mathbf{U}[3, :, 0] = 7$ initializes the third row of slice 0 to the value 7. That is, $\mathbf{U}[3, j, 0] = 7$ for every index j on the second axis $[0 : N_y + 1]$.

2.1 Discrete Partial Derivatives

In basic PDE course that covers the FDM method, we learn how a partial derivative can be approximated using three different formulas

$$u_x(x_i, y_j, t_k) = \frac{u(x_{i+1}, y_j, t_k) - u(x_i, y_j, t_k)}{\Delta x} \tag{2}$$

$$u_x(x_i, y_j, t_k) = \frac{u(x_i, y_j, t_k) - u(x_{i-1}, y_j, t_k)}{\Delta x} \tag{3}$$

$$u_x(x_i, y_j, t_k) = \frac{u(x_{i+1}, y_j, t_k) - u(x_{i-1}, y_j, t_k)}{2\Delta x} \tag{4}$$

The first formula is called the ***forward-difference formula***, the second the ***backward difference***, and the third the ***central difference***. The more compact structure of the difference grid enables us to write these formulas somewhat more concisely

$$\mathbf{U}_x[i, j, k] = \frac{\mathbf{U}[i + 1, j, k] - \mathbf{U}[i, j, k]}{\Delta x} \tag{5}$$

$$\mathbf{U}_x[i, j, k] = \frac{\mathbf{U}[i, j, k] - \mathbf{U}[i - 1, j, k]}{\Delta x} \tag{6}$$

$$\mathbf{U}_x[i, j, k] = \frac{\mathbf{U}[i + 1, j, k] - \mathbf{U}[i - 1, j, k]}{2\Delta x} \tag{7}$$

The expression $\mathbf{U}_x$ on the left-hand side is in fact a representation of the concept of partial differentiation of a difference grid, which can be defined formally by any of the expressions on the right-hand side. An operation of this kind is sometimes called ***discrete differentiation***, and is studied and investigated in a branch of discrete mathematics called Discrete Calculus. Software environments such as MATLAB or NumPy provide libraries for differentiation operations of this kind.

In the same manner, similar formulas can be obtained for the partial derivatives $\mathbf{U}_y$, $\mathbf{U}_t$, which we shall not write here, but will use later.

During the course (see also the course lecture notes), we shall prove the existence of similar formulas for second-order partial derivatives as well.

$$u_{xx}(x_i, y_j, t_k) = \frac{u(x_{i+1}, y_j, t_k) - 2u(x_i, y_j, t_k) + u(x_{i-1}, y_j, t_k)}{\Delta x^2} \tag{8}$$

$$u_{xx}(x_i, y_j, t_k) = \frac{u(x_i, y_j, t_k) - 2u(x_{i-1}, y_j, t_k) + u(x_{i-2}, y_j, t_k)}{\Delta x^2} \tag{9}$$

$$u_{xx}(x_i, y_j, t_k) = \frac{u(x_{i+2}, y_j, t_k) - 2u(x_{i+1}, y_j, t_k) + u(x_i, y_j, t_k)}{\Delta x^2} \tag{10}$$

which can be expressed similarly in terms of the difference grid

$$\mathrm{U}_{xx}[i, j, k] = \frac{\mathrm{U}[i+1, j, k] - 2\mathrm{U}[i, j, k] + \mathrm{U}[i-1, j, k]}{\Delta x^2} \tag{11}$$

$$\mathrm{U}_{xx}[i, j, k] = \frac{\mathrm{U}[i, j, k] - 2\mathrm{U}[i-1, j, k] + \mathrm{U}[i-2, j, k]}{\Delta x^2} \tag{12}$$

$$\mathrm{U}_{xx}[i, j, k] = \frac{\mathrm{U}[i+2, j, k] - 2\mathrm{U}[i+1, j, k] + \mathrm{U}[i, j, k]}{\Delta x^2} \tag{13}$$

We shall call formula (11) the ***central-difference formula***, formula (12) the ***backward-difference formula***, and formula (13) the ***forward-difference formula***. Similar formulas can be written for U_{yy}, U_{tt}, U_{xy}, and so forth. For example, we present a symbolic derivation of the forward formula for the mixed derivative u_{xy}

$$\begin{aligned} u_{xy}(x, y, t) &= \frac{\partial}{\partial y}\left[u_x(x, y, t)\right] = \frac{\partial}{\partial y}\left[\frac{u(x+\Delta x, y, t) - u(x, y, t)}{\Delta x}\right] \\ &= \frac{\frac{\partial}{\partial y}u(x+\Delta x, y, t) - \frac{\partial}{\partial y}u(x, y, t)}{\Delta x} \\ &= \frac{\frac{u(x+\Delta x, y+\Delta y, t) - u(x+\Delta x, y, t)}{\Delta y} - \frac{u(x, y+\Delta y, t) - u(x, y, t)}{\Delta y}}{\Delta x} \\ &= \frac{u(x+\Delta x, y+\Delta y, t) - u(x+\Delta x, y, t) - u(x, y+\Delta y, t) + u(x, y, t)}{\Delta x \Delta y} \end{aligned}$$

We can write a similar formula for the points on our grid

$$u_{xy}(x_i, y_j, t_k) = \frac{u(x_{i+1}, y_{j+1}, t_k) - u(x_{i+1}, y_j, t_k) - u(x_i, y_{j+1}, t_k) + u(x_i, y_j, t_k)}{\Delta x \Delta y} \tag{14}$$

which can be expressed in terms of the difference grid as follows

$$\mathrm{U}_{xy}[i, j, k] = \frac{\mathrm{U}[i+1, j+1, k] - \mathrm{U}[i+1, j, k] - \mathrm{U}[i, j+1, k] + \mathrm{U}[i, j, k]}{\Delta x \Delta y} \tag{15}$$

For precision, we emphasize that formulas (8)-(15), are not exact equalities in the usual sense of the partial derivatives as defined in classical calculus courses! They are merely approximations of the classical derivatives. However, since we have relinquished absolute accuracy from the outset, we are

free to redefine them for the purpose of developing numerical methods whose aim is only to obtain a good approximation.

In all the examples above, we focused on a function of three variables. Similar formulas can, of course, be written for functions with a different number of variables. We assume that the idea is clear and therefore shall not state this explicitly in such cases.

To conclude this section, we emphasize that in the context of functions, the difference grid refers only to the domain of the function, and not to its values. Although the function values are also discrete values, they are not part of the grid.

3.1 Solving Discrete Systems

In a PDE course (that covers FDM) we meet examples of solving PDEs using the finite difference method. We summarize the solution process in the following four steps. For convenience, assume that our grid is two-dimensional.

a. Construct a difference grid $\mathbf{U}$ over the domain of the given problem by choosing the number of subdivisions in each dimension of the problem domain.

b. Convert the differential equation into a difference equation on the grid $\mathbf{U}$ by replacing the partial derivatives with suitable difference formulas.

c. Convert the boundary conditions of the problem into equivalent conditions on the difference grid, and add the resulting conditions to the system of equations obtained in the preceding step.

d. Verify that the resulting system of equations is solvable. In most cases, our PDE is linear, and therefore we obtain a system of linear equations in the unknowns $\mathbf{U}[i, j]$ (assuming that our grid is two-dimensional). Verify that the number of equations is greater than or equal to the number of unknowns, and that the system is consistent and has a unique solution.

Example 1: For clarification, we present a simple example illustrating the principles of the method. We compute manually a discrete solution of size $\mathbf{4\times4}$ for a first-order linear PDE

$$\begin{cases} u_x + u_y = 0, & 0 < x < 3,\ 0 < y < 3 \\ u(x,0) = x, & 0 \le x \le 3 \\ u(0,y) = 2y, & 0 \le y \le 3 \\ u(x,3) = 6 - 2x, & 0 \le x \le 3 \\ u(3,y) = 3 - y, & 0 \le y \le 3 \end{cases} \tag{16}$$

We describe the process according to the steps above:

a. Grid Construction: Choose a uniform number of subdivisions $N = 3$ along both axes of the domain $[0,3]\times[0,3]$. Our grid spacings are $\Delta x = \Delta y = \frac{3}{N} = 1$, and therefore in this case the partition points are integers $x_i = i$, $y_j = j$. Observe that the number of partition points on each axis is $N+1 = 4$! Therefore our grid $\mathbf{U}$ is a two-dimensional grid of size $(N+1)\times(N+1) = 4\times4$, containing **16** nodes. We shall use forward-difference formulas for both partial derivatives

$$\mathbf{U}_x[i,j] = \frac{\mathbf{U}[i+1,j]-\mathbf{U}[i,j]}{\Delta x} \tag{17}$$

$$\mathbf{U}_y[i,j] = \frac{\mathbf{U}[i,j+1]-\mathbf{U}[i,j]}{\Delta y} \tag{18}$$

b. Conversion of the PDE into a Difference Equation: Replace the partial derivatives u_x, u_y with the difference formulas

$$U_x[i,j] + U_y[i,j] = \frac{\mathbf{U}[i+1,j]-\mathbf{U}[i,j]}{\Delta x} + \frac{\mathbf{U}[i,j+1]-\mathbf{U}[i,j]}{\Delta y} = 0$$

Since we chose to work with a uniform grid in which $\Delta x = \Delta y = 1$, we obtain

$$\mathbf{U}[i+1,j] - \mathbf{U}[i,j] + \mathbf{U}[i,j+1] - \mathbf{U}[i,j] = 0$$

and after simplification

$$\begin{aligned} &\mathbf{U}[i+1,j] - 2\mathbf{U}[i,j] + \mathbf{U}[i,j+1] = 0 \\ &i = 1,2 \quad j = 1,2 \end{aligned} \tag{19}$$

It is very important to note the restriction of the index values i, j to ***interior nodes***. In the usual theoretical setting, the PDE is restricted to interior points of the domain because the partial derivatives are not always guaranteed to exist on the boundary of the domain. In the discrete setting, this may be reflected by the fact that substituting boundary nodes into the difference equations is not always possible. In the present case, for example, when $i = 3$, the expression $i+1$ **falls outside the grid domain**, and therefore $\mathbf{U}[i+1,j]$ is undefined, so the forward derivative $U_x[i,j]$ is undefined!

For problems of this kind, it is advisable to draw a ***stencil*** for the equation, which characterizes the nodes that can be substituted into it. Figure 5 shows that equation (19) does not apply to the red boundary nodes, because placing the ***stencil center*** (i,j) at these points would cause the stencil to extend outside the grid.

Therefore, in many cases encountered in this booklet, the conversion of a PDE into a difference formula will apply only to interior nodes. In the present case, for example, the stencil shows that the difference equations can be applied at boundary nodes for which $i = 0$ or $j = 0$, but

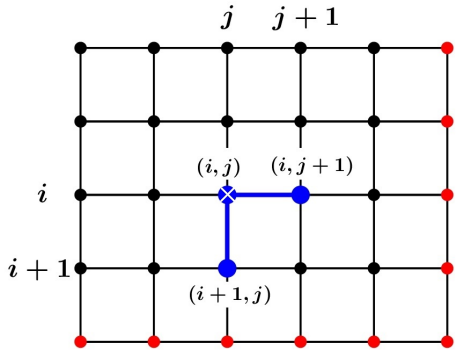


Figure 5: Stencil for the difference equation (19)

boundary conditions have already been specified at those nodes[3], and therefore we did not include $i, j = 0$ in the domain of equation (19). To solve the system, write equality (19) in the more convenient form

$$\mathbf{U}[i, j+1] = 2\mathbf{U}[i, j] - \mathbf{U}[i+1, j], \qquad i = 1, 2 \quad j = 1, 2 \tag{20}$$

This gives us 4 different linear equations in the unknowns $\mathbf{U}[i, j]$ (the interior nodes). Recall that our unknowns are all the grid values $\mathbf{U}[i, j]$ for $i = 0, 1, 2, 3$, $j = 0, 1, 2, 3$. We have 16 nodes and therefore 16 unknowns to solve for! 4 equations will not suffice.

c. Boundary Conditions: The 12 missing equations are to be obtained from the boundary conditions.

$$\begin{cases} \mathbf{U}[i, 0] = i, & i = 0, 1, 2, 3 \\ \mathbf{U}[i, 3] = 6 - 2i, & i = 0, 1, 2, 3 \\ \mathbf{U}[0, j] = 2j, & j = 0, 1, 2, 3 \\ \mathbf{U}[3, j] = 3 - j, & j = 0, 1, 2, 3 \end{cases}$$

Since we chose the small number of subdivisions $N = 3$, we can write a complete discrete specification of the boundary conditions

$\mathbf{U}[0, 0] = 0$	$\mathbf{U}[0, 1] = 2$	$\mathbf{U}[0, 2] = 4$	$\mathbf{U}[0, 3] = 6$
$\mathbf{U}[1, 0] = 1$			$\mathbf{U}[1, 3] = 4$
$\mathbf{U}[2, 0] = 2$			$\mathbf{U}[2, 3] = 2$
$\mathbf{U}[3, 0] = 3$	$\mathbf{U}[3, 1] = 2$	$\mathbf{U}[3, 2] = 1$	$\mathbf{U}[3, 3] = 0$

Observe that the boundary conditions overlap at the corners of the domain, and therefore there are 12 rather than 16. Thus, the boundary conditions actually provide us with solutions for 12 of the system's 16 unknowns! These are all the ***boundary nodes***. This is, of course, an exceptional case because our number of subdivisions is small. If, for example, $N = 10$, then we

[3] In fact, situations arise, including in industrial applications, in which the boundary conditions themselves conflict (at the corners of the boundary, for example). In such cases, the problem is resolved by allowing the later conditions to take precedence over the earlier ones. At the programming stage, this means that the later data "overwrite" the earlier data.

have a system of **121** equations, **40** boundary nodes, and **81** interior nodes.

It remains, therefore, to solve for **4** unknowns corresponding to the interior nodes of the grid:

$\mathbf{U[0,0]=0}$	$\mathbf{U[0,1]=2}$	$\mathbf{U[0,2]=4}$	$\mathbf{U[0,3]=6}$
$\mathbf{U[1,0]=1}$	$\mathbf{U[1,1]=?}$	$\mathbf{U[1,2]=?}$	$\mathbf{U[1,3]=4}$
$\mathbf{U[2,0]=2}$	$\mathbf{U[2,1]=?}$	$\mathbf{U[2,2]=?}$	$\mathbf{U[2,3]=2}$
$\mathbf{U[3,0]=3}$	$\mathbf{U[3,1]=2}$	$\mathbf{U[3,2]=1}$	$\mathbf{U[3,3]=0}$

In general, in every two-dimensional grid with a uniform number of subdivisions $\boldsymbol{N}$, there are $\boldsymbol{(N-1)^2}$ interior nodes and $\boldsymbol{4N}$ boundary nodes.

d. Solving the System of Equations: When the number of subdivisions is large, for example $\boldsymbol{N=999}$, we obtain a system of one million equations in one million unknowns! The only way to solve systems of this kind is by means of numerical algorithms, some of which we shall study in the following chapters. In the present example, we chose a small number of subdivisions $\boldsymbol{N=3}$ and a very simple PDE in order to give the student a concrete impression of how the method works. Such a partition clearly does not yield a realistic level of accuracy, and in the following chapters we shall encounter larger numbers of subdivisions in order to attain a practical level of accuracy.

We solve for the interior unknowns using formula (20) above

$$\begin{aligned}
\mathbf{U[1,1]} &= \mathbf{2U[1,0]-U[2,0]=2\cdot 1-2=0}\\
\mathbf{U[2,1]} &= \mathbf{2U[2,0]-U[3,0]=2\cdot 2-3=1}\\
\mathbf{U[1,2]} &= \mathbf{2U[1,1]-U[2,1]=2\cdot 0-1=-1}\\
\mathbf{U[2,2]} &= \mathbf{2U[2,1]-U[3,1]=2\cdot 1-2=0}
\end{aligned}$$

This completes the task of finding the discrete solution of size $\mathbf{4\times 4}$

$\mathbf{U[0,0]=0}$	$\mathbf{U[0,1]=2}$	$\mathbf{U[0,2]=4}$	$\mathbf{U[0,3]=6}$
$\mathbf{U[1,0]=1}$	$\mathbf{U[1,1]=0}$	$\mathbf{U[1,2]=-1}$	$\mathbf{U[1,3]=4}$
$\mathbf{U[2,0]=2}$	$\mathbf{U[2,1]=1}$	$\mathbf{U[2,2]=0}$	$\mathbf{U[2,3]=2}$
$\mathbf{U[3,0]=3}$	$\mathbf{U[3,1]=2}$	$\mathbf{U[3,2]=1}$	$\mathbf{U[3,3]=0}$

Further examples of this kind can be found in the exercises later in the booklet 1, 4, 10, 11. Practice with small grids is very helpful for understanding the method, and it is therefore recommended to solve each of them.

Summary: In general, for a uniform two-dimensional grid with $\boldsymbol{N}$ subdivisions, the finite difference method leads to a system of $\boldsymbol{(N+1)\times(N+1)}$ linear equations in $\boldsymbol{(N+1)\times(N+1)}$ unknowns. The unknowns of the system are the grid values $\mathbf{U}[\boldsymbol{i},\boldsymbol{j}]$, $\boldsymbol{i,j=0,1,2,\ldots,N}$. The equations are obtained from the difference equation for the PDE and from the boundary conditions. The boundary

conditions of the PDE provide immediate solutions for some or all of the grid's boundary nodes. Note that for a PDE of order greater than $\mathbf{1}$, the boundary conditions may also affect interior nodes! In the common cases, this is a sparse system of equations in which most off-diagonal coefficients vanish, and it can therefore be solved by simple algebraic manipulations. One of the common methods we shall encounter later in the booklet is a solution by recurrence formulas.

In the following chapters we shall also encounter grids of dimension $\mathbf{3}$, in which the subdivisions are not always uniform, and whose systems require greater creativity and programming skills to solve.

