Finite difference methods for diffusion processes

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 22, 2015

Note: **PRELIMINARY VERSION**

Contents

The famous *diffusion equation*, also known as the *heat equation*, reads

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

where $u(x,t)$ is the unknown function to be solved for, x is a coordinate in space, and t is time. The coefficient α is the *diffusion coefficient* and determines how fast u changes in time. A quick short form for the diffusion equation is $u_t = \alpha u_{xx}$.

Compared to the wave equation, $u_{tt} = c^2 u_{xx}$, which looks very similar, the diffusion equation features solutions that are very different from those of the wave equation. Also, the diffusion equation makes quite different demands to the numerical methods.

Typical diffusion problems may experience rapid change in the very beginning, but then the evolution of u becomes slower and slower. The solution is usually very smooth, and after some time, one cannot recognize the initial shape of u. This is in sharp contrast to solutions of the wave equation where the initial shape is preserved - the solution is basically a moving initial condition. The standard wave equation $u_{tt} = c^2 u_{xx}$ has solutions that propagates with speed c forever, without changing shape, while the diffusion equation converges to a *stationary solution* $\bar{u}(x)$ as $t \to \infty$. In this limit, $u_t = 0$, and $\bar{u}$ is governed by $\bar{u}''(x) = 0$. This stationary limit of the diffusion equation is called the *Laplace* equation and arises in a very wide range of applications throughout the sciences.

It is possible to solve for $u(x,t)$ using a explicit scheme, as we do in Section 1, but the time step restrictions soon become much less favorable than for an explicit scheme for the wave equation. And of more importance, since the solution u of the diffusion equation is very smooth and changes slowly, small time steps are not convenient and not required by accuracy as the diffusion process converges to a stationary state. Therefore, implicit schemes as described in Section 2 are popular, but these require solutions of systems of algebraic equations. We shall

use ready-made software for this purpose, but also program some simple iterative methods.

1 An explicit method for the 1D diffusion equation

1.1 The initial-boundary value problem for 1D diffusion

To obtain a unique solution of the diffusion equation, or equivalently, to apply numerical methods, we need initial and boundary conditions. The diffusion equation goes with one initial condition $u(x,0) = I(x)$, where I is a prescribed function. One boundary condition is required at each point on the boundary, which in 1D means that u must be known, u_x must be known, or some combination of them.

We shall start with the simplest boundary condition: $u = 0$. The complete initial-boundary value diffusion problem in one space dimension can then be specified as

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f, \quad x \in (0,L),\ t \in (0,T] \tag{1}$$

$$u(x,0) = I(x), \qquad x \in [0,L] \tag{2}$$

$$u(0,t) = 0, \qquad t > 0, \tag{3}$$

$$u(L,t) = 0, \qquad t > 0\,. \tag{4}$$

With only a first-order derivative in time, only one *initial condition* is needed, while the second-order derivative in space leads to a demand for two *boundary conditions*. We have added a source term $f = f(x,t)$ for convenience when testing implementations.

Diffusion equations like (1) have a wide range of applications throughout physical, biological, and financial sciences. One of the most common applications is propagation of heat, where $u(x,t)$ represents the temperature of some substance at point x and time t.

1.2 Forward Euler scheme

The first step in the discretization procedure is to replace the domain $[0,L]\times[0,T]$ by a set of mesh points. Here we apply equally spaced mesh points

$$x_i = i\Delta x, \quad i = 0,\ldots,N_x,$$

and

$$t_n = n\Delta t, \quad n = 0,\ldots,N_t\,.$$

Moreover, u_i^n denotes the mesh function that approximates $u(x_i, t_n)$ for $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$. Requiring the PDE (1) to be fulfilled at a mesh point (x_i, t_n) leads to the equation

$$\frac{\partial}{\partial t}u(x_i, t_n) = \alpha \frac{\partial^2}{\partial x^2}u(x_i, t_n) + f(x_i, t_n), \tag{5}$$

The next step is to replace the derivatives by finite difference approximations. The computationally simplest method arises from using a forward difference in time and a central difference in space:

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n \,. \tag{6}$$

Written out,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \,. \tag{7}$$

We have turned the PDE into algebraic equations, also often called discrete equations. The key property of the equations is that they are algebraic, which makes them easy to solve. As usual, we anticipate that u_i^n is already computed such that u_i^{n+1} is the only unknown in (7). Solving with respect to this unknown is easy:

$$u_i^{n+1} = u_i^n + F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t f_i^n, \tag{8}$$

where we have introduced the *mesh Fourier number*:

$$F = \alpha \frac{\Delta t}{\Delta x^2} \,. \tag{9}$$

F is the key parameter in the discrete diffusion equation.

Note that F is a *dimensionless* number that lumps the key physical parameter in the problem, α, and the discretization parameters Δx and Δt into a single parameter. All the properties of the numerical method are critically dependent upon the value of F (see Section 3 for details).

The computational algorithm then becomes

1. compute $u_i^0 = I(x_i)$for $i = 0, \ldots, N_x$
2. for $n = 0, 1, \ldots, N_t$:
   (a) apply (8) for all the internal spatial points $i = 1, \ldots, N_x - 1$
   (b) set the boundary values $u_i^{n+1} = 0$ for $i = 0$ and $i = N_x$

The algorithm is compactly fully specified in Python:

```
x = linspace(0, L, Nx+1)    # mesh points in space
dx = x[1] - x[0]
t = linspace(0, T, Nt+1)    # mesh points in time
dt = t[1] - t[0]
F = a*dt/dx**2
u   = zeros(Nx+1)           # unknown u at new time level
u_1 = zeros(Nx+1)           # u at the previous time level

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

for n in range(0, Nt):
    # Compute u at inner mesh points
    for i in range(1, Nx):
        u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               f(x[i], t[n])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0

    # Update u_1 before next step
    u_1[:]= u
```

Note that we use `a` for α in the code, motivated by easy visual mapping between the variable name and the mathematical symbol in formulas.

We need to state already now that the shown algorithm does not produce meaningful results unless $F \leq 1/2$. Why is explained in Section 3.

1.3 Implementation

The file `diffu1D_u0.py`[1] contains a complete function `solver_FE_simple` for solving the 1D diffusion equation with $u = 0$ on the boundary as specified in the algorithm above:

```
import numpy as np
import time

def solver_FE_simple(I, a, f, L, dt, F, T):
    """
    Simplest expression of the computational algorithm
    using the Forward Euler method and explicit Python loops.
    f must be a Python function of x and t. If None, a
    default f=0 is used.
    """
    import time
    t0 = time.clock()

    dt = float(dt)                # avoid integer division
    Nt = int(round(T/dt))
    t = np.linspace(0, T, Nt+1)   # mesh points in time
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)   # mesh points in space

    if f is None:
        f = lambda x, t: 0
```

[1] `http://tinyurl.com/nm5587k/diffu/diffu1D_u0.py`

```
    u   = np.zeros(Nx+1)
    u_1 = np.zeros(Nx+1)

    # Set initial condition u(x,0) = I(x)
    for i in range(0, Nx+1):
        u_1[i] = I(x[i])

    for n in range(0, Nt):
        # Compute u at inner mesh points
        for i in range(1, Nx):
            u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                   dt*f(x[i], t[n])

        # Insert boundary conditions
        u[0] = 0;  u[Nx] = 0

        # Switch variables before next step
        u_1, u = u, u_1

    t1 = time.clock()
    # Return u_1 as u since we set u_1=u above
    return u_1, x, t, t1-t0
```

A faster version, based on vectorization of the finite difference scheme, is available in the function `solver_FE`. The vectorized version replaces the explicit loop

```
for i in range(1, Nx):
    u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) \
           + dt*f(x[i], t[n])
```

by arithmetics on displaced slices of the `u` array:

```
u[1:Nx] = u_1[1:Nx] + F*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1]) \
          + dt*f(x[1:Nx], t[n])
# or
u[1:-1] = u_1[1:-1] + F*(u_1[0:-2] - 2*u_1[1:-1] + u_1[2:]) \
          + dt*f(x[1:-1], t[n])
```

For example, the vectorized version runs 70 times faster than the scalar version in a case with 100 time steps and a spatial mesh of 10^5 cells.

The `solver_FE` function also features a callback function such that the user can process the solution at each time level. The callback function looks like `user_action(u, x, t, n)`, where `u` is the array containing the solution at time level `n`, `x` holds all the spatial mesh points, while `t` holds all the temporal mesh points. Apart from the vectorized loop over the spatial mesh points, the callback function, and a bit more complicated setting of the source `f` it is not specified (`None`), the `solver_FE` is identical to `solver_FE_simple` above:

```
def solver_FE(I, a, f, L, dt, F, T,
              user_action=None, version='scalar'):
    """
    Vectorized implementation of solver_FE_simple.
    If version='vectorized', f must be a vectorized
    function of x and t (if f is None, a default version
    f=0 is made).
```

```
    """
    import time
    t0 = time.clock()

    dt = float(dt)                # avoid integer division
    Nt = int(round(T/dt))
    t = np.linspace(0, T, Nt+1)   # mesh points in time
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)   # mesh points in space

    if f is None:
        if version == 'scalar':
            f = lambda x, t: 0
        else:
            f = lambda x, t: np.zeros(len(x))

    u   = np.zeros(Nx+1)   # solution array
    u_1 = np.zeros(Nx+1)   # solution at t-dt

    # Set initial condition
    for i in range(0,Nx+1):
        u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, 0)

    for n in range(0, Nt):
        # Update all inner points
        if version == 'scalar':
            for i in range(1, Nx):
                u[i] = u_1[i] + F*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                       dt*f(x[i], t[n])

        elif version == 'vectorized':
            u[1:Nx] = u_1[1:Nx] +  \
                      F*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1]) + \
                      dt*f(x[1:Nx], t[n])
        else:
            raise ValueError('version=%s' % version)

        # Insert boundary conditions
        u[0] = 0;  u[Nx] = 0
        if user_action is not None:
            user_action(u, x, t, n+1)

        # Update u_1 before next step
        #u_1[:]= u
        u_1, u = u, u_1

    t1 = time.clock()
    # Return u_1 as solution since we set u_1=u above
    return u_1, x, t, t1-t0
```

1.4 Verification

Before thinking about running the functions in the previous section, we need to construct a suitable test example for verification. It appears that a manufactured solution that is linear in time and at most quadratic in space fulfills the Forward

Euler scheme exactly. With the restriction that $u = 0$ for $x = 0, L$, we can try the solution

$$u(x,t) = 5tx(L-x)\,.$$

Inserted in the PDE, it requires a source term

$$f(x,t) = 10\alpha t + 5x(L-x)\,.$$

Let us check that the manufactured `u` fulfills the scheme:

$$\begin{aligned}[D_t^+ u = \alpha D_x D_x u + f]_i^n &= [5x(L-x)D_t^+ t = 5t\alpha D_x D_x(xL - x^2) + 10\alpha t + 5x(L-x)]_i^n \\ &= [5x(L-x) = 5t\alpha(-2) + 10\alpha t + 5x(L-x)]_i^n\,.\end{aligned}$$

The computation of the source term, given any u, is easily automated with SymPy:

```
import sympy as sym
x, t, a, L = sym.symbols('x t a L')
u = x*(L-x)*5*t

def pde(u):
    return sym.diff(u, t) - a*sym.diff(u, x, x)

f = sym.simplify(pde(u))
```

Now we can choose any expression for `u` and automatically get the suitable source term `f`.

