Computational Physics Lectures: Partial differential equations

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Aug 23, 2017

Famous PDEs

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

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial^2 u}{\partial t^2}, \tag{1}$$

where A is a constant. The solution u depends on both spatial and temporal variables, viz. $u = u(x,t)$.

Famous PDEs, two dimension

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

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

Famous PDEs, diffusion equation

The diffusion equation whose one-dimensional version reads

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

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.

Famous PDEs, Laplace's equation

Another familiar equation from electrostatics is Laplace's equation, which looks similar to the wave equation in Eq. (1) except that we have set $A = 0$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{3}$$

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

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

Famous PDEs, Helmholtz' equation

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

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = \lambda u, \tag{5}$$

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

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0, \tag{6}$$

Famous PDEs, Schroedinger's equation in two dimensions

Schroedinger's equation

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

Famous PDEs, Maxwell's equations

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

$$\frac{\partial \mathbf{E}}{\partial t} = \text{curl}\mathbf{B},$$

and

$$-\text{curl}\mathbf{E} = \mathbf{B}$$

and

$$\text{div}\mathbf{E} = \text{div}\mathbf{B} = 0.$$

Famous PDEs, Euler's equations

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.

Famous PDEs, the Navier-Stokes' equations

Another example is the set of Navier-Stokes equations for incompressible, viscous flow

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

Famous PDEs, general equation in two dimensions

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. (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.

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

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

Diffusion equation

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. The constant D can be coupled with various physical constants, such as the diffusion constant or the specific heat and thermal conductivity discussed below.

Diffusion equation, famous laws

If we let u denote the concetration of a particle species, this results in Fick's law of diffusion. 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.

Diffusion equation, 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.,

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

Diffusion equation, heat equation in one dimension

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

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

Diffusion equation, dimensionless form

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 $T(\hat{x}, \hat{t}) \to u(x, t)$.

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.

Explicit Scheme, boundary conditions

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 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

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

Explicit Scheme, algorithm

If we 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.

Explicit Scheme, simplifications

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

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{7}$$

Explicit Scheme, solving the equations

Since all the discretized initial values

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

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. (7).

Explicit Scheme, simple case

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 $t_j = j\Delta t$

$$V_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{bmatrix}.$$

Explicit Scheme, matrix-vector formulation

This results in a matrix-vector multiplication

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

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

$$\mathbf{A} = \begin{bmatrix} 1-2\alpha & \alpha & 0 & 0\dots \\ \alpha & 1-2\alpha & \alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & \alpha & 1-2\alpha \end{bmatrix}$$

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

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

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.

Explicit Scheme, sketch of code

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
//  Armadillo notation for vectors
    u(0) = unew(0) = u(n) = unew(n) = 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 integration
```

```
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.
```

Explicit Scheme, stability condition

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 $\Delta x = 0.01$ (a rather frequent choice), 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 $\rho(\mathbf{A})$ of our matrix $\mathbf{A}$ satisfies the condition

$$\rho(\mathbf{A}) < 1. \tag{8}$$

Explicit Scheme, spectral radius and stability

The spectral radius is defined as

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

which is interpreted as the smallest number such that a circle with radius centered at zero in the complex plane contains all eigenvalues of $\mathbf{A}$. If the matrix is positive definite, the condition in Eq. (8) is always satisfied.

Explicit Scheme, eigenvalues and stability

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

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

with

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

Explicit Scheme, final stability analysis

The eigenvalues of $\mathbf{A}$ 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

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

Explicit Scheme, stability condition

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

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

or

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

which is nothing but

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

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

Our requirement in Eq. (8) results in

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

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

Explicit Scheme, general tridiagonal matrix

A more general tridiagonal matrix

$$\mathbf{A} = \begin{bmatrix} a & b & 0 & 0 & \ldots \\ c & a & b & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & b \\ 0 & 0 & \ldots & c & a \end{bmatrix},$$

has eigenvalues $\mu_i = a + s\sqrt{bc}\cos(i\pi/n + 1)$ with $i = 1 : n$.

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 $O(\Delta t)$.

Implicit Scheme

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 $\alpha = \Delta t/\Delta x^2$.

Implicit Scheme

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 $\mathbf{A}$

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

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

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

Implicit Scheme

It means that we can rewrite the problem as

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

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. 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.

Implicit Scheme

The implicit scheme is always stable since the spectral radius satisfies $\rho(\mathbf{A}) < 1$. 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 $\mathbf{A} = \hat{I} + \alpha\hat{B}$ 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 $\rho(\mathbf{A}^{-1}) < 1$ and the method is stable for all combinations of Δt and Δx.

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.

```
//  parts of the function for backward Euler
void backward_euler(int n, int tsteps, double delta_x, double alpha)
{
   double a, b, c;
   vec u(n+1); // This is u  of Au = y
   vec y(n+1); // Right side of matrix equation Au=y, the solution at a previous step

   // Initial conditions
   for (int i = 1; i < n; i++) {
      y(i) = u(i) = func(delta_x*i);
   }
   // Boundary conditions (zero here)
   y(n) = u(n) = u(0) = y(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, y, u, n+1);
      // boundary conditions
      u(0) = 0;
      u(n) = 0;
      // replace previous time solution with new
      for (int i = 0; i <= n; i++) {
         y(i) = u(i);
      }
```