2. The Heat Equation (1D/2D)

The heat equation is a partial differential equation that provides a mathematical model for the phenomenon of heat propagation in the physics of materials. The phenomenon can be studied on a one-dimensional metal rod (1D-Rod), a two-dimensional metal plate (2D-plate), or three-dimensional solids with volume (such as a box, a sphere, or a cylinder). The solution methods presented here apply to all these cases, but for simplicity we shall demonstrate the finite difference method using an example of a one-dimensional rod, and a second example involving a two-dimensional rectangular plate. Later we shall encounter additional examples from the fields of waves and finance.

Before proceeding to an example involving a large, practical grid, we shall practice once more with a simple $\mathbf{4\times4}$ grid that can be solved manually. Review Example 1 at the end of the preceding chapter and carry out the four steps described there once again in preparation for the remainder of this chapter.

Exercise 1: (1D Heat Equation with Growth/Decay)

Find a discrete solution of size $\mathbf{4\times4}$, similar to the one presented in Example 1.

$$\begin{cases} u_t = 4u_{xx} - u, & 0 < x < 3,\ 0 < t < 3 \\ u(0,t) = 0, & 0 \le t \le 3 \\ u(3,t) = 0, & 0 \le t \le 3 \\ u(x,0) = x(3-x), & 0 \le x \le 3 \end{cases}$$

Guidance: The general form of the classical heat equation is $u_t = \alpha u_{xx}$. If we add the term βu to the right-hand side, we obtain the general form of the heat equation with growth/decay coefficient β: $u_t = \alpha u_{xx} + \beta u$. Review Example 1 at the end of the preceding chapter and construct a discrete model of size 4×4 for this equation. Partition the x-axis and the time axis t into $N = 3$ parts, and construct a 4×4 table of grid values. It is recommended to use a central-difference derivative formula for u_{xx}, and a forward-difference derivative formula for u_t.

In the next example, we construct a general discrete model of size $(N+1)\times(N_t+1)$ for the classical one-dimensional heat equation $u_t = \alpha u_{xx}$, where N is the number of subdivisions of the rod interval $[0, L]$, and N_t is the number of subdivisions of the time interval $[0, T]$. An

appropriate choice of the parameters N, N_t will provide us with a grid that enables us to compute the temperature of the rod at the point x and time t to a practical level of accuracy.

Example 2: Using finite differences, we solve a typical one-dimensional (1D) heat problem on a rod of length $L = 1$, over a time interval of length $T = 0.2$ seconds.

$$\begin{cases} u_t = u_{xx}, & 0 < x < 1, \;\; 0 < t < 0.2 \\ u(0,t) = e^{-\pi^2 t}, & 0 \le t \le 0.2 \\ u(1,t) = -e^{-\pi^2 t}, & 0 \le t \le 0.2 \\ u(x,0) = \cos(\pi x), & 0 \le x \le 1 \end{cases}$$

The exact solution of the problem is $u(x,t) = e^{-\pi^2 t}\cos(\pi x)$. This can, of course, be verified manually by substitution. We can use this solution to validate the approximate numerical solution obtained by the finite difference method (we do this in the Colab notebook provided below).

Choose a number of subdivisions N for the interval $[0,1]$, and a number of subdivisions N_t for the time interval $[0, 0.2]$. Define our basic parameters

$$\Delta x = \frac{1}{N}, \quad \Delta t = \frac{0.2}{N_t}, \quad x_i = i\Delta x, \quad t_k = k\Delta t, \quad \mathrm{U}[i,k] = u(x_i, t_k)$$

We shall use the following discrete differentiation formulas

$$\mathrm{U}_{xx}[i,k] = \frac{\mathrm{U}[i+1,k] - 2\mathrm{U}[i,k] + \mathrm{U}[i-1,k]}{\Delta x^2} \tag{21}$$

$$\mathrm{U}_t[i,k] = \frac{\mathrm{U}[i,k+1] - \mathrm{U}[i,k]}{\Delta t} \tag{22}$$

We chose a central-difference formula for the second derivative U_{xx} and a forward-difference formula for the derivative U_t. We translate our heat problem into a discrete problem on the difference grid U.

$$\begin{cases} \mathrm{U}_t[i,k] = \mathrm{U}_{xx}[i,k], & 0 < i < N, \;\; 0 < k < N_t \\ \mathrm{U}[0,k] = e^{-\pi^2 t_k}, & 0 \le k \le N_t \\ \mathrm{U}[N,k] = -e^{-\pi^2 t_k}, & 0 \le k \le N_t \\ \mathrm{U}[i,0] = \cos(\pi x_i), & 0 \le i \le N \end{cases}$$

In general, the finite difference method leads to a system of $(N+1)\times(N_t+1)$ linear equations in $(N+1)\times(N_t+1)$ unknowns. The unknowns of the system are the grid values $\mathrm{U}[i,k]$, $i = 0,1,2,\ldots,N$, $k = 0,1,2,\ldots,N_t$. In most cases, this is a sparse system of equations in which most off-diagonal coefficients vanish, and therefore in many cases it can be solved using simple manipulations such as recurrence formulas, as we shall see in this example and later. Formula (22)

implies

$$\mathbf{U}[i,k+1] = \mathbf{U}[i,k] + \mathbf{U}_t[i,k]\Delta t$$

From the equality $\mathbf{U}_t[i,k] = \mathbf{U}_{xx}[i,k]$ we obtain

$$\mathbf{U}[i,k+1] = \mathbf{U}[i,k] + \mathbf{U}_{xx}[i,k]\Delta t$$

This is in fact a simple recurrence formula[4] that enables us to compute a solution

$$\boxed{\mathbf{U}[i,k+1] = \mathbf{U}[i,k] + \mathbf{U}_{xx}[i,k]\Delta t}$$

The recurrence formula enables us to compute the heat distribution along the rod at time $t = t_{k+1}$ from the heat distribution at time $t = t_k$. In other words, if the values $\mathbf{U}[i,k]$, $0 \le i \le N$, are known, then the recurrence formula enables us to determine the unknowns $\mathbf{U}[i,k+1]$ (for $0 < i < N$). Since the unknowns $\mathbf{U}[i,0]$ are provided by the boundary conditions, we can compute all the grid values $\mathbf{U}[i,k]$! This is a typical procedure for many problems of this kind, and later we shall see several more examples of it.

But the recurrence formula is not the end of the story! To ensure convergence of the discrete solution to the true solution as $N, N_t \to \infty$, we must verify that the grid spacings Δx, Δt satisfy appropriate stability conditions. For the one-dimensional heat equation, the condition is $\Delta t \le \frac{\Delta x^2}{2\alpha}$, where α is the thermal conductivity coefficient (a physical constant) appearing in the equation $u_t = \alpha u_{xx}$. In the present case $\alpha = 1$, and therefore our stability condition is

$$\Delta t \le \frac{\Delta x^2}{2}$$

or, equivalently,

$$\gamma = \frac{\Delta t}{\Delta x^2} \le 0.5$$

It is customary to use the parameter γ in problems of this kind. Later we shall encounter similar stability conditions in two-dimensional heat and wave problems. A general discussion of accuracy and stability is presented in Chapter 6. A specific explanation of the stability condition for the heat equation can be found in reference [8]. Complete Python code implementing this solution is provided in the Colab notebook linked to Figure 6, which displays its main section.

In the next exercise, we return to the heat equation with growth/decay encountered in Exercise 1, this time constructing a grid of general size $N \times N_t$ that can be used for a practical purpose.

[4] Verify that the expression $k+1$ is not hidden in the right-hand side! Otherwise, this is not a recurrence formula! Formula (21) should also be inspected to verify this.

```
# Boundary conditions
for i in range(0, N+1):
    U[i,0] = cos(pi * i*dx)           # u(x,0) = cos (\pi x)

for k in range(0, Nt+1):
    U[0,k] = exp(-pi**2 * k * dt)     # u(0,t) = e^{-\pi^2 t}
    U[N,k] = -exp(-pi**2 * k * dt)    # u(1,t) = -e^{-\pi^2 t}

def Solve(U):
  for k in range(0,Nt):
    for i in range(1,N):
      Uxx = (U[i+1,k] - 2*U[i,k] + U[i-1,k]) / (dx**2)
      U[i,k+1] = U[i,k] + Uxx * dt
```

Figure 6: Python code for the one-dimensional heat problem in Example 2

Exercise 2: (1D Heat Equation with Growth/Decay)

Solve the following problem using the finite difference method.

$$\begin{cases} u_t = 4u_{xx} - u, & 0 < x < 2,\ 0 < t < 1 \\ u(0,t) = 0, & 0 \le t \le 1 \\ u(2,t) = 0, & 0 \le t \le 1 \\ u(x,0) = x(2-x)\sin(3\pi x), & 0 \le x \le 2 \end{cases}$$

Guidance: The general form of the heat equation with growth/decay coefficient β is: $u_t = \alpha u_{xx} + \beta u$. Find a recurrence formula for numbers of subdivisions N, N_t, as in the preceding example. We encountered this equation in Exercise 1, so you may use the recurrence formula obtained there. It is recommended to use a copy of the Colab notebook from the preceding example to construct suitable Python code for validating the recurrence formula you obtained.

Example 3: Let D be a two-dimensional rectangular plate made of a homogeneous material of uniform density. For every point (x, y) on the plate D, $u(x, y, t)$ is the temperature at the point (x, y) at time t. The course treats the one-dimensional case (please consult the lecture notes concerning the working assumptions in this case). The two-dimensional case is similar: the function u is twice continuously differentiable with respect to the variables x, y at interior points of the domain D, continuously differentiable with respect to the time variable t, and satisfies the following partial differential equation

$$u_t = \alpha(u_{xx} + u_{yy}), \qquad (x, y) \in D, \quad 0 < t < \infty$$

where α is a positive real thermal conductivity constant of the material from which the plate is made. Figure 7 shows the thermal state of a rectangular plate at three different times.

As in the one-dimensional case, there are three types of boundary conditions: Dirichlet, Neumann, and a mixture of the two. In our examples, we restrict ourselves to Dirichlet conditions in which the boundary points are held at a fixed temperature (as a function of the point). In the example in

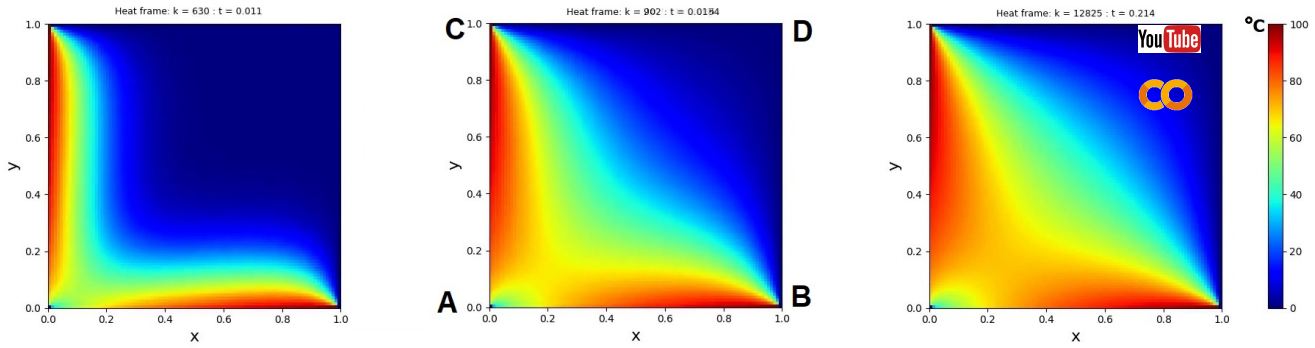


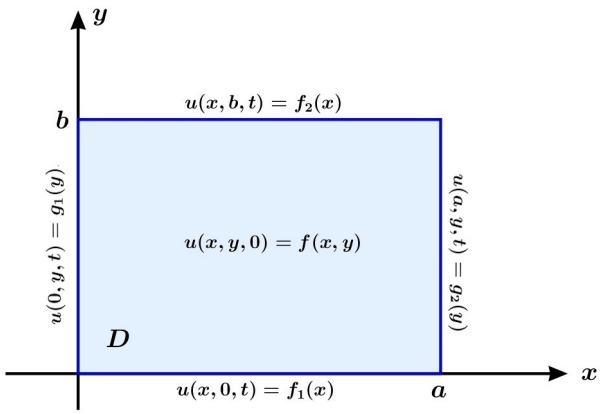
Figure 7: Heat maps on a rectangular plate at three times (the color map on the right ranges from $0°$ to $100°$ Celsius)

Figure 7, the boundary conditions are given by

$$\begin{cases} u(x,0,t) = f_1(x) = 100\sqrt[4]{x} & 0<x<1,\ 0<y<1,\ 0<t<\infty \\ u(x,1,t) = f_2(x) = 0 & 0<x<1,\ 0<y<1,\ 0<t<\infty \\ u(0,y,t) = g_1(y) = 100\sqrt[8]{y} & 0<x<1,\ 0<y<1,\ 0<t<\infty \\ u(0,y,t) = g_2(y) = 0 & 0<x<1,\ 0<y<1,\ 0<t<\infty \\ u(x,y,0) = f(x,y) = 0 & 0<x<1,\ 0<y<1 \end{cases} \tag{23}$$

Note that in the first four conditions, the temperatures of the boundary points are held fixed throughout the entire time interval t! Temperature changes occur only at interior points of the plate. The fifth condition specifies the initial temperature ($t=0$) of the interior points of the plate. Note that at time $t=0$ there is an incompatibility between the temperatures on the boundary of the plate (which are positive on the edges AB, AC) and the temperatures at the interior points (which are zero). This phenomenon is sometimes called a "heat bath". One can observe how the heat propagates from the two edges AB, AC toward the corner D.

Figure 8 presents a general description of the heat equation on a rectangular plate $D=[0,a]\times[0,b]$. The accompanying boundary conditions are Dirichlet conditions. As stated above, this is the only type that we shall consider in the present project. Examples of Neumann and mixed boundary conditions can be found in books such as [18], [17], [20].



$$\begin{cases} u_t = \alpha(u_{xx}+u_{yy}), & 0<x<a,\ 0<y<b,\ 0<t<\infty \\ u(x,y,0)=f(x,y), & 0<x<a,\ 0<y<b \\ u(x,0,t)=f_1(x), & 0\le x\le a,\ 0\le t<\infty \\ u(x,b,t)=f_2(x), & 0\le x\le a,\ 0\le t<\infty \\ u(0,y,t)=g_1(y), & 0\le y\le b,\ 0\le t<\infty \\ u(a,y,t)=g_2(y), & 0\le y\le b,\ 0\le t<\infty \end{cases}$$

Figure 8: The heat equation with Dirichlet boundary conditions on a two-dimensional plate

There are conditions on the functions f, f_1, f_2, g_1, g_2 that are required to guarantee the existence and uniqueness of a solution to our heat equation, which we shall not specify here. They can be found in the sources cited above.

To treat a PDE in three variables, we must work on a ***three-dimensional difference grid***. See Figure 9.

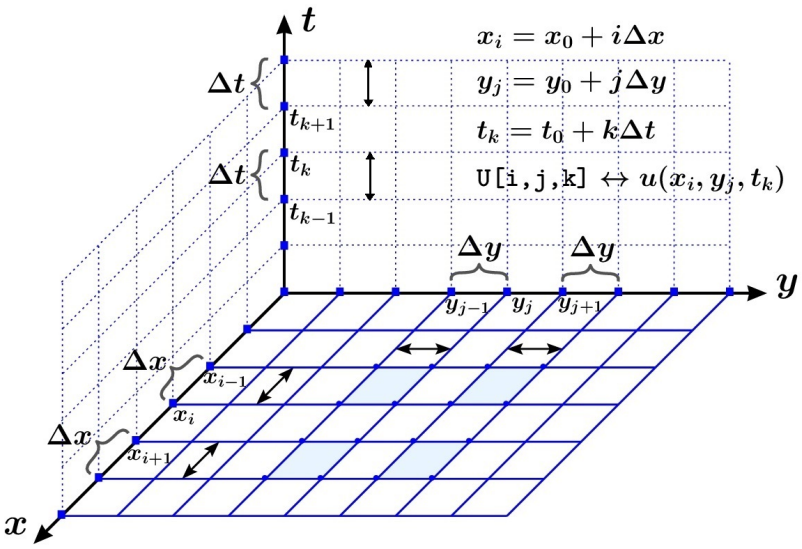


Figure 9: A three-dimensional grid for the heat equation in the variables x, y, t.

Example 4: To illustrate the method, we use it to solve the two-dimensional heat equation on the square $D = [0,4]\times[0,4]$ (that is, $a = b = 4$), over the time interval $[0, T]$.

$$\begin{cases} u_t = 0.1(u_{xx} + u_{yy}), & 0 < x < 4,\ 0 < y < 4,\ 0 < t < T \\ u(x,y,0) = 0, & 0 < x < 4,\ 0 < y < 4 \\ u(x,0,t) = 100\sqrt[3]{x/4}, & 0 \le x \le 4,\ 0 \le t \le T \\ u(x,4,t) = 100(0.7 + 0.3\sin\frac{5\pi x}{4}), & 0 \le x \le 4,\ 0 \le t \le T \\ u(0,y,t) = 100\sqrt{y/4}, & 0 \le y \le 4,\ 0 \le t \le T \\ u(4,y,t) = 0, & 0 \le y \le 4,\ 0 \le t \le T \end{cases} \tag{24}$$

The boundary conditions are described in Figure 10. The temperature along the right edge is zero, as is clearly indicated by the dark shade of blue near it. The figure also includes two links to a YouTube video and a Colab notebook for this example (at the lower right)[5].

Solving this problem by the standard method is possible but rather complicated because of the need to compute the Fourier series of $\sqrt{x}$ or $\sqrt[3]{y}$, which is quite a tedious task. By contrast, functions of this kind present no obstacle to the finite difference method.