The numerical code will need to access the `u` and `f` above as Python function. The exact solution is wanted as a Python function `u_exact(x, t)`, while the source term is wanted as `f(x, t)`. The parameters `a` and `L` in `u` and `f` above are symbols and must be replaced by `float` objects in a Python function. This can be done by redefining `a` and `L` as `float` objects and performing substitutions of symbols by numbers in `u` and `f`. The appropriate code looks like this:

```
a = 0.5
L = 1.5
u_exact = sym.lambdify(
    [x, t], u.subs('L', L).subs('a', a), modules='numpy')
f = sym.lambdify(
    [x, t], f.subs('L', L).subs('a', a), modules='numpy')
I = lambda x: u_exact(x, 0)
```

Here we also make a function `I` for the initial condition.

The idea now is that our manufactured solution should be exactly reproduced by the code (to machine precision). For this purpose we make a test function for comparing the exact and numerical solutions at the end of the time interval:

```
def test_solver_FE():
    # Define u_exact, f, I as explained above

    dx = L/3  # 3 cells
    F = 0.5
    dt = F*dx**2

    u, x, t, cpu = solver_FE_simple(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2)
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    assert diff < tol, 'max diff solver_FE_simple: %g' % diff

    u, x, t, cpu = solver_FE(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2,
        user_action=None, version='scalar')
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    assert diff < tol, 'max diff solver_FE, scalar: %g' % diff

    u, x, t, cpu = solver_FE(
        I=I, a=a, f=f, L=L, dt=dt, F=F, T=2,
        user_action=None, version='vectorized')
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    assert diff < tol, 'max diff solver_FE, vectorized: %g' % diff
```

We emphasize that the value `F=0.5` is critical: the tests above will fail if `F` has a larger value (this is because the Forward Euler scheme is unstable for $F > 1/2$).

1.5 Numerical experiments

When a test function like the one above runs silently without errors, we have some evidence for a correct implementation of the numerical method. The next step is to do some experiments with more interesting solutions.

We target a scaled diffusion problem where x/L is a new spatial coordinate and $\alpha t/L^2$ is a new time coordinate. The source term f is omitted, and u is scaled by $\max_{x\in[0,L]} |I(x)|$ (see Section 3.2 in [1] for details). The governing PDE is then

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

in the spatial domain $[0, L]$, with boundary conditions $u(0) = u(1) = 0$. Two initial conditions will be tested: a discontinuous plug,

$$I(x) = \begin{cases} 0, & |x - L/2| > 0.1 \\ 1, & \text{otherwise} \end{cases}$$

and a smooth Gaussian function,

$$I(x) = e^{-\frac{1}{2\sigma^2}(x-L/2)^2}.$$

The functions `plug` and `gaussian` in `diffu1D_u0.py`[2] run the two cases, respectively:

```
def plug(solver='solver_FE', F=0.5, dt=0.0002):
    """Plug profile as initial condition."""
    L = 1.
    a = 1
    T = 0.1

    def I(x):
        return 0 if abs(x-L/2.0) > 0.1 else 1

    u, cpu = viz(I, a, None, L, dt, F, T, umin=-0.1, umax=1.1,
                 solver=solver, animate=True, framefiles=True)
    return u

def gaussian(solver='solver_FE', F=0.5, dt=0.0002, sigma=0.1):
    """Gaussian profile as initial condition."""
    L = 1.
    a = 1
    T = 0.1

    def I(x):
        return np.exp(-0.5*((x-L/2.0)**2)/sigma**2)

    u, cpu = viz(I, a, None, L, dt, F, T, umin=-0.1, umax=1.1,
                 solver=solver, animate=True, framefiles=True)
    return u
```

These functions make use of the function `viz` for running the solver and visualizing the solution using a callback function with plotting:

```
def viz(I, a, f, L, dt, F, T, umin, umax,
        solver='solver_FE', animate=True, framefiles=True):

    solutions = []
    from scitools.std import plot, savefig

    def plot_u(u, x, t, n):
        if n == 0:
            # Store x and t first in solutions
            solutions.append(x)
            solutions.append(t)
        solutions.append(u.copy())
        plot(x, u, 'r-', axis=[0, L, umin, umax], title='t=%f' % t[n])
        if framefiles:
            savefig('tmp_frame%04d.png' % n)
            if n in [0, 2, 5, 10, 25, 50, 75, 100, 250, 500]: savefig('tmp_frame%04d.pdf' % n)
        if t[n] == 0:
            time.sleep(2)
        elif not framefiles:
            # It takes time to write files so pause is needed
            # for screen only animation
            time.sleep(0.2)

    user_action = plot_u if animate else lambda u,x,t,n: None

    u, x, t, cpu = eval(solver)(I, a, f, L, dt, F, T,
                                user_action=user_action)
    return solutions, cpu
```

[2] `http://tinyurl.com/nm5587k/diffu/diffu1D_u0.py`

Notice that this `viz` function stores all the solutions in a list `solutions` in the callback function. Modern computers have hardly any problem with storing a lot of such solutions for moderate values of N_x in 1D problems, but for 2D and 3D problems, this technique cannot be used and solutions must be stored in files.

hpl 1: Better to show the scalable file solution here?

Our experiments employs a time step $\Delta t = 0.0002$ and simulate for $t \in [0, 0.1]$. First we try the highest value of F: $F = 0.5$. This resolution corresponds to $N_x = 50$. A possible terminal command is

Terminal

```
Terminal> python -c 'from diffu1D_u0 import gaussian
> gaussian("solver_FE", F=0.5, dt=0.0002)'
```

The $u(x,t)$ curve as a function of x is shown in Figure 1 at four time levels (see also a movie[3]).

We see that the curves have saw-tooth waves in the beginning of the simulation. This non-physical noise is smoothed out with time, but solutions of the diffusion equations are known to be smooth, and this numerical solution is definitely not smooth. Lowering F helps: $F \leq 0.25$ gives a smooth solution, see Figure 2 (and a movie[4]).

Increasing F slightly beyond the limit 0.5, to $F = 0.51$, gives growing, non-physical instabilities, as seen in Figure 3.

Instead of a discontinuous initial condition we now try the smooth Gaussian function for $I(x)$. A simulation for $F = 0.5$ is shown in Figure 4. Now the numerical solution is smooth for all times, and this is true for any $F \leq 0.5$.

Experiments with these two choices of $I(x)$ reveal some important observations:

- The Forward Euler scheme leads to growing solutions if $F > \frac{1}{2}$.
- $I(x)$ as a discontinuous plug leads to a saw tooth-like noise for $F = \frac{1}{2}$, which is absent for $F \leq \frac{1}{4}$.
- The smooth Gaussian initial function leads to a smooth solution for all relevant F values ($F \geq \frac{1}{2}$).

[3] `http://tinyurl.com/opdfafk/pub/mov-diffu/diffu1D_u0_FE_plug/movie.ogg`
[4] `http://tinyurl.com/opdfafk/pub/mov-diffu/diffu1D_u0_FE_plug_F025/movie.ogg`

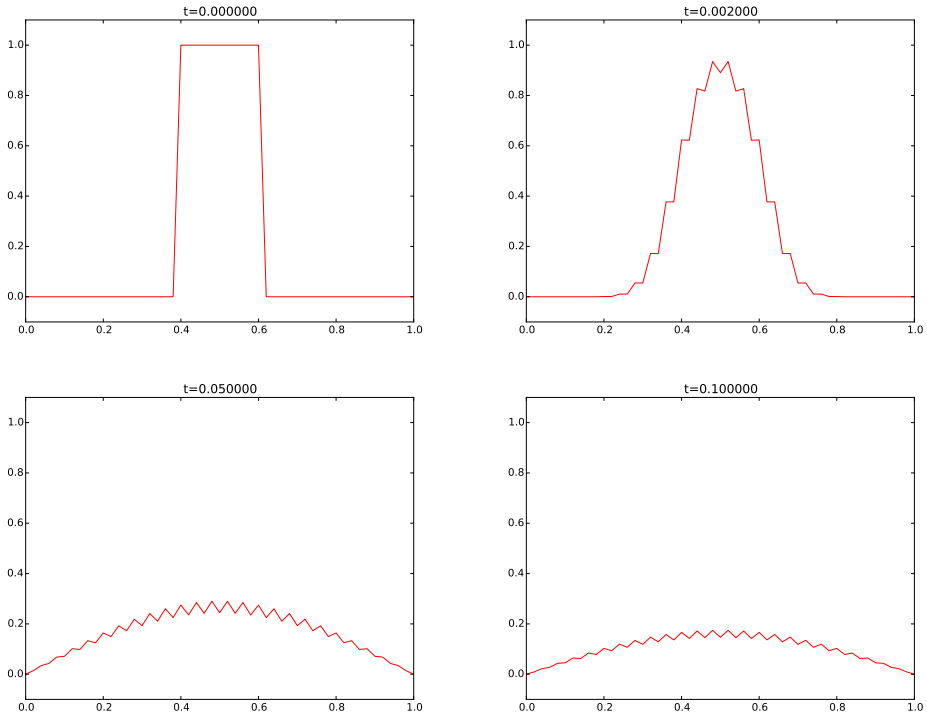

Figure 1: Forward Euler scheme for $F = 0.5$.

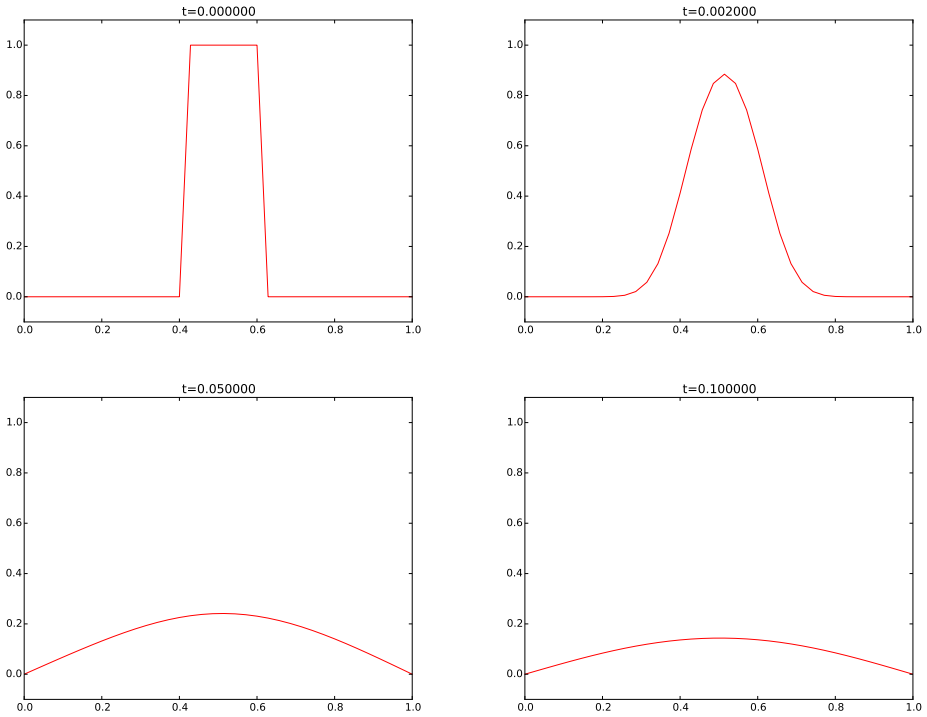

Figure 2: Forward Euler scheme for $F = 0.25$.

