Chapter 10
Partial differential equations

Abstract Partial differential equations play an important role in our modelling of physical processes, from diffusion of heat to our understanding of Tsunamis. In this chapter we present some of the basic methods using finite difference methods.

10.1 Introduction

In the Natural Sciences we often encounter problems with many variables constrained by boundary conditions and initial values. Many of these problems can be modelled as partial differential equations. One case which arises in many situations is the so-called wave equation whose one-dimensional form reads

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial^2 u}{\partial t^2}, \tag{10.1}$$

where A is a constant. The solution u depends on both spatial and temporal variables, viz. $u = u(x,t)$. In two dimension we have $u = u(x,y,t)$. We will, unless otherwise stated, simply use u in our discussion below. Familiar situations which this equation can model are waves on a string, pressure waves, waves on the surface of a fjord or a lake, electromagnetic waves and sound waves to mention a few. For e.g., electromagnetic waves we have the constant $A = c^2$, with c the speed of light. It is rather straightforward to extend this equation to two or three dimension. In two dimensions we have

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = A\frac{\partial^2 u}{\partial t^2},$$

In Chapter 12 we will see another case of a partial differential equation widely used in the Natural Sciences, namely the diffusion equation whose one-dimensional version we derived from a Markovian random walk. It reads

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial u}{\partial t}, \tag{10.2}$$

and A is in this case called the diffusion constant. It can be used to model a wide selection of diffusion processes, from molecules to the diffusion of heat in a given material.

Another familiar equation from electrostatics is Laplace's equation, which looks similar to the wave equation in Eq. (10.1) except that we have set $A = 0$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{10.3}$$

or if we have a finite electric charge represented by a charge density $\rho(\mathbf{x})$ we have the familiar Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(\mathbf{x}). \tag{10.4}$$

Other famous partial differential equations are the Helmholtz (or eigenvalue) equation, here specialized to two dimensions only

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = \lambda u, \tag{10.5}$$

the linear transport equation (in $2+1$ dimensions) familiar from Brownian motion as well

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0, \tag{10.6}$$

and Schrödinger's equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + f(x,y)u = \imath\frac{\partial u}{\partial t}.$$

Important systems of linear partial differential equations are the famous Maxwell equations

$$\frac{\partial \mathbf{E}}{\partial t} = \text{curl}\mathbf{B}; \qquad -\text{curl}\mathbf{E} = \mathbf{B}; \qquad \text{div}\mathbf{E} = \text{div}\mathbf{B} = 0.$$

Similarly, famous systems of non-linear partial differential equations are for example Euler's equations for incompressible, inviscid flow

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} = -Dp; \qquad \text{div}\mathbf{u} = 0,$$

with p being the pressure and

$$\nabla = \frac{\partial}{\partial x}e_x + \frac{\partial}{\partial y}e_y,$$

in the two dimensions. The unit vectors are e_x and e_y. Another example is the set of Navier-Stokes equations for incompressible, viscous flow

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} - \Delta\mathbf{u} = -Dp; \qquad \text{div}\mathbf{u} = 0.$$

Ref. [52] contains a long list of interesting partial differential equations.

In this chapter we focus on so-called finite difference schemes and explicit and implicit methods. The more advanced topic of finite element methods are not treated in this text. For texts with several numerical examples, see for example Refs. [48, 53].

As in the previous chapters we will focus mainly on widely used algorithms for solutions of partial differential equations. A text like Evans' [52] is highly recommended if one wishes to study the mathematical foundations for partial differential equations, in particular how to determine the uniqueness and existence of a solution. We assume that our problems are well-posed, strictly meaning that the problem has a solution, this solution is unique and the solution depends continuously on the data given by the problem. While Evans' text provides a rigorous mathematical exposition, the texts of Langtangen, Ramdas-Mohan, Winther and Tveito and Evans *et al.* contain a more practical algorithmic approach see Refs. [48,50,53,54].

A general partial differential equation with two given dimensions reads

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial^2 u}{\partial x \partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} = F(x,y,u,\frac{\partial u}{\partial x},\frac{\partial u}{\partial y}),$$

and if we set

$$B = C = 0,$$

we recover the $1+1$-dimensional diffusion equation which is an example of a so-called parabolic partial differential equation. With

$$B = 0, \qquad AC < 0$$

we get the $2+1$-dim wave equation which is an example of a so-called elliptic PDE, where more generally we have $B^2 > AC$. For $B^2 < AC$ we obtain a so-called hyperbolic PDE, with the Laplace equation in Eq. (10.3) as one of the classical examples. These equations can all be easily extended to non-linear partial differential equations and $3+1$ dimensional cases.

The aim of this chapter is to present some of the more familiar difference methods and their possible implementations.

10.2 Diffusion equation

The diffusion equation describes in typical applications the evolution in time of the density u of a quantity like the particle density, energy density, temperature gradient, chemical concentrations etc.

The basis is the assumption that the flux density ρ obeys the Gauss-Green theorem

$$\int_V \mathrm{div}\rho dx = \int_{\partial V} \rho \mathbf{n} dS,$$

where n is the unit outer normal field and V is a smooth region with the space where we seek a solution. The Gauss-Green theorem leads to

$$\mathrm{div}\rho = 0.$$

Assuming that the flux is proportional to the gradient ∇u but pointing in the opposite direction since the flow is from regions of high concetration to lower concentrations, we obtain

$$\rho = -D\nabla u,$$

resulting in

$$\mathrm{div}\nabla u = D\Delta u = 0,$$

which is Laplace's equation, an equation whose one-dimensional version we met in chapter 6. The constant D can be coupled with various physical constants, such as the diffusion constant or the specific heat and thermal conductivity discussed below. We will discuss the solution of the Laplace equation later in this chapter.

If we let u denote the concetration of a particle species, this results in Fick's law of diffusion, see Ref. [55]. If it denotes the temperature gradient, we have Fourier'slaw of heat conduction and if it refers to the electrostatic potential we have Ohm's law of electrical conduction.

Coupling the rate of change (temporal dependence) of u with the flux density we have

$$\frac{\partial u}{\partial t} = -\mathrm{div}\rho,$$

which results in

$$\frac{\partial u}{\partial t} = D\mathrm{div}\nabla u = D\Delta u,$$

the diffusion equation, or heat equation.

If we specialize to the heat equation, we assume that the diffusion of heat through some material is proportional with the temperature gradient $T(\mathbf{x},t)$ and using conservation of energy we arrive at the diffusion equation

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

where C is the specific heat and ρ the density of the material. Here we let the density be represented by a constant, but there is no problem introducing an explicit spatial dependence, viz.,

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}.$$

Setting all constants equal to the diffusion constant D, i.e.,

$$D = \frac{C\rho}{\kappa},$$

we arrive at

$$\nabla^2 T(\mathbf{x},t) = D\frac{\partial T(\mathbf{x},t)}{\partial t}.$$

Specializing to the $1+1$-dimensional case we have

$$\frac{\partial^2 T(x,t)}{\partial x^2} = D\frac{\partial T(x,t)}{\partial t}.$$

We note that the dimension of D is time/length^2. Introducing the dimensionless variables $\alpha\hat{x} = x$ we get

$$\frac{\partial^2 T(x,t)}{\alpha^2\partial \hat{x}^2} = D\frac{\partial T(x,t)}{\partial t},$$

and since α is just a constant we could define $\alpha^2 D = 1$ or use the last expression to define a dimensionless time-variable $\hat{t}$. This yields a simplified diffusion equation

$$\frac{\partial^2 T(\hat{x},\hat{t})}{\partial \hat{x}^2} = \frac{\partial T(\hat{x},\hat{t})}{\partial \hat{t}}.$$

It is now a partial differential equation in terms of dimensionless variables. In the discussion below, we will however, for the sake of notational simplicity replace $\hat{x} \to x$ and $\hat{t} \to t$. Moreover, the solution to the $1+1$-dimensional partial differential equation is replaced by $T(\hat{x},\hat{t}) \to u(x,t)$.