[5] There is an incompatibility of the boundary conditions at the points $(0,4,t)$, $(4,4,t)$, which may present a problem for the existence of an exact classical solution, but does not present a problem for the existence of an approximate solution by the finite difference method! From the algorithmic perspective, the order in which the boundary conditions are entered into the computation determines the final value of u at these points. See Section 6.2.

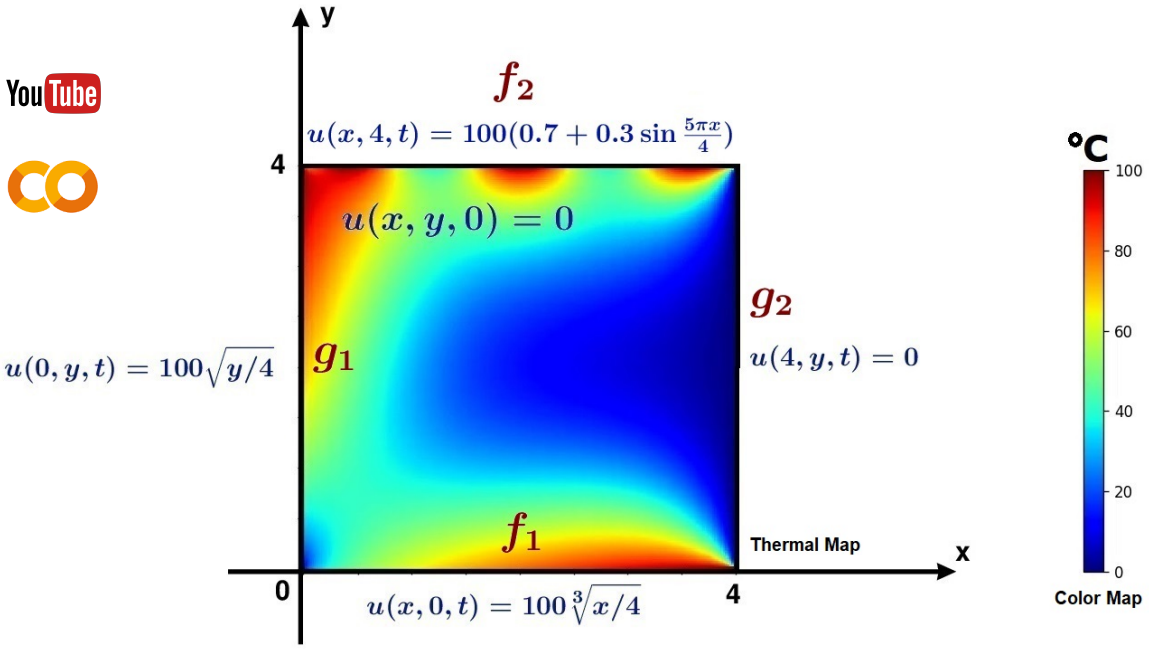


Figure 10: Boundary conditions for problem (24). Heat map at time $t = 5.910$.

For simplicity, we use the same number of subdivisions N for the interval $[0, 4]$ along both the x-axis and the y-axis. Therefore $\Delta x = \Delta y = \frac{4}{N}$. The number of subdivisions of the time axis $[0, T]$ will generally be determined by the number of images required to create a simulation video[6]. For the time being, denote it by N_t, and therefore $\Delta t = \frac{T}{N_t}$. In addition, the existence of a discrete solution with the desired level of approximation also depends on an appropriate choice of the numbers of subdivisions N, N_t, which are discussed more extensively in Chapter 6, and in the articles [16], [19], whose results we shall use in our examples but not discuss.

Assuming that the chosen subdivision sizes satisfy the required conditions, we may assume that the difference grid $\mathbf{U}$ converges to an approximate solution u in some sense: $\mathbf{U}[i,j,k] \sim u(i\Delta x, j\Delta y, k\Delta t)$. The approximation may be understood as pointwise convergence of $\mathbf{U}[i,j,k]$ to the value $u(i\Delta x, j\Delta y, k\Delta t)$, or as uniform convergence, and sometimes as convergence in another sense.

To this end, the equation and our boundary conditions must be translated into equivalent conditions on the difference grid $\mathbf{U}$. We begin with the differentiation formulas encountered in Section 1.2

$$\mathbf{U}_t[i,j,k] = \frac{\mathbf{U}[i,j,k+1] - \mathbf{U}[i,j,k]}{\Delta t}$$

$$\mathbf{U}_{xx}[i,j,k] = \frac{\mathbf{U}[i+1,j,k] - 2\mathbf{U}[i,j,k] + \mathbf{U}[i-1,j,k]}{\Delta x^2}$$

$$\mathbf{U}_{yy}[i,j,k] = \frac{\mathbf{U}[i,j+1,k] - 2\mathbf{U}[i,j,k] + \mathbf{U}[i,j-1,k]}{\Delta y^2}$$

[6] Sometimes, however, a large number of subdivisions is required to obtain good accuracy, in which case the simulation video will use only a sample of the images.

The choice among the three alternatives (backward, central, or forward formula) is critical for solving the problem. We chose a forward formula for $\mathbf{U}_t$ and a central formula for $\mathbf{U}_{xx}$, $\mathbf{U}_{yy}$, for a reason that will become clear later. Represent the heat equation in terms of the discrete function $\mathbf{U}$ (and recall once again that $\boldsymbol{\Delta y = \Delta x}$)

$$\mathbf{U}_t = \boldsymbol{\alpha}\left(\mathbf{U}_{xx} + \mathbf{U}_{yy}\right) \tag{25}$$

Express $\mathbf{U}_{xx} + \mathbf{U}_{yy}$ by means of the difference formula

$$\begin{aligned}
&\mathbf{U}_{xx}[i,j,k] + \mathbf{U}_{yy}[i,j,k] = \\
&= \frac{\mathbf{U}[i+1,j,k] - 2\mathbf{U}[i,j,k] + \mathbf{U}[i-1,j,k]}{\boldsymbol{\Delta x^2}} + \frac{\mathbf{U}[i,j+1,k] - 2\mathbf{U}[i,j,k] + \mathbf{U}[i,j-1,k]}{\boldsymbol{\Delta x^2}} \\
&= \frac{\mathbf{1}}{\boldsymbol{\Delta x^2}}\Big(\mathbf{U}[i-1,j,k] + \mathbf{U}[i+1,j,k] + \mathbf{U}[i,j-1,k] + \mathbf{U}[i,j+1,k] - 4\mathbf{U}[i,j,k]\Big)
\end{aligned}$$

The expression inside the parentheses in the last line

$$\boxed{\boldsymbol{\Delta}\mathbf{U} = \mathbf{U}[i-1,j,k] + \mathbf{U}[i+1,j,k] + \mathbf{U}[i,j-1,k] + \mathbf{U}[i,j+1,k] - 4\mathbf{U}[i,j,k]} \tag{26}$$

is called the discrete Laplace operator and in other contexts it is also called a stencil. This is a geometric pattern that is commonly represented in a diagram such as the one shown in Figure 11, which depicts the graphical pattern of the operator. We have obtained the following formula

$$\boxed{\mathbf{U}_{xx}[i,j,k] + \mathbf{U}_{yy}[i,j,k] = \frac{\boldsymbol{\Delta}\mathbf{U}}{\boldsymbol{\Delta x^2}}} \tag{27}$$

This formula is also useful for the Laplace equation and the wave equation, and is therefore worth noting. Substitute this into the discrete heat equation $\mathbf{U}_t = \boldsymbol{\alpha}(\mathbf{U}_{xx} + \mathbf{U}_{yy})$ (25) to obtain

$$\mathbf{U}_t[i,j,k] = \frac{\mathbf{U}[i,j,k+1] - \mathbf{U}[i,j,k]}{\boldsymbol{\Delta t}} = \frac{\boldsymbol{\alpha}\boldsymbol{\Delta}\mathbf{U}}{\boldsymbol{\Delta x^2}}$$

After a simple rearrangement, we obtain a ***recurrence formula*** for $\mathbf{U}$ in the time dimension $\boldsymbol{k}$

$$\mathbf{U}[i,j,k+1] = \mathbf{U}[i,j,k] + \frac{\boldsymbol{\alpha}\boldsymbol{\Delta t}}{\boldsymbol{\Delta x^2}}\,\boldsymbol{\Delta}\mathbf{U}$$

It is customary to denote

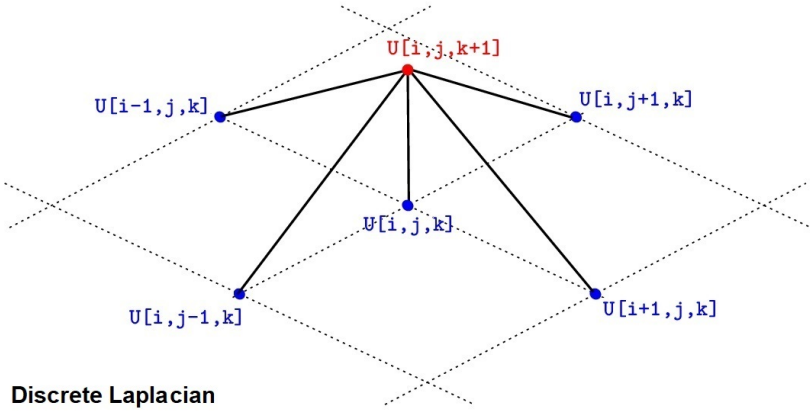
$$\gamma = \frac{\boldsymbol{\alpha}\boldsymbol{\Delta t}}{\boldsymbol{\Delta x^2}}$$

Our recurrence formula therefore becomes shorter and more readable

$$\mathbf{U}[i,j,k+1] = \mathbf{U}[i,j,k] + \gamma\boldsymbol{\Delta}\mathbf{U} \tag{28}$$

The meaning of this formula is that, given a heat map at time $\boldsymbol{t = t_k}$, we can compute the heat

map at the next time $\boldsymbol{t = t_{k+1}}$. Since the heat map at time $\boldsymbol{t_0 = 0}$ is given as part of the boundary conditions, we can compute all subsequent heat maps recursively!



Discrete Laplacian

```
DU = U[i+1,j,k] + U[i-1,j,k] + U[i,j+1,k] + U[i,j-1,k] - 4*U[i,j,k]
U[i,j,k+1] = U[i,j,k] + gamma * DU
```

Figure 11: Stencil for the recurrence formula used to solve the heat equation (Python code)

However, greater precision is needed: formula (28) is valid only for ***interior keys*** $\boldsymbol{(i,j)}$: $\boldsymbol{0 < i,j < N}$, because the expressions $\boldsymbol{i \pm 1}$, $\boldsymbol{j \pm 1}$ occur in $\boldsymbol{\Delta \mathbf{U}}$. Therefore, to obtain a unique solution, we need constraints derived from the boundary conditions. Figure 11 shows the dependence of a point on heat map $\boldsymbol{k+1}$ on points from the preceding heat map $\boldsymbol{k}$. This figure shows that only interior points of map $\boldsymbol{k+1}$ can be obtained from map $\boldsymbol{k}$. The boundary points are handled by the boundary conditions.

We now proceed to translate the boundary conditions into equivalent conditions on the difference grid $\mathbf{U}$. For example, the condition $\boldsymbol{u(x,y,0) = 0}$ must be written as follows

$$\mathbf{U}[i,j,0] = 0, \quad 0 < i < N, \;\; 0 < j < N$$

Once again, note that the boundary indices $\boldsymbol{i,j = 0}$ or $\boldsymbol{i,j = N}$ are not included in this constraint. In software environments such as MATLAB or NumPy this condition is written using a simple, short line of code

```
U[1:N,1:N,0] = 0
```

Similarly, the boundary conditions $\boldsymbol{u(x,0,t) = f_1(x)}$, $\boldsymbol{u(x,L,t) = f_2(x)}$ are translated into the grid conditions

$$\begin{aligned} \mathbf{U}[i,0,k] &= f_1(x_i), \quad 0 \le i \le N, \;\; 0 \le k \le N_t \\ \mathbf{U}[i,N,k] &= f_2(x_i), \quad 0 \le i \le N, \;\; 0 \le k \le N_t \end{aligned}$$

The boundary conditions $\boldsymbol{u(0,y,t) = g_1(x)}$, $\boldsymbol{u(L,y,t) = g_2(x)}$ are translated into the grid conditions

$$\begin{aligned} \mathbf{U}[0,j,k] &= g_1(y_j), \quad 0 \le j \le N, \;\; 0 \le k \le N_t \\ \mathbf{U}[N,j,k] &= g_2(y_j), \quad 0 \le j \le N, \;\; 0 \le k \le N_t \end{aligned}$$

where $x_i = i\Delta x$, $y_j = j\Delta y$.

After initializing the grid $\mathbf{U}$ with the boundary conditions, apply recurrence formula (28) for every k in the range $k = 1, 2, \ldots, N_t - 1$. Figure 12 shows how this is carried out using 5 lines of Python code (from the NumPy package). In writing the code, we relied on the excellent blog [3] by Nervadof on this subject, and we thank him for it. The complete Python code for the discrete solution of our problem can be viewed in the Colab notebook heat.ipynb linked to Figure 12.

```
def Solve(U):
  for k in range(0, num_frames-1):
    for i in range(1, N-1):
      for j in range(1, N-1):
        DU = U[i+1,j,k] + U[i-1,j,k] + U[i,j+1,k] + U[i,j-1,k] - 4*U[i,j,k]
        U[i,j,k+1] = U[i,j,k] + gamma * DU
```

Figure 12: Heat-equation simulation using code from NumPy

The expression `DU` appearing at the bottom of Figure 11 and Figure 12 is the code for our discrete Laplacian $\Delta\mathbf{U}$, which appears in several similar contexts as a discrete version of the continuous Laplacian $u_{xx} + u_{yy}$. We shall encounter it again in the wave equation.

Initialization of the three-dimensional array $\mathbf{U}$ requires familiarity and proficiency in the use of multidimensional arrays in mathematical programming environments such as MATLAB or NumPy. Since this is a mathematics course, we have created a more convenient interface that enables the boundary conditions to be defined without prior programming experience in these environments. Students with such experience may, of course, download our code package and examine the details more deeply. Figure 13 shows how the boundary conditions in our example are initialized.

```
f  = lambda x,y: 0
f1 = lambda x: 100 * (x/L) ** (1/3)
f2 = lambda x: 100 * (0.7 + 0.3*sin(5*pi*x/L))
g1 = lambda y: 100 * (y/L)**(1/2)
g2 = lambda y: 0
boundary_conditions(U, f, f1, f2, g1, g2, dx, dy)
```

Figure 13: Boundary conditions for the heat equation in code using NumPy

The first line defines the function $f(x, y)$ used to initialize $u(x, y, 0)$. The following four lines define the functions `f1`, `f2`, `g1`, `g2` for the boundary conditions. The initialization itself is performed in the final line using the function `boundary_conditions`, which handles all the intricate technical details required to initialize the array $\mathbf{U}$. The command `lambda` is a Python command for defining an anonymous function in a single line. More complicated functions can be constructed

using the standard mechanism of `def`.

3. Additional Examples

Having learned the method through two examples involving the heat equation, we now attempt to apply it to further examples of partial differential equations. We begin with a simple example of a first-order linear equation and leave two additional exercises for independent study.