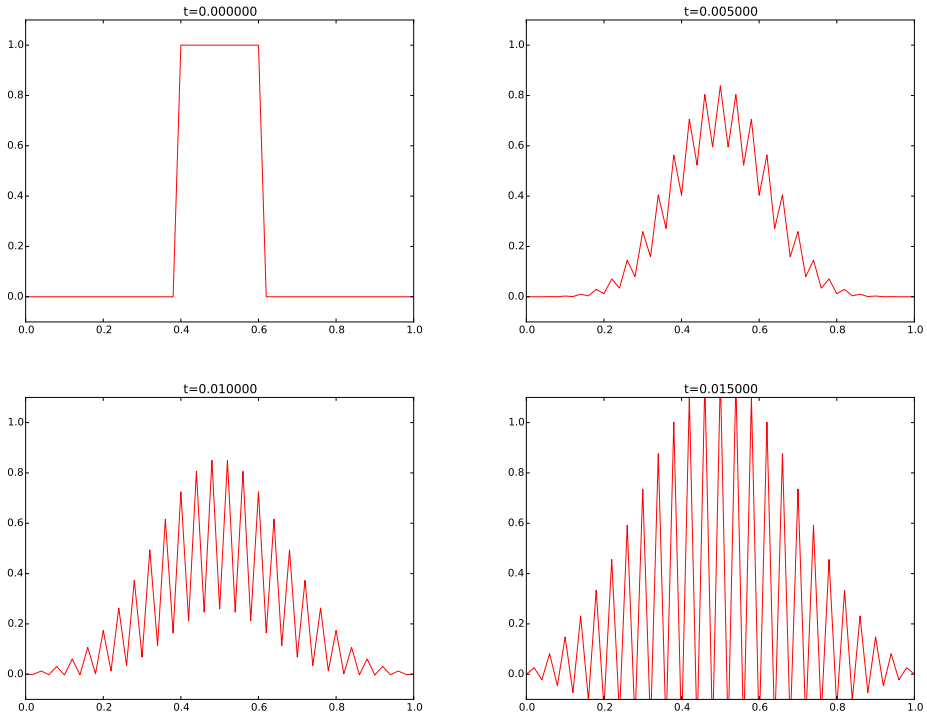

Figure 3: Forward Euler scheme for $F = 0.51$.

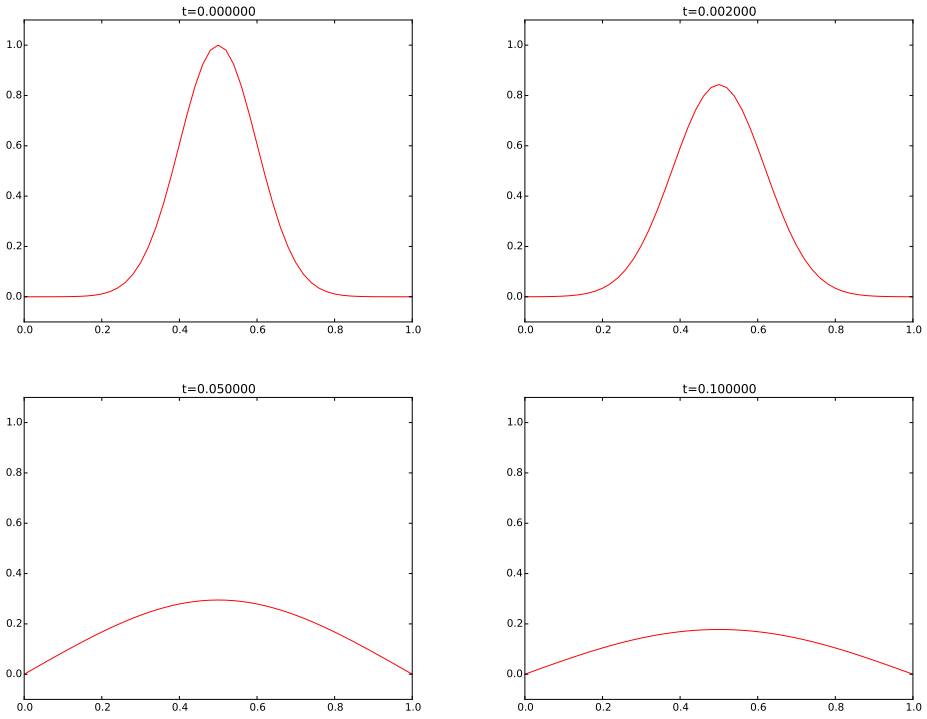

Figure 4: Forward Euler scheme for $F = 0.5$.

2 Implicit methods for the 1D diffusion equation

Simulations with the Forward Euler scheme shows that the time step restriction, $F \leq \frac{1}{2}$, which means $\Delta t \leq \Delta x^2/(2\alpha)$, may be relevant in the beginning of the diffusion process, when the solution changes quite fast, but as time increases, the process slows down, and a small Δt may be inconvenient. By using *implicit schemes*, which lead to a coupled system of linear equations to be solved at each time level, any size of Δt is possible (but the accuracy decreases with increasing Δt). The Backward Euler scheme, derived and implemented below, is the simplest implicit scheme for the diffusion equation.

2.1 Backward Euler scheme

We now apply a backward difference in time in (5), but the same central difference in space:

$$[D_t^- u = D_x D_x u + f]_i^n, \tag{10}$$

which written out reads

$$\frac{u_i^n - u_i^{n-1}}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \,. \tag{11}$$

Now we assume u_i^{n-1} is computed, but all quantities at the "new" time level n are unknown. This time it is not possible to solve with respect to u_i^n because this value couples to its neighbors in space, u_{i-1}^n and u_{i+1}^n, which are also unknown. Let us examine this fact for the case when $N_x = 3$. Equation (11) written for $i = 1, \ldots, Nx - 1 = 1, 2$ becomes

$$\frac{u_1^n - u_1^{n-1}}{\Delta t} = \alpha \frac{u_2^n - 2u_1^n + u_0^n}{\Delta x^2} + f_1^n \tag{12}$$

$$\frac{u_2^n - u_2^{n-1}}{\Delta t} = \alpha \frac{u_3^n - 2u_2^n + u_1^n}{\Delta x^2} + f_2^n \tag{13}$$

The boundary values u_0^n and u_3^n are known as zero. Collecting the unknown new values u_1^n and u_2^n on the left-hand side and multiplying by Δt gives

$$(1 + 2F)\, u_1^n - F u_2^n = u_1^{n-1} + \Delta t f_1^n, \tag{14}$$

$$-F u_1^n + (1 + 2F)\, u_2^n = u_2^{n-1} + \Delta t f_2^n \,. \tag{15}$$

This is a coupled 2×2 system of algebraic equations for the unknowns u_1^n and u_2^n. The equivalent matrix form is

$$\begin{pmatrix} 1+2F & -F \\ -F & 1+2F \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \end{pmatrix} = \begin{pmatrix} u_1^{n-1} + \Delta t f_1^n \\ u_2^{n-1} + \Delta t f_2^n \end{pmatrix}$$

Implicit vs. explicit methods.

Discretization methods that lead to a coupled system of equations for the unknown function at a new time level are said to be *implicit methods*. The counterpart, *explicit methods*, refers to discretization methods where there is a simple explicit formula for the values of the unknown function at each of the spatial mesh points at the new time level. From an implementational point of view, implicit methods are more comprehensive to code since they require the solution of coupled equations, i.e., a matrix system, at each time level.

In the general case, (11) gives rise to a coupled $(Nx-1)\times(Nx-1)$ system of algebraic equations for all the unknown u_i^n at the interior spatial points $i = 1, \ldots, Nx-1$. Collecting the unknowns on the left-hand side, (11) can be written

$$-Fu_{i-1}^n + (1+2F)\,u_i^n - Fu_{i+1}^n = u_{i-1}^{n-1}, \tag{16}$$

for $i = 1, \ldots, Nx-1$. One can either view these equations as a system for where the u_i^n values at the internal mesh points, $i = 1, \ldots, N_x - 1$, are unknown, or we may append the boundary values u_0^n and $u_{N_x}^n$ to the system. In the latter case, all u_i^n for $i = 0, \ldots, N_x$ are unknown and we must add the boundary equations to the $N_x - 1$ equations in (16):

$$u_0^n = 0, \tag{17}$$

$$u_{N_x}^n = 0\,. \tag{18}$$

A coupled system of algebraic equations can be written on matrix form, and this is important if we want to call up ready-made software for solving the system. The equations (16) and (17)–(18) correspond to the matrix equation

$$AU = b$$

where $U = (u_0^n, \ldots, u_{N_x}^n)$, and the matrix A has the following structure:

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ A_{1,0} & A_{1,1} & 0 & \ddots & & & & & \vdots \\ 0 & A_{2,1} & A_{2,2} & A_{2,3} & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & A_{i,i-1} & A_{i,i} & A_{i,i+1} & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & A_{N_x-1,N_x} \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & A_{N_x,N_x-1} & A_{N_x,N_x} \end{pmatrix} \tag{19}$$

The nonzero elements are given by

$$A_{i,i-1} = -F \tag{20}$$
$$A_{i,i} = 1 + 2F \tag{21}$$
$$A_{i,i+1} = -F \tag{22}$$

for the equations for internal points, $i = 1, \ldots, N_x - 1$. The equations for the boundary points correspond to

$$A_{0,0} = 1, \tag{23}$$
$$A_{0,1} = 0, \tag{24}$$
$$A_{N_x,N_x-1} = 0, \tag{25}$$
$$A_{N_x,N_x} = 1\,. \tag{26}$$

The right-hand side b is written as

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{N_x} \end{pmatrix} \tag{27}$$

with

$$b_0 = 0, \tag{28}$$
$$b_i = u_i^{n-1}, \quad i = 1, \ldots, N_x - 1, \tag{29}$$
$$b_{N_x} = 0\,. \tag{30}$$