```
      //  You may consider printing the solution at regular time intervals
      ....   // print statements
   }  // end time iteration
   ...
}
```

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

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j}-2u_{i,j}+u_{i+1,j}\right)+\frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1}-2u_{i,j-1}+u_{i-1,j-1}\right)=\frac{1}{\Delta t}\left(u_{i,j}-u_{i,j-1}\right), \tag{9}$$

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.

Derivation of CN scheme

To derive the Crank-Nicolson equation, we start with the forward Euler scheme and Taylor expand $u(x, t + \Delta t)$, $u(x + \Delta x, t)$ and $u(x - \Delta x, t)$

$$\begin{aligned}
u(x+\Delta x,t) &= u(x,t)+\frac{\partial u(x,t)}{\partial x}\Delta x+\frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathcal{O}(\Delta x^3), \\
u(x-\Delta x,t) &= u(x,t)-\frac{\partial u(x,t)}{\partial x}\Delta x+\frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathcal{O}(\Delta x^3), \\
u(x,t+\Delta t) &= u(x,t)+\frac{\partial u(x,t)}{\partial t}\Delta t+\mathcal{O}(\Delta t^2).
\end{aligned} \tag{10}$$

Taylor expansions

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

$$\begin{aligned}
\left[\frac{\partial u(x,t)}{\partial t}\right]_{\text{approx}} &= \frac{\partial u(x,t)}{\partial t}+\mathcal{O}(\Delta t), \\
\left[\frac{\partial^2 u(x,t)}{\partial x^2}\right]_{\text{approx}} &= \frac{\partial^2 u(x,t)}{\partial x^2}+\mathcal{O}(\Delta x^2).
\end{aligned} \tag{11}$$

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

Error in CN 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 $t' = t + \Delta t/2$.

$$u(x+\Delta x, t+\Delta t) = u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x-\Delta x, t+\Delta t) = u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x+\Delta x, t) = u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x-\Delta x, t) = u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x,t+\Delta t) = u(x,t') + \frac{\partial u(x,t')}{\partial t}\frac{\Delta_t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3)$$

$$u(x,t) = u(x,t') - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3)$$

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

$$\left[\frac{\partial u(x,t')}{\partial t}\right]_{\text{approx}} = \frac{\partial u(x,t')}{\partial t} + \mathcal{O}(\Delta t^2), \tag{12}$$

$$\left[\frac{\partial^2 u(x,t')}{\partial x^2}\right]_{\text{approx}} = \frac{\partial^2 u(x,t')}{\partial x^2} + \mathcal{O}(\Delta x^2).$$

Truncation errors and stability

The following table summarizes the three methods.

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

Rewrite of CN scheme

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite Eq. (9) 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

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

Final CN equations

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

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

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.

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.

Parts of Code for the Crank-Nicolson Scheme

We illustrate this in the following part of our program.

```
void crank_nicolson(int n, int tsteps, double delta_x, double alpha)
{
   double a, b, c;
   vec u(n+1); // This is u in Au = r
   vec r(n+1); // Right side of matrix equation Au=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 < n; i++) {
         r(i) = alpha*u(i-1) + (2 - 2*alpha)*u(i) + alpha*u(i+1);
      }
      r(0) = 0;
      r(n) = 0;
      //  Then solve the tridiagonal matrix
      tridiag(a, b, c, r, u, xsteps+1);
      u(0) = 0;
      u(n) = 0;
      //  Eventual print statements etc
      ....
}
```

Python code for solving the one-dimensional diffusion equation

The following Python code sets up and solves the diffusion equation for all three methods discussed.

```
    # Code for solving the 1+1 dimensional diffusion equation
    # du/dt = ddu/ddx on a rectangular grid of size L x (T*dt),
    # with with L = 1, u(x,0) = g(x), u(0,t) = u(L,t) = 0

    import numpy, sys, math
    from  matplotlib import pyplot as plt
    import numpy as np

    def forward_step(alpha,u,uPrev,N):
        """
        Steps forward-euler algo one step ahead.
        Implemented in a separate function for code-reuse from crank_nicolson()
        """

        for x in xrange(1,N+1): #loop from i=1 to i=N
            u[x] = alpha*uPrev[x-1] + (1.0-2*alpha)*uPrev[x] + alpha*uPrev[x+1]

    def forward_euler(alpha,u,N,T):
        """
        Implements the forward Euler sheme, results saved to
        array u
        """

        #Skip boundary elements
```

```
    for t in xrange(1,T):
        forward_step(alpha,u[t],u[t-1],N)

def tridiag(alpha,u,N):
    """
    Tridiagonal gaus-eliminator, specialized to diagonal = 1+2*alpha,
    super- and sub- diagonal = - alpha
    """
    d = numpy.zeros(N) + (1+2*alpha)
    b = numpy.zeros(N-1) - alpha

    #Forward eliminate
    for i in xrange(1,N):
        #Normalize row i (i in u convention):
        b[i-1] /= d[i-1];
        u[i] /= d[i-1] #Note: row i in u = row i-1 in the matrix
        d[i-1] = 1.0
        #Eliminate
        u[i+1] += u[i]*alpha
        d[i] += b[i-1]*alpha
    #Normalize bottom row
    u[N] /= d[N-1]
    d[N-1] = 1.0

    #Backward substitute
    for i in xrange(N,1,-1): #loop from i=N to i=2
        u[i-1] -= u[i]*b[i-2]
        #b[i-2] = 0.0 #This is never read, why bother...

def backward_euler(alpha,u,N,T):
    """
    Implements backward euler scheme by gaus-elimination of tridiagonal matrix.
    Results are saved to u.
    """
    for t in xrange(1,T):
        u[t] = u[t-1].copy()
        tridiag(alpha,u[t],N) #Note: Passing a pointer to row t, which is modified in-place

def crank_nicolson(alpha,u,N,T):
    """
    Implents crank-nicolson scheme, reusing code from forward- and backward euler
    """
    for t in xrange(1,T):
        forward_step(alpha/2,u[t],u[t-1],N)
        tridiag(alpha/2,u[t],N)

def g(x):
    """Initial condition u(x,0) = g(x), x \in [0,1]"""
    return numpy.sin(math.pi*x)