Example 5: Find a discrete solution $u(x,y)$ in the form of a recurrence formula on a suitable difference grid for the following partial differential equation (first order) on the unit square $[0,1]\times[0,1]$.

$$\begin{cases} xu_x + (1+y)u_y = 0, & 0 < x < 1,\ 0 < y < 1 \\ u(x,0) = 2x, & 0 \le x \le 1 \\ u(0,y) = 0, & 0 \le y \le 1 \\ u(x,1) = x, & 0 \le x \le 1 \\ u(1,y) = \frac{2}{1+y}, & 0 \le y \le 1 \end{cases} \tag{29}$$

Solution: This is a first-order linear PDE that can also be solved using the standard methods taught at the beginning of the course. The closed-form solution is $u(x,y) = \frac{2x}{1+y}$, and it can therefore be used to validate the numerical solution obtained by the finite difference method. Let N be the common number of subdivisions along both axes of our square domain $D = [0,1]\times[0,1]$. Denote $\Delta x = \Delta y = \frac{1}{N}$. We shall try using a backward formula for the derivative with respect to y, and a forward formula for the derivative with respect to y:

$$\mathrm{U}_x[i,j] = \frac{\mathrm{U}[i,j] - \mathrm{U}[i-1,j]}{\Delta x}$$

$$\mathrm{U}_y[i,j] = \frac{\mathrm{U}[i,j+1] - \mathrm{U}[i,j]}{\Delta y}$$

Recall that $x = i\Delta x$, $y = j\Delta y$, and substitute into the equation $xu_x + (1+y)u_y = 0$

$$x\mathrm{U}_x + (1+y)\mathrm{U}_y = i\Delta x\mathrm{U}_x[i,j] + (1+j\Delta y)\mathrm{U}_y[i,j] = 0$$

We have obtained the difference equation

$$i\Delta x\mathrm{U}_x[i,j] + (1+j\Delta y)\mathrm{U}_y[i,j] = 0$$

The equation involves the discrete derivatives U_x, U_y, and in this case the appropriate formulas must be used to extract a recurrence formula from it:

$$\begin{aligned} & i\Delta x\mathrm{U}_x[i,j] + (1+j\Delta y)\mathrm{U}_y[i,j] = 0 \\ \Rightarrow\quad & i\Delta x \cdot \frac{\mathrm{U}[i,j] - \mathrm{U}[i-1,j]}{\Delta x} + (1+j\Delta y)\frac{\mathrm{U}[i,j+1] - \mathrm{U}[i,j]}{\Delta y} = 0 \end{aligned}$$

$$
\begin{aligned}
\Rightarrow \quad & i\cdot(\mathbf{U}[i,j]-\mathbf{U}[i-1,j])+(1+j\Delta y)\frac{\mathbf{U}[i,j+1]-\mathbf{U}[i,j]}{\Delta y}=0\\
\Rightarrow \quad & i\Delta y\cdot(\mathbf{U}[i,j]-\mathbf{U}[i-1,j])+(1+j\Delta y)\,(\mathbf{U}[i,j+1]-\mathbf{U}[i,j])=0\\
\Rightarrow \quad & (1+j\Delta y)\,(\mathbf{U}[i,j+1]-\mathbf{U}[i,j])=i\Delta y\cdot\mathbf{U}[i-1,j]-i\Delta y\cdot\mathbf{U}[i,j]\\
\Rightarrow \quad & (1+j\Delta y)\mathbf{U}[i,j+1]=(1+j\Delta y)\mathbf{U}[i,j]+i\Delta y\cdot\mathbf{U}[i-1,j]-i\Delta y\cdot\mathbf{U}[i,j]\\
\Rightarrow \quad & (1+j\Delta y)\mathbf{U}[i,j+1]=[1+(j-i)\Delta y]\cdot\mathbf{U}[i,j]+i\Delta y\cdot\mathbf{U}[i-1,j]\\
\Rightarrow \quad & \mathbf{U}[i,j+1]=\frac{[1+(j-i)\Delta y]\cdot\mathbf{U}[i,j]+i\Delta y\cdot\mathbf{U}[i-1,j]}{(1+j\Delta y)}
\end{aligned}
$$

We have obtained the recurrence formula

$$\boxed{\mathbf{U}[i,j+1]=\frac{[1+(j-i)\Delta y]\cdot\mathbf{U}[i,j]+i\Delta y\cdot\mathbf{U}[i-1,j]}{(1+j\Delta y)}}$$

The formula is valid for $i=1,2,\ldots,N-1$, $j=0,1,\ldots,N-2$. Using the boundary conditions on $\mathbf{U}[i,0]$, $\mathbf{U}[0,j]$, we can compute $\mathbf{U}$ at all the remaining keys. The Python code for the solution shown in Figure 14 is short and elegant.

```
dx = dy = 1/N
U = np.zeros((N+1, N+1))

# Boundary conditions
for i in range(0, N+1):
    U[i,0] = 2*i*dx             # u(x,0) = 2x
    U[i,N] = i*dx               # u(x,1) = x

for j in range(0, N+1):
    U[0,j] = 0                  # u(0,y) = 0
    U[N,j] = 2 / (1+j*dy)       # u(1,y) = 2/(1+y)

def Solve(U):
  for i in range(1,N):
    for j in range(0,N-1):
      U[i,j+1] = ((1 + (j-i)*dx)*U[i,j] + i*dx*U[i-1,j]) / (1 + j*dx)
```

Figure 14: Python code for Example 5

Exercise 3: Find a discrete solution $u(x,y)$ in the form of a recurrence formula on a suitable difference grid for the following partial differential equation (first order) on the unit square $[0,1]\times[0,1]$.

$$
\begin{cases}
xu_x-yu_y=u, & 0<x<1,\ 0<y<1\\
u(x,0)=x, & 0\le x\le 1\\
u(0,y)=0, & 0\le y\le 1\\
u(x,1)=\frac{x}{1+x}, & 0\le x\le 1\\
u(1,y)=\frac{1}{1+y}, & 0\le y\le 1
\end{cases}
\tag{30}
$$

Find a closed-form solution to the problem using the methods learned at the beginning of the course, and compare it with the discrete solution at the grid points.

Guidance: It is recommended first to find a closed-form solution to the problem using the standard methods learned in the course. This will help validate the discrete solution. Choose different numbers of subdivisions N_x, N_y for the two axes of the domain $[0,1]\times[0,1]$. There are three discrete derivative formulas for U_x: backward, forward, and central, and three for U_y. Thus, a total of 9 cases must be examined. In most cases, the resulting recurrence formulas do not work for various reasons (such as division by zero). The recommended way to overcome these problems is to construct a "reverse" recurrence formula: instead of advancing from 1 to N, advance from N to 1! That is, advance from step i to step $i-1$! To write the Python code, it is recommended to copy the Colab notebook from the preceding exercise and build the code for this exercise upon it. Based on our experiments, there is a minor stability problem near the point $(0,1)$. To minimize it, choose a number of subdivisions N_x that is significantly larger than N_y (for example, by a factor of 1000).

We present additional exercises involving various forms of the one-dimensional wave equation (some of them). As before, we shall practice once again with a simple 4×4 grid that can be solved manually. Review Example 1 at the end of the preceding chapter and carry out the four steps described there once again in preparation for the remainder of this chapter. This time, a small surprise is expected in connection with the boundary condition on the first derivative, and it is therefore strongly recommended not to skip this exercise.

Exercise 4: Find a discrete solution of size 4×4, similar to the one presented in Example 1, for the following one-dimensional wave equation

$$\begin{cases} u_{tt} = 4u_{xx}, & 0<x<3,\ 0<t<3 \\ u(0,t)=0, & 0\le t\le 3 \\ u(3,t)=1, & 0\le t\le 3 \\ u(x,0)=x^2(3-x), & 0\le x\le 3 \\ u_t(x,0)=1, & 0\le x\le 3 \end{cases}$$

Guidance: Review Example 1 at the end of the preceding chapter and construct a discrete model of size 4×4 for our equation. It is recommended to use central-derivative formulas for u_{xx}, u_{tt}, and a forward-derivative formula for u_t. Note that the condition $u_t(x,0)=1$ requires special treatment!

Exercise 5: Use the difference formula obtained in the solution of Exercise 4 to solve the following classical one-dimensional wave equation by the finite difference method

$$\begin{aligned} &u_{tt} = 4u_{xx}, && 0<x<1,\ 0<t<8 \\ &u(0,t)=0, && 0\le t\le 8 \\ &u(1,t)=0, && 0\le t\le 8 \\ &u(x,0)=\sin(\pi x), && 0\le x\le 1 \\ &u_t(x,0)=0, && 0\le x\le 1 \end{aligned}$$

Guidance: Note that the length of the string and the time interval differ from those in Exercise 4, and therefore adjustments are

required. The boundary conditions have also changed. For programming enthusiasts: it is recommended to construct a Colab notebook and check whether the solution appears reasonable by means of graphs depicting the state of the string at different times.

Exercise 6: Find a finite difference solution for the one-dimensional wave equation with a **damping** coefficient

$$\begin{aligned} &u_{tt} + 2u_t = 4u_{xx}, && 0 < x < 1,\ \ 0 < t < 8 \\ &u(0,t) = 0, && 0 \le t \le 8 \\ &u(1,t) = 0, && 0 \le t \le 8 \\ &u(x,0) = \sin(\pi x), && 0 \le x \le 1 \\ &u_t(x,0) = 0, && 0 \le x \le 1 \end{aligned}$$

Obtain a solution in the form of a recurrence formula similar to the one obtained for the heat equation (or Exercise 5).

Exercise 7: Find a finite difference formula for the ***Klein-Gordon equation***

$$\begin{aligned} &u_{tt} - c^2 u_{xx} + m^2 u = 0, && 0 < x < 1,\ \ 0 < t < 1 \\ &u(0,t) = 0, && 0 \le t \le 1 \\ &u(1,t) = 0, && 0 \le t \le 1 \\ &u(x,0) = \sin(4\pi x), && 0 \le x \le 1 \\ &u_t(x,0) = 0, && 0 \le x \le 1 \end{aligned}$$

This is a one-dimensional version of a wave equation from physics of quantum fields. Obtain a solution in the form of a recurrence formula similar to the one obtained for the heat equation (or Exercise 5).

Guidance: In various physics books, the equation is also called the ***Klein-Fock-Gordon*** equation or the ***Klein-Gordon-Fock*** equation. This is a wave equation from quantum relativity, and it is also related to the Schrödinger equation and to the Schrödinger field.

Exercise 8: (Schrödinger Equation)

Solve the one-dimensional **Schrödinger equation** with the following boundary conditions using the finite difference method

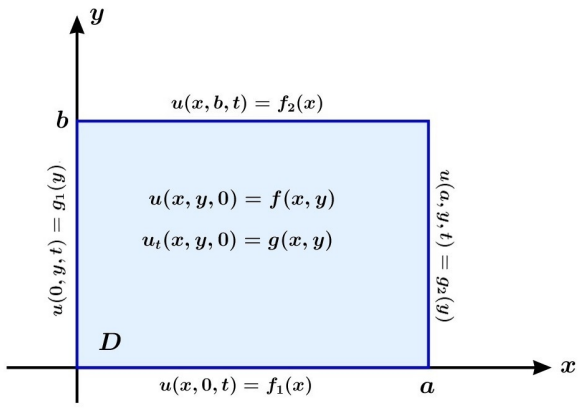
$$\begin{aligned} &i\hbar\psi_t = -\frac{\hbar^2}{2m}\psi_{xx} + V(x,t)\psi, && a < x < b,\ \ 0 < t < T \\ &\psi(a,t) = 0, && 0 \le t \le T \\ &\psi(b,t) = 0, && 0 \le t \le T \\ &\psi(x,0) = f(x), && a \le x \le b \end{aligned}$$

This is a one-dimensional version of Schrödinger's wave equation known from quantum physics. Find a solution in the form of a recurrence formula similar to the one obtained for the heat equation (or Exercise 5).

Guidance: This is another type of one-dimensional wave equation from quantum physics. The function $\psi(x,t)$ is called Schrödinger's ***wave function*** for a **particle** of mass m. Surprisingly, the function $\psi(x,t)$ takes complex values, even though its arguments (x,t) are real! It turns out that this does not present an obstacle to the finite difference method, and we proceed as usual. Note the imaginary unit $i=\sqrt{-1}$ appearing at the beginning of the equation, and take care not to confuse it with the index i (it is therefore recommended to use the indices j, k for the difference grid). The parameter m denotes the mass of the particle. The constant $\hbar$ is related to Planck's constant (Max Plank). Since the physical aspect of the problem is not important for us here, we may assume that $\hbar=1$. The function $V(x,t)$ is the particle's potential-energy function, which is given in advance, and therefore we can use its discrete values $V(x_j,t_k)$ in the recurrence formula. Likewise, the function $f(x)$ is given as the boundary condition $\psi(x,0)=f(x)$ at time $t=0$, and therefore we can use its discrete values $f(x_j)$ in the recurrence formula. As in the preceding examples, the magnitude $\gamma=\frac{\Delta t}{2m\Delta x^2}$ that may arise during the solution is of great importance. The stability condition (the **CFL condition**) for convergence of the discrete solution is: $\gamma\le 0.5$. Find a recurrence formula of the form $\mathbf{U}[j,k+1]=\Psi\big(U[i,k],V(x_i,t_k),f,\Delta x,\Delta t\big)$, where $\mathbf{U}$ is the difference grid of $\psi(x,y)$, and the expression Ψ on the right-hand side is an algebraic expression involving the parameters computed at the preceding step k.

4. The Wave Equation in the Plane (2D)

The course lecture notes treat the one-dimensional wave equation on a one-dimensional string (1D). The analytical treatment of the two-dimensional case (2D-Membrane) is similar, but is restricted to very specific domains and initial/boundary conditions. In this section, we shall see how to use the finite difference method (see Part IV of the lecture notes) to obtain a numerical solution of the equation without restrictions of this kind. For simplicity, we shall consider only the example of a two-dimensional rectangle $\boldsymbol{D=[0,a]\times[0,b]}$ made of a homogeneous material of uniform density (see Figure 15).



$$\begin{cases} u_{tt}=c^2(u_{xx}+u_{yy}), & 0<x<a,\ 0<y<b,\ 0<t<\infty \\ u(x,y,0)=f(x,y), & 0<x<a,\ 0<y<b \\ u_t(x,y,0)=g(x,y), & 0<x<a,\ 0<y<b \\ u(x,0,t)=f_1(x), & 0\le x\le a,\ 0\le t<\infty \\ u(x,b,t)=f_2(x), & 0\le x\le a,\ 0\le t<\infty \\ u(0,y,t)=g_1(y), & 0\le y\le b,\ 0\le t<\infty \\ u(a,y,t)=g_2(y), & 0\le y\le b,\ 0\le t<\infty \end{cases}$$

Figure 15: The wave equation with Dirichlet boundary conditions on a two-dimensional surface

For every point (x,y) on the surface D, $u(x,y,t)$ is the height coordinate z of the point at time t. The function $u(x,y,t)$ is twice continuously differentiable with respect to the variables x, y at interior points of the domain D, twice continuously differentiable with respect to the time

dimension t, and satisfies the following partial differential equation

$$u_{tt} = c^2(u_{xx} + u_{yy}), \quad (x, y) \in D, \quad 0 < t < \infty$$

where c is a physical constant determined by the type and density of the material.

As with the heat equation, there are various types of boundary/initial conditions for the wave equation. We shall discuss the common type in which the height and velocity of every interior point of the surface at time $t = 0$ are given by two functions $f(x, y)$, $g(x, y)$. The boundary points of D remain at rest and do not move along the time axis. For example, if the surface D is the rectangle $[0, a]\times[0, b]$, then the boundary conditions are as shown on the right-hand side of Figure 15

1.4 Solution Using the Finite Difference Method

As in the heat-equation example, for simplicity we restrict ourselves to a square domain $a = b = L$ over a finite time interval $[0, T]$. Choose a number of subdivisions N for the interval $[0, L]$, and a number of subdivisions N_t for the time interval $[0, T]$. As in the preceding section, construct a three-dimensional grid $\mathbf{U}$ of the form

$$(N + 1)\times(N + 1)\times(Nt + 1)$$

Recall once again that partitioning any axis into N subintervals creates $N + 1$ nodes!

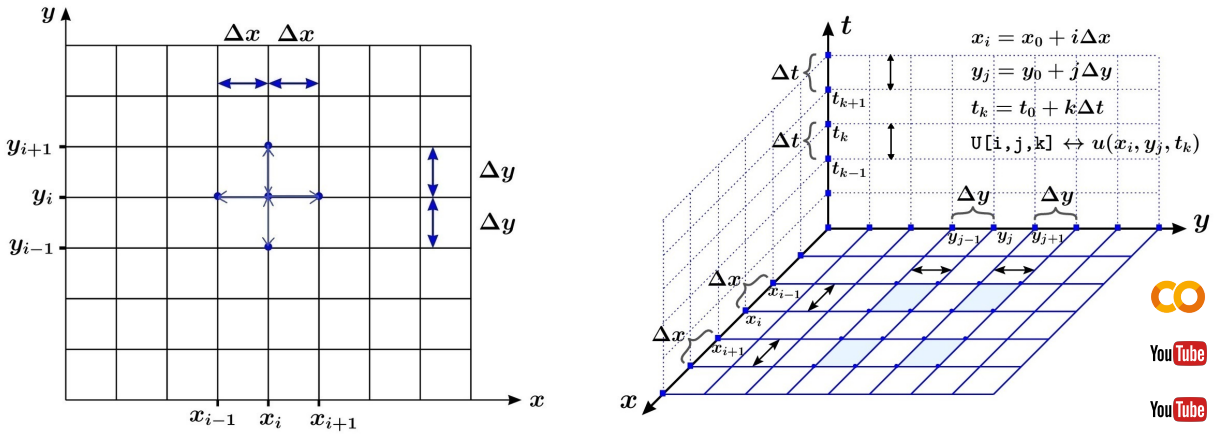


Figure 16: A multidimensional difference grid

Based on the differentiation formulas developed in the first section, we can use the following differentiation formulas

$$\mathbf{U}_{tt}[i, j, k] = \frac{\mathbf{U}[i, j, k] - 2\mathbf{U}[i, j, k - 1] + \mathbf{U}[i, j, k - 2]}{\Delta t^2} \tag{31}$$

$$\mathbf{U}_{xx}[i, j, k] = \frac{\mathbf{U}[i + 1, j, k] - 2\mathbf{U}[i, j, k] + \mathbf{U}[i - 1, j, k]}{\Delta x^2} \tag{32}$$

$$\mathbf{U}_{yy}[i,j,k] = \frac{\mathbf{U}[i,j+1,k] - 2\mathbf{U}[i,j,k] + \mathbf{U}[i,j-1,k]}{\Delta x^2} \tag{33}$$

Note that we chose a backward-difference formula for $\mathbf{U}_{tt}$, and central-difference formulas for $\mathbf{U}_{xx}$, $\mathbf{U}_{yy}$. From formula (31), it is easy to isolate $\mathbf{U}[i,j,k]$

$$\mathbf{U}[i,j,k] = 2\mathbf{U}[i,j,k-1] - \mathbf{U}[i,j,k-2] + \mathbf{U}_{tt}[i,j,k]\Delta t^2$$

We are given that $\mathbf{U}_{tt} = c^2\left(\mathbf{U}_{xx} + \mathbf{U}_{yy}\right)$ Therefore

$$\mathbf{U}[i,j,k] = 2\mathbf{U}[i,j,k-1] - \mathbf{U}[i,j,k-2] + c^2\Big(\mathbf{U}_{xx}[i,j,k] + \mathbf{U}_{yy}[i,j,k]\Big)\Delta t^2$$

Using the discrete Laplacian formula (27) encountered on page 19,

$$c^2\Big(\mathbf{U}_{xx}[i,j,k] + \mathbf{U}_{yy}[i,j,k]\Big) = \frac{c^2\boldsymbol{\Delta}\mathbf{U}}{\Delta x^2}$$

we obtain a recurrence formula along the time axis k

$$\mathbf{U}[i,j,k] = 2\mathbf{U}[i,j,k-1] - \mathbf{U}[i,j,k-2] + \frac{c^2\Delta t^2}{\Delta x^2}\boldsymbol{\Delta}\mathbf{U}$$

It is customary to denote

$$\gamma = \frac{c\Delta t}{\Delta x}$$

and we obtain a shorter and clearer recurrence formula

$$\boxed{\mathbf{U}[i,j,k] = 2\mathbf{U}[i,j,k-1] - \mathbf{U}[i,j,k-2] + \gamma^2\boldsymbol{\Delta}\mathbf{U}} \tag{34}$$

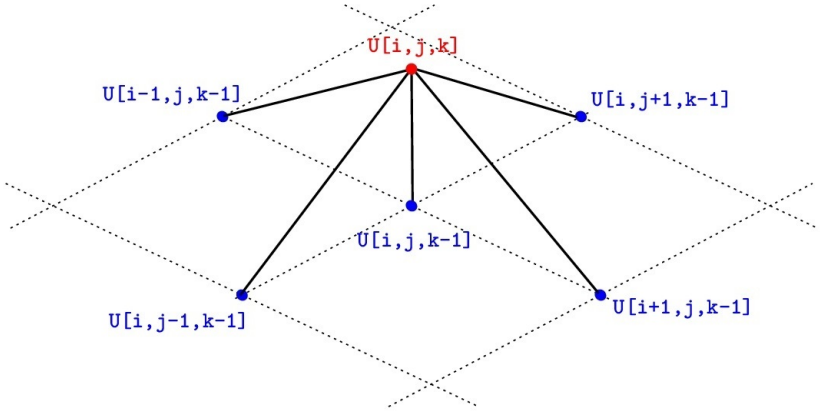
The expression $\boldsymbol{\Delta}\mathbf{U}$ is the discrete Laplace operator which we already encountered in the discussion of the heat equation! It appears repeatedly in equations of this kind. In software environments, this is a data structure called a ***Stencil***, which is commonly represented by a geometric diagram such as the one in Figure 17.