We observe that the matrix A contains quantities that do not change in time. Therefore, A can be formed once and for all before we enter the recursive formulas for the time evolution. The right-hand side b, however, must be updated at each time step. This leads to the following computational algorithm, here sketched with Python code:

```
x = linspace(0, L, Nx+1)   # mesh points in space
dx = x[1] - x[0]
t = linspace(0, T, N+1)    # mesh points in time
u   = zeros(Nx+1)          # unknown u at new time level
u_1 = zeros(Nx+1)          # u at the previous time level

# Data structures for the linear system
A = zeros((Nx+1, Nx+1))
b = zeros(Nx+1)

for i in range(1, Nx):
    A[i,i-1] = -F
    A[i,i+1] = -F
    A[i,i] = 1 + 2*F
A[0,0] = A[Nx,Nx] = 1

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

import scipy.linalg

for n in range(0, Nt):
    # Compute b and solve linear system
    for i in range(1, Nx):
        b[i] = -u_1[i]
    b[0] = b[Nx] = 0
    u[:] = scipy.linalg.solve(A, b)

    # Update u_1 before next step
    u_1[:] = u
```

2.2 Sparse matrix implementation

We have seen from (19) that the matrix A is tridiagonal. The code segment above used a full, dense matrix representation of A, which stores a lot of values we know are zero beforehand, and worse, the solution algorithm computes with all these zeros. With $N_x + 1$ unknowns, the work by the solution algorithm is $\frac{1}{3}(N_x + 1)^3$ and the storage requirements $(N_x + 1)^2$. By utilizing the fact that A is tridiagonal and employing corresponding software tools, the work and storage demands can be proportional to N_x only.

The key idea is to apply a data structure for a tridiagonal or sparse matrix. The `scipy.sparse` package has relevant utilities. For example, we can store the nonzero diagonals of a matrix. The package also has linear system solvers that operate on sparse matrix data structures. The code below illustrates how we can store only the main diagonal and the upper and lower diagonals.

```
# Representation of sparse matrix and right-hand side
main  = zeros(Nx+1)
lower = zeros(Nx-1)
upper = zeros(Nx-1)
b     = zeros(Nx+1)

# Precompute sparse matrix
main[:] = 1 + 2*F
lower[:] = -F  #1
upper[:] = -F  #1
# Insert boundary conditions
main[0] = 1
main[Nx] = 1

A = scipy.sparse.diags(
    diagonals=[main, lower, upper],
    offsets=[0, -1, 1], shape=(Nx+1, Nx+1),
    format='csr')
print A.todense()  # Check that A is correct

# Set initial condition
for i in range(0,Nx+1):
    u_1[i] = I(x[i])

for n in range(0, Nt):
    b = u_1
    b[0] = b[-1] = 0.0  # boundary conditions
    u[:] = scipy.sparse.linalg.spsolve(A, b)
    u_1[:] = u
```

The `scipy.sparse.linalg.spsolve` function utilizes the sparse storage structure of `A` and performs in this case a very efficient Gaussian elimination solve.

The program `diffu1D_u0.py`[5] contains a function `solver_BE`, which implements the Backward Euler scheme sketched above. As mentioned in Section 1.2, the functions `plug` and `gaussian` runs the case with $I(x)$ as a discontinuous plug or a smooth Gaussian function. All experiments point to two characteristic features of the Backward Euler scheme: 1) it is always stable, and 2) it always gives a smooth, decaying solution.

2.3 Crank-Nicolson scheme

The idea in the Crank-Nicolson scheme is to apply centered differences in space and time, combined with an average in time. We demand the PDE to be fulfilled at the spatial mesh points, but in between the points in the time mesh:

$$\frac{\partial}{\partial t}u(x_i, t_{n+\frac{1}{2}}) = \alpha\frac{\partial^2}{\partial x^2}u(x_i, t_{n+\frac{1}{2}}) + f(x_i, t_{n+\frac{1}{2}}),$$

for $i = 1, \ldots, N_x - 1$ and $n = 0, \ldots, N_t - 1$.

With centered differences in space and time, we get

$$[D_t u = \alpha D_x D_x u + f]_i^{n+\frac{1}{2}}\,.$$

On the right-hand side we get an expression

[5] `http://tinyurl.com/nm5587k/diffu/diffu1D_u0.py`

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

This expression is problematic since $u_i^{n+\frac{1}{2}}$ is not one of the unknown we compute. A possibility is to replace $u_i^{n+\frac{1}{2}}$ by an arithmetic average:

$$u_i^{n+\frac{1}{2}} \approx \frac{1}{2}\left(u_i^n + u_i^{n+1}\right)\,.$$

In the compact notation, we can use the arithmetic average notation $\overline{u}^t$:

$$[D_t u = \alpha D_x D_x \overline{u}^t + f]_i^{n+\frac{1}{2}}\,.$$

We can also use an average for $f_i^{n+\frac{1}{2}}$:

$$[D_t u = \alpha D_x D_x \overline{u}^t + \overline{f}^t]_i^{n+\frac{1}{2}}\,.$$

After writing out the differences and average, multiplying by Δt, and collecting all unknown terms on the left-hand side, we get

$$u_i^{n+1} - \frac{1}{2}F(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}) = u_i^n + \frac{1}{2}F(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \\ \frac{1}{2}f_i^{n+1} + \frac{1}{2}f_i^n\,. \tag{31}$$

Also here, as in the Backward Euler scheme, the new unknowns u_{i-1}^{n+1}, u_i^{n+1}, and u_{i+1}^{n+1} are coupled in a linear system $AU = b$, where A has the same structure as in (19), but with slightly different entries:

$$A_{i,i-1} = -\frac{1}{2}F \tag{32}$$

$$A_{i,i} = \frac{1}{2} + F \tag{33}$$

$$A_{i,i+1} = -\frac{1}{2}F \tag{34}$$

for the equations for internal points, $i = 1, \ldots, N_x - 1$. The equations for the boundary points correspond to

$$A_{0,0} = 1, \tag{35}$$

$$A_{0,1} = 0, \tag{36}$$

$$A_{N_x,N_x-1} = 0, \tag{37}$$

$$A_{N_x,N_x} = 1\,. \tag{38}$$

The right-hand side b has entries

$$b_0 = 0, \tag{39}$$

$$b_i = u_i^{n-1} + \frac{1}{2}(f_i^n + f_i^{n+1}), \quad i = 1, \ldots, N_x - 1, \tag{40}$$

$$b_{N_x} = 0\,. \tag{41}$$

2.4 The θ rule

For the equation

$$\frac{\partial u}{\partial t} = G(u),$$

where $G(u)$ is some a spatial differential operator, the θ-rule looks like

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \theta G(u_i^{n+1}) + (1-\theta)G(u_i^n)\,.$$

The important feature of this time discretization scheme is that we can implement one formula and then generate a family of well-known and widely used schemes:

- $\theta = 0$ gives the Forward Euler scheme in time
- $\theta = 1$ gives the Backward Euler scheme in time
- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme in time

Applied to the 1D diffusion problem, the θ-rule gives

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

This scheme also leads to a matrix system with entries

$$A_{i,i-1} = -F\theta, \quad A_{i,i} = 1 + 2F\theta \quad , A_{i,i+1} = -F\theta,$$

while right-hand side entry b_i is

$$b_i = u_i^n + F(1-\theta)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \Delta t\theta f_i^{n+1} + \Delta t(1-\theta) f_i^n\,.$$

The corresponding entries for the boundary points are as in the Backward Euler and Crank-Nicolson schemes listed earlier.

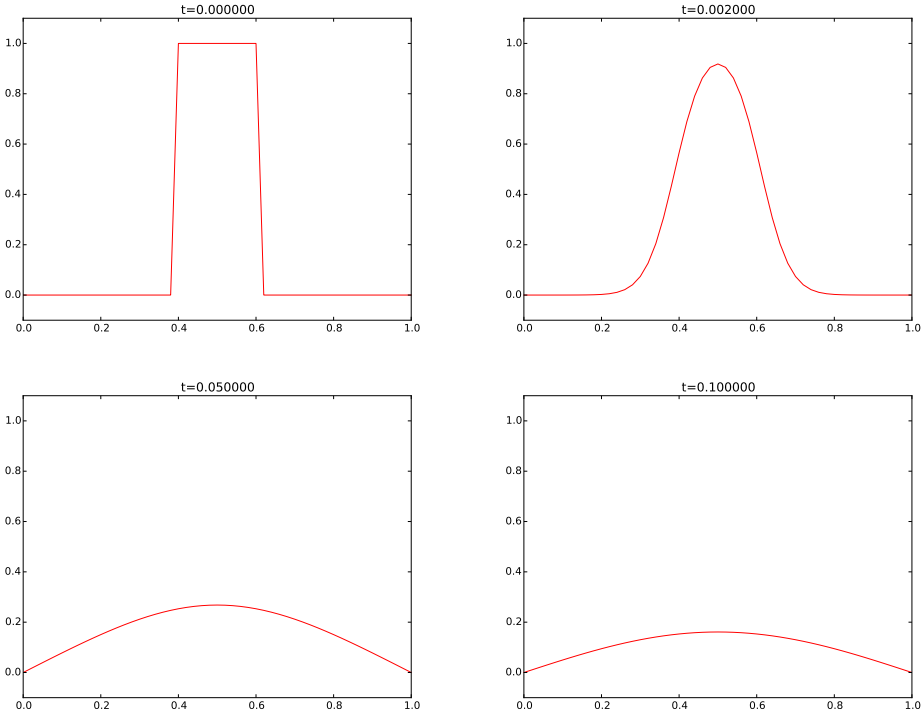

Figure 5: Backward Euler scheme for $F = 0.5$.

2.5 Experiments

We can repeat the experiments from Section 1.5 to see if the Backward Euler or Crank-Nicolson schemes have problems with sawtooth-like noise when starting with a discontinuous initial condition. We can also verify that we can have $F > \frac{1}{2}$, which in practice often means choosing larger time steps.

The Backward Euler scheme always produces smooth solutions for any F. Figure 5 shows one example. The Crank-Nicolson method produces smooth solutions for small F, $F \leq \frac{1}{2}$, but small noise is more and more evident as F increases. Figures 6 and 7 demonstrates the effect for $F = 3$ and $F = 10$, respectively. Section 3 explains why such noise occur.