# Number of integration points along x-axis
N       =   100
# Step length in time
dt      =   0.01
# Number of time steps till final time
T       =   100
# Define method to use 1 = explicit scheme, 2= implicit scheme, 3 = Crank-Nicolson
method  =   2

#dx = 1/float(N+1)
u = numpy.zeros((T,N+2),numpy.double)
```

```
(x,dx) = numpy.linspace (0,1,N+2, retstep=True)
alpha = dt/(dx**2)

#Initial codition
u[0,:] = g(x)
u[0,0] = u[0,N+1] = 0.0 #Implement boundaries rigidly

if   method == 1:
    forward_euler(alpha,u,N,T)
elif method == 2:
    backward_euler(alpha,u,N,T)
elif method == 3:
    crank_nicolson(alpha,u,N,T)
else:
    print "Please select method 1,2, or 3!"
    import sys
    sys.exit(0)
# To do: add movie
```

Solution for the One-dimensional Diffusion Equation

It cannot be repeated enough, it is always useful to find cases where one can compare the numerical results 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

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

Solution for the One-dimensional Diffusion Equation

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.

Solution for the One-dimensional Diffusion Equation

We call this constant $-\lambda^2$. This gives us the two differential equations,

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

with general solutions

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

Solution for the One-dimensional Diffusion Equation

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

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

Solution for the One-dimensional Diffusion Equation

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.

Explict scheme for the diffusion equation in two dimensions

The $2+1$-dimensional diffusion equation, with the diffusion constant $D = 1$, is given by

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

where we have $u = u(x, y, t)$. We assume that we have a square lattice of length L with equally many mesh points in the x and y directions.

We discretize again position and time using now

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

which we rewrite as, in its discretized version,

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

where $x_i = x_0 + ih$, $y_j = y_0 + jh$ and $t_l = t_0 + l\Delta t$, with $h = L/(n+1)$ and Δt the time step.

Explict scheme for the diffusion equation in two dimensions

We have defined our domain to start $x(y) = 0$ and end at $X(y) = L$. The second derivative with respect to y reads

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

We use again the so-called forward-going Euler formula for the first derivative in time. In its discretized form we have

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

resulting in

$$u^{l+1}_{i,j} = u^l_{i,j} + \alpha\left[u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1} - 4u^l_{i,j}\right],$$

where the left hand side, with the solution at the new time step, is the only unknown term, since starting with $t = t_0$, the right hand side is entirely determined by the boundary and initial conditions. We have $\alpha = \Delta t/h^2$. This scheme can be implemented using essentially the same approach as we used in Eq. (7).

Laplace's and Poisson's Equations

Laplace's equation reads

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

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.

Laplace's and Poisson's Equations, discretized version

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

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

Laplace's and Poisson's Equations, final discretized version

Inserting in Laplace's equation we obtain

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1}+u_{i,j-1}+u_{i+1,j}+u_{i-1,j}\right]. \tag{13}$$

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

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1}+u_{i,j-1}+u_{i+1,j}+u_{i-1,j}\right]+\frac{h^2}{4}\rho_{i,j}. \tag{14}$$

Laplace's and Poisson's Equations, boundary conditions

The boundary condtions read

$$u_{i,0} = g_{i,0} \qquad 0 \le i \le n+1,$$

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

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

and

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

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 $u_{i,j}$.

Scheme for solving Laplace's (Poisson's) equation

We rewrite Eq. (14)

$$4u_{i,j} = [u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}] - h^2\rho_{i,j} = \Delta_{ij} - \tilde{\rho}_{ij}, \qquad (15)$$

where we have defined

$$\Delta_{ij} = [u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}],$$

and

$$\tilde{\rho}_{ij} = h^2\rho_{i,j}.$$

Scheme for solving Laplace's (Poisson's) equation

In order to illustrate how we can transform the last equations into a linear algebra problem of the type $\mathbf{Ax} = \mathbf{w}$, with $\mathbf{A}$ a matrix and $\mathbf{x}$ and $\mathbf{w}$ unknown and known vectors respectively, let us also for the sake of simplicity assume that the number of points $n = 3$. We assume also that $u(x, y) = g(x, y)$ on the border $\delta\Omega$.

The inner values of the function u are then given by

$$\begin{aligned}
4u_{11} - u_{21} - u_{01} - u_{12} - u_{10} &= -\tilde{\rho}_{11} \\
4u_{12} - u_{02} - u_{22} - u_{13} - u_{11} &= -\tilde{\rho}_{12} \\
4u_{21} - u_{11} - u_{31} - u_{22} - u_{20} &= -\tilde{\rho}_{21} \\
4u_{22} - u_{12} - u_{32} - u_{23} - u_{21} &= -\tilde{\rho}_{22}.
\end{aligned}$$

Scheme for solving Laplace's (Poisson's) equation

If we isolate on the left-hand side the unknown quantities u_{11}, u_{12}, u_{21} and u_{22}, that is the inner points not constrained by the boundary conditions, we can rewrite the above equations as a matrix $\mathbf{A}$ times an unknown vector $\mathbf{x}$, that is

$$Ax = b,$$

or in more detail

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix} = \begin{bmatrix} u_{01} + u_{10} - \tilde{\rho}_{11} \\ u_{13} + u_{02} - \tilde{\rho}_{12} \\ u_{31} + u_{20} - \tilde{\rho}_{21} \\ u_{32} + u_{23} - \tilde{\rho}_{22} \end{bmatrix}.$$

Scheme for solving Laplace's (Poisson's) equation

