The 2D Wave Equation

Hans Petter Langtangen[*]

November 25, 2001

Contents

1 Introduction

Simulation of physical phenomena often involves numerical solution of partial differential equations (PDEs). This is particularly the case in mechanics, geophysics, astrophysics, and many engineering disciplines, as well as in parts of physics, geology, biology, and finance. When discussing high-performance computing aspects of numerical solution of PDEs, it is convenient to work with a model problem with as simple mathematics and numerics as possible, yet with physical and large-scale simulation relevance. The 2D or 3D standard, linear wave equation constitute such a problem. The wave equation is one of the simplest PDEs in physics from a mathematical point of view, and it can be solved by the simplest numerical methods for PDEs, namely explicit finite difference schemes. Fortunately, the resulting simulation code can (especially with a proper choice of boundary conditions) contain important constructions for exemplifying basic high-performance computing aspects, e.g.,

- large, nested do-loops
- if-tests and function calls inside/outside loops
- 5- or 7-point stensils and cache problems

This note describes in detail the description of an appropriate model problem to be used as a main example in a high-performance computing course.

> **NOTE: You do not need to read or understand the mathematics in Section 2 to carry out the exercises. Hence, you can jump directly to Section 3, which describes a sample code in Fortran 77, and then you can**

[*]Dept. of Scientific Computing, Simula Research Laboratory, and Dept. of Informatics, University of Oslo. `hpl@ifi.uio.no`.

start with the exercises in Section 4. If you are an experienced programmer, you will understand the basics of the exercises and the code without understanding the mathematical details behind the code.

2 Problem description

2.1 The mathematical model

Wave propagation is a physical process of fundamental importance; just think of water waves, sound waves for oral communication, and electromagnetic waves for transmission of light as well as radio and TV programs. The simplest mathematical model for waves consists of a partial differential equation, to so-called *wave equation*,

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [\lambda \nabla u] \,. \tag{1}$$

This equation for the unknown function u can describe, for example, water surface waves, electromagnetic waves, earthquake waves, waves on a drum, or sound waves in air. When it comes to water and earthquake waves, equation (1) is actually an approximation to the real-world physics, valid under certain assumptions; the water waves need to be long compared to the depth of water. This is satisfied for destructive flood waves generated by natural hazards such as avalanches, slides, earthquakes.

Understanding the right-hand side of equation (1) requires familiarity with vector calculus. We can write out the term $\nabla \cdot [\lambda \nabla u]$ explicitly for a two-dimensional application of the wave equation, i.e., when u depends on two spatial coordinates, x and y:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right) \,. \tag{2}$$

The parameter $\sqrt{\lambda}$ represents the velocity of the waves and depends in general on the properties of the medium in which wave the propagation takes place. In the case of water waves, $\lambda(x,y) = gH(x,y)$, where g is the acceleration of gravity and $H(x,y)$ is the still-water depth. The function $u(x,y,t)$ is the elevation of the surface, and $u = 0$ corresponds to still water. If H is constant, λ is constant and can be moved outside the spatial derivative. This results in a simpler equation,

$$\frac{\partial^2 u}{\partial t^2} = \lambda\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \,. \tag{3}$$

The right-hand side is the Laplacian of u, often denoted by $\nabla^2 u$. Hence, an alternative form of (3) is

$$\frac{\partial^2 u}{\partial t^2} = \lambda \nabla^2 u \,.$$

Some readers might recall this wave equation from other contexts.

Equation (2) is defined in a *domain* Ω, i.e., a two-dimensional area in which we want to simulate the wave propagation. This can be a harbor or just a part of a big ocean basin. The boundary of the domain is often denoted by $\partial\Omega$.

There are an infinite number of solutions to 2 unless we specify additional conditions. These conditions consists of *boundary conditions* and *initial conditions.* For the water wave application,

$$\frac{\partial u}{\partial n} \equiv \nabla u \cdot \boldsymbol{n} = 0 \tag{4}$$

is an appropriate boundary condition that we will make use of. The vector $\boldsymbol{n}$ is an outward unit normal to the boundary $\partial\Omega$. The condition (4) is to be applied at every point on $\partial\Omega$. If the boundary is the line $x = 0$, $\boldsymbol{n}$ is a unit vector along the x axis, and $\partial u/\partial n$ simply becomes

$\partial u/\partial x$. Physically, equation (4) tells that waves are perfectly reflected from the boundary, which is relevant if the boundary is, e.g., a coastline with steep hills. A simpler boundary condition is

$$u = 0\,. \tag{5}$$

Initial conditions specify the shape of u at initial time $t = 0$ throughout the domain Ω:

$$u(x, y, 0) = I(x, y)\,. \tag{6}$$

The physical interpretation of this equation is that the shape of the water surface is described by the function $I(x, y)$. In addition to this condition, we need to specify the time derivative of u at $t = 0$ as well. This time derivative is, from a physical point of view, the velocity of the water surface at $t = 0$. For simplicity, we assume that the surface is at rest (zero velocity):

$$\frac{\partial}{\partial t} u(x, y, 0) = 0\,. \tag{7}$$

The assumption of zero velocity can easily be relaxed.