2.6 The Laplace and Poisson equation

The Laplace equation, $\nabla^2 u = 0$, or the Poisson equation, $-\nabla^2 u = f$, occur in numerous applications throughout science and engineering. In 1D these equations read $u''(x) = 0$ and $-u''(x) = f(x)$, respectively. We can solve 1D variants of the Laplace equations with the listed software, because we can interpret $u_{xx} = 0$ as the limiting solution of $u_t = \alpha u_{xx}$ when u reach a steady state limit where

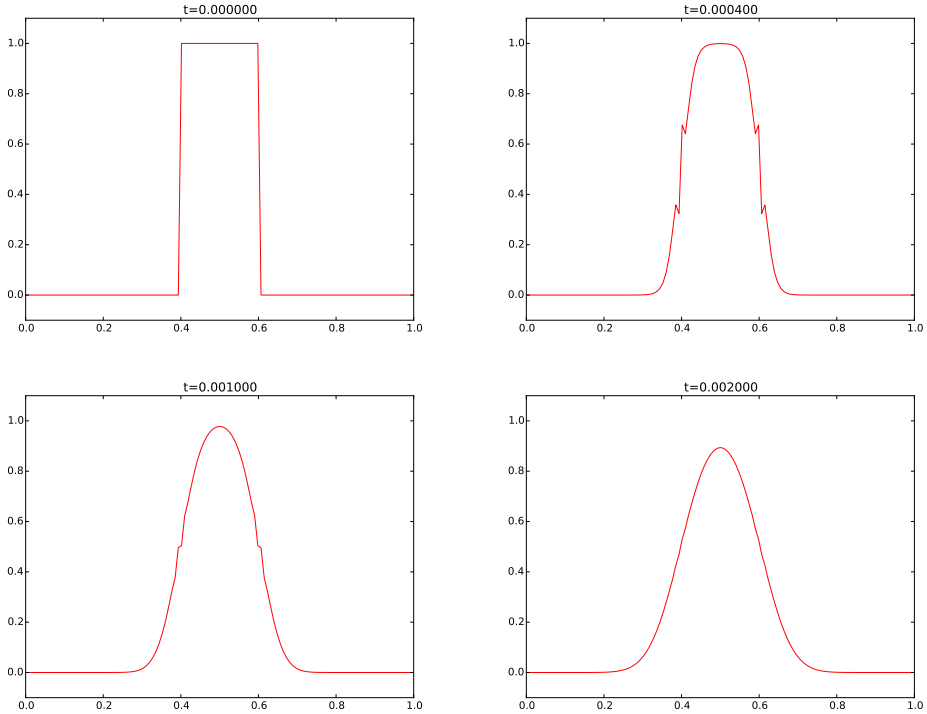

Figure 6: Crank-Nicolson scheme for $F = 3$.

$u_t \to 0$. Similarly, Poisson's equation $-u_{xx} = f$ arises from solving $u_t = u_{xx} + f$ and letting $t \to$ so $u_t \to 0$.

Technically in a program, we can simulate $t \to \infty$ by just taking one large time step: $\Delta t \to \infty$. In the limit the Backward Euler scheme gives

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

which is nothing but the discretization $[-D_x D_x u = f]_i^{n+1} = 0$ of $-u_{xx} = f$.

The result above means that the Backward Euler scheme can solve the limit equation directly and hence produce a solution of the 1D Laplace equation. With the Forward Euler scheme we must do the time stepping since $\Delta t > \Delta x^2/\alpha$ is illegal and leads to instability. We may interpret this time stepping as solving the equation system from $-u_{xx} = f$ by iterating on a time pseudo time variable.

hpl 2: Better to say the last sentence when we treat iterative methods.

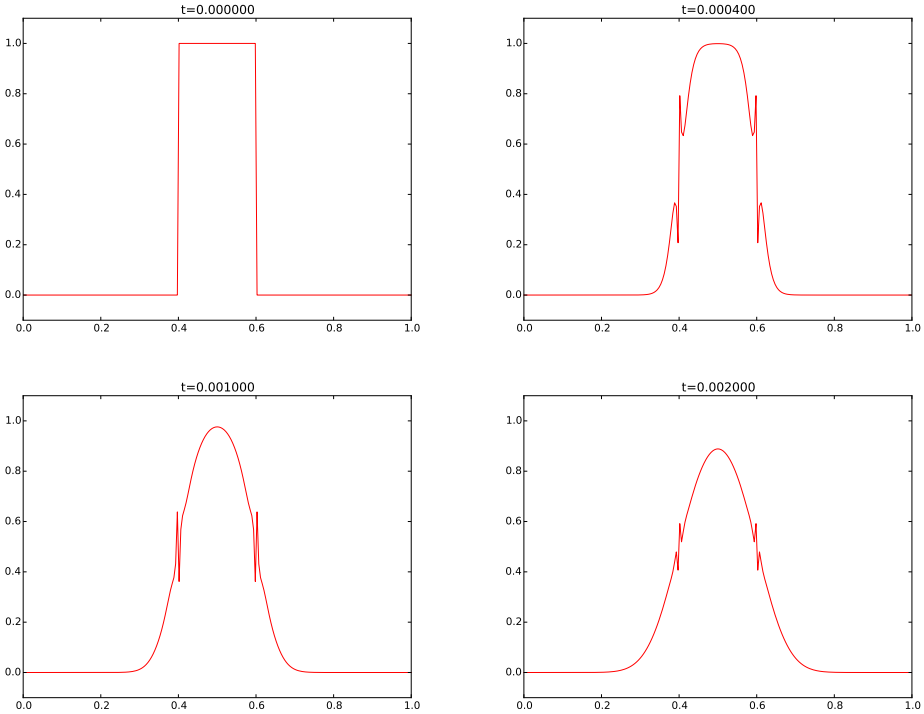

Figure 7: Crank-Nicolson scheme for $F = 10$.

3 Analysis of schemes for the diffusion equation

The numerical experiments in Sections 2.5 and 1.5 reveal that there are some numerical problems with the Forward Euler and Crank-Nicolson schemes: sawtooth-like noise is sometimes present in solutions that are, from a mathematical point of view, expected to be smooth. This section presents a mathematical analysis that explains the observed behavior and arrives at criteria for obtaining numerical solutions that reproduce the qualitative properties of the exact solutions. In short, we shall explain what is observed in Figures 1, 4, 2, 3, 5, 6, and 7.

3.1 Properties of the solution

A particular characteristic of diffusive processes, governed by an equation like

$$u_t = \alpha u_{xx}, \tag{42}$$

is that the initial shape $u(x,0) = I(x)$ spreads out in space with time, along with a decaying amplitude. Three different examples will illustrate the spreading of u in space and the decay in time.

Similarity solution. The diffusion equation (42) admits solutions that depend on $\eta = (x-c)/\sqrt{4\alpha t}$ for a given value of c. One particular solution is

$$u(x,t) = a\,\mathrm{erf}(\eta) + b, \tag{43}$$

where

$$\mathrm{erf}(\eta) = \frac{2}{\sqrt{\pi}} \int_0^\eta e^{-\zeta^2} d\zeta, \tag{44}$$

is the *error function*, and a and b are arbitrary constants. The error function lies in $(-1,1)$, is odd around $\eta = 0$, and goes relatively quickly to ± 1:

$$\begin{aligned}
\lim_{\eta\to-\infty} \mathrm{erf}(\eta) &= -1, \\
\lim_{\eta\to\infty} \mathrm{erf}(\eta) &= 1, \\
\mathrm{erf}(\eta) &= -\mathrm{erf}(-\eta), \\
\mathrm{erf}(0) &= 0, \\
\mathrm{erf}(2) &= 0.99532227, \\
\mathrm{erf}(3) &= 0.99997791\,.
\end{aligned}$$

As $t \to 0$, the error function approaches a step function centered at $x = c$. For a diffusion problem posed on the unit interval $[0,1]$, we may choose the step at $x = 1/2$ (meaning $c = 1/2$), $a = -1/2$, $b = 1/2$. Then

$$u(x,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right)\right) = \frac{1}{2}\mathrm{erfc}\left(\frac{x-\frac{1}{2}}{\sqrt{4\alpha t}}\right), \tag{45}$$

where we have introduced the *complementary error function* $\mathrm{erfc}(\eta) = 1 - \mathrm{erf}(\eta)$. The solution (45) implies the boundary conditions

$$u(0,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{-1/2}{\sqrt{4\alpha t}}\right)\right), \tag{46}$$

$$u(1,t) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{1/2}{\sqrt{4\alpha t}}\right)\right). \tag{47}$$

For small enough t, $u(0,t) \approx 1$ and $u(1,t) \approx 1$, but as $t \to \infty$, $u(x,t) \to 1/2$ on $[0,1]$.

Solution for a Gaussian pulse. The standard diffusion equation $u_t = \alpha u_{xx}$ admits a Gaussian function as solution:

$$u(x,t) = \frac{1}{\sqrt{4\pi\alpha t}} \exp\left(-\frac{(x-c)^2}{4\alpha t}\right). \tag{48}$$

At $t = 0$ this is a Dirac delta function, so for computational purposes one must start to view the solution at some time $t = t_\epsilon > 0$. Replacing t by $t_\epsilon + t$ in (48) makes it easy to operate with a (new) t that starts at $t = 0$ with an initial condition with a finite width. The important feature of (48) is that the standard deviation σ of a sharp initial Gaussian pulse increases in time according to $\sigma = \sqrt{2\alpha t}$, making the pulse diffuse and flatten out.

Solution for a sine component. For example, (42) admits a solution of the form

$$u(x,t) = Qe^{-at}\sin(kx)\ . \tag{49}$$

The parameters Q and k can be freely chosen, while inserting (49) in (42) gives the constraint

$$a = -\alpha k^2\,.$$

A very important feature is that the initial shape $I(x) = Q\sin kx$ undergoes a damping $\exp(-\alpha k^2 t)$, meaning that rapid oscillations in space, corresponding to large k, are very much faster dampened than slow oscillations in space, corresponding to small k. This feature leads to a smoothing of the initial condition with time.

The following examples illustrates the damping properties of (49). We consider the specific problem

$$\begin{aligned} u_t &= u_{xx}, \quad x \in (0,1),\ t \in (0,T], \\ u(0,t) &= u(1,t) = 0, \quad t \in (0,T], \\ u(x,0) &= \sin(\pi x) + 0.1\sin(100\pi x)\,. \end{aligned}$$

The initial condition has been chosen such that adding two solutions like (49) constructs an analytical solution to the problem:

$$u(x,t) = e^{-\pi^2 t}\sin(\pi x) + 0.1e^{-\pi^2 10^4 t}\sin(100\pi x)\,. \tag{50}$$

Figure 8 illustrates the rapid damping of rapid oscillations $\sin(100\pi x)$ and the very much slower damping of the slowly varying $\sin(\pi x)$ term. After about $t = 0.5\cdot 10^{-4}$ the rapid oscillations do not have a visible amplitude, while we have to wait until $t \sim 0.5$ before the amplitude of the long wave $\sin(\pi x)$ becomes very small.