The right hand side is constrained by the values at the boundary plus the known function $\tilde{\rho}$. For a two-dimensional equation it is easy to convince oneself that for larger sets of mesh points, we will not have more than five function values for every row of the above matrix. For a problem with $n+1$ mesh points, our matrix $\mathbf{A} \in \mathbb{R}^{(n+1)\times(n+1)}$ leads to $(n-1)\times(n-1)$ unknown function values u_{ij}. This means that, if we fix the endpoints for the two-dimensional case (with a square lattice) at $i(j) = 0$ and $i(j) = n+1$, we have to solve the equations for $1 \geq i(j)len$.

Since the matrix is rather sparse but is not on a tridiagonal form, elimination methods like the LU decomposition discussed, are not very practical. Rather, iterative schemes like Jacobi's method or the Gauss-Seidel are preferred. The above matrix is also always diagonally dominant, a necessary condition for these iterative solvers to converge.

Scheme for solving Laplace's (Poisson's) equation using Jacobi's iterative method

In setting up for example Jacobi's method, it is useful to rewrite the matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L},$$

with $\mathbf{D}$ being a diagonal matrix with 4 as the only value, $\mathbf{U}$ is an upper triangular matrix and $\mathbf{L}$ a lower triangular matrix. In our case we have

$$\mathbf{D} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix},$$

and

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Scheme for solving Laplace's (Poisson's) equation, with Jacobi's method

We assume now that we have an estimate for the unknown functions u_{11}, u_{12}, u_{21} and u_{22}. We will call this the zeroth value and label it as $u_{11}^{(0)}$, $u_{12}^{(0)}$, $u_{21}^{(0)}$ and $u_{22}^{(0)}$. We can then set up an iterative scheme where the next solution is defined

in terms of the previous one as

$$
\begin{aligned}
u_{11}^{(1)} =& \frac{1}{4}(b_1 - u_{12}^{(0)} - u_{21}^{(0)}) \\
u_{12}^{(1)} =& \frac{1}{4}(b_2 - u_{11}^{(0)} - u_{22}^{(0)}) \\
u_{21}^{(1)} =& \frac{1}{4}(b_3 - u_{11}^{(0)} - u_{22}^{(0)}) \\
u_{22}^{(1)} =& \frac{1}{4}(b_4 - u_{12}^{(0)} - u_{21}^{(0)}),
\end{aligned}
$$

where we have defined the vector

$$
\mathbf{b} = \begin{bmatrix} u_{01} + u_{10} - \tilde{\rho}_{11} \\ u_{13} + u_{02} - \tilde{\rho}_{12} \\ u_{31} + u_{20} - \tilde{\rho}_{21} \\ u_{32} + u_{23} - \tilde{\rho}_{22} \end{bmatrix}.
$$

Scheme for solving Laplace's (Poisson's) equation, final rewrite

We can rewrite the equations in a more compact form in terms of the matrices $\mathbf{D}$, $\mathbf{L}$ and $\mathbf{U}$ as, after $r+1$ iterations,

$$
\mathbf{x}^{(r+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(r)}\right), \tag{16}
$$

where the unknown functions are now defined in terms of

$$
\mathbf{x} = \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix}.
$$

If we wish to implement Gauss-Seidel's algorithm, the set of equations to solve are then given by

$$
\mathbf{x}^{(r+1)} = -(\mathbf{D} + \mathbf{L})^{-1}\left(\mathbf{b} - \mathbf{U}\mathbf{x}^{(r)}\right), \tag{17}
$$

or alternatively as

$$
\mathbf{x}^{(r+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(r+1)} - \mathbf{U}\mathbf{x}^{(r)}\right).
$$

Jacobi Algorithm for solving Laplace's Equation

It is thus fairly straightforward to extend this equation to the three-dimensional case. Whether we solve Eq. (13) or Eq. (14), 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. (13) requires the knowledge of u at all of the neighbouring points in order to calculate u at any given point.

Jacobi Algorithm for solving Laplace's Equation

The way we solve these equations is based on an iterative scheme based on the Jacobi method or the Gauss-Seidel method or the relaxation methods.

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. (13) or Eq. (14) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (13). 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.

Jacobi Algorithm for solving Laplace's Equation, the algorithm

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 $j = 1:n$
2. Use Eq. (13) 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 to the next step.
4. Update the new value of u for the given iteration
5. Go to step 2

Jacobi Algorithm for solving Laplace's Equation, simple example

A simple example may help in understanding this method. We consider a condensator with parallel plates separated at a distance L resulting in for example the voltage differences $u(x,0) = 200sin(2\pi x/L)$ and $u(x,1) = -200sin(2\pi x/L)$. 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. (13) using Jacobi's method. Only the part which computes Eq. (13) is included here.

```
....
//  We define the step size for a square lattice with n+1 points
       double h = (xmax-xmin)/(n+1);
       double L = xmax-xmin;   // The length of the lattice
//  We allocate space for the vector u and the temporary vector to
//  be upgraded in every iteration
       mat  u( n+1, n+1);  // using Armadillo to define matrices
       mat  u_temp( n+1, n+1);  // This is the temporary value
       u = 0.  //  This is also our initial guess for all unknown values
```

```
//  We need to set up the  boundary conditions.  Specify for various cases
    .....
//  The iteration algorithm starts here
       iterations = 0;
       while( (iterations <= max_iter) && ( diff > 0.00001) ){
          u_temp = u; diff = 0.;
          for (j = 1; j<= n,j++){
              for(l = 1; l <= n; 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((n),2.0);
       }   // end while loop
```

Jacobi Algorithm for solving Laplace's Equation, to observe

The important part of the algorithm is applied in the function which sets up the two-dimensional Laplace equation. There we have a 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 a given maximum. We need also to provide a convergence tolerance. In the above program example we have fixed this to be 0.00001. Depending on the type of applications one may have to change both the number of maximum iterations and the tolerance.

Python code for solving the two-dimensional Laplace equation

The following Python code sets up and solves the Laplace equation in two dimensions.