The partial differential equations with the boundary and initial conditions constitute the complete mathematical model for simulating water waves[1]. In particular, we need to specify the depth function $H(x, y)$ and the initial surface shape $I(x, y)$ before the simulation can take place.

2.2 The numerical method

In the old days, clever mathematicians tried to find solutions to equations like (2) by paper and pencil. This is an extremely challenging task and in general impossible for a depth function $H(x, y)$ taken from a sea map. Solution in terms of a computer is, however, quite simple. A fairly short program is capable of calculating $u(x, y, t)$ with sufficient accuracy in seconds or minutes for any choice of $H(x, y)$ and initial shape $I(x, y)$.

The numerical solution method to partial differential equations to be used here is based on *finite differences*. Finite difference methods are quite easy to explain and program if the domain Ω has a rectangular shape, which we will assume in the rest of the document. We write

$$\Omega = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]\,.$$

The finite difference method is based on some fundamental ideas:

1. The domain Ω is represented by a *grid*, which is a rectangular set of points (x_i, y_j):

$$x_i = x_{\min} + (i-1)\Delta x, \quad y_j = y_{\min} + (j-1)\Delta y, \tag{8}$$

   for $i = 1, \ldots, n_x$, and $j = 1, \ldots, n_y$. The points form a rectangular array with equal spacings Δx and Δy between the points, see figure 1 (in this figure, $\Delta x = \Delta y$ but in general we can have $\Delta x \neq \Delta y$). The value of n_x and n_y in figure 1 is 9 and 6, respectively. The size the grid spacings is determined by

$$\Delta x = \frac{x_{\max} - x_{\min}}{n_x - 1}, \quad \Delta x = \frac{y_{\max} - y_{\min}}{n_y - 1}\,.$$

2. Also in time a grid is introduced:

$$t_0 = 0,\ t_1 = \Delta t,\ t_2 = 2\Delta t,\ t_3 = 3\Delta t, \ldots$$

   or expressed in terms of an index ℓ: $t_\ell = \ell\Delta t$, $\ell = 0, 1, \ldots$

[1]The term "water waves" refers in this document to water waves where the typical wave length is much greater than the water depth. If this assumption is violated, the solution to the wave equation will be qualitatively wrong.

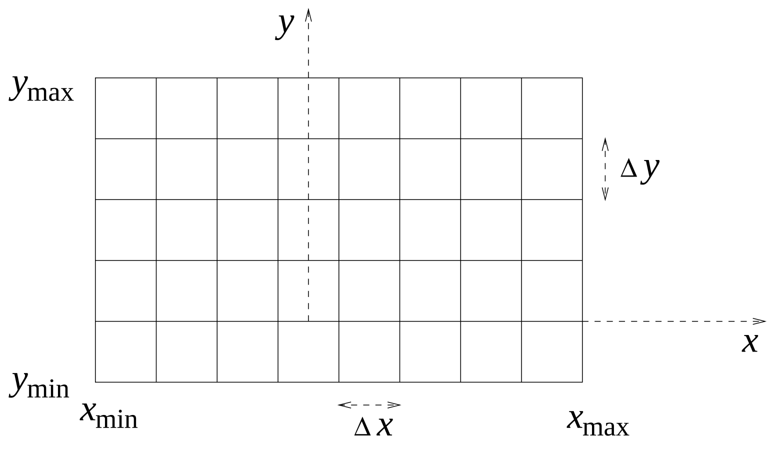

Figure 1: Example on a two-dimensional grid suitable for finite difference solution of partial differential equations.