This time, the recurrence formula is based on two preceding states of the wave at times $k-1$, $k-2$. Thus, to apply the formula and obtain a complete solution, we need the state of the wave at times $k=0$, $k=1$. The state of the wave at time $k=0$ is given by the boundary condition $u(x,y,0) = f(x,y)$. To obtain the state of the wave at time $k=1$, we use the second boundary condition $u_t(x,y,0) = g(x,y)$.

$$\mathbf{U}_t[i,j,k] = \frac{\mathbf{U}[i,j,k+1] - \mathbf{U}[i,j,k]}{\Delta t}$$

Substitute $k=0$ to obtain

$$\mathbf{U}_t[i,j,0] = \frac{\mathbf{U}[i,j,1] - \mathbf{U}[i,j,0]}{\Delta t} = g(x_i, y_j)$$




```
DU = U[i-1,j,k-1] + U[i+1,j,k-1] + U[i,j-1,k-1] + U[i,j+1,k-1] - 4*U[i,j,k-1]
U[i,j,k] = 2*U[i,j,k-1] - U[i,j,k-2] + gamma*DU
```

Figure 17: Stencil for the recurrence formula used to solve the wave equation (Python code)

Therefore, the state of the wave at time $\boldsymbol{k=1}$ is given by

$$\mathbf{U}[i,j,1] = \mathbf{U}[i,j,0] + \mathbf{U}_t[i,j,0]\Delta t = f(x_i,y_j) + g(x_i,y_j)\Delta t$$

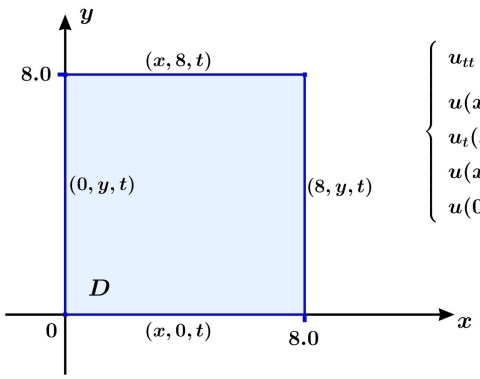
At this stage, we therefore have everything required to write an algorithm for simulating the states of the wave at every time $\boldsymbol{k=0,1,2,\ldots,N_t-1}$. Figure 18 presents Python code consisting of only 5 lines for the recurrence formula.

```
def Solve(U):
  Nrows, Ncols, Nt = U.shape
  for k in range(2, Nt):
    for i in range(1, Nrows-1):
      for j in range(1, Ncols-1):
        DU = U[i-1,j,k-1] + U[i+1,j,k-1] + U[i,j-1,k-1] + U[i,j+1,k-1] - 4*U[i,j,k-1]
        U[i,j,k] = 2*U[i,j,k-1] - U[i,j,k-2] + gamma**2 * DU
```

Figure 18: NumPy code for solving the wave equation

Note that this code does not compute the values of $\mathbf{U}[i,j,k]$ for the boundary keys $i=0,N$ or $j=0,N$! This is because the range `range(1, N)` does not include the values 0, N! The boundary points are handled by the boundary conditions in the stage preceding application of the recurrence formula. Complete NumPy code for solving a typical example of the wave equation, with explanations of all the parameters involved, is presented in the following example.

Example 6: We solve the following wave equation using the finite difference method



$$\begin{cases} u_{tt} = u_{xx} + u_{yy}, & 0 < x < 8,\ 0 < y < 8,\ 0 < t < \infty \\ u(x,y,0) = 8e^{-8\left((x-4)^2+(y-4)^2\right)} \sin\frac{\pi x}{4} \sin\frac{\pi y}{4}, & 0 < x < 8,\ 0 < y < 8 \\ u_t(x,y,0) = 0, & 0 < x < 8,\ 0 < y < 8 \\ u(x,0,t) = u(x,8,t) = 0, & 0 \le x \le 8,\ 0 \le t < \infty \\ u(0,y,t) = u(8,y,t) = 0, & 0 \le y \le 8,\ 0 \le t < \infty \end{cases}$$

This time, our square domain is $\boldsymbol{D = [0,8]\times[0,8]}$. Therefore, to obtain a high level of accuracy, we must use a finer partition than in the case of the heat equation.

Boundary conditions: All four functions $\boldsymbol{f_1(x)}$, $\boldsymbol{f_2(x)}$, $\boldsymbol{g_1(x)}$, $\boldsymbol{g_2(x)}$ vanish on the boundary of the square. The initial function $\boldsymbol{f(x,y)}$ is an interesting product of a Gaussian function and two trigonometric functions (inspired by the YouTube example [23]).

$$\boldsymbol{u(x,y,0) = 8e^{-8\left((x-4)^2+(y-4)^2\right)} \sin\frac{\pi x}{4} \sin\frac{\pi y}{4}}$$

As can be seen in the linked simulation video, this function produces two upper "blips" and two lower blips, which subsequently replicate themselves in various forms. The wave velocity $\boldsymbol{u_t(x,y,0)}$ at time $\boldsymbol{t = 0}$ is zero at every point of the domain.

In our simulation, we chose the number of subdivisions $\boldsymbol{N = 160}$, and therefore $\boldsymbol{\Delta x = \Delta y = 0.05}$. All other details of the simulation are specified and explained in the Colab notebook linked to Figures 18, 19.

```
L = 8.0                       # Length of square membrane (LxL)
N = 50                        # Grid division size
c = 1.0                       # Phisical wave speed constant
fps = 30                      # Frames per second (for video simulation)
seconds = 20                  # Simulation time (video duration)
num_frames = int(seconds*fps) # Total number of time frames (defined in wave2sim.py module)
dx = dy = L/N                 # Step sizes = x,y grid units
gamma = 0.4                   # CFL (Courant/Friedrich/Lewy) condition is met: gamma = c*dt/dx <= sqrt(2)/2
#gamma = c*dt/dx              # This is the standard definition of gamma. See class notes or project booklet
# We prefer to define gamma explicitly to ensure CFL condition, and then use it to define dt
dt = gamma * dx/c             # Automatic choice for dt (See link below)
T = seconds * fps * dt        # Physical time (the real time)
U = np.zeros((N+1, N+1, num_frames))  # Our FDM array container (grid)
```

Figure 19: Parameters of the wave-equation simulation

We explain some of the parameters:

a. `CFL` - Courant/Friedrich/Lewy condition This is a stability condition that guarantees that the discrete solution converges to a true or exceptional solution. For the wave equation, the stability

condition[7] is

$$\left(\frac{c\Delta t}{\Delta x}\right)^2+\left(\frac{c\Delta t}{\Delta y}\right)^2\le 1$$

In our case, the grid is uniform (that is, $\Delta x=\Delta y$), and therefore the stability condition is

$$\gamma=\frac{c\Delta t}{\Delta x}\le\frac{\sqrt{2}}{2}$$

The condition enables us to choose an "optimal" value of the time step Δt as a function of Δx, Δy, which guarantees a stable solution; in addition, the resulting value of Δt usually yields a simulation at a speed that is pleasant for the human eye. Further details about this important condition can be learned from the blog [19].

b. To guarantee that the CFL condition holds, we decided to begin with the concrete choice $\gamma=0.4$, which lies near the center of the interval $(0,\frac{\sqrt{2}}{2})$ of values permitted by the CFL condition, and then define Δt using the formula $\Delta t=\frac{\gamma\Delta x}{c}$.

c. The parameter `fps` (frames per second) denotes the number of frames per second in the simulation video. In most cases, the recommended value for the human eye is $\mathbf{fps}=30$.

d. As we have already seen for the heat equation, a distinction must be made between the ***simulation time*** `seconds`, which is also the duration of the video, and the ***physical time*** $\mathbf{T}$, which is the actual duration over which the wave phenomenon occurs, and which may be very short (a microsecond, for example) or very long (a million seconds, for example)[8]. In most cases, we shall not want to watch such a video for more than **30** to **60** seconds, and the simulation must therefore be accelerated or slowed down accordingly. The acceleration and deceleration can be controlled by choosing suitable subdivision sizes and/or filtering some of the frames (sampling).

e. Our difference grid $\mathbf{U}$ is a NumPy three-dimensional array whose shape is

$$(N+1)\times(N+1)\times\mathbf{num_frames}$$

The number of frames in the simulation, `num_frames`, is in fact equal to the product `seconds*fps` (the simulation time multiplied by the number of frames per second). The physical time `T` is `seconds*fps*dt` and not necessarily `num_frames*dt`! This is because in more complex simulations there is no requirement to use all the frames, and only a small sample is used (for example, one frame out of every ten). For simplicity, in the present case we use all the frames.

To complete the picture, we must understand how the boundary conditions are entered into the grid $\mathbf{U}$. In most examples found online or in many textbooks, this is performed by direct manipulation of the keys of the array $\mathbf{U}$. However, this type of work is suitable only for experienced programmers

[7] We relied on a ChatGPT prompt that was verified by code. See the document: https://samyzaf.com/fdm/gpt_wave2d_cfl.pdf
[8] The unit of time in all cases is the second

familiar with the intricacies of multidimensional arrays in NumPy. To overcome this obstacle, our Python package fdmtools, created for the course, includes a more convenient interface that saves us from dealing with overly complex programming mechanisms.

```
f  = lambda x,y: 8*exp(-8*((x-4)**2 + (y-4)**2)) * sin(pi*x/4) * sin(pi*y/4)
g  = lambda x,y: 0
f1 = lambda x: 0
f2 = lambda y: 0
g1 = lambda y: 0
g2 = lambda y: 0
```

Figure 20: Definition of the boundary functions for the grid

Except for the command `lambda` which denotes a short function, everything looks like the definition of ordinary mathematical functions. The final command, **`RunSimulation`**, in the code in the Colab notebook also enters these conditions into the grid, runs the simulation, and constructs the grid $\mathbf{U}$. This frees us from dealing with many technical details, and should make it easier to run additional examples encountered in the following exercises.

Exercise 9: Solve the following wave equation using the finite difference method and perform a simulation for a physical time of $T = 30$ seconds. Verify that the simulation displays at least 6 cycles of the wave.

$$\begin{cases} u_{tt} = u_{xx} + u_{yy}, & 0 < x < 2,\ 0 < y < 2,\ 0 < t < \infty \\ u(x,y,0) = xy(2-x)(2-y), & 0 < x < 2,\ 0 < y < 2 \\ u_t(x,y,0) = xy, & 0 < x < 2,\ 0 < y < 2 \\ u(x,0,t) = u(x,2,t) = 0, & 0 \le x \le 2,\ 0 < t < \infty \\ u(0,y,t) = u(2,y,t) = 0, & 0 \le y \le 2,\ 0 < t < \infty \end{cases}$$

Note that $c = 1.0$.

Guidance: Copy the Colab notebook wave.ipynb to a new notebook wave_ex9.ipynb and make the changes required to run the simulation. Note that this time the second boundary condition $u_t(x,y,0)$ is not zero! This example therefore illustrates the effect that the initial velocity of the wave at time $t = 0$ has on its subsequent behavior. An example of such a simulation can be viewed by clicking the YouTube button above.

This is a square domain $D = [0,2]\times[0,2]$ with simple boundary conditions: All four functions $f_1(x)$, $f_2(x)$, $g_1(x)$, $g_2(x)$ vanish on the boundary of the square. The state of the membrane at time $t = 0$ is $xy(2-x)(2-y)$, and the initial velocity at time $t = 0$ is $u_t(x,y,0) = xy$.

Solving this problem by classical methods requires constructing a two-dimensional Fourier series expansion of the function $g(x,y) = xy(2-x)(2-y)$, which we shall not present here because of the time and labor involved. Interested students are invited to consult the books [17], [18], [20], [22].

We conclude the chapter with the Poisson equation, named after the French mathematician Siméon Denis Poisson. This is, in fact, a nonhomogeneous Laplace equation. Unlike the examples

encountered thus far, solving it by the finite difference method requires a higher degree of ingenuity. The resulting system of equations is not easy to solve and requires advanced diagonalization methods from linear algebra, which we shall not discuss in this booklet because of space limitations. We shall content ourselves with two exercises involving the manual construction of small difference grids of size $\mathbf{4\times4}$ and $\mathbf{5\times5}$, of the kind encountered in Example 1 in the introductory chapter.

Exercise 10: Two-dimensional Poisson equation (S. D. Poisson). Review Example 1 in the introductory chapter, and find a discrete solution of size $\mathbf{4\times4}$ for the Poisson equation with the accompanying boundary conditions.

$$\begin{cases} u_{xx}+u_{yy}=10e^{x+\frac{y}{3}}, & 0<x<3,\ 0<y<3 \\ u(x,0)=9e^{x}, & 0\le x\le 3 \\ u(x,3)=9e^{x+1}, & 0\le x\le 3 \\ u(0,y)=9e^{\frac{y}{3}}, & 0\le y\le 3 \\ u(3,y)=9e^{3+\frac{y}{3}}, & 0\le y\le 3 \end{cases}$$

Exercise 11: Two-dimensional Poisson equation (S. D. Poisson). Review Example 1 in the introductory chapter, and find a discrete solution of size $\mathbf{5\times5}$ for the Poisson equation with the accompanying boundary conditions.



$$\begin{cases} u_{xx}+u_{yy}=17e^{x+\frac{y}{4}}, & 0<x<4,\ 0<y<4 \\ u(x,0)=16e^{x}, & 0\le x\le 4 \\ u(x,4)=16e^{x+1}, & 0\le x\le 4 \\ u(0,y)=16e^{\frac{y}{4}}, & 0\le y\le 4 \\ u(4,y)=16e^{4+\frac{y}{4}}, & 0\le y\le 4 \end{cases}$$

Note that this time the number of subdivisions is $N=4$, and therefore the grid has size $\mathbf{5\times5}$. This is not a simple problem. The recommended activity is detailed in the guidance.

Guidance: Difficult problem. This is a system of $\mathbf{25}$ equations in $\mathbf{25}$ unknowns. Nine of them are solved by the boundary conditions, leaving a reduced system of $\mathbf{16}$ equations in $\mathbf{16}$ unknowns. The order of the unknowns in the system is important. The customary order is called ***lexicographic***:

$$\mathbf{U[1,1],\ U[1,2],\ U[1,3],\ U[1,4],\ U[2,1],\ U[2,2],\ U[2,3],\ \ldots\ U[4,1],\ U[4,2],\ U[4,3],\ U[4,4],}$$

It is recommended only to compute the coefficient matrix of the reduced system. The matrix has a distinctive symmetric structure of a known type, and diagonalization methods for it can be found with additional effort.

5. The Black-Scholes-Merton Equation

The equation known as Black-Scholes-Merton, or **BSM** for short, is a partial differential equation for calculating the prices of financial derivatives (such as options PUT or CALL options in the stock market) as a function of the underlying asset price s and the maturity time t. The equation is based on an economic model called the Black–Scholes model developed by the economists Fischer Black, Myron Scholes, in 1997. Myron Scholes and Robert Merton were awarded the Nobel Prize in Economics for their contribution to this subject (Fischer Black died in 1995).

The value v of an option with strike price K (strike price) is a function $v = v(s,t)$ of the asset price s (usually a stock) and the maturity time t. In the Black-Scholes model, the function $v(s,t)$ must satisfy the following partial differential equation, called the ***Black-Scholes-Merton*** equation:

$$\boxed{v_t + rsv_s + \tfrac{1}{2}\sigma^2 s^2 v_{ss} = rv} \qquad \text{YouTube} \qquad (35)$$

The constant r is the average interest rate over the maturity period, and σ is a statistical constant that measures the ***volatility*** of the asset price. It is recommended to watch the YouTube video above for a brief explanation of the financial derivatives treated by the equation. This equation arises from physical considerations involving heat diffusion, Brownian motion, and stochastic processes. A detailed explanation is provided in Chapter 7.

As in the treatment of the heat and wave equations, given an option with strike price K, construct a difference-grid model $\mathbf{V}[i,k]$ for $v(s,t)$ as follows

- The grid $\mathbf{V}[i,k]$ is constructed over the rectangle $[0,S]\times[0,T]$.
- $[0,S]$ is the range of asset prices.
- $S = S_{max}$ is the upper bound on the asset price: usually $S = 2K$ or $S = 3K$.
- $[0,T]$ is the range of times for which we want to compute the option value.