```
    # Solves the 2d Laplace equation using relaxation method

    import numpy, math

    def relax(A, maxsteps, convergence):
        """
        Relaxes the matrix A until the sum of the absolute differences
        between the previous step and the next step (divided by the number of
        elements in A) is below convergence, or maxsteps is reached.

        Input:
         - A: matrix to relax
         - maxsteps, convergence: Convergence criterions

        Output:
         - A is relaxed when this method returns
        """

        iterations = 0
        diff = convergence +1

        Nx = A.shape[1]
        Ny = A.shape[0]
```

```
    while iterations < maxsteps and diff > convergence:
        #Loop over all *INNER* points and relax
        Atemp = A.copy()
        diff = 0.0

        for y in xrange(1,Ny-1):
            for x in xrange(1,Ny-1):
                A[y,x] = 0.25*(Atemp[y,x+1]+Atemp[y,x-1]+Atemp[y+1,x]+Atemp[y-1,x])
                diff  += math.fabs(A[y,x] - Atemp[y,x])

        diff /=(Nx*Ny)
        iterations += 1
        print "Iteration #", iterations, ", diff =", diff;


def boundary(A,x,y):
    """
    Set up boundary conditions

    Input:
     - A: Matrix to set boundaries on
     - x: Array where x[i] = hx*i, x[last_element] = Lx
     - y: Eqivalent array for y

    Output:
     - A is initialized in-place (when this method returns)
    """

    #Boundaries implemented (condensator with plates at y={0,Lx}, DeltaV = 200):
    # A(x,0)  =  100*sin(2*pi*x/Lx)
    # A(x,Ly) = -100*sin(2*pi*x/Lx)
    # A(0,y)  = 0
    # A(Lx,y) = 0

    Nx = A.shape[1]
    Ny = A.shape[0]
    Lx = x[Nx-1] #They *SHOULD* have same sizes!
    Ly = x[Nx-1]


    A[:,0]    =   100*numpy.sin(math.pi*x/Lx)
    A[:,Nx-1] = - 100*numpy.sin(math.pi*x/Lx)
    A[0,:]    = 0.0
    A[Ny-1,:] = 0.0


#Main program

import sys

# Input parameters
Nx      = 100
Ny      = 100
maxiter = 1000

x = numpy.linspace(0,1,num=Nx+2) #Also include edges
y = numpy.linspace(0,1,num=Ny+2)
A = numpy.zeros((Nx+2,Ny+2))

boundary(A,x,y)
```

```
#Remember: as solution "creeps" in from the edges,
#number of steps MUST AT LEAST be equal to
#number of inner meshpoints/2 (unless you have a better
#estimate for the solution than zeros() )
relax(A,maxiter,0.00001)

# To do: add visualization
```

Jacobi's algorithm extended to the diffusion equation in two dimensions

Let us know implement the implicit scheme and show how we can extend the previous algorithm for solving Laplace's or Poisson's equations to the diffusion equation as well. As the reader will notice, this simply implies a slight redefinition of the vector **b** defined in Eq. (16).

To see this, let us first set up the diffusion in two spatial dimensions, with boundary and initial conditions. The $2+1$-dimensional diffusion equation (with dimensionless variables) reads for a function $u = u(x, y, t)$

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

Jacobi's algorithm extended to the diffusion equation in two dimensions

We assume that we have a square lattice of length L with equally many mesh points in the x and y directions. Setting the diffusion constant $D = 1$ and using the shorthand notation $u_{xx} = \partial^2 u/\partial x^2$ etc for the second derivatives and $u_t = \partial u/\partial t$ for the time derivative, we have, with a given set of boundary and initial conditions,

$$\begin{array}{cc} u_t = u_{xx} + u_{yy} & x, y \in (0, L), t > 0 \\ u(x, y, 0) = g(x, y) & x, y \in (0, L) \\ u(0, y, t) = u(L, y, t) = u(x, 0, t) = u(x, L, t)0 & t > 0 \end{array}$$

Jacobi's algorithm extended to the diffusion equation in two dimensions, discretizing

We discretize again position and time, and use the following approximation for the second derivatives

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

which we rewrite as, in its discretized version,

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

where $x_i = x_0 + ih$, $y_j = y_0 + jh$ and $t_l = t_0 + l\Delta t$, with $h = L/(n+1)$ and Δt the time step.

Jacobi's algorithm extended to the diffusion equation in two dimensions, the second derivative

The second derivative with respect to y reads

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

We use now the so-called backward going Euler formula for the first derivative in time. In its discretized form we have

$$u_t \approx \frac{u^l_{i,j} - u^{l-1}_{i,j}}{\Delta t},$$

resulting in

$$u^l_{i,j} + 4\alpha u^l_{i,j} - \alpha\left[u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right] = u^{l-1}_{i,j},$$

where the right hand side is the only known term, since starting with $t = t_0$, the right hand side is entirely determined by the boundary and initial conditions. We have $\alpha = \Delta t/h^2$.

Jacobi's algorithm extended to the diffusion equation in two dimensions

For future time steps, only the boundary values are determined and we need to solve the equations for the interior part in an iterative way similar to what was done for Laplace's or Poisson's equations. To see this, we rewrite the previous equation as

$$u^l_{i,j} = \frac{1}{1+4\alpha}\left[\alpha(u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}) + u^{l-1}_{i,j}\right],$$

or in a more compact form as

$$u^l_{i,j} = \frac{1}{1+4\alpha}\left[\alpha\Delta^l_{ij} + u^{l-1}_{i,j}\right], \tag{18}$$

with $\Delta^l_{ij} = \left[u^l_{i,j+1} + u^l_{i,j-1} + u^l_{i+1,j} + u^l_{i-1,j}\right]$. This equation has essentially the same structure as Eq. (15), except that the function ρ_{ij} is replaced by the solution at a previous time step $l-1$. Furthermore, the diagonal matrix elements are now given by $1+4\alpha$, while the non-zero non-diagonal matrix elements equal α. This matrix is also positive definite, meaning in turn that iterative schemes like the Jacobi or the Gauss-Seidel methods will converge to the desired solution after a given number of iterations.

Solving project 1 again but now with Jacobi's method

Let us revisit project 1 and the Thomas algorithm for solving a system of tridiagonal matrices for the equation