3.2 Example: Diffusion of a discontinues profile

We shall see how different schemes predict the evolution of a discontinuous initial condition:

$$u(x,0) = \begin{cases} U_L, & x < L/2 \\ U_R, & x \geq L/2 \end{cases}$$

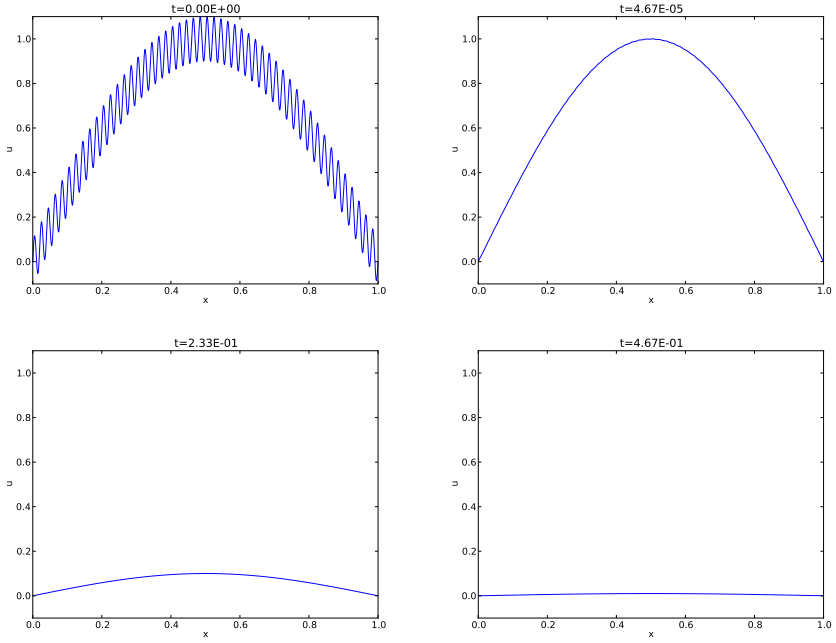

Figure 8: Evolution of the solution of a diffusion problem: initial condition (upper left), 1/100 reduction of the small waves (upper right), 1/10 reduction of the long wave (lower left), and 1/100 reduction of the long wave (lower right).

Such a discontinuous initial condition may arise when two insulated blocks of metals at different temperature are brought in contact at $t = 0$. Alternatively, signaling in the brain is based on release of a huge ion concentration on one side of a synapse, which implies diffusive transport of a discontinuous concentration function.

More to be written...

3.3 Analysis of discrete equations

A counterpart to (49) is the complex representation of the same function:

$$u(x,t) = Qe^{-at}e^{ikx},$$

where $i = \sqrt{-1}$ is the imaginary unit. We can add such functions, often referred to as wave components, to make a Fourier representation of a general solution of the diffusion equation:

$$u(x,t) \approx \sum_{k\in K} b_k e^{-\alpha k^2 t} e^{ikx}, \tag{51}$$

where K is a set of an infinite number of k values needed to construct the solution. In practice, however, the series is truncated and K is a finite set of k

values need build a good approximate solution. Note that (50) is a special case of (51) where $K = \{\pi, 100\pi\}$, $b_\pi = 1$, and $b_{100\pi} = 0.1$.

The amplitudes b_k of the individual Fourier waves must be determined from the initial condition. At $t = 0$ we have $u \approx \sum_k b_k \exp(ikx)$ and find K and b_k such that

$$I(x) \approx \sum_{k \in K} b_k e^{ikx}. \tag{52}$$

(The relevant formulas for b_k come from Fourier analysis, or equivalently, a least-squares method for approximating $I(x)$ in a function space with basis $\exp(ikx)$.)

Much insight about the behavior of numerical methods can be obtained by investigating how a wave component $\exp(-\alpha k^2 t)\exp(ikx)$ is treated by the numerical scheme. It appears that such wave components are also solutions of the schemes, but the damping factor $\exp(-\alpha k^2 t)$ varies among the schemes. To ease the forthcoming algebra, we write the damping factor as A^n. The exact amplification factor corresponding to A is $A_\mathrm{e} = \exp(-\alpha k^2 \Delta t)$.

3.4 Analysis of the finite difference schemes

We have seen that a general solution of the diffusion equation can be built as a linear combination of basic components

$$e^{-\alpha k^2 t} e^{ikx}.$$

A fundamental question is whether such components are also solutions of the finite difference schemes. This is indeed the case, but the amplitude $\exp(-\alpha k^2 t)$ might be modified (which also happens when solving the ODE counterpart $u' = -\alpha u$). We therefore look for numerical solutions of the form

$$u_q^n = A^n e^{ikq\Delta x} = A^n e^{ikx}, \tag{53}$$

where the amplification factor A must be determined by inserting the component into an actual scheme.

Stability. The exact amplification factor is $A_\mathrm{e} = \exp(-\alpha^2 k^2 \Delta t)$. We should therefore require $|A| < 1$ to have a decaying numerical solution as well. If $-1 \leq A < 0$, A^n will change sign from time level to time level, and we get stable, non-physical oscillations in the numerical solutions that are not present in the exact solution.

Accuracy. To determine how accurately a finite difference scheme treats one wave component (53), we see that the basic deviation from the exact solution is reflected in how well A^n approximates A_e^n, or how well A approximates A_e. We can plot A_e and the various expressions for A, and we can make Taylor expansions of A/A_e to see the error more analytically.

3.5 Analysis of the Forward Euler scheme

The Forward Euler finite difference scheme for $u_t = \alpha u_{xx}$ can be written as

$$[D_t^+ u = \alpha D_x D_x u]_q^n \,.$$

Inserting a wave component (53) in the scheme demands calculating the terms

$$e^{ikq\Delta x}[D_t^+ A]^n = e^{ikq\Delta x} A^n \frac{A-1}{\Delta t},$$

and

$$A^n D_x D_x [e^{ikx}]_q = A^n \left(-e^{ikq\Delta x} \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right)\right) .$$

Inserting these terms in the discrete equation and dividing by $A^n e^{ikq\Delta x}$ leads to

$$\frac{A-1}{\Delta t} = -\alpha \frac{4}{\Delta x^2} \sin^2\left(\frac{k\Delta x}{2}\right),$$

and consequently

$$A = 1 - 4F \sin^2\left(\frac{k\Delta x}{2}\right), \tag{54}$$

where

$$F = \frac{\alpha \Delta t}{\Delta x^2} \tag{55}$$

is the *numerical Fourier number*. The complete numerical solution is then

$$u_q^n = \left(1 - 4F \sin^2\left(\frac{k\Delta x}{2}\right)\right)^n e^{ikq\Delta x} \,. \tag{56}$$

Stability. We easily see that $A \leq 1$. However, the A can be less than -1, which will lead to growth of a numerical wave component. The criterion $A \geq -1$ implies

$$4F \sin^2(p/2) \leq 2 \,.$$

The worst case is when $\sin^2(p/2) = 1$, so a sufficient criterion for stability is

$$F \leq \frac{1}{2}, \tag{57}$$

or expressed as a condition on Δt:

$$\Delta t \leq \frac{\Delta x^2}{2\alpha} \,. \tag{58}$$

Note that halving the spatial mesh size, $\Delta x \rightarrow \frac{1}{2}\Delta x$, requires Δt to be reduced by a factor of $1/4$. The method hence becomes very expensive for fine spatial meshes.

Accuracy. Since A is expressed in terms of F and the parameter we now call $p = k\Delta x/2$, we should also express A_{e} by F and p. The exponent in A_{e} is $-\alpha k^2\Delta t$, which equals $-Fk^2\Delta x^2 = -F4p^2$. Consequently,

$$A_{\mathrm{e}} = \exp\left(-\alpha k^2\Delta t\right) = \exp\left(-4Fp^2\right).$$

All our A expressions as well as A_{e} are now functions of the two dimensionless parameters F and p.

Computing the Taylor series expansion of A/A_{e} in terms of F can easily be done with aid of `sympy`:

```
def A_exact(F, p):
    return exp(-4*F*p**2)

def A_FE(F, p):
    return 1 - 4*F*sin(p)**2

from sympy import *
F, p = symbols('F p')
A_err_FE = A_FE(F, p)/A_exact(F, p)
print A_err_FE.series(F, 0, 6)
```

The result is

$$\frac{A}{A_{\mathrm{e}}} = 1 - 4F\sin^2 p + 2Fp^2 - 16F^2p^2\sin^2 p + 8F^2p^4 + \cdots$$

Recalling that $F = \alpha\Delta t/\Delta x$, $p = k\Delta x/2$, and that $\sin^2 p \leq 1$, we realize that the dominating error terms are at most

$$1 - 4\alpha\frac{\Delta t}{\Delta x^2} + \alpha\Delta t - 4\alpha^2\Delta t^2 + \alpha^2\Delta t^2\Delta x^2 + \cdots.$$

3.6 Analysis of the Backward Euler scheme

Discretizing $u_t = \alpha u_{xx}$ by a Backward Euler scheme,

$$[D_t^- u = \alpha D_x D_x u]_q^n,$$

and inserting a wave component (53), leads to calculations similar to those arising from the Forward Euler scheme, but since

$$e^{ikq\Delta x}[D_t^- A]^n = A^n e^{ikq\Delta x}\frac{1 - A^{-1}}{\Delta t},$$

we get

$$\frac{1 - A^{-1}}{\Delta t} = -\alpha\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right),$$

and then

$$A = \left(1 + 4F\sin^2 p\right)^{-1}. \tag{59}$$

The complete numerical solution can be written

$$u_q^n = \left(1 + 4F\sin^2 p\right)^{-n} e^{ikq\Delta x}\,. \tag{60}$$

Stability. We see from (59) that $0 < A < 1$, which means that all numerical wave components are stable and non-oscillatory for any $\Delta t > 0$.

3.7 Analysis of the Crank-Nicolson scheme

The Crank-Nicolson scheme can be written as

$$[D_t u = \alpha D_x D_x \overline{u}^x]_q^{n+\frac{1}{2}},$$

or

$$[D_t u]_q^{n+\frac{1}{2}} = \frac{1}{2}\alpha\left([D_x D_x u]_q^n + [D_x D_x u]_q^{n+1}\right)\,.$$

Inserting (53) in the time derivative approximation leads to

$$[D_t A^n e^{ikq\Delta x}]^{n+\frac{1}{2}} = A^{n+\frac{1}{2}} e^{ikq\Delta x}\frac{A^{\frac{1}{2}} - A^{-\frac{1}{2}}}{\Delta t} = A^n e^{ikq\Delta x}\frac{A-1}{\Delta t}\,.$$

Inserting (53) in the other terms and dividing by $A^n e^{ikq\Delta x}$ gives the relation

$$\frac{A-1}{\Delta t} = -\frac{1}{2}\alpha\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)(1+A),$$

and after some more algebra,

$$A = \frac{1 - 2F\sin^2 p}{1 + 2F\sin^2 p}\,. \tag{61}$$

The exact numerical solution is hence

$$u_q^n = \left(\frac{1 - 2F\sin^2 p}{1 + 2F\sin^2 p}\right)^n e^{ikp\Delta x}\,. \tag{62}$$

Stability. The criteria $A > -1$ and $A < 1$ are fulfilled for any $\Delta t > 0$.