3. We compute numerical approximations to $u(x_i, y_j, t_\ell)$, i.e., u at the space-time grid points. The approximations are denoted by $u_{i,j}^\ell$.
4. The governing partial differential equation (2) is supposed to be satisified at all space-time points (x_i, y_j, t_ℓ), $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$, $\ell = 0, 1, \ldots$. This is an approximation as equation (2) in the mathematical model is to be fulfilled at all the inifinte number of points in Ω and for all $t > 0$.
5. Derivatives in the partial differential equation is replaced by following finite difference approximations:

$$\frac{\partial^2}{\partial t^2}u(x_i, y_j, t_\ell) \approx \frac{u_{i,j}^{\ell-1} - 2u_{i,j}^\ell + u_{i,j}^{\ell+1}}{\Delta t}, \tag{9}$$

$$\frac{\partial}{\partial x}\left(\lambda(x_i, y_j)\frac{\partial}{\partial x}u(x_i, y_j, t_\ell)\right) \approx \frac{1}{\Delta x}\left(\lambda_{i+\frac{1}{2},j}\frac{u_{i+1,j}^\ell - u_{i,j}^\ell}{\Delta x} - \lambda_{i-\frac{1}{2},j}\frac{u_{i,j}^\ell - u_{i-1,j}^\ell}{\Delta x}\right) \tag{10}$$

$$\frac{\partial}{\partial y}\left(\lambda(x_i, y_j)\frac{\partial}{\partial y}u(x_i, y_j, t_\ell)\right) \approx \frac{1}{\Delta y}\left(\lambda_{i,j+\frac{1}{2}}\frac{u_{i,j+1}^\ell - u_{i,j}^\ell}{\Delta y} - \lambda_{i,j-\frac{1}{2}}\frac{u_{i,j}^\ell - u_{i,j-1}^\ell}{\Delta y}\right) \tag{11}$$

When λ is constant, the formulas simplify to (hopefully) well-known formulas for the second derivative:

$$\frac{\partial^2}{\partial x^2}u(x_i, y_j, t_\ell) \approx \frac{u_{i-1,j}^\ell - 2u_{i,j}^\ell + u_{i+1,j}^\ell}{\Delta x^2}\,.$$

6. At each time level we assume that u from previous time levels are already computed, i.e., they are known values.

Inserting the finite difference approximations into the wave equation results in what we refer to as the finite difference equations or simply the discrete equations. We make the important observation that there is only one term involving time level $\ell+1$ in the discrete equations. Since we assume that everything on the previous time levels ℓ and $\ell - 1$ are known, there is actually only one unknown value, $u_{i,j}^{\ell+1}$. We can solve the discrete equations with respect to this value and get something like

$$u_{i,j}^{\ell+1} = \text{known terms from previous time levels}$$

This means, algorithmically, that we can just run through all the grid points $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$, and compute new approximations to u from a simple formula. It is common

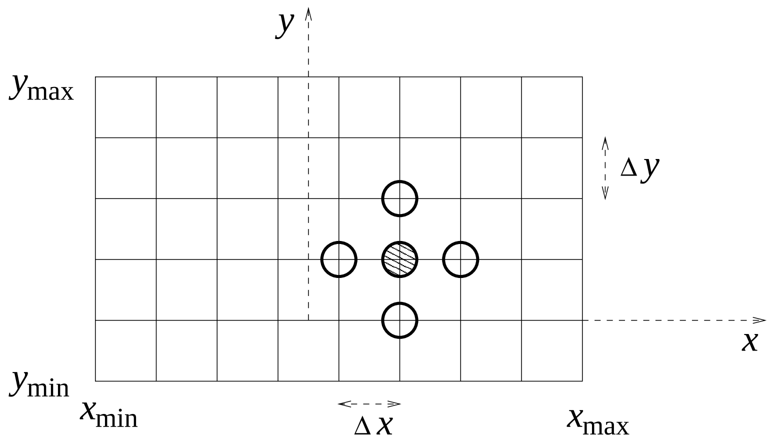

Figure 2: A new u value time level $\ell+1$ depends on values at previous time levels ℓ and $\ell-1$ at the same point and values at time level ℓ at the nieghboring points to the north, south, east, and west.

to classify such methods as *explicit* schemes. For other partial differential equations and other finite difference discretizations, new u values at time level $\ell+1$ will in general be coupled with each other in a linear system of algebraic equations (matrix system). This is referred to as an *implicit* scheme.

A notation, which is convenient when translating a computational algorithm to program statements in a code, is to set

$$u^{+}_{i,j} \equiv u^{\ell+1}_{i,j}, \quad u_{i,j} \equiv u^{\ell}_{i,j}, \quad u^{-}_{i,j} \equiv u^{\ell-1}_{i,j} \,.$$

We can now write the discrete equations compactly:

$$u^{+}_{i,j} = 2u_{i,j} - u_{i,j} + [\triangle u]_{i,j} \tag{12}$$

where

$$\begin{aligned} [\triangle u]_{i,j} \quad \equiv \quad & \left(\frac{\Delta t}{\Delta x}\right)^2 (\lambda_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - \lambda_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})) + \\ & \left(\frac{\Delta t}{\Delta y}\right)^2 (\lambda_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - \lambda_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1}))\,. \end{aligned} \tag{13}$$

Assuming that $u_{i,j}$ and $u^{-}_{i,j}$ are known for all (i,j) values in the grid, equation (12) gives the new u value at time level $\ell+1$ at the grid point with index i and j.