10.2.1 Explicit Scheme

In one dimension we have the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = g(x) \qquad 0 < x < L$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \qquad t \geq 0,$$

and

$$u(L,t) = b(t) \qquad t \geq 0,$$

where $a(t)$ and $b(t)$ are two functions which depend on time only, while $g(x)$ depends only on the position x. Our next step is to find a numerical algorithm for solving this equation. Here we recur to our familiar equal-step methods discussed in Chapter 3 and introduce different step lengths for the space-variable x and time t through the step length for x

$$\Delta x = \frac{1}{n+1}$$

and the time step length Δt. The position after i steps and time at time-step j are now given by

$$\begin{cases} t_j = j\Delta t & j \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n+1 \end{cases}$$

If we then use standard approximations for the derivatives we obtain

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}$$

with a local approximation error $O(\Delta t)$ and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2},$$

with a local approximation error $O(\Delta x^2)$. Our approximation is to higher order in coordinate space. This can be justified since in most cases it is the spatial dependence which causes numerical problems. These equations can be further simplified as

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t},$$

and

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

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}.$$

Defining $\alpha = \Delta t/\Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{10.7}$$

Since all the discretized initial values

$$u_{i,0} = g(x_i),$$

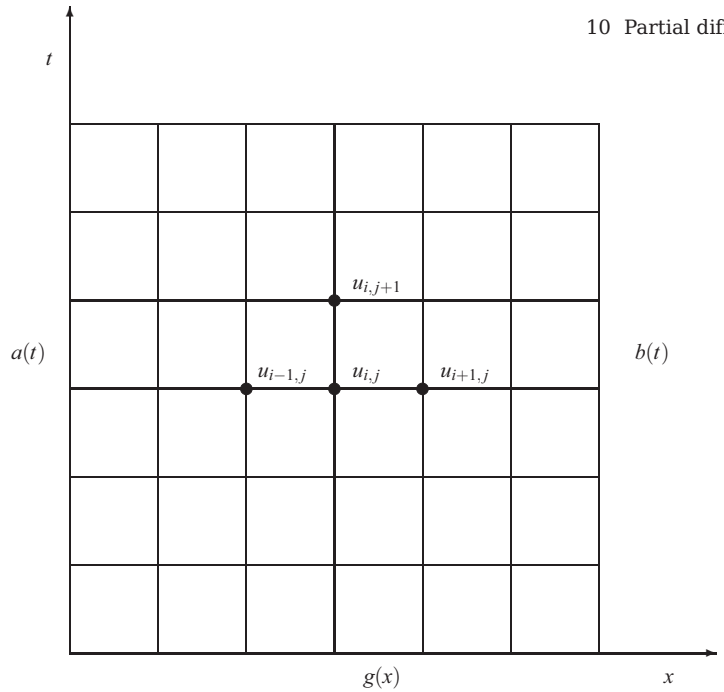

Fig. 10.1 Discretization of the integration area used in the solution of the $1+1$-dimensional diffusion equation. This discretization is often called calculational molecule.

are known, then after one time-step the only unknown quantity is $u_{i,1}$ which is given by

$$u_{i,1} = \alpha u_{i-1,0} + (1-2\alpha)u_{i,0} + \alpha u_{i+1,0} = \alpha g(x_{i-1}) + (1-2\alpha)g(x_i) + \alpha g(x_{i+1}).$$

We can then obtain $u_{i,2}$ using the previously calculated values $u_{i,1}$ and the boundary conditions $a(t)$ and $b(t)$. This algorithm results in a so-called explicit scheme, since the next functions $u_{i,j+1}$ are explicitely given by Eq. (10.7). The procedure is depicted in Fig. 10.1.

We specialize to the case $a(t) = b(t) = 0$ which results in $u_{0,j} = u_{n+1,j} = 0$. We can then reformulate our partial differential equation through the vector V_j at the time $t_j = j\Delta t$

$$V_j = \begin{pmatrix} u_{1,j} \\ u_{2,j} \\ \cdots \\ u_{n,j} \end{pmatrix}.$$

This results in a matrix-vector multiplication

$$V_{j+1} = \hat{A} V_j$$

with the matrix $\hat{A}$ given by

$$\hat{A} = \begin{pmatrix} 1-2\alpha & \alpha & 0 & 0\ldots \\ \alpha & 1-2\alpha & \alpha & 0\ldots \\ \cdots & \cdots & \cdots & \cdots \\ 0\ldots & 0\ldots & \alpha & 1-2\alpha \end{pmatrix}$$

which means we can rewrite the original partial differential equation as a set of matrix-vector multiplications

$$V_{j+1} = \hat{A} V_j = \cdots = \hat{A}^{j+1} V_0,$$

where V_0 is the initial vector at time $t = 0$ defined by the initial value $g(x)$. In the numerical implementation one should avoid to treat this problem as a matrix vector multiplication since the matrix is triangular and at most three elements in each row are different from zero.

It is rather easy to implement this matrix-vector multiplication as seen in the following piece of code

```
// First we set initialise the new and old vectors
// Here we have chosen the boundary conditions to be zero.
// n+1 is the number of mesh points in x
   u[0] = unew[0] = u[n] = unew = 0.0;
   for (int i = 1; i < n; i++) {
     x = i*step;
     // initial condition
     u[i] = func(x);
     // intitialise the new vector
     unew[i] = 0;
   }
  // Time iteration
  for (int t = 1; t <= tsteps; t++) {
    for (int i = 1; i < n; i++) {
      // Discretized diff eq
      unew[i] = alpha * u[i-1] + (1 - 2*alpha) * u[i] + alpha * u[i+1];
    }
  // note that the boundaries are not changed.
```

However, although the explicit scheme is easy to implement, it has a very weak stability condition, given by

$$\Delta t/\Delta x^2 \leq 1/2.$$

This means that if $\Delta x^2 = 0.01$, then $\Delta t = 5 \times 10^{-5}$. This has obviously bad consequences if our time interval is large. In order to derive this relation we need some results from studies of iterative schemes. If we require that our solution approaches a definite value after a certain amount of time steps we need to require that the so-called spectral radius $\rho(\hat{A})$ of our matrix $\hat{A}$ satisfies the condition

$$\rho(\hat{A}) < 1, \tag{10.8}$$

see for example chapter 10 of Ref. [28] or chapter 4 of [23] for proofs. The spectral radius is defined as

$$\rho(\hat{A}) = \max\left\{|\lambda| : \det(\hat{A} - \lambda\hat{I}) = 0\right\},$$

which is interpreted as the smallest number such that a circle with radius centered at zero in the complex plane contains all eigenvalues of $\hat{A}$. If the matrix is positive definite, the condition in Eq. (10.8) is always satisfied.