```
//  Solves linear equations for simple tridiagonal matrix using the iterative Jacobi method
....
// Begin main program
int main(int argc, char *argv[]){
     // missing statements, see code link above

      mat A = zeros<mat>(n,n);
      // Set up arrays for the simple case
      vec b(n);  vec x(n);
      A(0,1) = -1;  x(0) = h;  b(0) =  hh*f(x(0));
      x(n-1) = x(0)+(n-1)*h; b(n-1) = hh*f(x(n-1));
      for (int i = 1; i < n-1; i++){
        x(i) = x(i-1)+h;
        b(i) = hh*f(x(i));
        A(i,i-1)  = -1.0;
        A(i,i+1)  = -1.0;
      }
     A(n-2,n-1) = -1.0; A(n-1,n-2) = -1.0;
  // solve Ax = b by iteration with a random starting vector
     int maxiter = 100; double diff = 1.0;
     double epsilon = 1.0e-10;  int iter = 0;
      vec SolutionOld  = randu<vec>(n);
      vec SolutionNew  = zeros<vec>(n);
      while (iter <= maxiter || diff > epsilon){
        SolutionNew = (b -A*SolutionOld)*0.5;
        iter++; diff = fabs(sum(SolutionNew-SolutionOld)/n);
        SolutionOld = SolutionNew;
      }
      vec solution = SolutionOld;}
```

Program to solve Jacobi's method in two dimension

The following program sets up the diffusion equation solver in two spatial dimensions using Jacobi's method. Note that we have skipped a loop over time. This has to be inserted in order to perform the calculations.

```
/* Simple program for solving the two-dimensional diffusion
   equation or Poisson equation using Jacobi's iterative method
   Note that this program does not contain a loop over the time
   dependence.
*/

#include <iostream>
#include <iomanip>
#include <armadillo>
using namespace std;
using namespace arma;

int JacobiSolver(int, double, double, mat &, mat &, double);

int main(int argc, char * argv[]){
  int Npoints = 40;
  double ExactSolution;
  double dx = 1.0/(Npoints-1);
```

```
  double dt = 0.25*dx*dx;
  double tolerance = 1.0e-14;
  mat A = zeros<mat>(Npoints,Npoints);
  mat q = zeros<mat>(Npoints,Npoints);

  // setting up an additional source term
  for(int i = 0; i < Npoints; i++)
    for(int j = 0; j < Npoints; j++)
      q(i,j) = -2.0*M_PI*M_PI*sin(M_PI*dx*i)*sin(M_PI*dx*j);

  int itcount = JacobiSolver(Npoints,dx,dt,A,q,tolerance);

  // Testing against exact solution
  double sum = 0.0;
  for(int i = 0; i < Npoints; i++){
    for(int j=0;j < Npoints; j++){
      ExactSolution = -sin(M_PI*dx*i)*sin(M_PI*dx*j);
      sum += fabs((A(i,j) - ExactSolution));
    }
  }
  cout << setprecision(5) << setiosflags(ios::scientific);
  cout << "Jacobi method with error " << sum/Npoints << " in " << itcount << " iterations" << endl
}
```

The Jacobi solver function

```
// Function for setting up the iterative Jacobi solver
int JacobiSolver(int N, double dx, double dt, mat &A, mat &q, double abstol)
{
  int MaxIterations = 100000;
  mat Aold = zeros<mat>(N,N);

  double D = dt/(dx*dx);

  for(int i=1;  i < N-1; i++)
    for(int j=1; j < N-1; j++)
      Aold(i,j) = 1.0;

  // Boundary Conditions -- all zeros
  for(int i=0; i < N; i++){
    A(0,i) = 0.0;
    A(N-1,i) = 0.0;
    A(i,0) = 0.0;
    A(i,N-1) = 0.0;
  }
  // Start the iterative solver
  for(int k = 0; k < MaxIterations; k++){
    for(int i = 1; i < N-1; i++){
      for(int j=1; j < N-1; j++){
        A(i,j) = dt*q(i,j) + Aold(i,j) +
          D*(Aold(i+1,j) + Aold(i,j+1) - 4.0*Aold(i,j) +
             Aold(i-1,j) + Aold(i,j-1));
      }
    }
    double sum = 0.0;
    for(int i = 0; i < N;i++){
      for(int j = 0; j < N;j++){
        sum += (Aold(i,j)-A(i,j))*(Aold(i,j)-A(i,j));
        Aold(i,j) = A(i,j);
      }
```

```
    }
    if(sqrt (sum) <abstol){
      return k;
    }
  }
  cerr << "Jacobi: Maximum Number of Interations Reached Without Convergence\n";
  return MaxIterations;
}
```

Parallel Jacobi

In order to parallelize the Jacobi method we need to introduce to new **MPI** functions, namely *MPIGather* and *MPIAllgather*.

Here we present a parallel implementation of the Jacobi method without an explicit link to the diffusion equation. Let us go back to the plain Jacobi method and implement it in parallel.