The "geometry" of (12) is interesting to observe: When computing a new u value at a spatial point with index i and j, we make use of previous u values at this point and the four neighboring points (west, east, north, and south). This is illustrated in figure 2. This observation, together with the fact that the order in which we visit the spatial indices i and j in (12) is irrelevant, forms the basis for constructing an algorithm for wave simulation on parallel computers.

There is one immediate basic problem with (12): If we apply it to the first time level, $\ell = 1$, it will involve $u^{-}_{i,j}$, i.e., u values at time level -1 ($t = -\Delta t$), which are unknown to us. The reason why we run into this problem is that we have not applied the initial condition $\partial u/\partial t = 0$ at $t = 0$. A typical finite difference approximation to this condition reads

$$\frac{\partial}{\partial t} u(x_i, y_j, 0) \approx \frac{u^{1}_{i,j} - u^{-1}_{i,j}}{2\Delta t} = 0,$$

which implies

$$u^{-1}_{i,j} = u^{1}_{i,j} \,.$$

Hence, we can just substitute $u^{-}_{i,j}$ by $u^{+}_{i,j}$ in (12) at the first time level. Alternatively, we can use (12) as it stands, even for the first time level, if we just define u at time level -1 to be

$$u^{-1}_{i,j} = u_{i,j} + \frac{1}{2}[\triangle u]_{i,j}\,.$$

Another problem with (12) is that we have not incorporated the boundary values. For simplicity, we assume that $u = 0$ on the boundaries. We can then just use the zero value when (12) involves u at a boundary. The complete computational recipe is summarized in algorithm 1.

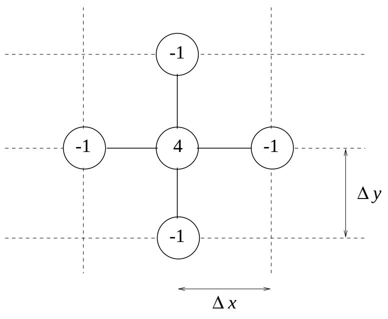

Figure 3: Illustration of the finite difference stencil for approximating the Laplace operator $\nabla^2 u$ in a regular grid.

Algorithm 1

Explicit scheme for the two-dimensional wave equation with $u = 0$ on the boundary.

```
define $u^+_{i,j}$ , $u_{i,j}$ and $u^-_{i,j}$ to represent $u^{\ell+1}_{i,j}$, $u^{\ell}_{i,j}$ and $u^{\ell-1}_{i,j}$, resp.
define $[\triangle u]_{i,j}$ as in (13)
define $(i,j) \in \bar{\mathcal{I}}$ to be $i = 1,\ldots,n_x$, $j = 1,\ldots,n_y$
define $(i,j) \in \mathcal{I}$ to be $i = 2,\ldots,n_x - 1$, $j = 2,\ldots,n_y - 1$
set $u_{i,j} = 0, \quad (i,j) \in \bar{\mathcal{I}}$
SET THE INITIAL CONDITIONS:
$u_{i,j} = I(x_i, y_j), \quad (i,j) \in \mathcal{I}$
DEFINE THE VALUE OF THE ARTIFICIAL QUANTITY $u^-_{i,j}$:
$u^-_{i,j} = u_{i,j} + \frac{1}{2}[\triangle u]_{i,j}, \quad (i,j) \in \mathcal{I}$
$t = 0$
while time $t \le t_{\text{stop}}$
    $t \leftarrow t + \Delta t$
    UPDATE ALL INNER POINTS:
    $u^+_{i,j} = 2u_{i,j} - u^-_{i,j} + [\triangle u]_{i,j}, \quad (i,j) \in \mathcal{I}$
    INITIALIZE FOR NEXT STEP:
    $u^-_{i,j} = u_{i,j}, \quad u_{i,j} = u^+_{i,j}, \quad (i,j) \in \mathcal{I}$
```

Notice that we do not explicitly set $u_{i,j} = 0$ at the boundary. Instead, we set $u_{i,j} = 0$ initially and never touch the boundary values.