3.8 Summary of accuracy of amplification factors

We can plot the various amplification factors against $p = k\Delta x/2$ for different choices of the F parameter. Figures 9, 10, and 11 show how long and small waves are damped by the various schemes compared to the exact damping. As long as all schemes are stable, the amplification factor is positive, except for Crank-Nicolson when $F > 0.5$.

The effect of negative amplification factors is that A^n changes sign from one time level to the next, thereby giving rise to oscillations in time in an animation of the solution. We see from Figure 9 that for $F = 20$, waves with $p \geq \pi/2$

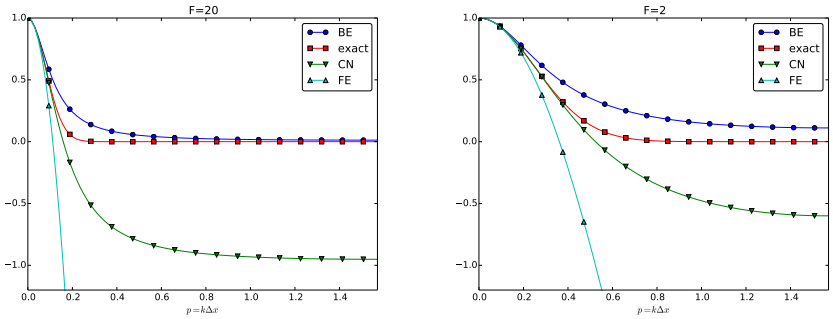

Figure 9: Amplification factors for large time steps.

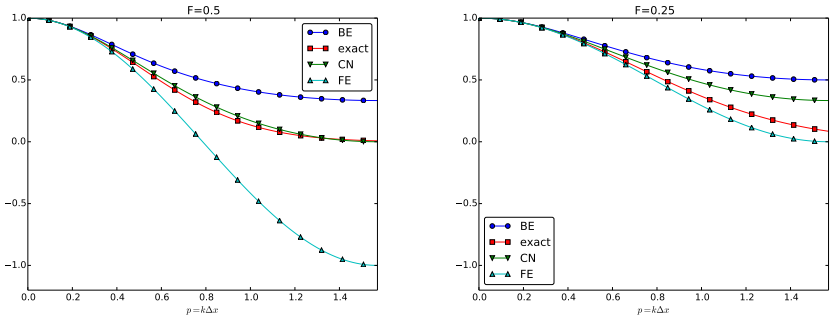

Figure 10: Amplification factors for time steps around the Forward Euler stability limit.

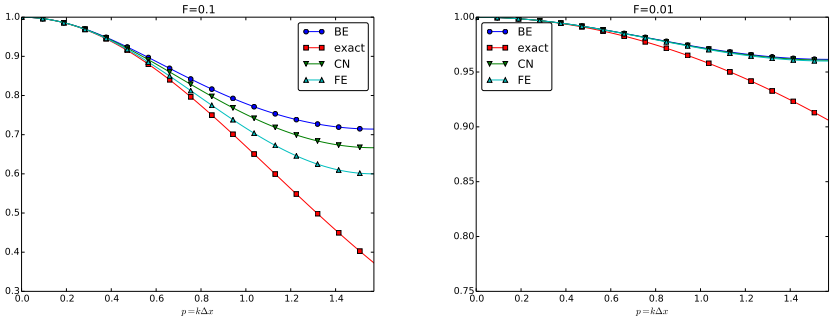

Figure 11: Amplification factors for small time steps.

undergo a damping close to -1, which means that the amplitude does not decay and that the wave component jumps up and down in time. For $F = 2$ we have a damping of a factor of 0.5 from one time level to the next, which is very much smaller than the exact damping. Short waves will therefore fail to be effectively dampened. These waves will manifest themselves as high frequency oscillatory noise in the solution.

A value $p = \pi/4$ corresponds to four mesh points per wave length of e^{ikx}, while $p = \pi/2$ implies only two points per wave length, which is the smallest number of points we can have to represent the wave on the mesh.

To demonstrate the oscillatory behavior of the Crank-Nicolson scheme, we choose an initial condition that leads to short waves with significant amplitude. A discontinuous $I(x)$ will in particular serve this purpose.

Run $F =$ .....

Exercise 1: Explore symmetry in a 1D problem

This exercise simulates the exact solution (48). Suppose for simplicity that $c = 0$.

a) Formulate an initial-boundary value problem that has (48) as solution in the domain $[-L, L]$. Use the exact solution (48) as Dirichlet condition at the boundaries. Simulate the diffusion of the Gaussian peak. Observe that the solution is symmetric around $x = 0$.

b) Show from (48) that $u_x(c, t) = 0$. Since the solution is symmetric around $x = c = 0$, we can solve the numerical problem in half of the domain, using a *symmetry boundary condition* $u_x = 0$ at $x = 0$. Set up the initial-boundary value problem in this case. Simulate the diffusion problem in $[0, L]$ and compare with the solution in a).

Solution.

$$
\begin{aligned}
u_t &= \alpha u_x x, \\
u_x(0,t) &= 0, \\
u(L,t) &= \frac{1}{\sqrt{4\pi\alpha t}} \exp\left(-\frac{x^2}{4\alpha t}\right).
\end{aligned}
$$

Filename: `diffu_symmetric_gaussian`.

Exercise 2: Investigate approximation errors from a $u_x = 0$ boundary condition

We consider the problem solved in Exercise 1 part b). The boundary condition $u_x(0, t) = 0$ can be implemented in two ways: 1) by a standard symmetric finite difference $[D_{2x}u]_i^n = 0$, or 2) by a one-sided difference $[D^+u = 0]_i^n = 0$. Investigate the effect of these two conditions on the convergence rate in space.

Hint. If you use a Forward Euler scheme, choose a discretization parameter $h = \Delta t = \Delta x^2$ and assume the error goes like $E \sim h^r$. The error in the scheme is $\mathcal{O}(\Delta t, \Delta x^2)$ so one should expect that the estimated r approaches 1. The question is if a one-sided difference approximation to $u_x(0, t) = 0$ destroys this convergence rate.
Filename: `diffu_onesided_fd`.

Exercise 3: Experiment with open boundary conditions in 1D

We address diffusion of a Gaussian function as in Exercise 1, in the domain $[0, L]$, but now we shall explore different types of boundary conditions on $x = L$. In real-life problems we do not know the exact solution on $x = L$ and must use something simpler.

a) Imagine that we want to solve the problem numerically on $[0, L]$, with a symmetry boundary condition $u_x = 0$ at $x = 0$, but we do not know the exact solution and cannot of that reason assign a correct Dirichlet condition at $x = L$. One idea is to simply set $u(L, t) = 0$ since this will be an accurate approximation before the diffused pulse reaches $x = L$ and even thereafter it might be a satisfactory condition if the exact u has a small value. Let u_{e} be the exact solution and let u be the solution of $u_t = \alpha u_{xx}$ with an initial Gaussian pulse and the boundary conditions $u_x(0, t) = u(L, t) = 0$. Derive a diffusion problem for the error $e = u_{\mathrm{e}} - u$. Solve this problem numerically using an exact Dirichlet condition at $x = L$. Animate the evolution of the error and make a curve plot of the error measure

$$E(t) = \sqrt{\frac{\int_0^L e^2 dx}{\int_0^L u dx}}\,.$$

Is this a suitable error measure for the present problem?

b) Instead of using $u(L, t) = 0$ as approximate boundary condition for letting the diffused Gaussian pulse move out of our finite domain, one may try $u_x(L, t) = 0$ since the solution for large t is quite flat. Argue that this condition gives a completely wrong asymptotic solution as $t \to 0$. To do this, integrate the diffusion equation from 0 to L, integrate u_{xx} by parts (or use Gauss' divergence theorem in 1D) to arrive at the important property

$$\frac{d}{dt}\int_0^L u(x, t)dx = 0,$$

implying that $\int_0^L u dx$ must be constant in time, and therefore

$$\int_0^L u(x, t)dx = \int_0^L I(x)dx\,.$$

The integral of the initial pulse is 1.

c) Another idea for an artificial boundary condition at $x = L$ is to use a cooling law

$$-\alpha u_x = q(u - u_S), \tag{63}$$

where q is an unknown heat transfer coefficient and u_S is the surrounding temperature in the medium outside of $[0, L]$. (Note that arguing that u_S is

approximately $u(L,t)$ gives the $u_x = 0$ condition from the previous subexercise that is qualitatively wrong for large t.) Develop a diffusion problem for the error in the solution using (63) as boundary condition. Assume one can take $u_S = 0$ "outside the domain" since $u_\mathrm{e} \to 0$ as $x \to \infty$. Find a function $q = q(t)$ such that the exact solution obeys the condition (63). Test some constant values of q and animate how the corresponding error function behaves. Also compute $E(t)$ curves as defined above.
Filename: `diffu_open_BC`.

Exercise 4: Simulate a diffused Gaussian peak in 2D/3D

a) Generalize (48) to multi dimensions by assuming that one-dimensional solutions can be multiplied to solve $u_t = \alpha\nabla^2 u$. Set $c = 0$ such that the peak of the Gaussian is at the origin.

b) One can from the exact solution show that $u_x = 0$ on $x = 0$, $u_y = 0$ on $y = 0$, and $u_z = 0$ on $z = 0$. The approximately correct condition $u = 0$ can be set on the remaining boundaries (say $x = L$, $y = L$, $z = L$), cf. Exercise 3. Simulate a 2D case and make an animation of the diffused Gaussian peak.

c) The formulation in b) makes use of symmetry of the solution such that we can solve the problem in the first quadrant (2D) or octant (3D) only. To check that the symmetry assumption is correct, formulate the problem without symmetry in a domain $[-L, L] \times [L, L]$ in 2D. Use $u = 0$ as approximately correct boundary condition. Simulate the same case as in b), but in a four times as large domain. Make an animation and compare it with the one in b).
Filename: `diffu_symmetric_gaussian_2D`.

Exercise 5: Examine stability of a diffusion model with a source term

Consider a diffusion equation with a linear u term:

$$u_t = \alpha u_{xx} + \beta u\,.$$

a) Derive in detail a Forward Euler scheme, a Backward Euler scheme, and a Crank-Nicolson for this type of diffusion model. Thereafter, formulate a θ-rule to summarize the three schemes.

b) Assume a solution like (49) and find the relation between a, k, α, and β.

Hint. Insert (49) in the PDE problem.

c) Calculate the stability of the Forward Euler scheme. Design numerical experiments to confirm the results.

Hint. Insert the discrete counterpart to (49) in the numerical scheme. Run experiments at the stability limit and slightly above.

d) Repeat c) for the Backward Euler scheme.

e) Repeat c) for the Crank-Nicolson scheme.

f) How does the extra term bu impact the accuracy of the three schemes?

Hint. For analysis of the accuracy, compare the numerical and exact amplification factors, in graphs and/or by Taylor series expansion.
Filename: `diffu_stability_uterm`.