For clarity, we emphasize once again: **The function `v(s,t)` computes the value of an option on an asset whose current price is `s`, with strike price `K`, at maturity time `t`.**

To construct the discrete model $\mathbf{V}$, choose a number of subdivisions N_s for the price interval $[0,S]$, and a number of subdivisions N_t for the time interval $[0,T]$. We obtain two lists of nodes

$$s_i = i\Delta s, \quad i = 0,1,2,\ldots,N_s, \quad \Delta s = \frac{S}{N_s}$$

$$t_k = k\Delta t, \quad k = 0,1,2,\ldots,N_t, \quad \Delta t = \frac{T}{N_t}$$

and, of course, define $\mathbf{V}[i,k] = v(s_i,t_k)$, where $v(s,t)$ is a possible solution of the equation

(subject to suitable boundary conditions). Define the discrete derivatives $\mathbf{V}_s$, $\mathbf{V}_{ss}$, $\mathbf{V}_t$ as before. For simplicity, restrict the discussion to European call options only (CALL Option). This option gives the buyer the right to purchase the asset at a price fixed in advance, $\mathtt{K}$ (Strike Price or Exercise Price), at a time fixed in advance t (measured from the time of purchase $t = 0$). The transaction can take place only at the maturity time t, and not earlier as permitted by the American model.

1.5 Solution Using the Finite Difference Method

The differential equation for a call option is

$$\boxed{-v_t + rsv_s + \tfrac{1}{2}\sigma^2 s^2 v_{ss} = rv} \qquad \text{YouTube} \qquad (36)$$

Observe that for call options, the equation begins with the term $-v_t$ rather than v_t, as it is usually presented (see [6]). The solution presented here is based on the blog [4] by Antoni Smolski, who also kindly provided complete Python code in a Jupyter notebook, on which we based our Colab notebook. Another important source, written in the C++ programming language and containing an excellent account of the theory and code, is reference [6].

We shall use central-difference formulas for $\mathbf{V}_s$, $\mathbf{V}_{ss}$, and a forward-difference formula for $\mathbf{V}_t$

$$\mathbf{V}_s[i,k] = \frac{\mathbf{V}[i+1,k] - \mathbf{V}[i-1,k]}{2\Delta s} \qquad (37)$$

$$\mathbf{V}_{ss}[i,k] = \frac{\mathbf{V}[i+1,k] - 2\mathbf{V}[i,k] + \mathbf{V}[i-1,k]}{\Delta s^2} \qquad (38)$$

$$\mathbf{V}_t[i,k] = \frac{\mathbf{V}[i,k+1] - \mathbf{V}[i,k]}{\Delta t} \qquad (39)$$

In difference notation, differential equation (36) is

$$-\mathbf{V}_t[i,k] + r(i\Delta s)V_s[i,k] + \frac{1}{2}\sigma^2(i\Delta s)^2\mathbf{V}_{ss}[i,k] = r\mathbf{V}[i,k]$$

Therefore

$$-\mathbf{V}_t[i,k] = r\mathbf{V}[i,k] - r(i\Delta s)V_s[i,k] - \frac{1}{2}\sigma^2(i\Delta s)^2\mathbf{V}_{ss}[i,k]$$

Therefore

$$-\frac{\mathbf{V}[i,k+1] - \mathbf{V}[i,k]}{\Delta t} = r\mathbf{V}[i,k] - r(i\Delta s)V_s[i,k] - 0.5\sigma^2(i\Delta s)^2\mathbf{V}_{ss}[i,k]$$

and after a small rearrangement, we obtain

$$\mathbf{V}[i,k+1] = \mathbf{V}[i,k] - \Delta t\left(r\mathbf{V}[i,k] - r(i\Delta s)V_s[i,k] - 0.5\sigma^2(i\Delta s)^2\mathbf{V}_{ss}[i,k]\right)$$

Verify that the index $k+1$ does not appear on the right-hand side (inspect formulas (37), (38) once again). We have therefore obtained a recurrence formula for computing the option value at time t_{k+1} from its value at time t_k!

It is customary to denote

$$\gamma = r\mathrm{V}[i,k] - r(i\Delta s)V_s[i,k] - 0.5(i\sigma\Delta s)^2\mathrm{V}_{ss}[i,k]$$

Our recurrence formula therefore reduces to the following formula

$$\boxed{\mathrm{V}[i,k+1] = \mathrm{V}[i,k] - \gamma\Delta t} \tag{40}$$

From this formula, we learn how the value of an option on an asset priced at s_i at time t_{k+1} depends on the three preceding values s_{i-1}, s_i, s_{i+1} at time t_k. At any given time t_k and asset price s_i, there is an equal probability that at the next time t_{k+1} the asset price will fall to s_{i-1}, rise to s_{i+1}, or remain unchanged at s_i. The recurrence formula weights these three possibilities (see the corresponding stencil in Figure 21).

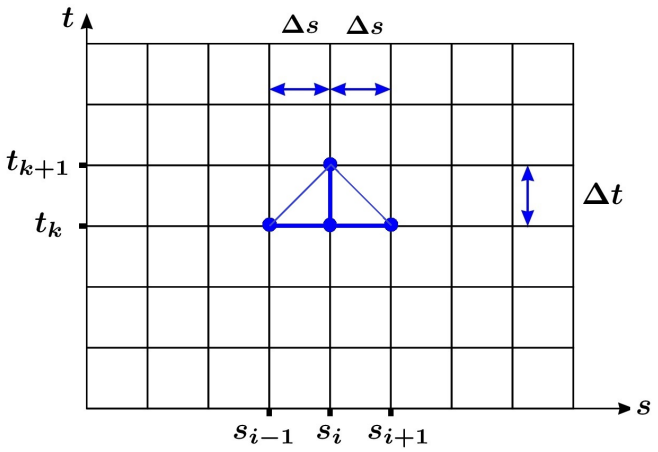


Figure 21: Stencil for the recurrence formula of the BSM equation

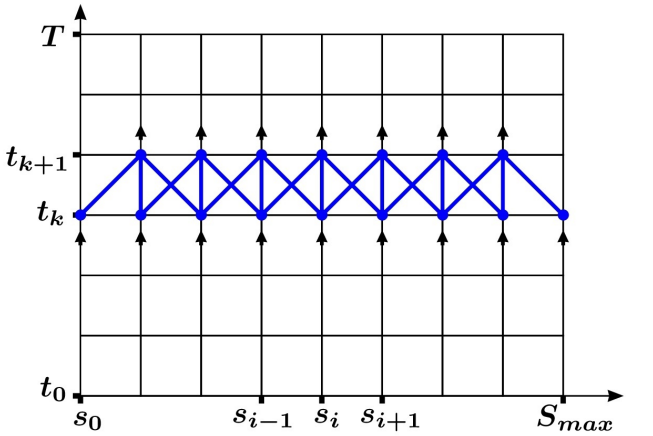


Figure 22: Transition from level k to level $k+1$ in the recurrence formula

One advantage of the finite difference method is that the required boundary conditions can be inferred from the form of the stencil. From differentiation formulas (38), (37), it is clear that they cannot be applied at the boundary indices $i = 0$ or $i = N_s$[9]. Therefore, the recurrence formula is valid only for $0 < i < N_s$ (see Figure 22). Thus, the values $\mathrm{V}(0,k)$, $\mathrm{V}(N_s,k)$ must be specified as boundary conditions. Likewise, to apply the formula, level $k = 0$ of the grid must be specified in advance. Determining these conditions accurately is critical to obtaining a high level of accuracy. We explain below how these conditions are determined. In summary: given boundary conditions for $t_0 = 0$, $s = 0$, $s = S$, this formula yields the complete solution at every time!

It is not difficult to write suitable Python code for this formula, as we have already learned from

[9] In most cases, recurrence formulas are applicable only to interior keys of the grid

the examples involving the heat and wave equations. The entire Black-Scholes-Merton procedure is summarized by the following six lines of code:

```
for k in range(0, Nt):      # Time loop
    # And now ... Nobel prize winning diff equation ;)
    for i in range(1, Ns):  # Asset loop
        Vs = (V[i+1,k] - V[i-1,k]) / (2*ds)              # Vs[i,k] as central difference
        Vss = (V[i+1,k] - 2*V[i,k] + V[i-1,k]) / (ds**2)  # Vss[i,k] as central difference
        gamma = r*V[i,k] - r*(i*ds) * Vs - 0.5 * sigma**2 * (i*ds)**2 * Vss
        V[i,k+1] = V[i,k] - dt * gamma
```

Figure 23: NumPy code for solving the Black-Scholes-Merton equation

The complete Python code with explanations can be found in our Colab notebook, which is based on Antoni Smolski's notebook.

2.5 Boundary Conditions

In addition to Antoni Smolski's excellent blog, we also used reference [5] to treat the boundary conditions of our discrete model. As is customary in most cases, we work over a price interval $[\mathbf{0}, \boldsymbol{S}]$, where $\boldsymbol{S} = \mathbf{3}\boldsymbol{K}$ is the maximum asset price[10], and $\boldsymbol{K}$ is the desired strike price. Three conditions must be considered

a. $\underline{\boldsymbol{t} = \mathbf{0}}$: Purchasing an option for a very short time, such as $\boldsymbol{t} = \mathbf{0}$, (when the option is purchased and exercised simultaneously) is not a realistic case, but simple logic suggests that nothing should happen in such a transaction. That is, if $\boldsymbol{s} \geq \boldsymbol{K}$, then the option costs $\boldsymbol{s} - \boldsymbol{K}$, and at the same instant the buyer pays $\boldsymbol{K}$ to purchase the stock, so the transaction ends with zero profit for both parties. If $\boldsymbol{s} < \boldsymbol{K}$, then the option costs $\mathbf{0}$, and the buyer will naturally decline to purchase the stock because the offered price $\boldsymbol{K}$ exceeds its market price. Therefore, the boundary condition in this case, for every $\boldsymbol{s}$ in the price interval $[\mathbf{0}, \boldsymbol{S}]$, is

$$\boldsymbol{v}(\boldsymbol{s}, \mathbf{0}) = \begin{cases} \mathbf{0}, & \boldsymbol{s} < \boldsymbol{K} \\ \boldsymbol{s} - \boldsymbol{K}, & \boldsymbol{s} \geq \boldsymbol{K} \end{cases}$$

or, more concisely,

$$\boldsymbol{v}(\boldsymbol{s}, \mathbf{0}) = \mathbf{max}\{\boldsymbol{s} - \boldsymbol{K}, \mathbf{0}\}$$

as we shall see shortly in the code version.

b. $\underline{\boldsymbol{s} = \mathbf{0}}$: When the asset price is $\boldsymbol{s} = \mathbf{0}$, a strike-price offer $\boldsymbol{K} > \mathbf{0}$ is unrealistic (the company has gone bankrupt), and therefore, formally, the option value is also $\mathbf{0}$. That is, $\boldsymbol{v}(\mathbf{0}, \boldsymbol{t}) = \mathbf{0}$ for every $\mathbf{0} \leq \boldsymbol{t} \leq \boldsymbol{T}$.

c. $\underline{\boldsymbol{s} = \boldsymbol{S}}$: As stated above, $\boldsymbol{S} = \mathbf{3}\boldsymbol{K}$ is the maximum asset price included in our discrete model

[10] In more practical applications, it is customary to define $\boldsymbol{S}$ as the stock value lying $\mathbf{8}$ standard deviations from the mean price.

$\mathbf{V}$. If the asset price is very high relative to the desired strike price $\boldsymbol{K}$, simple logic suggests that the option value $\boldsymbol{v(S,t)}$ should be correspondingly high. Without considering interest, the value should be $\boldsymbol{v(S,t) = S - K}$. The assumption is that the asset price will remain near this price, and therefore the option buyer must pay the difference $\boldsymbol{S - K = 2K}$ in advance. However, since the strike price $\boldsymbol{K}$ is paid at time $\boldsymbol{t}$, interest must also be taken into account

$$\boldsymbol{v(S,t) = S - Ke^{-rt}}$$

The core Python code for solving the problem by the finite difference method (including the boundary conditions) is presented in Figure 24.

```
for i in range(Ns+1):
    V[i,0] = max(i*ds - K, 0)       # Initial condition at time t=0

for k in range(0, Nt+1):            # Time loop
    V[0,k] = 0                      # Boundary condition at s=0
    V[Ns,k] = S - K*exp(-r*k*dt)    # Boundary condition at s=S (Smax)

for k in range(0, Nt):      # Time loop
    # And now ... Nobel prize winning diff equation ;)
    for i in range(1, Ns):  # Asset loop
        Vs = (V[i+1,k] - V[i-1,k]) / (2*ds)              # Vs[i,k] as central difference
        Vss = (V[i+1,k] - 2*V[i,k] + V[i-1,k]) / (ds**2)  # Vss[i,k] as central difference
        gamma = r*V[i,k] - r*(i*ds) * Vs - 0.5 * sigma**2 * (i*ds)**2 * Vss
        V[i,k+1] = V[i,k] - dt * gamma
```

Figure 24: NumPy code for solving the Black-Scholes-Merton equation (including boundary conditions)

The complete code of the **`Solve`** function for the BSM equation is presented in the Colab notebook bsm.ipynb. Note that the **`data`** object returned by this function is called a DataFrame. A great deal of information about the discrete model $\mathbf{V}$ can be extracted from it using simple queries. This is the basic component of the Python pandas package for data processing and analysis. It can also be exported to an Excel table and analyzed using the tools provided by Excel.

```
data
```

	0.00	3.75	7.50	11.25	15.00	18.75	22.50	26.25	30.00	33.75	...	116.25	120.00	123.75	127.50	131.25	135.00	138.75	142.50	146.25	150.00
0.000	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	...	66.250	70.000	73.750	77.500	81.250	85.000	88.750	92.500	96.250	100.000
0.004	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	...	66.259	70.009	73.759	77.509	81.259	85.009	88.759	92.509	96.259	100.009
0.007	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	...	66.268	70.018	73.768	77.518	81.268	85.018	88.768	92.518	96.268	100.018
0.011	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	...	66.276	70.026	73.776	77.526	81.276	85.026	88.776	92.526	96.276	100.026
0.014	0.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	...	66.285	70.035	73.785	77.535	81.285	85.035	88.785	92.535	96.285	100.035
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2.986	0.0	0.001	0.025	0.147	0.465	1.050	1.933	3.119	4.592	6.333	...	74.485	78.063	81.651	85.247	88.850	92.459	96.072	99.690	103.311	106.934
2.989	0.0	0.001	0.025	0.147	0.467	1.053	1.937	3.124	4.599	6.340	...	74.495	78.072	81.660	85.256	88.858	92.467	96.081	99.698	103.319	106.942
2.993	0.0	0.001	0.025	0.148	0.468	1.055	1.941	3.129	4.605	6.347	...	74.504	78.082	81.669	85.265	88.867	92.475	96.089	99.706	103.326	106.949
2.996	0.0	0.001	0.025	0.149	0.470	1.058	1.945	3.134	4.611	6.355	...	74.513	78.091	81.678	85.273	88.876	92.484	96.097	99.714	103.334	106.957
3.000	0.0	0.001	0.025	0.149	0.471	1.060	1.949	3.139	4.617	6.362	...	74.523	78.100	81.687	85.282	88.884	92.492	96.105	99.722	103.342	106.965

855 rows × 41 columns

Figure 25: pandas table of the model $\mathbf{V}$

In addition to the many operations that can be performed on a Pandas table, a heat map can be drawn in which the option value $v(s,t)$ is represented by a corresponding temperature over the rectangular grid domain (see Figure 26). Another visualization can be obtained using a three-dimensional surface that can easily be generated from our table (see the Colab notebook). See Figure 28.

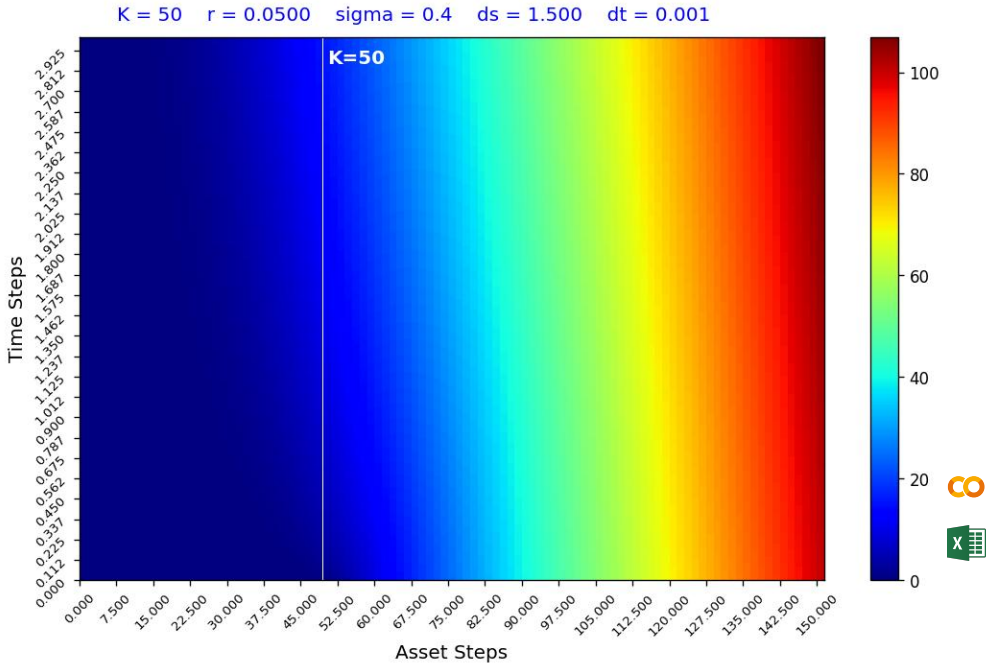


Figure 26: Heat map of the tabular representation of $\mathbf{V}$



Figure 27: Information website for the AAPL stock

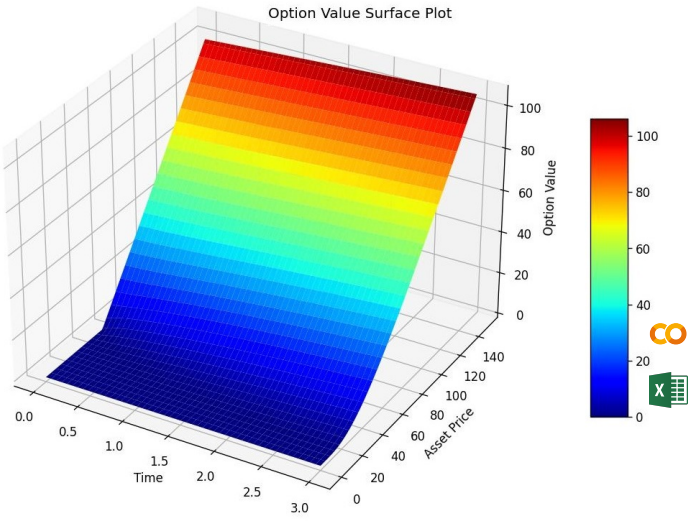


Figure 28: Visualization of $\mathbf{V}$ using a three-dimensional surface

A heat map and a three-dimensional surface provide a general impression of the behavior of the option value over a wide range of asset prices and times, but are not practical for computing $v(s,t)$ for specific values of the price and maturity time. For this purpose, we have the Excel file mentioned above, but it is more effective to submit focused queries to the `data` object as can be seen

in the accompanying Colab notebook.