We can obtain closed-form expressions for the eigenvalues of $\hat{A}$. To achieve this it is convenient to rewrite the matrix as

$$\hat{A} = \hat{I} - \alpha\hat{B},$$

with

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\ldots \\ -1 & 2 & -1 & 0\ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0\ldots & 0\ldots & -1 & 2 \end{pmatrix}.$$

The eigenvalues of $\hat{A}$ are $\lambda_i = 1 - \alpha\mu_i$, with μ_i being the eigenvalues of $\hat{B}$. To find μ_i we note that the matrix elements of $\hat{B}$ are

$$b_{ij} = 2\delta_{ij} - \delta_{i+1j} - \delta_{i-1j},$$

meaning that we have the following set of eigenequations for component i

$$(\hat{B}\hat{x})_i = \mu_i x_i,$$

resulting in

$$(\hat{B}\hat{x})_i = \sum_{j=1}^{n} \left(2\delta_{ij} - \delta_{i+1j} - \delta_{i-1j}\right) x_j = 2x_i - x_{i+1} - x_{i-1} = \mu_i x_i.$$

If we assume that x can be expanded in a basis of $x = (sin(\theta), sin(2\theta), \ldots, sin(n\theta))$ with $\theta = l\pi/n+1$, where we have the endpoints given by $x_0 = 0$ and $x_{n+1} = 0$, we can rewrite the last equation as

$$2sin(i\theta) - sin((i+1)\theta) - sin((i-1)\theta) = \mu_i sin(i\theta),$$

or

$$2\,(1 - cos(\theta))sin(i\theta) = \mu_i sin(i\theta),$$

which is nothing but

$$2\,(1 - cos(\theta))x_i = \mu_i x_i,$$

with eigenvalues $\mu_i = 2 - 2cos(\theta)$.

Our requirement in Eq. (10.8) results in

$$-1 < 1 - \alpha 2\,(1 - cos(\theta)) < 1,$$

which is satisfied only if $\alpha < (1 - cos(\theta))^{-1}$ resulting in $\alpha \leq 1/2$ or $\Delta t/\Delta x^2 \leq 1/2$.

10.2.2 Implicit Scheme

In deriving the equations for the explicit scheme we started with the so-called forward formula for the first derivative, i.e., we used the discrete approximation

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}.$$

However, there is nothing which hinders us from using the backward formula

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t},$$

still with a truncation error which goes like $O(\Delta t)$. We could also have used a midpoint approximation for the first derivative, resulting in

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t},$$

with a truncation error $O(\Delta t^2)$. Here we will stick to the backward formula and come back to the latter below. For the second derivative we use however

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2},$$

and define again $\alpha = \Delta t/\Delta x^2$. We obtain now

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} - \alpha u_{i+1,j}.$$

Here $u_{i,j-1}$ is the only unknown quantity. Defining the matrix $\hat{A}$

$$\hat{A} = \begin{pmatrix} 1+2\alpha & -\alpha & 0 & 0\dots \\ -\alpha & 1+2\alpha & -\alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & -\alpha & 1+2\alpha \end{pmatrix},$$

we can reformulate again the problem as a matrix-vector multiplication

$$\hat{A}V_j = V_{j-1}$$

meaning that we can rewrite the problem as

$$V_j = \hat{A}^{-1}V_{j-1} = \hat{A}^{-1}\left(\hat{A}^{-1}V_{j-2}\right) = \cdots = \hat{A}^{-j}V_0.$$

This is an implicit scheme since it relies on determining the vector $u_{i,j-1}$ instead of $u_{i,j+1}$. If α does not depend on time t, we need to invert a matrix only once. Alternatively we can solve this system of equations using our methods from linear algebra discussed in chapter 6. These are however very cumbersome ways of solving since they involve $\sim O(N^3)$ operations for a $N \times N$ matrix. It is much faster to solve these linear equations using methods for tridiagonal matrices, since these involve only $\sim O(N)$ operations. The function `tridag` of Ref. [36] is suitbale for these tasks.

The implicit scheme is always stable since the spectral radius satisfies $\rho(\hat{A}) < 1$. We could have inferred this by noting that the matrix is positive definite, viz. all eigenvalues are larger than zero. We see this from the fact that $\hat{A} = \hat{I} + \alpha\hat{B}$ has eigenvalues $\lambda_i = 1 + \alpha(2 - 2cos(\theta))$ which satisfy $\lambda_i > 1$. Since it is the inverse which stands to the right of our iterative equation, we have $\rho(\hat{A}^{-1}) < 1$ and the method is stable for all combinations of Δt and Δx. The calculational molecule for the implicit scheme is shown in Fig. 10.2.

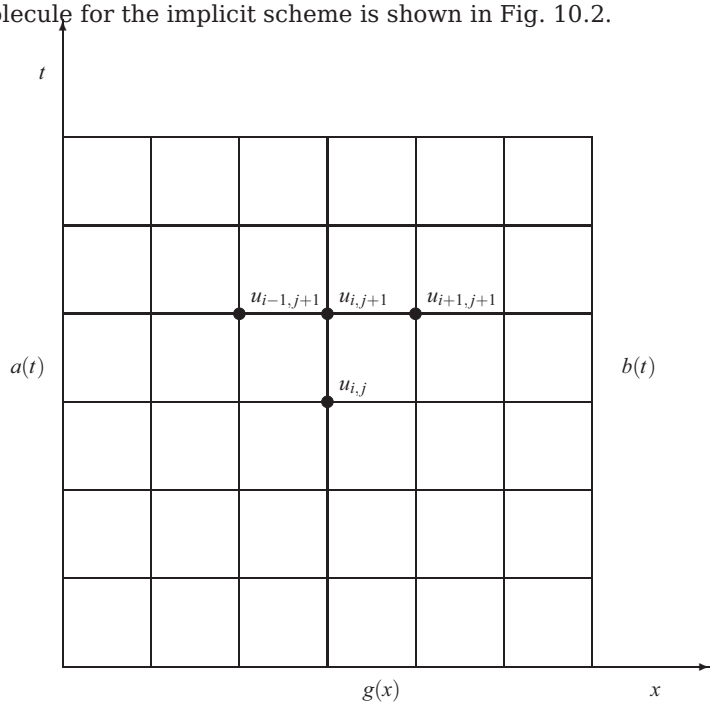

Fig. 10.2 Calculational molecule for the implicit scheme.

10.2.2.1 Program Example for Implicit Equation

We show here parts of a simple example of how to solve the one-dimensional diffusion equation using the implicit scheme discussed above. The program uses the function to solve linear equations with a tridiagonal matrix discussed in chapter 6.

```
// parts of the function for backward Euler
void backward_euler(int xsteps, int tsteps, double delta_x, double alpha)
{
  double *v, *r, a, b, c;

  v = new double[xsteps+1]; // This is u
  r = new double[xsteps+1]; // Right side of matrix equation Av=r

  // Initialize vectors
  for (int i = 0; i < xsteps; i++) {
    r[i] = v[i] = func(delta_x*i);
  }
  r[xsteps] = v[xsteps] = 0;
  // Matrix A, only constants
  a = c = - alpha;
  b = 1 + 2*alpha;
  // Time iteration
  for (int t = 1; t <= tsteps; t++) {
    // here we solve the tridiagonal linear set of equations
    tridag(a, b, c, r, v, x_steps+1);
    // boundary conditions
    v[0] = 0;
    v[xsteps] = 0;
    for (int i = 0; i <= x_steps; i++) {
  r[i] = v[i];
    }
  }
  ...
}
// Function used to solve systems of equations for tridiagonal matrices
void tridag(double a, double b, double c, double *r, double *u, int n)
{
  double bet, *gam;
  gam = new double[n];
  bet = b;
  // forward substitution
  u[0]=r[0]/bet;
  for (int j=1;j<n;j++) {
    gam[j] = c/bet;
    bet = b - a*gam[j];
    if (bet == 0.0) {cout << "Error 2 in tridag" << endl;}
    u[j] = (r[j] - a*u[j-1])/bet;
  }
  // backward substitution
  for (int j=n-2; j>=0; j--) {u[j] -= gam[j+1]*u[j+1];}
  delete [] gam;
}
```

10.2.3 Crank-Nicolson scheme

It is possible to combine the implicit and explicit methods in a slightly more general approach. Introducing a parameter θ (the so-called θ-rule) we can set up an equation

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j}-2u_{i,j}+u_{i+1,j}\right)+\frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1}-2u_{i,j-1}+u_{i-1,j-1}\right)=\frac{1}{\Delta t}\left(u_{i,j}-u_{i,j-1}\right), \tag{10.9}$$

which for $\theta = 0$ yields the forward formula for the first derivative and the explicit scheme, while $\theta = 1$ yields the backward formula and the implicit scheme. These two schemes are called the backward and forward Euler schemes, respectively. For $\theta = 1/2$ we obtain a new scheme after its inventors, Crank and Nicolson. This scheme yields a truncation in time which goes like $O(\Delta t^2)$ and it is stable for all possible combinations of Δt and Δx.