```
//  Main program first
#include <mpi.h>

//  Omitted statements
int main(int argc, char * argv[]){
  int i,j, N = 20;
  double **A,*x,*q;
  int totalnodes,mynode;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  if(mynode==0){

  }
  ParallelJacobi(mynode,totalnodes,N,A,x,q,1.0e-14);
  if(mynode==0){
    for(int i = 0; i < N; i++)
      cout << x[i] << endl;

  }
  MPI_Finalize();
}
```

Parallel Jacobi

Here follows the parallel implementation of the Jacobi algorithm

```
int ParallelJacobi(int mynode, int numnodes, int N, double **A, double *x, double *b, double absto
  int i,j,k,i_global;
  int maxit = 100000;
  int rows_local,local_offset,last_rows_local,*count,*displacements;
  double sum1,sum2,*xold;
  double error_sum_local, error_sum_global;
  MPI_Status status;

  rows_local = (int) floor((double)N/numnodes);
  local_offset = mynode*rows_local;
  if(mynode == (numnodes-1))
```

```
  rows_local = N - rows_local*(numnodes-1);

/*Distribute the Matrix and R.H.S. among the processors */
if(mynode == 0){
  for(i=1;i<numnodes-1;i++){
    for(j=0;j<rows_local;j++)
      MPI_Send(A[i*rows_local+j],N,MPI_DOUBLE,i,j,MPI_COMM_WORLD);
    MPI_Send(b+i*rows_local,rows_local,MPI_DOUBLE,i,rows_local,
             MPI_COMM_WORLD);
  }
  last_rows_local = N-rows_local*(numnodes-1);
  for(j=0;j<last_rows_local;j++)
    MPI_Send(A[(numnodes-1)*rows_local+j],N,MPI_DOUBLE,numnodes-1,j,
             MPI_COMM_WORLD);
  MPI_Send(b+(numnodes-1)*rows_local,last_rows_local,MPI_DOUBLE,numnodes-1,
           last_rows_local,MPI_COMM_WORLD);
}
else{
  A = CreateMatrix(rows_local,N);
  x = new double[rows_local];
  b = new double[rows_local];
  for(i=0;i<rows_local;i++)
    MPI_Recv(A[i],N,MPI_DOUBLE,0,i,MPI_COMM_WORLD,&status);
  MPI_Recv(b,rows_local,MPI_DOUBLE,0,rows_local,MPI_COMM_WORLD,&status);
}


xold = new double[N];
count = new int[numnodes];
displacements = new int[numnodes];


//set initial guess to all 1.0
for(i=0; i<N; i++){
  xold[i] = 1.0;
}

for(i=0;i<numnodes;i++){
  count[i] = (int) floor((double)N/numnodes);
  displacements[i] = i*count[i];
}
count[numnodes-1] = N - ((int)floor((double)N/numnodes))*(numnodes-1);

for(k=0; k<maxit; k++){
  error_sum_local = 0.0;
  for(i = 0; i<rows_local; i++){
    i_global = local_offset+i;
    sum1 = 0.0; sum2 = 0.0;
    for(j=0; j < i_global; j++)
      sum1 = sum1 + A[i][j]*xold[j];
    for(j=i_global+1; j < N; j++)
      sum2 = sum2 + A[i][j]*xold[j];

    x[i] = (-sum1 - sum2 + b[i])/A[i][i_global];
    error_sum_local += (x[i]-xold[i_global])*(x[i]-xold[i_global]);
  }

  MPI_Allreduce(&error_sum_local,&error_sum_global,1,MPI_DOUBLE,
                MPI_SUM,MPI_COMM_WORLD);
  MPI_Allgatherv(x,rows_local,MPI_DOUBLE,xold,count,displacements,
                 MPI_DOUBLE,MPI_COMM_WORLD);
```

```
    if(sqrt(error_sum_global)<abstol){
      if(mynode == 0){
        for(i=0;i<N;i++)
          x[i] = xold[i];
      }
      else{
        DestroyMatrix(A,rows_local,N);
        delete[] x;
        delete[] b;
      }
      delete[] xold;
      delete[] count;
      delete[] displacements;
      return k;
    }
  }

  cerr << "Jacobi: Maximum Number of Interations Reached Without Convergence\n";
  if(mynode == 0){
    for(i=0;i<N;i++)
      x[i] = xold[i];
  }
  else{
    DestroyMatrix(A,rows_local,N);
    delete[] x;
    delete[] b;
  }
  delete[] xold;
  delete[] count;
  delete[] displacements;

  return maxit;
}
```

Parallel Jacobi

Here follows the parallel implementation of the diffusion equation using OpenMP

```
/* Simple program for solving the two-dimensional diffusion
   equation or Poisson equation using Jacobi's iterative method
   Note that this program does not contain a loop over the time
   dependence. It uses OpenMP to parallelize
*/

#include <iostream>
#include <iomanip>
#include <armadillo>
#include <omp.h>
using namespace std;
using namespace arma;

int JacobiSolver(int, double, double, mat &, mat &, double);

int main(int argc, char * argv[]){
  int Npoints = 100;
  double ExactSolution;
  double dx = 1.0/(Npoints-1);
  double dt = 0.25*dx*dx;
  double tolerance = 1.0e-8;
```