We remark that if λ is constant and $\Delta x = \Delta y = h$, the numerical scheme simplifies greatly, and the Laplace term $\nabla \cdot [\lambda \nabla u] = \lambda \nabla^2 u$ takes the well-known discrete form

$$[\triangle u]_{i,j} = \lambda \left(\frac{\Delta t}{h}\right)^2 (-u_{i-1,j} - u_{i,j-1} - u_{i+1,j} - u_{i,j+1} + 4u_{i,j}) \ . \tag{14}$$

This formula for approximating $\lambda \nabla^2 u$ can be graphically exposed as in figure 3. The circles denote the points in the grid that are used in the approximation, and the numbers reflect the weight of the point in the finite difference formula. One often refers to such a graphical representation as a *finite difference stencil* or a *computational molecule.* In the more general case when λ is not constant, the same stencil arises, except that the weights are different.

Algorithm 1 assumes that $u = 0$ on the boundary. We shall now demonstrate how to implement a more complicated boundary condition, $\partial u/\partial n = 0$, in the scheme. The boundary condition is discretized by a centered difference at the boundary. For example, at the line $i = 1$

we then require

$$\frac{u^\ell_{2,j} - u^\ell_{0,j}}{\Delta x} = 0 \quad \Rightarrow \quad u^\ell_{0,j} = u^\ell_{2,j}\,.$$

Notice that this involves a fictitious value $u^\ell_{0,j}$ outside the grid. Using the discrete finite difference equation (12) at the same boundary point, with $u^\ell_{0,j} = u^\ell_{2,j}$ from the boundary condition, enables elimination of the fictitious value. The $[\triangle u]_{i,j}$ operator is then modified to

$$[\triangle u]_{1,j:i-1\to i+1} \equiv \left(\frac{\Delta t}{\Delta x}\right)^2 (\lambda_{1+\frac{1}{2},j}(u_{2,j} - u_{1,j}) - \lambda_{1-\frac{1}{2},j}(u_{1,j} - u_{2,j})) +$$
$$\left(\frac{\Delta t}{\Delta y}\right)^2 (\lambda_{1,j+\frac{1}{2}}(u_{1,j+1} - u_{1,j}) - \lambda_{1,j-\frac{1}{2}}(u_{1,j} - u_{1,j-1})),\ j \neq 1, n_y\,. \tag{15}$$

At the boundary $i = n_x$ we would then apply the modification $[\triangle u]_{n_x,j:i+1\to i-1}$. Similarly, for $j = 1$ and $j = n_y$ we replace the original $[\triangle u]_{i,j}$ operator by $[\triangle u]_{i,1:j-1\to j+1}$ and $[\triangle u]_{i,n_y:j+1\to j-1}$, respectively. The corner points of the grid require modification of both indices, for example, the $i = j = 1$ point leads to $[\triangle u]_{1,1:i-1\to i+1,j-1\to j+1}$.

Algorithm 2 precisely explains the updating of internal and boundary points in terms of a function WAVE(u^+, u, u^-, a, b, c). As a special case, the call WAVE$(u^+, u, u^-, 1, 1, 1)$ reproduces the original finite difference scheme with modifications due to homogeneous Neumann conditions $\partial u/\partial n = 0$.

Algorithm 2

```
Basic finite difference updating formula for the two-dimensional wave equation with $\frac{\partial u}{\partial n} = 0$ on the boundary.

define $[\triangle u]_{i,j}$ as in (13)
define $[\triangle u]_{1,j:i-1\to i+1}$, $[\triangle u]_{n_x,j:i+1\to i-1}$, $[\triangle u]_{i,1:j-1\to j+1}$,
    and $[\triangle u]_{i,n_y:j+1\to j-1}$ according to (15)
define $(i,j) \in \mathcal{I}$ to be $i = 2,\ldots,n_x - 1$, $j = 2,\ldots,n_y - 1$
define $u^+_{i,j}$ , $u_{i,j}$ and $u^-_{i,j}$ to represent $u^{\ell+1}_{i,j}$, $u^\ell_{i,j}$ and $u^{\ell-1}_{i,j}$, resp.
function WAVE$(u^+, u, u^-, a, b, c)$:
    UPDATE ALL INNER POINTS:
    $u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j}, \quad (i,j) \in \mathcal{I}$
    UPDATE BOUNDARY POINTS:
    $i = 1;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\to i+1},\ j = 2,\ldots,n_y - 1$
    $i = n_x;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\to i-1},\ j = 2,\ldots,n_y - 1$
    $j = 1;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:j-1\to j+1},\ i = 2,\ldots,n_x - 1$
    $j = n_y;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:j-1\to j+1},\ i = 2,\ldots,n_x - 1$
    UPDATE CORNER POINTS ON THE BOUNDARY:
    $i = 1,\ j = 1;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\to i+1,j-1\to j+1}$
    $i = n_x,\ j = 1;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\to i-1,j-1\to j+1}$
    $i = 1,\ j = n_y;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\to i+1,j+1\to j-1}$
    $i = n_x,\ j = n_y;\ u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\to i-1,j+1\to j-1}$
```