Exercise 12: On 20.07.2020, the price of AAPL stock was **100** dollars. The interest rate in the United States that year was **4.25%**, and the stock's volatility level that year was **0.2**.

a. Construct a discrete model $\boldsymbol{V}$ of AAPL stock for a period of **4** years with a strike price of **100** dollars (Call Option).

b. Use the model to generate a price table for options over the four years following the date above. After constructing the table, determine which options were exercised at a profit and what the profit was in each case. Data about the stock can be obtained by clicking Figure 27.

c. Construct a discrete model $\boldsymbol{V}$ of AAPL stock for a period of **4** years with a strike price of **110**.

d. Using the stock data for the years 2020-2023, compare the profits from selling the option in the two cases. Present a profit table for the three years 2020-2023 (after deducting costs and interest, of course).

Guidance: Take $r = 0.0425$, $\sigma = 0.2$. To obtain a reasonable level of accuracy, take at least $N_s = 100$. A time unit of one year can be used, and therefore our time interval is $[0, 4]$ (that is, $T = 4$). The stock-price interval is $[0, 300]$ ($S = 300$). The table should include only the times $t = 1.0, 2.0, 3.0, 4.0$. For parts c-d, construct an entirely new model in which the strike price is $K = 110$. The profit on the option at the exercise time equals the stock price at exercise minus the strike price and the option cost.

6. Appendix: Accuracy, Stability, and Convergence

The central question concerning the discrete solution obtained by the finite difference method is: What conditions guarantee a desired level of accuracy relative to the true solution?

For simplicity, the discussion in this project is restricted to linear partial differential equations of order 2. Our principal aim is to present the idea and learn the method through simple examples. The list of references at the end of the booklet includes books and articles that discuss more general equations. In Part III of the course notes, we shall see that the general form of a linear PDE of order 2 is

$$a(x,y)u_{xx} + 2b(x,y)u_{xy} + c(x,y)u_{yy} + d(x,y)u_x + e(x,y)u_y + f(x,y)u = g(x,y) \quad (41)$$

where $a(x,y)$, $b(x,y)$, $c(x,y)$, $d(x,y)$, $e(x,y)$, $f(x,y)$, $g(x,y)$ are twice continuously differentiable functions on a given rectangular domain D. If (x_i, y_j) are the nodes of a grid over the rectangle D, with the subdivision $i = 0, 1, 2, \ldots, N_x$, $j = 0, 1, 2, \ldots, N_y$, we can define, as before,

$$\mathrm{U}[i,j] = u(x_i, y_j), \quad \mathrm{A}[i,j] = a(x_i, y_j), \quad \mathrm{B}[i,j] = b(x_i, y_j), \quad \mathrm{C}[i,j] = c(x_i, y_j),$$
$$\mathrm{D}[i,j] = d(x_i, y_j), \quad \mathrm{E}[i,j] = e(x_i, y_j), \quad \mathrm{F}[i,j] = f(x_i, y_j), \quad \mathrm{G}[i,j] = g(x_i, y_j)$$

After passing to the discrete variables in equation (41), we obtain a system of $(N_x-1)\times(N_y-1)$ equations (interior keys only)

$$\begin{aligned}\mathbf{A}[i,j]\,\mathbf{U}_{xx}[i,j]+2\mathbf{B}[i,j]\,\mathbf{U}_{xy}[i,j]+\mathbf{C}[i,j]\,\mathbf{U}_{yy}[i,j]\\ +\mathbf{D}[i,j]\,\mathbf{U}_{x}[i,j]+\mathbf{E}[i,j]\,\mathbf{U}_{y}[i,j]+\mathbf{F}[i,j]\,\mathbf{U}[i,j]=\mathbf{G}[i,j]\end{aligned}$$

Using differentiation formulas of the form (8)-(15) encountered in the first chapter, we convert all the partial derivatives into expressions involving only the unknowns $\mathbf{U}[i,j]$, and thereby obtain a system of equations in the unknowns $\mathbf{U}[i,j]$. Note that the number of unknowns is $(N_x+1)\times(N_y+1)$, equal to the number of nodes in the grid. Therefore, this system must be supplemented by the equations derived from the boundary conditions on the rectangle D (Dirichlet, Neumann, or mixed) in order to obtain a complete system of equations that guarantees a unique solution.

1.6 Accuracy and Stability

Assuming that the hypotheses of the existence and uniqueness theorem for equation (41) (with boundary conditions) hold, we are guaranteed the existence of a unique continuous solution $u(x,y)$ of the equation on the rectangle D. We return to the question with which we began: Are there conditions guaranteeing the existence of a discrete solution $\mathbf{U}[i,j]$ at every required level of accuracy? For example, we may require the distance between $u(x_i,y_j)$ and $\mathbf{U}[x_i,y_j]$ to be bounded by a given ε:

$$\left|\mathbf{U}[x_i,y_j]-u(x_i,y_j)\right|<\varepsilon,\quad i=0,1,2,\ldots,N_x,\quad j=0,1,2,\ldots,N_y$$

More precisely: is there pointwise convergence $\mathbf{U}[x_i,y_j]\to u(x_i,y_j)$ as the numbers of subdivisions N_x, N_y tend to infinity? The requirement of ***stability*** poses a more stringent challenge: Are there conditions guaranteeing that as the partition becomes finer (larger values of N_x, N_y), the distance between $\mathbf{U}$ and u also decreases? The answer to these questions depends on the type of equation and its accompanying conditions. In most cases known to us, the answer is affirmative, but we shall not present formulations or proofs here, apart from several examples and links to sources.

For the two-dimensional heat and wave equations, we encountered the Courant/Friedrich/Lewy (CFL) condition, which guarantees stability when the grid subdivision parameters satisfy it. Recall, for example, that in Chapter 2 we saw that a sufficient stability condition for the heat equation on a rod of length L over a time interval of duration T is

$$\gamma=\frac{\alpha\Delta t}{\Delta x^2}\le 0.5$$

where $\Delta x=\frac{L}{N}$, $\Delta t=\frac{T}{N_t}$. An explanation is included in reference [8].

A stability condition for the two-dimensional heat equation is

$$\gamma = \frac{\alpha \Delta t}{\Delta x^2} + \frac{\alpha \Delta t}{\Delta y^2} \leq 0.5$$

When $\Delta x = \Delta y$ (as is usually the case), the condition becomes $\gamma = \frac{\alpha \Delta t}{\Delta x^2} \leq 0.25$.

A stability condition for the one-dimensional wave equation is

$$\gamma = \frac{c \Delta t}{\Delta x} \leq 1$$

where c is the physical constant of the equation. A stability condition for the two-dimensional wave equation is

$$\left(\frac{c \Delta t}{\Delta x}\right)^2 + \left(\frac{c \Delta t}{\Delta y}\right)^2 \leq 1$$

For a uniform grid ($\Delta x = \Delta y$), the stability condition is: $\gamma = \frac{c \Delta t}{\Delta x} \leq \frac{\sqrt{2}}{2}$.

A stability proof for a general first-order ordinary differential equation can be found in Chapter 8 (Theorem 8.2) of [21]. It is recommended to examine this case, since it contains an idea typical of stability proofs for various types of PDEs as well.

Similar stability conditions exist for various types of partial differential equations. We shall limit ourselves to the examples listed above. Students interested in studying convergence and stability for higher-order PDEs in greater depth are invited to consult the lecture notes [16], [8], [11] (Section 1.4.7), and the article [19], which discusses the CFL condition.

2.6 Exceptional Discrete Solutions

In addition to approximations of exact solutions, there are also exceptional discrete solutions that do not approximate any actual solution. This occurs in cases involving equations and boundary conditions that are not necessarily solvable in a purely mathematical sense, but for which something resembling a solution can nevertheless be produced in a numerical sense. We encountered such an example at the beginning of Chapter 2, in heat equation (23). This equation includes contradictory boundary conditions at boundary points[11], and therefore there are jumps (discontinuities) at these points that preclude the existence of an exact mathematical solution based on the known existence and uniqueness theorems. Nevertheless, the finite difference method works in these cases as well and yields numerical discrete solutions that are quite good in practical terms and may even be of practical importance in the technology industry.

The existence of exceptional discrete solutions may sound problematic, but it is in fact another advantage of the finite difference method, since incompatibility between boundary conditions may

[11] The contradiction is resolved by entering the conditions into the algorithm in their order of appearance, so that the final value of the function is determined by the last boundary condition entered into the algorithm that applies to the point.

arise precisely in models of practical scenarios, and the numerical method is the only one that yields practical solutions in such cases.

7. Appendix: The Theoretical Basis of the BSM Equation

1.7 Stochastic Processes

The principal theoretical foundation of the Black-Scholes-Merton equation lies at the heart of a special branch of probability theory called the study of stochastic processes. The underlying assumption of the Black-Scholes-Merton model is that the price function of a financial asset is in fact a stochastic process.

Stochastic processes constitute an important branch of probability theory that studies the laws and behavioral patterns of random processes such as the motion of a gas molecule (Brownian Motion), diffusion, heat propagation, the growth of bacterial populations, the motion of a financial asset's price, and so forth. This is a broad subject to which many academic institutions devote an entire course, so it is recommended merely to gain an initial impression from the links above.

Within our limited framework, we shall content ourselves with the following brief characterization: A stochastic process $\boldsymbol{S = S(t)}$ is a time-dependent function of $\boldsymbol{t}$ characterized by the following stochastic differential equation

$$\boldsymbol{dS = \mu S\, dt + \sigma S\, dW} \tag{42}$$

where $\boldsymbol{W = W(t)}$ is a Wiener process (Wiener process), which is the completely random component of the stochastic process, while $\boldsymbol{\mu S\, dt}$ is the expected drift in the value of $\boldsymbol{S}$. For an asset price, this is the interest rate or the expected rate of increase (or decrease) in the asset's value in the market environment in which it is traded. Thus, this is a combination of a process governed by a predictable law and a completely random process.

Let us elaborate: a stochastic process consists of two components

a. Deterministic component: $\boldsymbol{\mu S\, dt}$

This describes its expected process of change (or drift). The value $\boldsymbol{\mu}$ denotes an expected growth coefficient of the variable $\boldsymbol{S = S(t)}$. For a financial asset, this is the expected growth rate of the market in which the asset is traded (as observed over a significant historical period). Therefore, the expression $\boldsymbol{\mu S\, dt}$ computes the expected change in $\boldsymbol{S}$ over the time interval $\boldsymbol{dt}$ (without the random component).

b. Random component: $\boldsymbol{\sigma S\, dW}$

This component has three parts: $\boldsymbol{S}$ is the asset price at time $\boldsymbol{t}$; $\boldsymbol{\sigma}$ is the asset's level of volatility or sensitivity; and $\boldsymbol{W(t)}$ is a random change process (noise), usually called a Wiener process

(Wiener process). Another name for this process is Brownian motion (Brownian motion) (in the broader sense, beyond that of random molecular motion). The component $\sigma S\, dW$ is therefore the random component (noise) added to the preceding component. For example, in addition to a predetermined growth rate whose effect is clear and highly predictable, the price of a financial asset (such as a stock, an electrical appliance, or food) may be affected by thousands of other factors, such as: social/cultural conditions, political conditions, geopolitical conditions, and the behavior of competing companies; the moods of and relationships among directors and production managers; weather events, personal conflicts, psychological crises, and so forth. There is no practical way to quantify and predict the effect of these factors on the asset price, and the simplest way to model this component is by means of a random function $W(t)$ called a Wiener process. This is a completely random function of time t (advancing in completely random steps). For each unit of time Δt, the increment ΔW is a completely random real number in the interval $[-\Delta t, \Delta t]$ that is independent of the previous history of $W(t)$. The probability assigned to each real value of ΔW follows a normal distribution with variance Δt; therefore, the standard deviation is $\sqrt{\Delta t}$.

Figure 29 shows the form of the density graph of the Gaussian normal distribution with central value (that is, mean) μ and standard deviation σ. Since this is a differential equations course, we cannot expand on the subject beyond this point, and we recommend learning more about it by clicking the Wikipedia button on the right-hand side of the figure, which provides a good summary of the subject.

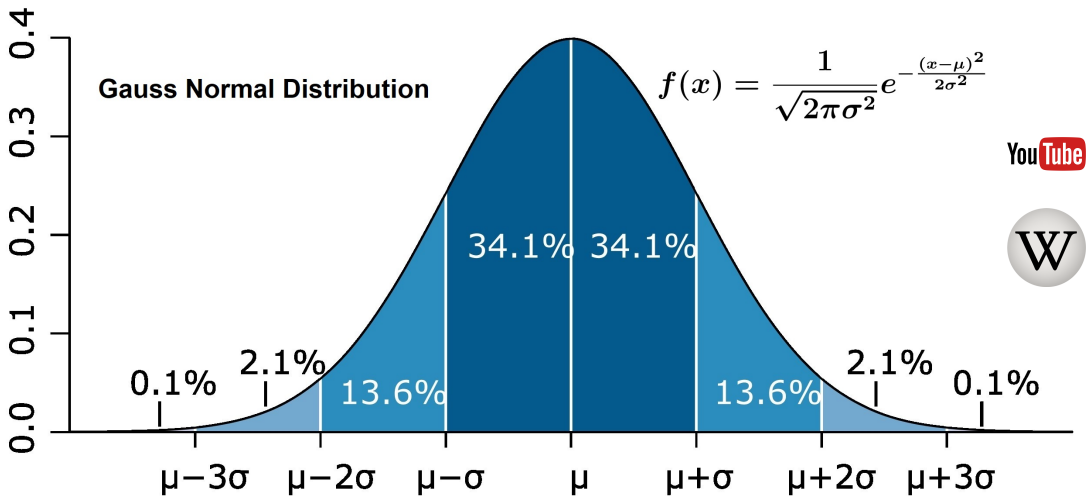


Figure 29: Normal distribution (Gaussian bell) with mean μ and standard deviation σ

In addition to the link to the Wikipedia page, there is also a link to an excellent YouTube video from the 3Blue1Brown channel, which explains the subject from the perspective of a teaching aid called a Galton board (Galton Board). Many probability and statistics courses use this teaching aid

to illustrate the fundamental principles of probability theory and the normal distribution. Viewing these videos is highly recommended for gaining a deep understanding of the notion of randomness at the heart of a Wiener process.

2.7 Statistical Background

On the left-hand side of Figure 30, 400 graphs of random functions are shown from the interval $[0, 4]$ to the range $[-4, 4]$. The color of each graph is determined by its final value at the last point $t = 4$ (according to a standard color map: blue near zero, red near 4 or -4). It is easy to see that the distribution of the graphs resembles the Gaussian bell (a normal distribution). After viewing the videos about the Galton board, we can see that the graphs are actually possible trajectories of a ball on a horizontal, continuous Galton board.