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite the latter equation as

$$-\alpha u_{i-1,j}+(2+2\alpha)u_{i,j}-\alpha u_{i+1,j}=\alpha u_{i-1,j-1}+(2-2\alpha)u_{i,j-1}+\alpha u_{i+1,j-1},$$

or in matrix-vector form as

$$\left(2\hat{I}+\alpha\hat{B}\right)V_j=\left(2\hat{I}-\alpha\hat{B}\right)V_{j-1},$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

$$\hat{B}=\begin{pmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & & 2 \end{pmatrix}$$

We can rewrite the Crank-Nicolson scheme as follows

$$V_j=\left(2\hat{I}+\alpha\hat{B}\right)^{-1}\left(2\hat{I}-\alpha\hat{B}\right)V_{j-1}.$$

We have already obtained the eigenvalues for the two matrices $\left(2\hat{I}+\alpha\hat{B}\right)$ and $\left(2\hat{I}-\alpha\hat{B}\right)$. This means that the spectral function has to satisfy

$$\rho(\left(2\hat{I}+\alpha\hat{B}\right)^{-1}\left(2\hat{I}-\alpha\hat{B}\right))<1,$$

meaning that

$$\left|((2+\alpha\mu_i)^{-1}(2-\alpha\mu_i)\right|<1,$$

and since $\mu_i = 2-2cos(\theta)$ we have $0<\mu_i<4$. A little algebra shows that the algorithm is stable for all possible values of Δt and Δx.

The calculational molecule for the Crank-Nicolson scheme is shown in Fig. 10.3.

10.2.3.1 Parts of Code for the Crank-Nicolson Scheme

We can code in an efficient way the Crank-Nicolson algortihm by first multplying the matrix

$$\tilde{V}_{j-1}=\left(2\hat{I}-\alpha\hat{B}\right)V_{j-1},$$

with our previous vector V_{j-1} using the matrix-vector multiplication algorithm for a tridiagonal matrix, as done in the forward-Euler scheme. Thereafter we can solve the equation

$$\left(2\hat{I}+\alpha\hat{B}\right)V_j=\tilde{V}_{j-1},$$

using our method for systems of linear equations with a tridiagonal matrix, as done for the backward Euler scheme.

We illustrate this in the following part of our program.

```
void crank_nicolson(int xsteps, int tsteps, double delta_x, double alpha)
{
  double *v, a, b, c, *r;

  v = new double[xsteps+1]; // This is u
  r = new double[xsteps+1]; // Right side of matrix equation Av=r
  ....
  // setting up the matrix
  a = c = - alpha;
  b = 2 + 2*alpha;

  // Time iteration
  for (int t = 1; t <= tsteps; t++) {

    // Calculate r for use in tridag, right hand side of the Crank Nicolson method
    for (int i = 1; i < xsteps; i++) {
  r[i] = alpha*v[i-1] + (2 - 2*alpha)*v[i] + alpha*v[i+1];
    }
    r[0] = 0;
    r[xsteps] = 0;
    // Then solve the tridiagonal matrix
    tridag(a, b, c, r, v, xsteps+1);
    v[0] = 0;
    v[xsteps] = 0;
    ....
}
```

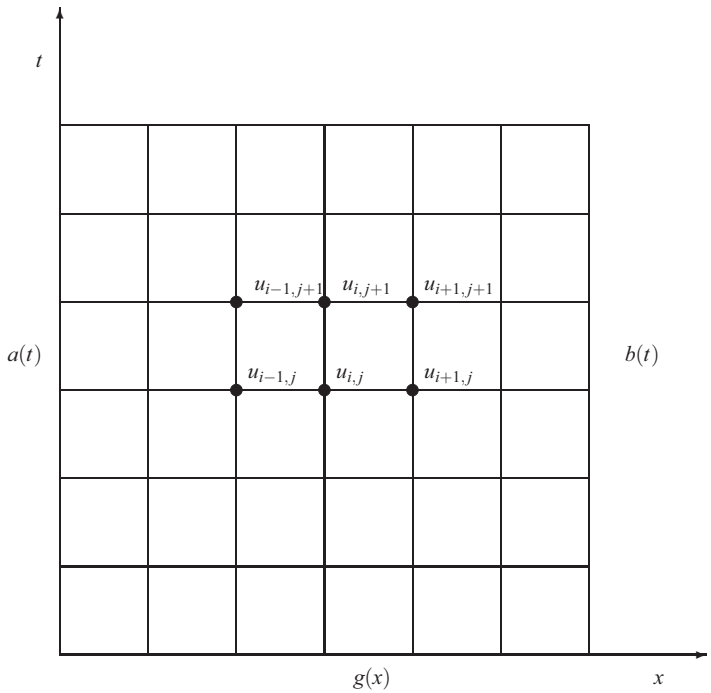

Fig. 10.3 Calculational molecule for the Crank-Nicolson scheme.

10.2.4 Numerical Truncation

We start with the forward Euler scheme and Taylor expand $u(x,t+\Delta t)$, $u(x+\Delta x,t)$ and $u(x-\Delta x,t)$

$$\begin{aligned} u(x+\Delta x,t) &= u(x,t)+\tfrac{\partial u(x,t)}{\partial x}\Delta x+\tfrac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathscr{O}(\Delta x^3), \\ u(x-\Delta x,t) &= u(x,t)-\tfrac{\partial u(x,t)}{\partial x}\Delta x+\tfrac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathscr{O}(\Delta x^3), \\ u(x,t+\Delta t) &= u(x,t)+\tfrac{\partial u(x,t)}{\partial t}\Delta t+\mathscr{O}(\Delta t^2). \end{aligned} \tag{10.10}$$

With these Taylor expansions the approximations for the derivatives takes the form

$$\begin{aligned} \left[\tfrac{\partial u(x,t)}{\partial t}\right]_{\text{approx}} &= \tfrac{\partial u(x,t)}{\partial t}+\mathscr{O}(\Delta t), \\ \left[\tfrac{\partial^2 u(x,t)}{\partial x^2}\right]_{\text{approx}} &= \tfrac{\partial^2 u(x,t)}{\partial x^2}+\mathscr{O}(\Delta x^2). \end{aligned} \tag{10.11}$$

It is easy to convince oneself that the backward Euler method must have the same truncation errors as the forward Euler scheme.

For the Crank-Nicolson scheme we also need to Taylor expand $u(x+\Delta x,t+\Delta t)$ and $u(x-\Delta x,t+\Delta t)$ around $t'=t+\Delta t/2$.

$$\begin{aligned} u(x+\Delta x,t+\Delta t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial x}\Delta x+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}+ \\ &\quad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3) \\ u(x-\Delta x,t+\Delta t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial x}\Delta x+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}- \\ &\quad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3) \\ u(x+\Delta x,t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial x}\Delta x-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}- \\ &\quad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3) \\ u(x-\Delta x,t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial x}\Delta x-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}+ \\ &\quad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3) \\ u(x,t+\Delta t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2+\mathscr{O}(\Delta t^3) \\ u(x,t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2+\mathscr{O}(\Delta t^3) \end{aligned}$$

We now insert these expansions in the approximations for the derivatives to find

$$\begin{aligned} \left[\tfrac{\partial u(x,t')}{\partial t}\right]_{\text{approx}} &= \tfrac{\partial u(x,t')}{\partial t}+\mathscr{O}(\Delta t^2), \\ \left[\tfrac{\partial^2 u(x,t')}{\partial x^2}\right]_{\text{approx}} &= \tfrac{\partial^2 u(x,t')}{\partial x^2}+\mathscr{O}(\Delta x^2). \end{aligned} \tag{10.12}$$

The following table summarizes the three methods.