Using the WAVE function from Algorithm 3, we can devise a compact description of all the computational tasks for the discrete two-dimensional wave equation with homogenous Neumann conditions. The steps are listed in Algorithm 3.

Algorithm 3

Complete scheme for the two-dimensional wave equation with $\frac{\partial u}{\partial n} = 0$ on the boundary.

```
define quantities in Algorithm 2
define $(i,j) \in \bar{\mathcal{I}}$ to be $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$
set $u_{i,j} = 0, \quad (i,j) \in \bar{\mathcal{I}}$
SET THE INITIAL CONDITIONS:
$u_{i,j} = I(x_i, y_j), \quad (i,j) \in \mathcal{I}$
DEFINE THE VALUE OF THE ARTIFICIAL QUANTITY $u^-_{i,j}$:
$WAVE(u^-, u, u^-, 0.5, 0, 0.5)$
$t = 0$
while time $t \leq t_{\mathrm{stop}}$
    $t \leftarrow t + \Delta t$
    UPDATE ALL POINTS:
    $WAVE(u^+, u, u^-, 1, 1, 1)$
    INITIALIZE FOR NEXT STEP:
    $u^-_{i,j} = u_{i,j}, \quad u_{i,j} = u^+_{i,j}, \quad (i,j) \in \mathcal{I}$
```

At this point a newcomer to partial differential equations and finite difference schmes might be overwhelmed and confused by all the indices and book-keeping in these algorithms. Surely, the amount of details is significant, and hands-on work with the formulas, either with pencil and paper or in a computer code, is necessary to become familiar with the ideas and the notation. We should, however, point out that the underlying ideas are quite simple; the whole business here is mostly a matter of becoming familiar with the index notation. A very good exercise for repeating the material and working towards a better understanding is to devise an algorithm for the case where $u = 0$ at the whole boundary, except the line $x = x_{\max}$. Readers who are experienced with finite difference schemes will do this in a minute, while newcomers need to carefully go through the previous pages and find the right elements that are needed in the algorithm for this slightly perturbed problem.

2.3 Implementation of the algorithm

The basic operations in algorithm 3 consist of running through all grid points,

$$i = 1, \ldots, n_x, \quad j = 1, \ldots, n_y$$

and computing some formula at each grid point. The formula typically involves $u^+_{i,j}$, $u_{i,j}$, and $u^-_{i,j}$ values. If we store the latter quantities in three two-dimensional arrays, where i and j become indices in the array, the basic operation consists of two nested loops and manipulation on the arrays inside the loop. This type of operation can be programmed in almost any computer language, since two-dimensional arrays are data structures found in most languages.

3 Implementation in Fortran 77

A rough implementation of the numerical problem described in the previous section has been realized in Fortran 77. It is not necessary to be familiar with the numerics in detail in order to play around with the code. We remark that the program applies a rough first-order approximation to the initial condition $\partial u/\partial t = 0$, the other numerical expressions are as outlined in the section 2.3.

The program consists of two files, `main.f` and `F77WAVE.fcp`. The `main.f` file contains

- a main program, where data structures are declared and some constants set, before the main routine `timeloop` is called,
- a function `h` implementing an expression for the $\lambda(x, y)$ function (here just 1.0 is returned),
- a function `bell` for defining the initial condition (a Gaussian bell),
- a function `setIC` for initializing arrays based on the problem's initial condition,
- a function `timeloop`, which runs the finite difference scheme for a given number of time steps,
- a function `dump` for dumping data to file.