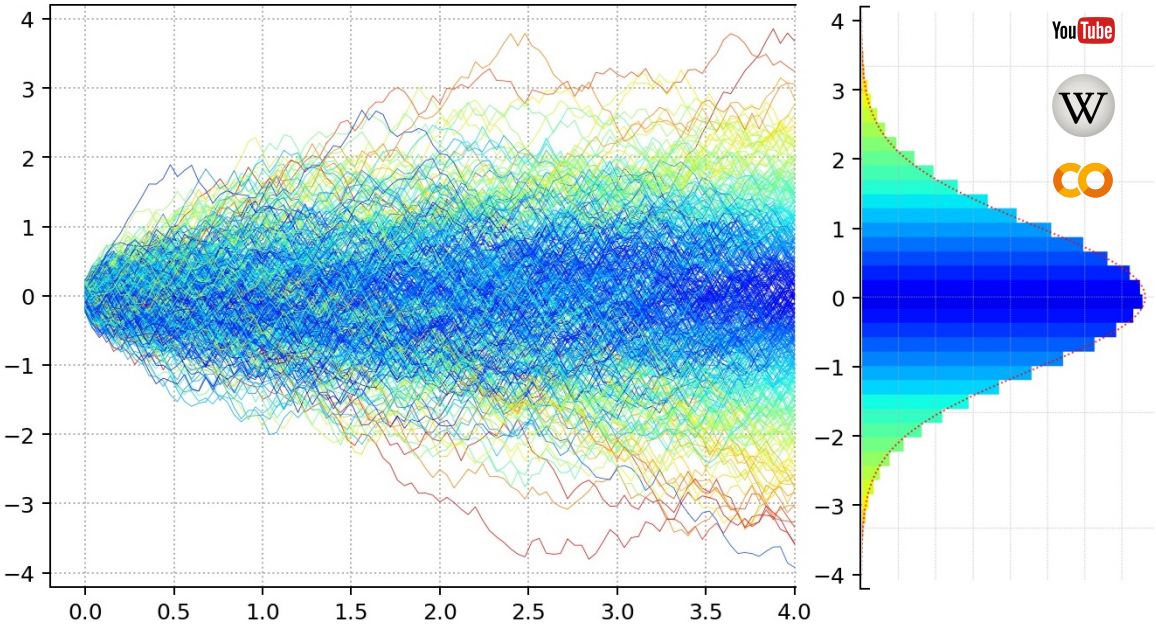


Figure 30: 400 random graphs $f : [0, 4] \to [-4, 4]$

The values of each random function lie in the range $[-4, 4]$. At every time t in the interval $[0, 4]$, the value of $f(t+\Delta t)$ is determined by the value $f(t)$ plus a random value from the closed interval $[-\Delta t, \Delta t]$. Each of the functions was generated recursively using the following code:

$$f(t + \Delta t) = f(t) + \mathbf{Random}(-\Delta t, \Delta t)$$

The function $\mathbf{Random}(-\Delta t, \Delta t)$ returns a random value in the interval $[-\Delta t, \Delta t]$; therefore, each time we construct such an f, we obtain a new function (with very high probability). The random functions are sorted according to the final value of the graph at $t = 4$ (the "maturity time of the option"). Note that the formal definition of a Wiener function is a random function for every $\Delta t > 0$! The definition does not make clear what such functions look like or what properties they have. For example, it can be proved that they are continuous at "most" points (except for a set of

measure zero), but are not differentiable at "most" points. We cannot elaborate further within our limited framework.

A function $\boldsymbol{W(t)}$ of the type described above is the core of the random component of a stochastic process: the random part of a stochastic process is obtained by multiplying $\boldsymbol{W(t)}$ by a sensitivity coefficient (volatility) $\boldsymbol{\sigma}$ and an asset price $\boldsymbol{S(t)}$. A Wiener process is characterized by the fact that the distribution of the function **Random** is **normal**. On the right-hand side of Figure 30, and also in Figure 31, a bar chart is shown that partitions the random graphs into $\mathbf{50}$ bins according to their terminal value $\boldsymbol{f(4)}$. The bar chart also serves as a color map.

The chart in Figure 31 presents the distribution of $\mathbf{500,000}$ random functions on the interval $\mathbf{[0,4]}$. We chose this large quantity to verify that the bar chart does indeed become progressively closer to a Gaussian bell. The graphs were classified into $\mathbf{50}$ different bins according to their terminal value $\boldsymbol{f(4)}$. For example, we can see that most of the functions belong to the blue bins near zero.

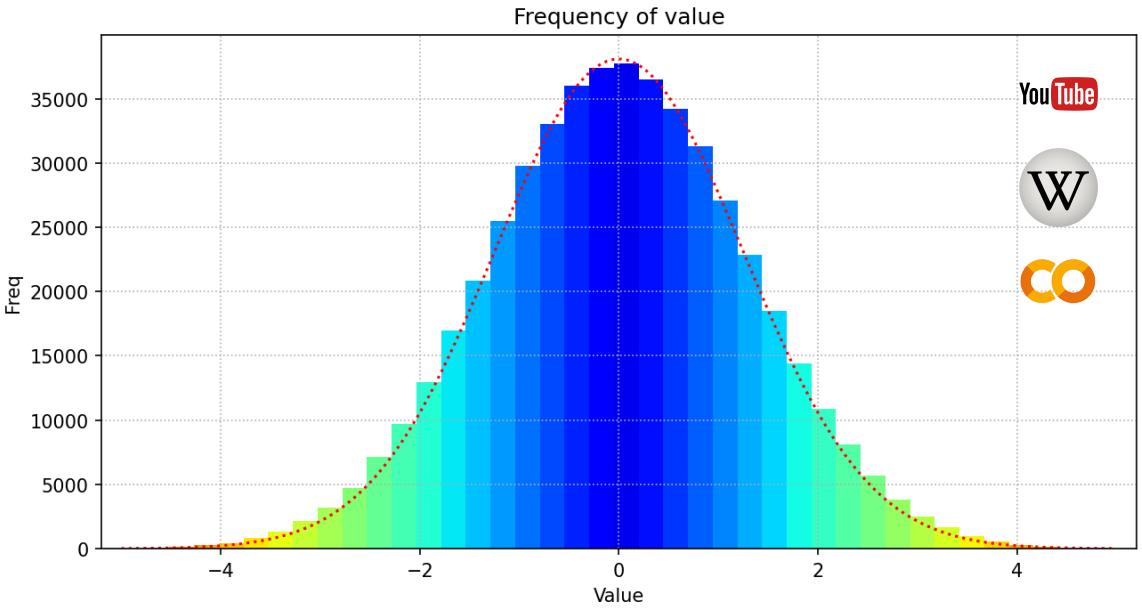


Figure 31: The distribution of the random graphs according to their final value is normal

3.7 Itô's Lemma (Kiyoshi Itô 1915-2008)

Another, more general name for the stochastic process described above is an Itô process, named after the Japanese mathematician Kiyoshi Itô ("The most famous Japanese in Wall Street"). The Japanese mathematician **Kiyoshi Itô** devoted a substantial part of his research to stochastic processes in financial markets and developed an entire branch of mathematics concerned with the differential and integral calculus of stochastic functions. Stochastic functions are characterized by graphs that behave randomly and therefore are not differentiable in the ordinary sense; consequently, they cannot be understood using the standard tools of classical calculus (in fact, the notion of continuity does not apply to them in the sense familiar to us). One of the most important results of his research

for the Black-Scholes-Merton model is a **"stochastic chain rule"** called Itô's lemma, which we shall use to construct the BSM differential equation. Recall that the chain rule enables us to compute the derivative of a composite function. In our case, we wish to differentiate the option value function $V(S,t)$ as a composite function $V(S(t),t)$, where $S(t)$ is the stock price at time t.

Theorem 1: (Itô's Lemma) Let $x = x(t)$ be a given Itô process. That is, there exist functions $a(x,t)$ and $b(x,t)$ such that

$$dx = a(x,t)dt + b(x,t)dW \tag{43}$$

Let $y = f(x,t)$ be a continuously differentiable function of both variables. Then

$$dy = \left[\frac{\partial f}{\partial x}a(x,t) + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}b(x,y)^2\right]dt + \frac{\partial f}{\partial x}b(x,t)dW \tag{44}$$

The Itô process $x(t)$ is characterized by equation (43), in which $a(x,t)$ and $b(x,t)$ are continuous functions of the variable x and the time variable t, while $W(t)$ is a Wiener process. As in a stochastic process, the first component $a(x,t)$ is the expected deterministic component, whereas the component $b(x,t)dW$ is the random component. The importance of Itô's lemma, which, as noted, is a form of the chain rule, is that it preserves the clear separation of the derivative of the composite stochastic process $f(x(t),t)$ into two components: one component with order and regularity that can be handled by classical Newtonian calculus, and a second, completely random component that is handled using statistical/probabilistic reasoning (Itô calculus).

We cannot prove this lemma within the limited scope of the course. We note only that the proof is based on the Taylor-series expansion of a function $f(x,t)$ of two variables. The main formula (44) consists of two parts: the deterministic part (the part inside the square brackets), which is in fact the first three terms of the Taylor series of $f(x,t)$, and the random part $\frac{\partial f}{\partial x}b(x,t)dW$. A complete proof of Itô's lemma can be found in Section 5.4 of the book [15].

Applying Itô's lemma to the function $V(S,t)$ in equation (42), we obtain

$$dV = \left[\frac{\partial V}{\partial S}\mu S + \frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\sigma^2 S^2\right]dt + \frac{\partial V}{\partial S}\sigma S\, dW \tag{45}$$

The equation we have obtained is called a stochastic differential equation, and it is rather difficult to solve because of the random component dW, which we have no tools to handle within our framework.

4.7 The Black-Scholes-Merton Strategy

The brilliant idea of Black-Scholes-Merton (for which they were awarded the Nobel Prize in Economics) was to neutralize the random component by constructing a theoretical investment portfolio combining derivatives (options) and ordinary shares in such a way that the portfolio is perfectly balanced at every point in time (with no loss or gain other than adjustment by the expected interest rate r). The idea is to purchase an option on a given stock together with a balancing quantity of ordinary shares of the same stock, so that a loss on one purchase is offset by a gain on the other. More precisely, the random component of one purchase is canceled by the random component of the other purchase, and as a result, the random component of the portfolio as a whole disappears from the equation! Our stochastic equation will therefore become an ordinary equation that we may be able to solve using standard mathematical methods.

We shall present only a schematic description of the idea, without entering into the finer technical details. Let $S(t)$ be the stock price at time t, and let $V(S,t)$ be the price of a put option (PUT) on a stock with strike price S at time t. Our speculative portfolio will consist of only two purchases:

a. a long position in one put option (PUT) with strike price $S(0)$ (that is, the current stock price at the initial time $t=0$);

b. a continuous holding of $\alpha = \frac{\partial V}{\partial S}$ ordinary shares at every given time t.

The theoretical assumption is that fractional shares may be purchased or sold at any given time, since the value of α may be any real number, including an irrational one! Another theoretical assumption is that no fees are charged for purchase and sale transactions; otherwise, our simulation is not feasible.

The value Π of such a portfolio at any given time is

$$\Pi = V - \alpha S \tag{46}$$

The instantaneous change in Π is[12]

$$d\Pi = dV - \alpha\, dS \tag{47}$$

[12] Differentiation in every case is with respect to the variable t. However, this equality is not fully justified because we have ignored the fact that $\alpha = \frac{\partial V}{\partial S}$ is also a function of t! We should therefore have differentiated the product αS according to the product rule. The full justification for ignoring the derivative of α is explained at length in Section 5.6 of the book [15].

© Rabayev/Zafrany (danielr@technion.ac.il/samyz@technion.ac.il)

Combining (42), (45), and (47) gives

$$\begin{aligned} d\Pi &= -\alpha\, dS + \left(\alpha\,\mu S + \frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt + \alpha\,\sigma S\, dW \\ &= -\alpha\,(\mu S\, dt + \sigma S\, dW) + \left(\alpha\,\mu S + \frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt + \alpha\,\sigma S\, dW \\ &= \left(\frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt \end{aligned}$$

We have therefore obtained

$$d\Pi = \left(\frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt \tag{48}$$

As we had hoped, the random component dW has disappeared from the equation! It is therefore no longer stochastic! In other words, the change in the portfolio value does not depend on the random component dW [13]. Note that the portfolio must be updated at every time increment Δt because the derivative $\alpha = \frac{\partial V}{\partial S}$ changes at every instant. That is, at every time increment Δt, we must ensure that the number of shares in the portfolio equals $\alpha = \frac{\partial V}{\partial S}$ at that instant (by purchasing or selling fractional shares). Of course, this is only a thought experiment, since in the real market it is not possible to purchase fractional parts of an option at every given time.

Assuming that market behavior is completely normal (with no crashes, arbitrage opportunities, economic crises, and so forth), the expected change in the market value should correspond to the expected interest rate r in the market environment. That is, the expected change is $d\Pi = r\Pi dt$; hence, on the basis of equation (48), we can write

$$r\Pi dt = \left(\frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt$$

By equation (46),

$$r\Pi dt = r\left(V - \frac{\partial V}{\partial S}\,S\right) dt = \left(\frac{\partial V}{\partial t} + \frac{1}{2}\,\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2\right) dt$$

and from this we obtain the Black-Scholes-Merton equation

$$\frac{\partial V}{\partial t} + \frac{1}{2}\,\frac{\partial^2 V}{\partial S^2}\,\sigma^2 S^2 + \frac{\partial V}{\partial S}\,rS - rV = 0$$

This equation is much simpler than equation (45), and can be handled using standard tools from the theory of partial differential equations. It turns out that for European options there is an analytic solution obtained through a substitution that transforms the equation into the heat equation. See page 7 of the paper [13], Section 4 of the paper [14], and Section 5.8 of the book [15]. For American options, no such solution exists, and the only known solution is a numerical one obtained using the

[13] More precisely, it is absorbed into the equation because our portfolio consists of two assets whose random components cancel each other.

finite-difference method, as presented earlier.

We are deeply grateful to Dr. Daniel Hackmann whose lucid paper [13] provides the best schematic explanation of the subject among all the sources we examined (after an extensive search).

8. Bibliography

1. **Numerical Partial Differential Equations: Finite Difference Methods.**
   James W. Thomas. Springer; 1995th edition (September 11, 1995).
   https://link.springer.com/book/10.1007/978-1-4899-7278-1
   https://www.amazon.com/-/he/J-W-Thomas/dp/0387979999
2. **Introductory Finite Difference Methods for PDEs.**
   D. M. Causon, C. G. Mingham. University of Manchester, England, 2010).
   https://www.cs.man.ac.uk/~fumie/tmp/introductory-finite-difference-methods-for-pdes.pdf
   https://www.bookboon.com
3. **Solving 2D Heat Equation Numerically using Python.** *G. Nervadof.* Level Up Coding.
   https://levelup.gitconnected.com/solving-2d-heat-equation-numerically-using-python-3334004aa01a
4. **Explicit Finite Difference with Black-Scholes Formula (with code).**
   Antoni Smolski. Medium.com Blog.
   https://antonismolski.medium.com/implementation-of-black-scholes-formula-using-finite-difference-method-with-code-965fd0539808
5. **Option pricing using the Black-Scholes model, without the formula.**
   R. Diter. Medium.com Blog.
   https://diter.medium.com/option-pricing-using-the-black-scholes-model-without-the-formula-e09235f75fb7
6. **C++ Explicit Euler Finite Difference Method for Black Scholes.**
   QuantStart Team. QuantStart Newsletter.
   https://www.quantstart.com/articles/C-Explicit-Euler-Finite-Difference-Method-for-Black-Scholes
7. **Solution of the Black-Scholes Equation by Finite Difference Schemes.**
   Ganga Ram, Kedar Nath Uprety, Harihar Khanal. Nepal Journals Online.
   Google Indexed Nepal Journal Paper
8. **Finite-Difference Method for the 1D Heat Equation.** *Padmanabhan Seshaiyer.*
   George Mason University. Dept. of Mathematical Sciences.
   https://math.gmu.edu/~pseshaiy/F12/m679/m679_F12_notes_hw1.pdf
9. **The 2D wave equation.** *Ingo Berg.* Recreational Programming.
   https://beltoforion.de/en/recreational_mathematics/2d-wave-equation.php
10. **Finite difference methods for 2D and 3D wave equations.** *Hans Petter Langtangen.*
    https://hplgit.github.io.
    https://hplgit.github.io/num-methods-for-PDEs/doc/pub/wave/sphinx/._main_wave005.html
    https://hplgit.github.io/num-methods-for-PDEs/doc/pub/wave/sphinx
11. **Finite Difference Computing with PDEs - A Modern Software Approach.**

Hans Petter Langtangen, Svein Linge. Springer Link.
https://link.springer.com/book/10.1007/978-3-319-55456-3
https://hplgit.github.io/fdm-book/doc/pub/book/pdf/fdm-book-4print.pdf

12. **Programming for Computations - A Gentle Introduction to Numerical Simulations with Python.** *Svein Linge, Hans Petter Langtangen.* Springer Link.
https://link.springer.com/book/10.1007/978-3-319-32428-9
https://hplgit.github.io/prog4comp/doc/pub/p4c_Python.pdf

13. **Solving the Black Scholes Equation using a Finite Difference Method.**
Daniel Hackmann. Personal Webpage.
http://www.danhackmann.com/BlackScholes7.pdf
http://samyzaf.com/fdm/BlackScholes7.pdf

14. **Stochastic Calculus and the Black–Scholes–Merton Model.**
Noah Fischer. Univ. of Chicago.
https://math.uchicago.edu/~may/REU2021/REUPapers/Fischer.pdf

15. **The Concepts of Mathematical Finance.** *M. S. Joshi.* Cambridge University Press.
https://assets.cambridge.org/97805215/14088/frontmatter/9780521514088_frontmatter.pdf

16. **Finite Difference Methods for Differential Equations.**
Randall J. LeVeque. Dept. of Applied Math. University of Washington.
https://edisciplinas.usp.br/pluginfile.php/41896/mod_resource/content/1/LeVeque%20Finite%20Diff.pdf

17. **An Introduction to Partial Differential Equations.** *Y. Pinchover, J. Rubinstein.* Cambridge University Press, 4th printing, 2008.

18. **Partial Differential Equations (4th Edition).** *Fritz John.* Springer 1971, ISBN 10: 0387900217, ISBN 13: 9780387900216.

19. **Understanding the Importance of the CFL Condition in CFD Simulations.**
Anthony Massobrio. NEURAL CONCEPT (BLOG).
https://www.neuralconcept.com/post/understanding-the-importance-of-the-cfl-condition-in-cfd-simulations

20. **DIFFERENTIAL EQUATIONS.** *Paul Dawkins.* Lamar University.
https://tutorial.math.lamar.edu/Classes/DE/DE.aspx

21. **Ordinary Differential Equations (English textbook).**
Samy Zafrany. Personal Webpage.
https://samyzaf.com

22. **Fourier Series and Integral Transforms (English textbook).**
Samy Zafrany. Personal Webpage.

https://samyzaf.com

23. 2D Wave Equation Simulation using Python. *Abel Flores Prieto.* YouTube.

https://www.youtube.com/watch?v=iS_msvfhB1I

https://github.com/abelfp/wave_equation_simulation

Dedicated to the memory of James Harris Simons
April 25, 1938 – May 10, 2024
A mathematician, and philanthropist.
"The mathematician that cracked Wall Street"