Scheme:	*Truncation Error:*	*Stability requirements:*
Crank-Nicolson	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t^2)$	Stable for all Δt and Δx.
Backward Euler	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t)$	Stable for all Δt and Δx.
Forward Euler	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t)$	$\Delta t \leq \frac{1}{2}\Delta x^2$

Table 10.1 Comparison of the different schemes.

10.2.5 Solution for the One-dimensional Diffusion Equation

It cannot be repeated enough, it is always useful to find cases where one can compare the numerics and the developed algorithms and codes with closed-form solutions. The above case is also particularly simple. We have the following partial differential equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

with initial conditions

$$u(x,0) = g(x) \qquad 0 < x < L.$$

The boundary conditions are

$$u(0,t) = 0 \qquad t \geq 0, \qquad\qquad u(L,t) = 0 \qquad t \geq 0,$$

We assume that we have solutions of the form (separation of variable)

$$u(x,t) = F(x)G(t).$$

which inserted in the partial differential equation results in

$$\frac{F''}{F} = \frac{G'}{G},$$

where the derivative is with respect to x on the left hand side and with respect to t on right hand side. This equation should hold for all x and t. We must require the rhs and lhs to be equal to a constant. We call this constant $-\lambda^2$. This gives us the two differential equations,

$$F'' + \lambda^2 F = 0; \qquad\qquad G' = -\lambda^2 G,$$

with general solutions

$$F(x) = A\sin(\lambda x) + B\cos(\lambda x); \qquad\qquad G(t) = Ce^{-\lambda^2 t}.$$

To satisfy the boundary conditions we require $B = 0$ and $\lambda = n\pi/L$. One solution is therefore found to be

$$u(x,t) = A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}.$$

But there are infinitely many possible n values (infinite number of solutions). Moreover, the diffusion equation is linear and because of this we know that a superposition of solutions will also be a solution of the equation. We may therefore write

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}.$$

The coefficient A_n is in turn determined from the initial condition. We require

$$u(x,0) = g(x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L).$$

The coefficient A_n is the Fourier coefficients for the function $g(x)$. Because of this, A_n is given by (from the theory on Fourier series)

$$A_n = \frac{2}{L}\int_0^L g(x)\sin(n\pi x/L)\mathrm{d}x.$$

Different $g(x)$ functions will obviously result in different results for A_n. A good discussion on Fourier series and their links with partial differential equations can be found in Ref. [50].

10.3 Laplace's and Poisson's Equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{10.13}$$

with possible boundary conditions $u(x,y) = g(x,y)$ on the border $\delta\Omega$. There is no time-dependence. We seek a solution in the region Ω and we choose a quadratic mesh with equally many steps in both directions. We could choose the grid to be rectangular or following polar coordinates r, θ as well. Here we choose equal steps lengths in the x and the y directions. We set

$$h = \Delta x = \Delta y = \frac{L}{n+1},$$

where L is the length of the sides and we have $n+1$ points in both directions.

The discretized version reads

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2},$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2},$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2},$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2},$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{10.14}$$

This is our final numerical scheme for solving Laplace's equation. Poisson's equation adds only a minor complication to the above equation since in this case we have

$$u_{xx} + u_{yy} = -\rho(x,y),$$

and we need only to add a discretized version of $\rho(\mathbf{x})$ resulting in

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \frac{h^2}{4}\rho_{i,j}. \tag{10.15}$$

The boundary condtions read

$$\begin{aligned} u_{i,0} &= g_{i,0} \qquad 0 \le i \le n+1, \\ u_{i,L} &= g_{i,0} \qquad 0 \le i \le n+1, \\ u_{0,j} &= g_{0,j} \qquad 0 \le j \le n+1, \end{aligned}$$

and

$$u_{L,j} = g_{L,j} \qquad 0 \le j \le n+1.$$

The calculational molecule for the Laplace operator of Eq. (10.14) is shown in Fig. 10.4.

With $n+1$ mesh points the equations for u result in a system of $(n+1)^2$ linear equations in the $(n+1)^2$ unknown $u_{i,j}$. One can show that there exist unique solutions for the Laplace and Poisson problems, see for example Ref. [50] for proofs. However, solving these equations using for example the LU decomposition techniques discussed in chapter 6 becomes inefficient since the matrices are sparse. The relaxation techniques discussed below are more efficient.

10.3.1 Jacobi Algorithm for solving Laplace's Equation

It is fairly straightforward to extend this equation to the three-dimensional case. Whether we solve Eq. (10.14) or Eq. (10.15), the solution strategy remains the same. We know the values of u at $i = 0$ or $i = n+1$ and at $j = 0$ or $j = n+1$ but we cannot start at one of the boundaries and work our way into and across the system since Eq. (10.14) requires the knowledge of u at all of the neighbouring points in order to calculate u at any given point.

The way we solve these equations is based on an iterative scheme based on the Jacobi method or the Gauss-Seidel method discussed in chapter 6.

Implementing Jacobi's method is rather simple. We start with an initial guess for $u_{i,j}^{(0)}$ where all values are known. To obtain a new solution we solve Eq. (10.14) or Eq. (10.15) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (10.14). This solution is in turn used to obtain a new and improved $u_{i,j}^{(2)}$. We continue this process till we obtain a result which satisfies some specific convergence criterion. Summarized, this algorithm reads

1. Make an initial guess for $u_{i,j}$ at all interior points (i,j) for all $i = 1:n$ and $j = 1:n$

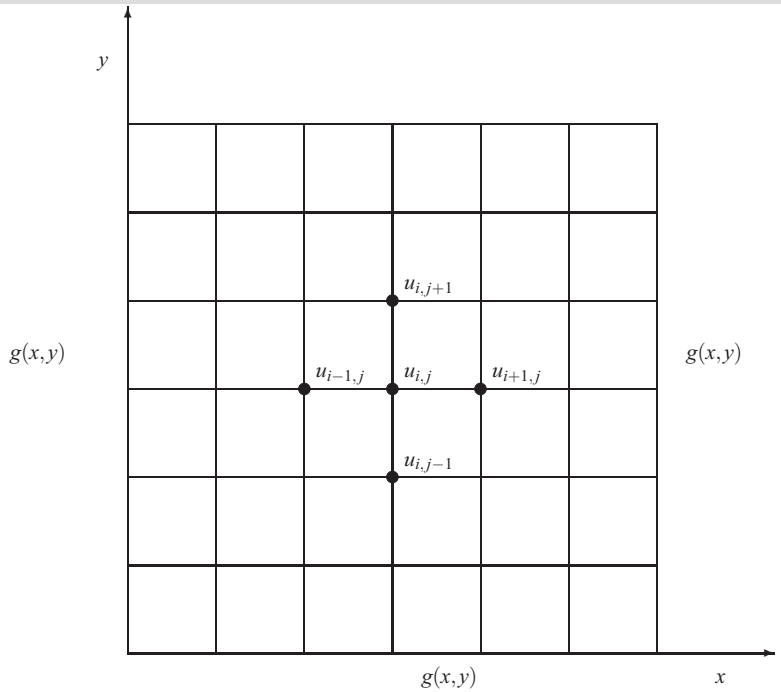

Fig. 10.4 Five-point calculational molecule for the Laplace operator of Eq. (10.14). The border $\delta\Omega$ defines the boundary condition $u(x,y) = g(x,y)$.