```
  mat A = zeros<mat>(Npoints,Npoints);
  mat q = zeros<mat>(Npoints,Npoints);

  int thread_num;
  omp_set_num_threads(4);
  thread_num = omp_get_max_threads ();
  cout << "  The number of processors available = " << omp_get_num_procs () << endl ;
  cout << "  The number of threads available    = " << thread_num <<  endl;
  // setting up an additional source term
  for(int i = 0; i < Npoints; i++)
    for(int j = 0; j < Npoints; j++)
      q(i,j) = -2.0*M_PI*M_PI*sin(M_PI*dx*i)*sin(M_PI*dx*j);

  int itcount = JacobiSolver(Npoints,dx,dt,A,q,tolerance);

  // Testing against exact solution
  double sum = 0.0;
  for(int i = 0; i < Npoints; i++){
    for(int j=0;j < Npoints; j++){
      ExactSolution = -sin(M_PI*dx*i)*sin(M_PI*dx*j);
      sum += fabs((A(i,j) - ExactSolution));
    }
  }
  cout << setprecision(5) << setiosflags(ios::scientific);
  cout << "Jacobi error is " << sum/Npoints << " in " << itcount << " iterations" << endl;
}


// Function for setting up the iterative Jacobi solver
int JacobiSolver(int N, double dx, double dt, mat &A, mat &q, double abstol)
{
  int MaxIterations = 100000;


  double D = dt/(dx*dx);
  // initial guess
  mat Aold = randu<mat>(N,N);
  // Boundary conditions, all zeros
  for(int i=0; i < N; i++){
    A(0,i) = 0.0;
    A(N-1,i) = 0.0;
    A(i,0) = 0.0;
    A(i,N-1) = 0.0;
  }
  double sum = 1.0;
  int k = 0;
  // Start the iterative solver
  while (k < MaxIterations && sum > abstol){
  int i, j;
  sum = 0.0;
  // Define parallel region
# pragma omp parallel default(shared) private (i, j) reduction(+:sum)
  {
     # pragma omp for
    for(i = 1; i < N-1; i++){
      for(j = 1; j < N-1; j++){
        A(i,j) = dt*q(i,j) + Aold(i,j) +
          D*(Aold(i+1,j) + Aold(i,j+1) - 4.0*Aold(i,j) +
             Aold(i-1,j) + Aold(i,j-1));
      }
    }
```

```
    for(i = 0; i < N;i++){
      for(j = 0; j < N;j++){
        sum += fabs(Aold(i,j)-A(i,j));
        Aold(i,j) = A(i,j);
      }
    }
    sum /= (N*N);
  } //end parallel region
  k++;
  } //end while loop
  return k;
}
```

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

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

Wave Equation in two Dimensions, discretizing

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

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right). \tag{19}$$

Wave Equation in two Dimensions

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 $u_{i,+1} = u_{i,-1}$.

Wave Equation in two Dimensions

If we insert this condition in Eq. (19) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right). \tag{20}$$

We need seemingly two different equations, one for the first time step given by Eq. (20) and one for all other time-steps given by Eq. (19). However, it suffices to use Eq. (19) 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.

Wave Equation in two Dimensions

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

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

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

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

Wave Equation in two Dimensions

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

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t^2}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \quad (21)$$

where again we have an explicit scheme with $u^{l+1}_{i,j}$ as the only unknown quantity.

Wave Equation in two Dimensions

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.

Analytical Solution for the two-dimensional wave equation

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

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

Analytical Solution for the two-dimensional wave equation, first step

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

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

Analytical Solution for the two-dimensional wave equation,

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

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

Analytical Solution for the two-dimensional wave equation, separation of variables

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

Analytical Solution for the two-dimensional wave equation, separation of variables

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

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

Analytical Solution for the two-dimensional wave equation, boundary conditions

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 $\lambda_{mn} = c\pi/L\sqrt{m^2 + n^2}$.

Analytical Solution for the two-dimensional wave equation, separation of variables and solutions

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

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

Analytical Solution for the two-dimensional wave equation, final steps

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

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

and

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

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

Python code for solving the two-dimensional wave equation

The following Python code sets up and solves the two-dimensional wave equation for all three methods discussed.

```
#Program which solves the 2+1-dimensional wave equation by a finite difference scheme

from numpy import *
#Define the grid
N = 31
h = 1.0 / (N-1)
dt = .0005
t_steps = 10000
x,y = ndgrid(linspace(0,1,N),linspace(0,1,N),sparse=False)

alpha = dt**2 / h**2

#Initial conditions with du/dt = 0
u = sin(x*pi)*cos(y*pi-pi/2)
u_old = zeros(u.shape,type(u[0,0]))
for i in xrange(1,N-1):
    for j in xrange(1,N-1):
        u_old[i,j] = u[i,j] + (alpha/2)*(u[i+1,j] - 4*u[i,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
u_new = zeros(u.shape,type(u[0,0]))

#We don't necessarily want to plot every time step. We plot every n'th step where
n = 100
plotnr = 0

#Iteration over time steps
for k in xrange(t_steps):
    for i in xrange(1,N-1): #1 - N-2 because we don't want to change the boundaries
        for j in xrange(1,N-1):
            u_new[i,j] = 2*u[i,j] - u_old[i,j] + alpha*(u[i+1,j] - 4*u[i,j] + u[i-1,j] + u[i,j+1]

    #Prepare for next time step by manipulating pointers
    temp = u_new
    u_new = u_old
    u_old = u
    u = temp
#To do: Make movie
```