The `timeloop` function calls another function `F77WAVE` for implementing the WAVE function defined in section 2.3. The WAVE function is just the core finite difference scheme for the wave equation, i.e., the "heart" of this simulation code. The `F77WAVE` function is written in Fotran 77, but we have used a C preprocessor macro to define an inline function `LaplaceU`, corresponding to $[\triangle u]_{i,j}$ in section 2.2:

```
#define LaplaceU(u,lambda,i,j,im1,ip1,jm1,jp1) \
  (dt*dt)/(dx*dx)*\
  ( 0.5*(lambda(ip1,j  )+lambda(i  ,j  ))*(u(ip1,j  )-u(i  ,j  )) \
   -0.5*(lambda(i  ,j  )+lambda(im1,j  ))*(u(i  ,j  )-u(im1,j  )))\
 +(dt*dt)/(dy*dy)*\
  ( 0.5*(lambda(i  ,jp1)+lambda(i  ,j  ))*(u(i  ,jp1)-u(i  ,j  )) \
   -0.5*(lambda(i  ,j  )+lambda(i  ,jm1))*(u(i  ,j  )-u(i  ,jm1)))
```

The equivalent mathematical expression is

$$\begin{aligned}[\triangle u]_{i,j} \quad \equiv \quad & \left(\frac{\Delta t}{\Delta x}\right)^2 (\lambda_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - \lambda_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})) + \\ & \left(\frac{\Delta t}{\Delta y}\right)^2 (\lambda_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - \lambda_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1}))\,.\end{aligned}$$

The symbols `ip1` ("i plus 1"), `im1` ("i minus 1"), and so on corresponds to $i + 1$, $i - 1$ etc.

A special script, `fcpp.py`, translates Fortran 77 code with C macros into plain Fortran 77. The resulting file is named `F77WAVE.f`. With the `LaplaceU` macro the code becomes much more readable, and it is easier to play around with modifications of the implementation of the finite difference scheme.

The script `make.sh` runs `fcpp.py` and compiles the application with profiling turned on (plain Unix `gprof`).

When working with high-performance computing issues related to this wave simulation code, we shall modify the `F77WAVE.fcp` file only. The core of this file is the WAVE function:

```
      SUBROUTINE F77WAVE(up, u, um, lambda, a, b, c, nx, ny,
     >                   dt, dx, dy)
      IMPLICIT LOGICAL (A-Z)
      INTEGER nx, ny
      REAL*8 up(nx,ny), u(nx,ny), um(nx,ny), lambda(nx,ny)
      REAL*8 a, b, c
      REAL*8 dt, dx, dy
      INTEGER i,j

      DO 20 j = 2, ny-1
         DO 10 i = 2, nx-1
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >           c*LaplaceU(u,lambda,i,j,i-1,i+1,j-1,j+1)
 10      CONTINUE
 20   CONTINUE

C     Boundary points:

      i=1
      DO 30 j = 2, ny-1
```

```
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >          c*LaplaceU(u,lambda,i,j,i+1,i+1,j-1,j+1)
 30      CONTINUE

         i=nx
         DO 40 j = 2, ny-1
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >          c*LaplaceU(u,lambda,i,j,i-1,i-1,j-1,j+1)
 40      CONTINUE

         j=1
         DO 50 i = 2, nx-1
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >          c*LaplaceU(u,lambda,i,j,i-1,i+1,j+1,j+1)
 50      CONTINUE

         j=ny
         DO 60 i = 2, nx-1
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >          c*LaplaceU(u,lambda,i,j,i-1,i+1,j-1,j-1)
 60      CONTINUE

C        Corners:
         i=1
         j=1
         up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >      c*LaplaceU(u,lambda,i,j,i+1,i+1,j+1,j+1)

         i=nx
         j=1
         up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >      c*LaplaceU(u,lambda,i,j,i-1,i-1,j+1,j+1)

         i=1
         j=ny
         up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >      c*LaplaceU(u,lambda,i,j,i+1,i+1,j-1,j-1)

         i=nx
         j=ny
         up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >      c*LaplaceU(u,lambda,i,j,i-1,i-1,j-1,j-1)
         RETURN
         END
```

In the implementation we have paid attention to constructing loops with the first index as the fastest index. This ensures that arrays are accessed in the way they are stored in Fortran. We have also treated the boundary points in separate loops. In some of the first exercises you are asked to rewrite the code in less efficient ways and measure the efficiency.

4 Exercises

CPU-Time Measurements

In most of the exercises below, the purpose is to perform some actions and measure the impact on the CPU time. The CPU time can be measure in different ways. The simplest is to use the program (or built-in shell command) `time`:

```
time app
```

The output from `time` when running the program `app` may look as follows:

```
real    0m4.314s
user    0m3.950s
sys     0m0.020s
```

Here, `real` is elapsed time (the time you have waited for the program to finish), `user` is user time (e.g., arithmetics in the code), and `sys` is system time (e.g., I/O in the code). The CPU time is the sum of the user time and the system time.