2. Use Eq. (10.14) to compute u^m at all interior points (i,j). The index m stands for iteration number m.
3. Stop if prescribed convergence threshold is reached, otherwise continue on next step.
4. Update the new value of u for the given iteration
5. Go to step 2

A simple example may help in visualizing this method. We consider a condensator with parallel plates separated at a distance L resulting in e.g., the voltage differences $u(x,0) = 100sin(2\pi x/L)$ and $u(x,1) = -100sin(2\pi x/L)$. These are our boundary conditions and we ask what is the voltage u between the plates? To solve this problem numerically we provide below a C++ program which solves iteratively Eq. (10.14). Only the part which computes Eq. (10.14) is included here.

```
....
// define the step size
     double h = (xmax-xmin)/(ndim+1);
     length = xmax-xmin;
// allocate space for the vector u and the temporary vector to
// be upgraded in every iteration
     Matrix u( ndim, ndim); // using Armadillo to define matrices
     Matrix u_temp( ndim, ndim);
     double pi = acos(-1.);
! set up of initial conditions at t = 0 and boundary conditions
     u = 0.
     for(i=0; i < ndim; i++){
       x = i*h*pi/length;
       u(i,1) = func(x);
       u(i,ndim) = -func(x);
     }
// iteration algorithm starts here
     iterations = 0;
     while( (iterations <= 20) && ( diff > 0.00001) ){
       u_temp = u; diff = 0.;
       for (j = 1; j< ndim - 1,j++){
          for(l = 1; l < ndim -1; l++){
            u(j,l) = 0.25*(u_temp(j+1,l)+u_temp(j-1,l)+ &
                       u_temp(j,l+1)+u_temp(j,l-1));
            diff += fabs(u_temp(i,j)-u(i,j));
          }
       }
       iterations++;
       diff /= pow((ndim+1.0),2.0);
     }  // end while loop
```

The important part of the algorithm is applied in the function which sets up the two-dimensional Laplace equation. There we have a do-while statement which tests the difference between the temporary vector and the solution $u_{i,j}$. Moreover, we have fixed the number of iterations to be at most 20. This is sufficient for the above problem, but for more general applications you need to test the convergence of the algorithm.

While the Jacobi iteration scheme is very simple and parallelizable, its slow convergence rate renders it impractical for any "real world" applications. One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation. This leads to the Successive Over Relaxation scheme, see chapter 19.5 on relaxation methods for boundary value problems of Ref. [36].

10.4 Wave Equation in two Dimensions

The $1+1$-dimensional wave equation reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2},$$

with $u = u(x,t)$ and we have assumed that we operate with dimensionless variables. Possible boundary and initial conditions with $L = 1$ are

$$\begin{cases} u_{xx} = u_{tt} & x \in (0,1), t > 0 \\ u(x,0) = g(x) & x \in (0,1) \\ u(0,t) = u(1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x \in (0,1) \end{cases}.$$

We discretize again time and position,

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

and

$$u_{tt} \approx \frac{u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)}{\Delta t^2},$$

which we rewrite as

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

and

$$u_{tt} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2},$$

resulting in

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right). \tag{10.16}$$

If we assume that all values at times $t = j$ and $t = j-1$ are known, the only unknown variable is $u_{i,j+1}$ and the last equation yields thus an explicit scheme for updating this quantity. We have thus an explicit finite difference scheme for computing the wave function u. The only additional complication in our case is the initial condition given by the first derivative in time, namely $\partial u/\partial t|_{t=0} = 0$. The discretized version of this first derivative is given by

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t},$$

and at $t = 0$ it reduces to

$$u_t \approx \frac{u_{i,+1} - u_{i,-1}}{2\Delta t} = 0,$$

implying that $u_{i,+1} = u_{i,-1}$. If we insert this condition in Eq. (10.16) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right). \tag{10.17}$$

We need seemingly two different equations, one for the first time step given by Eq. (10.17) and one for all other time-steps given by Eq. (10.16). However, it suffices to use Eq. (10.16) for all times as long as we provide $u(i,-1)$ using

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

in our setup of the initial conditions.

The situation is rather similar for the $2+1$-dimensional case, except that we now need to discretize the spatial y-coordinate as well. Our equations will now depend on three variables whose discretized versions are now

$$\begin{cases} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{cases},$$

and we will let $\Delta x = \Delta y = h$ and $n_x = n_y$ for the sake of simplicity. The equation with initial and boundary conditions reads now

$$\begin{cases} u_{xx} + u_{yy} = u_{tt} & x,y \in (0,1), t > 0 \\ u(x,y,0) = g(x,y) & x,y \in (0,1) \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x,y \in (0,1) \end{cases}.$$

We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2},$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2},$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2},$$

which we merge into the discretized $2+1$-dimensional wave equation as

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t^2}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \tag{10.18}$$

where again we have an explicit scheme with $u^{l+1}_{i,j}$ as the only unknown quantity. It is easy to account for different step lengths for x and y. The partial derivative is treated in much the same way as for the one-dimensional case, except that we now have an additional index due to the extra spatial dimension, viz., we need to compute $u^{-1}_{i,j}$ through

$$u^{-1}_{i,j} = u^0_{i,j} + \frac{\Delta t}{2h^2}\left(u^0_{i+1,j} - 4u^0_{i,j} + u^0_{i-1,j} + u^0_{i,j+1} + u^0_{i,j-1}\right),$$

in our setup of the initial conditions.

10.4.1 Closed-form Solution

We develop here the closed-form solution for the $2+1$ dimensional wave equation with the following boundary and initial conditions

$$\begin{cases} c^2(u_{xx} + u_{yy}) = u_{tt} & x,y \in (0,L), t > 0 \\ u(x,y,0) = f(x,y) & x,y \in (0,L) \\ u(0,0,t) = u(L,L,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = g(x,y) & x,y \in (0,L) \end{cases}.$$

Our first step is to make the ansatz

$$u(x,y,t) = F(x,y)G(t),$$

resulting in the equation

$$FG_{tt} = c^2(F_{xx}G + F_{yy}G),$$

or

$$\frac{G_{tt}}{c^2G} = \frac{1}{F}(F_{xx} + F_{yy}) = -\nu^2.$$

The lhs and rhs are independent of each other and we obtain two differential equations

$$F_{xx} + F_{yy} + F\nu^2 = 0,$$

and

$$G_{tt} + Gc^2\nu^2 = G_{tt} + G\lambda^2 = 0,$$

with $\lambda = c\nu$. We can in turn make the following ansatz for the x and y dependent part

$$F(x,y) = H(x)Q(y),$$

which results in

$$\frac{1}{H}H_{xx} = -\frac{1}{Q}(Q_{yy} + Q\nu^2) = -\kappa^2.$$

Since the lhs and rhs are again independent of each other, we can separate the latter equation into two independent equations, one for x and one for y, namely

$$H_{xx} + \kappa^2 H = 0,$$

and

$$Q_{yy} + \rho^2 Q = 0,$$

with $\rho^2 = \nu^2 - \kappa^2$.

The second step is to solve these differential equations, which all have trigonometric functions as solutions, viz.

$$H(x) = A\cos(\kappa x) + B\sin(\kappa x),$$

and

$$Q(y) = C\cos(\rho y) + D\sin(\rho y).$$

The boundary conditions require that $F(x,y) = H(x)Q(y)$ are zero at the boundaries, meaning that $H(0) = H(L) = Q(0) = Q(L) = 0$. This yields the solutions

$$H_m(x) = \sin(\frac{m\pi x}{L}) \qquad Q_n(y) = \sin(\frac{n\pi y}{L}),$$

or

$$F_{mn}(x,y) = \sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

With $\rho^2 = \nu^2 - \kappa^2$ and $\lambda = c\nu$ we have an eigenspectrum $\lambda = c\sqrt{\kappa^2 + \rho^2}$ or $\lambda_{mn} = c\pi/L\sqrt{m^2 + n^2}$. The solution for G is

$$G_{mn}(t) = B_{mn}\cos(\lambda_{mn}t) + D_{mn}\sin(\lambda_{mn}t),$$

with the general solution of the form

$$u(x,y,t) = \sum_{mn=1}^{\infty} u_{mn}(x,y,t) = \sum_{mn=1}^{\infty} F_{mn}(x,y)G_{mn}(t).$$

The final step is to determine the coefficients B_{mn} and D_{mn} from the Fourier coefficients. The equations for these are determined by the initial conditions $u(x,y,0) = f(x,y)$ and $\partial u/\partial t|_{t=0} = g(x,y)$. The final expressions are

$$B_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdyf(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}),$$

and

$$D_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdyg(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

Inserting the particular functional forms of $f(x,y)$ and $g(x,y)$ one obtains the final closed-form expressions.

10.5 Exercises

10.1. Consider the two-dimensional wave equation for a vibrating membrane given by the following initial and boundary conditions

$$\begin{cases} u_{xx}+u_{yy}=u_{tt} & x,y\in(0,1),t>0 \\ u(x,y,0)=sin(x)cos(y) & x,y\in(0,1) \\ u(0,0,t)=u(1,1,t)=0 & t>0 \\ \partial u/\partial t|_{t=0}=0 & x,y\in(0,1) \end{cases}.$$

1. Find the closed-form solution for this equation using the technique of separation of variables.
2. Write down the algorithm for solving this equation and set up a program to solve the discretized wave equation. Compare your results with the closed-form solution. Use a quadratic grid.
3. Consider thereafter a $2+1$ dimensional wave equation with variable velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla(\lambda(x,y)\nabla u).$$

If λ is constant, we obtain the standard wave equation discussed in the two previous points. The solution $u(x,y,t)$ could represent a model for water waves. It represents then the surface elevation from still water. The function λ simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines. You can discretize

$$\nabla(\lambda(x,y)\nabla u) = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

as follows using again a quadratic domain for x and y:

$$\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right)\approx\frac{1}{\Delta x}\left(\lambda_{i+1/2,j}\left[\frac{u^l_{i+1,j}-u^l_{i,j}}{\Delta x}\right]-\lambda_{i-1/2,j}\left[\frac{u^l_{i,j}-u^l_{i-1,j}}{\Delta x}\right]\right),$$

and

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right)\approx\frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1}-u^l_{i,j}}{\Delta y}\right]-\lambda_{i,j-1/2}\left[\frac{u^l_{i,j}-u^l_{i,j-1}}{\Delta y}\right]\right).$$

Convince yourself that this equation has the same truncation error as the expressions used in a) and b) and that they result in the same equations when λ is a constant.

4. Develop an algorithm for solving the new wave equation and write a program which implements it.

10.2. We are looking at a one-dimensional problem

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, t > 0, x \in [0,L]$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = 0 \qquad 0 < x < L$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = 0 \qquad t > 0,$$

and

$$u(L,t) = 1 \qquad t > 0.$$

The function $u(x,t)$ can be the temperature gradient of a the rod or represent the fluid velocity in a direction parallel to the plates, that is normal to the x-axis. In the latter case, for small t, only the part of the fluid close to the moving plate is set in significant motion, resulting in a thin boundary layer at $x = L$. As time increases, the velocity approaches a linear variation with x. In this case, which can be derived from the incompressible Navier-Stokes, the above equations constitute a model for studying friction between moving surfaces separated by a thin fluid film.

In this project we want to study the numerical stability of three methods for partial differential equations (PDEs). These methods are

- The explicit forward Euler algorithm with discretized versions of time given by a forward formula and a centered difference in space resulting in

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}.$$

- The implicit Backward Euler with

$$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2},$$

- Finally we use the implicit Crank-Nicolson scheme with a time-centered scheme at $(x,t+\Delta t/2)$

$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t}.$$

The corresponding spatial second-order derivative reads

$$u_{xx} \approx \frac{1}{2}\left(\frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}+\right.$$

$$\left.\frac{u(x_i+\Delta x,t_j+\Delta t)-2u(x_i,t_j+\Delta t)+u(x_i-\Delta x,t_j+\Delta t)}{\Delta x^2}\right).$$

Note well that we are using a time-centered scheme wih $t+\Delta t/2$ as center.

1. Write down the algorithms for these three methods and the equations you need to implement. For the implicit schemes show that the equations lead to a tridiagonal matrix system for the new values.
2. Find the truncation errors of these three schemes and investigate their stability properties. Find also the closed-form solution to the continuous problem. A useful hint here is to solve for $v(x,t)=u(x,t)-x$ instead. The boundary conditions for $v(x,t)$ are simpler, $v(0,t)=v(1,t)=0$ and the initial conditions are $v(x,0)=-x$.
3. Implement the three algorithms in the same code and perform tests of the solution for these three approaches for $\Delta x = 1/10$, $\Delta x = 1/100$ using Δt as dictated by the stability limit of the explicit scheme. Study the solutions at two time points t_1 and t_2 where $u(x,t_1)$ is smooth but still significantly curved and $u(x,t_2)$ is almost linear, close to the stationary state.
4. Compare the solutions at t_1 and t_2 with the closed-form result for the continuous problem. Which of the schemes would you classify as the best?
5. Generalize this problem to two dimensions and write down the algorithm for the forward and backward Euler approaches. Write a program which solves the diffusion equation in $2+1$ dimensions. The program should allow for general boundary and initial conditions.

10.3. In this project will first study the simple two-dimensional wave equation and compare our numerical solution with closed-form results. Thereafter we introduce a simple model for a tsunami.

Consider first the two-dimensional wave equation for a vibrating square membrane given by the following initial and boundary conditions

$$\begin{cases} \lambda\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=\frac{\partial^2 u}{\partial t^2} & x,y\in[0,1], t\geq 0 \\ u(x,y,0)=sin(\pi x)sin(2\pi y) & x,y\in(0,1) \\ u=0 \text{ boundary} & t\geq 0 \\ \partial u/\partial t|_{t=0}=0 & x,y\in(0,1) \end{cases}.$$

The boundary is defined by $x=0$, $x=1$, $y=0$ and $y=1$.

- Find the closed-form solution for this equation using the technique of separation of variables.
- Write down the algorithm for the explicit method for solving this equation and set up a program to solve the discretized wave equation. Describe in particular how you treat the boundary conditions and initial conditions. Compare your results with the closed-form solution. Use a quadratic grid.
  Check your results as function of the number of mesh points and in particular against the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\lambda}} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