The preferred way to measure CPU time in these exercises is to run `prof` or `gprof`. With `gprof`, you will find the CPU time at the last line of the `cumulative seconds` column in the table of the CPU-time consumption of the various functions. This command picks out the relevant number:

```
gprof app | perl -ne 'if (/MAIN__$/)
                      { $cpu=(split /\s+/)[2]; print "CPU=$cpu\n"; }'
```

This command is available as the script `cpu.sh`. To display the table of CPU-time consumption of the various functions, you can type

```
gprof app | head -12
```

which displays the first 12 lines of the `gprof` output.

If you want to have a look at the solution produced by the Fortran 77 program, you can run the script `Verify/test1.sh`. The plotting requires that you have the X11 plotting program `plotmtv` installed.

Exercises

1. *Determine the appropriate problem size.* CPU-time measurements are not reliable unless they last for some (say at least 10) seconds. In the present code example, the grid size and the number of time steps govern the CPU time. We have fixed the number of time steps at 40, so the parameter to tune is the number of grid points in each spatial direction. This parameter is set in the top of the `main.f.orig` file:

   ```
         PARAMETER (n=501)
   ```

   Compile the code by

   ```
   make.sh
   ```

   and run it:

   ```
   time app
   ```

   Adjust the value of `n` in the `PARAMETER` statement until you have a CPU time between 10 and 30 seconds.

2. *Computations with and without I/O.* Measure the CPU time of the original implementation of the wave simulation code. Then "uncomment" the calls to `dump`, i.e., write the solution to file at every time step. Compile and measure the CPU time of a run. In the rest of the exercises, we shall *not* write results to file so make sure you insert the comment again before proceeding.

3. *Function calls inside loops.* The $\lambda(x, y)$ coefficient in the wave equation is in our implementation stored in an array `lambda`. Alternatively, we could call a function, here `h(x,y)`. Modify the `LaplaceU` macro to call `h` instead of using `lambda`. First try to replace `lambda` by `h(0,0)` (take a copy of the original `F77WAVE.fcp` file!):

   ```
   #define LaplaceU(u,lambda,i,j,im1,ip1,jm1,jp1) \
     (dt*dt)/(dx*dx)*\
     ( 0.5*(h(0,0) + h(0,0))*(u(ip1,j  )-u(i  ,j  )) \
      -0.5*(h(0,0) + h(0,0))*(u(i   ,j  )-u(im1,j  )))\
    +(dt*dt)/(dy*dy)*\
     ( 0.5*(h(0,0) + h(0,0))*(u(i   ,jp1)-u(i  ,j  )) \
      -0.5*(h(0,0) + h(0,0))*(u(i   ,j  )-u(i  ,jm1)))
   ```

   In addition you need to declare `h` in the `F77WAVE` function as `REAL*8`. Compare timings with the original code.

   Smart compilers will notice that `h(0,0)` is a constant inside the loop and evaluate `h(0,0)` just once outside the loop. Replace `lambda(i,j)` by `h((i-1)*delta,(j-1)*delta)`, where `delta` is the cell size, which must be initialized before the loops in the `F77WAVE` function:

```
REAL*8 delta
delta = 10.0/(n-1)
```

This time the `h` needs to be called inside the loop, unless the compiler is so smart that it detects that the function declaration of `h` is simple (just 1 is returned) and inlines that function. The timings will uncover how smart your compiler is.

4. *If-tests inside loops.* Instead of splitting the finite difference scheme into a loop over the internal grid points, `i,j=2,...,n-1`, we can merge all loops and insert appropriate if-tests. This is a typical coding habit of novice numerical programmers:

```
      DO 20 j = 1, ny
         DO 10 i = 1, nx
            if (i .ge. 2 .and. i .le. nx-1 .and. j .ge. 2
     >          .and. j .le. ny-1) then
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     >          c*LaplaceU(u,lambda,i,j,i-1,i+1,j-1,j+1)
            else if ...
C           treat boundary points...
 10      CONTINUE
 20   CONTINUE
```

   Implement just one such if-test, i.e., the one given above with `else if` replaced by `end if`. Is your compiler sufficiently smart to replace the loop by new limits? Or do you get an increase in the CPU time?

5 Projects

1. Extend the code to simulation of acoustic waves in a heterogenous three-dimensional box. The governing PDE is the same,

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (\lambda \nabla u). \quad (16)$$

   The boundary conditions are also the same: $\partial u/\partial n = 0$.

2. Repeat the tests from the exercises with the two-dimensional version of the program. Comment upon differences in the results.

3. Reimplement the program in C or C++. Repeat the efficiency tests.