where Δt, Δx and Δy are the chosen step lengths. In our case $\Delta x = \Delta y$. It can be useful to make animations of the results.
An example of a simple code which solves this problem using the explicit scheme is listed here.

```
int main ( int argc, char * argv[] )
{
 // terminal input
 if ( argc < 4 ) {
  cout << "\n\nToo few input arguments. Please provide\n"
 << "spatial resolution, time step and\n"
 << "final time.\n\n"
 << "Ex: proj4b 100 0.005 10\n\n";
  return 0;
 }

 int n;
 double tStep, tFinal;
 n = atoi(argv[1]);
 tStep = atof(argv[2]);
 tFinal = atof(argv[3]);
 double h = 1.0/(((double) n) - 1.0);
 // variables
 double ** u;
 double ** uLast;
 double ** uNext;
 u = new double * [n];
 uLast = new double * [n];
 uNext = new double * [n];

 double * x;
 double * y;
 x = new double [n];
 y = new double [n];

 for ( int i = 0; i < n; i++ ) {
  u[i] = new double [n];
  uLast[i] = new double [n];
  uNext[i] = new double [n];
  x[i] = i*h;
  y[i] = x[i];
 }
 // initializing
 for ( int i = 0; i < n; i++ ) { // setting initial step
  for ( int j = 0; j < n; j++ ) {
   uLast[i][j] = sin(PI*x[i])*sin(2*PI*y[j]);
  }
 }

 for ( int i = 1; i < (n-1); i++ ) { // setting first step using the initial derivative
  for ( int j = 1; j < (n-1); j++ ) {
   u[i][j] = uLast[i][j] - ((tStep*tStep)/(2.0*h*h))*
(4*uLast[i][j] - uLast[i+1][j] - uLast[i-1][j] - uLast[i][j+1] - uLast[i][j-1]);
  }
  u[i][0] = 0; // setting boundaries once and for all
  u[i][n-1] = 0;
  u[0][i] = 0;
  u[n-1][i] = 0;
```

```
  uNext[i][0] = 0;
  uNext[i][n-1] = 0;
  uNext[0][i] = 0;
  uNext[n-1][i] = 0;
 }

 // iterating in time
 double t = 0.0 + tStep;
 int iter = 0;

 while ( t < tFinal ) {
  iter ++;
  t = t + tStep;
  for ( int i = 1; i < (n-1); i++ ) { // computing next step
   for ( int j = 1; j < (n-1); j++ ) {
uNext[i][j] = 2*u[i][j] - uLast[i][j] - ((tStep*tStep)/(h*h))*
  (4*u[i][j] - u[i+1][j] - u[i-1][j] - u[i][j+1] - u[i][j-1]);
   }
  }
  for ( int i = 1; i < (n-1); i++ ) { // shifting results down
   for ( int j = 1; j < (n-1); j++ ) {
uLast[i][j] = u[i][j];
u[i][j] = uNext[i][j];
   }
  }
 }
 // computing error and printing to screen
 double error;
 double errorTmp;
 for ( int i = 0; i < n; i++ ) {
  for ( int j = 0; j < n; j++ ) {
   errorTmp = u[i][j] - sin(PI*x[i])*sin(2*PI*y[j])*cos(sqrt(5)*PI*t);
   error += errorTmp*errorTmp;
  }
 }
 error = sqrt(error)/((double) n);
 cout << "\n\nRMS Error: " << setprecision(8) << error << endl
    << "Iterations: " << iter
    << "\n\n\n";

 // deallocating memory
 for ( int i = 0; i < n; i++ ) {
  delete [] u[i];
  delete [] uLast[i];
  delete [] uNext[i];
 }
 delete [] u;
 delete [] uLast;
 delete [] uNext;

 delete [] x;
 delete [] y;



 // finishing without error
 return 0;
}
```

We modify now the wave equation in order to consider a $2+1$ dimensional wave equation with a position dependent velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (\lambda(x,y)\nabla u).$$

If λ is constant, we obtain the standard wave equation discussed in the two previous points. The solution $u(x,y,t)$ could represent a model for water waves. It represents then the surface elevation from still water. We will model λ as

$$\lambda = gH(x,y),$$

with g being the acceleration of gravity and $H(x,y)$ is the still water depth.

The function $H(x,y)$ simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines as discussed in point d) below. We have assumed that the vertical motion is negligible and that we deal with long wavelenghts $\tilde{\lambda}$ compared with the depth of the sea H, that is $\tilde{\lambda}/H \gg 1$. We will also neglect Coriolis effects.

You can discretize

$$\nabla \cdot (\lambda(x,y)\nabla u) = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

as follows using again a quadratic domain for x and y:

$$\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Delta x}\left(\lambda_{i+1/2,j}\left[\frac{u^l_{i+1,j}-u^l_{i,j}}{\Delta x}\right] - \lambda_{i-1/2,j}\left[\frac{u^l_{i,j}-u^l_{i-1,j}}{\Delta x}\right]\right),$$

and

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right) \approx \frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1}-u^l_{i,j}}{\Delta y}\right] - \lambda_{i,j-1/2}\left[\frac{u^l_{i,j}-u^l_{i,j-1}}{\Delta y}\right]\right).$$

- Show that this equation has the same truncation error as the expressions used in a) and b) and that they result in the same equations when λ is a constant.

We assume that we can approximate the coastline with a quadratic grid. As boundary condition at the coastline we will employ

$$\frac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n} = 0,$$

where $\partial u/\partial n$ is the derivative in the direction normal to the boundary.

We are going to model the impact of an earthquake on sea water. This is normally modelled via an elevation of the sea bottom. We will assume that the movement of the sea bottom is very rapid compared with the period of the propagating waves. This means that we can approximate the bottom elevation with an initial surface elevation. The initial conditions are then given by (with L the length of the grid)

$$u(x,y,0) = f(x,y) \qquad x,y \in (0,L),$$

and

$$\partial u/\partial t|_{t=0} = 0 \qquad x,y \in (0,L).$$

We will approximate the initial elevation with the function

$$f(x,y) = A_0 \exp\left(-\left[\frac{x-x_c}{\sigma_x}\right]^2 - \left[\frac{y-y_c}{\sigma_y}\right]^2\right),$$

where A_0 is the elevation of the surface and is typically $1-2$ m. The variables σ_x and σ_y represent the extensions of the surface elevation. In this project we will let $\sigma_x = 80$ km and $\sigma_y = 200$ km. The 2004 tsunami had extensions of approximately 200 and 1000 km, respectively.

The variables x_c and y_c represent the epicentre of the earthquake.

We need also to model the sea bottom and the function $\lambda(x,y) = gH(x,y)$. We assume that we can model the sea bottom with a water depth of 5000 m and a surface elevation of 2 m. The sea bottom towards one of the coastlines has a shape with an inclination of $\theta = 1$ degree and depth where the earthquake takes place of 5000 m. This gives the following model for $\lambda(x,y) = gH(x,y) = gH(x)$ with $H_0 = 5000$ m

```
for ( int i = 0; i < (2*n+1); i++ ) {
  if ( (i-1)*(h/2.0) < X_0 ) {
    lambda[i] = G*H_0; // lambda depends only on x
  } else {
    lambda[i] = G*(H_0 - ((i-1)*(h/2.0)-X_0)*0.0174550649282176);
  }
 }
```

Here X_0 is the point where the sea bed changes (with respect to shore). Your tasks are as follows:

- Develop an algorithm for solving the new wave equation and write a program which implements it. Pay in particular attention to the implementation of the boundary conditions and the initial conditions. Figure out how to deal with the fictitious values in time and space for the discretized functions. You need also to find the functional form of $H(x,y) = H(x)$.
  Be careful to scale the equations properly. With the depth of 5000 m, extensions $\sigma_x = 80$ km and $\sigma_y = 200$ km you need to figure out the proper dimensions of the grid $L \times L$. Scale the equations so that you can use dimensionless quantities.
  With the above parameters, initial values and boundary conditions, study the temporal evolution of the wave towards the coastline. Comment your results. It can be useful to make animations of the results (a simple recipe with gnuplot and python for this is available under the project link for project 4 at the webpage).
  It also important that you keep in mind the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\max\lambda(x,y)}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2}$$

- We keep now the same shape of the sea bottom and the same parameters as in d), but we shift the center of the earthquake to the right with 40 km. Which one of the two earthquakes will produce the largest impact (wave elevation) at the coastline? Comment your results.

10.4. Consider a condensator with parallel plates separated at a distance L resulting in the voltage differences $u(x,0) = 100sin(2\pi x/L)$ and $u(x,1) = -100sin(2\pi x/L)$. These are our boundary conditions. Write a program which obtains the voltage u between the plates using both the Jacobi method and the Gauss-Seidel method. Parallelize your program as detailed in chapter 6 and study the stability of your solutions as functions of the number of mesh points. How does your parallel code scale?