4 Diffusion in heterogeneous media

Diffusion in heterogeneous media will normally imply a non-constant diffusion coefficient $\alpha = \alpha(x)$. A 1D diffusion model with such a variable diffusion coefficient reads

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(x)\frac{\partial u}{\partial x^2}\right), \quad x \in (0,L),\ t \in (0,T] \tag{64}$$

$$u(x,0) = I(x), \qquad x \in [0,L] \tag{65}$$

$$u(0,t) = U_L, \qquad t > 0, \tag{66}$$

$$u(L,t) = U_0, \qquad t > 0\,. \tag{67}$$

A short form of the diffusion equation with variable coefficients is $u_t = (\alpha u_x)_x$.

4.1 Discretization

We can discretize (64) by a *theta*-rule in time and centered differences in space.

4.2 Stationary solution

As $t \to \infty$, the solution of the above problem will approach a stationary limit where $\partial u/\partial t = 0$. The governing equation is then

$$\frac{d}{dx}\left(\alpha\frac{du}{dx}\right) = 0, \tag{68}$$

with boundary conditions $u(0) = U_0$ and $u(L) = u_L$. It is possible to obtain an exact solution of (68) for any α. Integrating twice and applying the boundary conditions to determine the integration constants gives

$$u(x) = U_0 + (U_L - U_0)\frac{\int_0^x (\alpha(\xi))^{-1}d\xi}{\int_0^L (\alpha(\xi))^{-1}d\xi}\,. \tag{69}$$

4.3 Piecewise constant medium

Consider a medium built of M layers. The boundaries between the layers are denoted by $b_0, \ldots, b_M$, where $b_0 = 0$ and $b_M = L$. If the material in each layer potentially differs from the others, but is otherwise constant, we can express α as a *piecewise constant function* according to

$$\alpha(x) = \begin{cases} \alpha_0, & b_0 \leq x < b_1, \\ \vdots & \\ \alpha_i, & b_i \leq x < b_{i+1}, \\ \vdots & \\ \alpha_0, & b_{M-1} \leq x \leq b_M. \end{cases} \tag{70}$$

The exact solution (69) in case of such a piecewise constant α function is easy to derive. Assume that x is in the m-th layer: $x \in [b_m, b_{m+1}]$. In the integral $\int_0^x (a(\xi))^{-1} d\xi$ we must integrate through the first $m-1$ layers and then add the contribution from the remaining part $x - b_m$ into the m-th layer:

$$u(x) = U_0 + (U_L - U_0) \frac{\sum_{j=0}^{m-1} (b_{j+1} - b_j)/\alpha(b_j) + (x - b_m)/\alpha(b_m)}{\sum_{j=0}^{M-1} (b_{j+1} - b_j)/\alpha(b_j)} \tag{71}$$

Remark. It may sound strange to have a discontinuous α in a differential equation where one is to differentiate, but a discontinuous α is compensated by a discontinuous u_x such that αu_x is continues and therefore can be differentiated as $(\alpha u_x)_x$.

4.4 Implementation

Programming with piecewise function definition quickly becomes cumbersome as the most naive approach is to test for which interval x lies, and then start evaluating a formula like (71). In Python, vectorized expressions may help to speed up the computations. The convenience classes `PiecewiseConstant` and `IntegratedPiecewiseConstant` were made to simplify programming with functions like (4.3) and expressions like (71). These utilities not only represent piecewise constant functions, but also *smoothed* versions of them where the discontinuities can be smoothed out in a controlled fashion. This is advantageous in many computational contexts (although seldom for pure finite difference computations of the solution u).

The `PiecewiseConstant` class is created by sending in the domain as a 2-tuple or 2-list and a `data` object describing the boundaries $b_0, \ldots, b_M$ and the corresponding function values $\alpha_0, \ldots, \alpha_{M-1}$. More precisely, `data` is a nested list, where `data[i][0]` holds b_i and `data[i][1]` holds the corresponding value α_i, for $i = 0, \ldots, M-1$. Given b_i and α_i in arrays `b` and `a`, it is easy to fill out the nested list `data`.

In our application, we want to represent α and $1/\alpha$ as piecewise constant function, in addition to the $u(x)$ function which involves the integrals of $1/\alpha$. A class creating the functions we need and a method for evaluating u, can take the form

```
class SerialLayers:
    """
    b: coordinates of boundaries of layers, b[0] is left boundary
    and b[-1] is right boundary of the domain [0,L].
    a: values of the functions in each layer (len(a) = len(b)-1).
    U_0: u(x) value at left boundary x=0=b[0].
    U_L: u(x) value at right boundary x=L=b[0].
    """

    def __init__(self, a, b, U_0, U_L, eps=0):
        self.a, self.b = np.asarray(a), np.asarray(b)
        self.eps = eps  # smoothing parameter for smoothed a
        self.U_0, self.U_L = U_0, U_L

        a_data = [[bi, ai] for bi, ai in zip(self.b, self.a)]
        domain = [b[0], b[-1]]
        self.a_func = PiecewiseConstant(domain, a_data, eps)

        # inv_a = 1/a is needed in formulas
        inv_a_data = [[bi, 1./ai] for bi, ai in zip(self.b, self.a)]
        self.inv_a_func = \
             PiecewiseConstant(domain, inv_a_data, eps)
        self.integral_of_inv_a_func = \
             IntegratedPiecewiseConstant(domain, inv_a_data, eps)
        # Denominator in the exact formula is constant
        self.inv_a_0L = self.integral_of_inv_a_func(b[-1])

    def __call__(self, x):
        solution = self.U_0 + (self.U_L-self.U_0)*\
                   self.integral_of_inv_a_func(x)/self.inv_a_0L
        return solution
```

A visualization method is also convenient to have. Below we plot $u(x)$ along with $\alpha(x)$ (which works well as long as $\max \alpha(x)$ is of the same size as $\max u = \max(U_0, U_L)$).

```
class SerialLayers:
    ...

    def plot(self):
        x, y_a = self.a_func.plot()
        x = np.asarray(x); y_a = np.asarray(y_a)
        y_u = self.u_exact(x)
        import matplotlib.pyplot as plt
        plt.figure()
        plt.plot(x, y_u, 'b')
        plt.hold('on')  # Matlab style
        plt.plot(x, y_a, 'r')
        ymin = -0.1
        ymax = 1.2*max(y_u.max(), y_a.max())
        plt.axis([x[0], x[-1], ymin, ymax])
        plt.legend(['solution $u$', 'coefficient $a$'], loc='upper left')
        if self.eps > 0:
            plt.title('Smoothing eps: %s' % self.eps)
        plt.savefig('tmp.pdf')
```

```
        plt.savefig('tmp.png')
        plt.show()
```

Figure 12 shows the case where

```
b = [0, 0.25, 0.5, 1]   # material boundaries
a = [0.2, 0.4, 4]       # material values
U_0 = 0.5;  U_L = 5     # boundary conditions
```

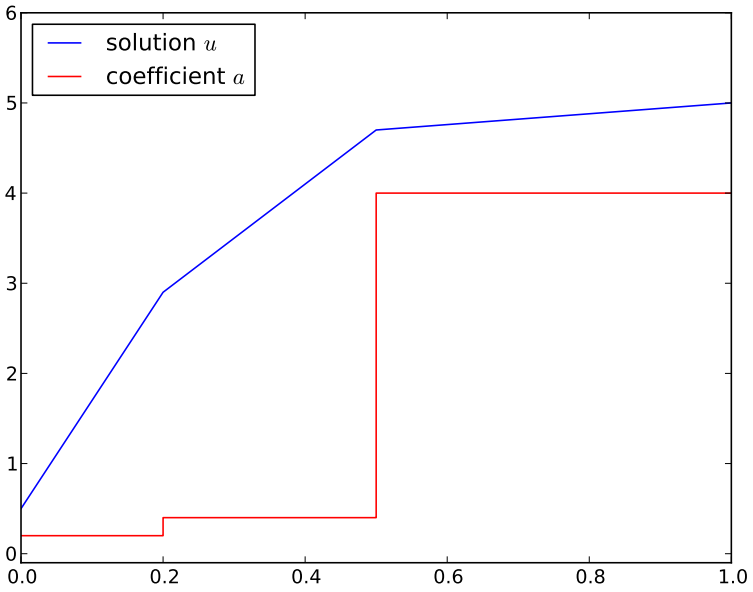

Figure 12: Solution of the stationary diffusion equation corresponding to a piecewise constant diffusion coefficient.

By adding the `eps` parameter to the constructor of the `SerialLayers` class, we can experiment with smoothed versions of α and see the (small) impact on u. Figure 13 shows the result.

4.5 Diffusion equation in axi-symmetric geometries

Suppose we have a diffusion process taking care in a straight tube with radius R. We assume axi-symmetry such that u is just a function of r and t. A model problem is

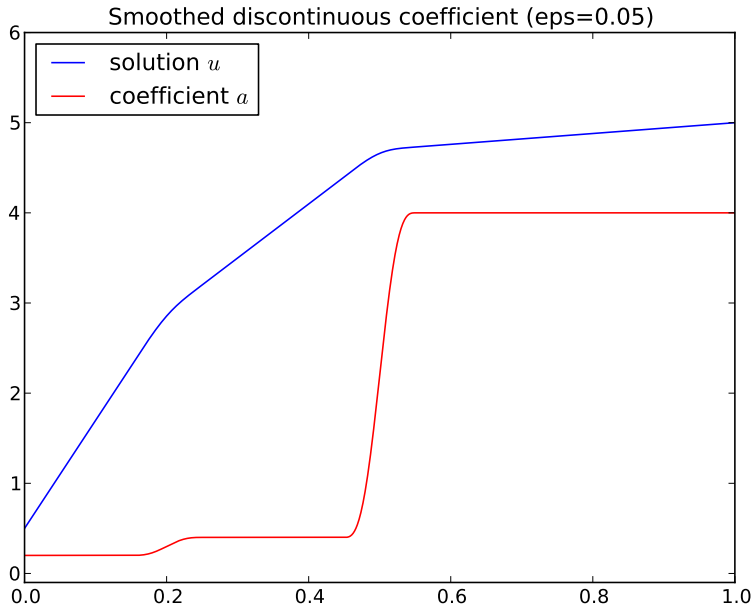

Figure 13: Solution of the stationary diffusion equation corresponding to a *smoothed* piecewise constant diffusion coefficient.

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha(r)\frac{\partial u}{\partial r}\right) + f(t), \qquad r \in (0,R),\ t \in (0,T], \tag{72}$$

$$\frac{\partial u}{\partial r}(0,t) = 0, \qquad t \in (0,T], \tag{73}$$

$$u(R,t) = 0, \qquad t \in (0,T], \tag{74}$$

$$u(r,0) = I(r), \qquad r \in [0,R]. \tag{75}$$

The condition (73) is a necessary symmetry condition at $r = 0$, while (74) could be any Dirichlet or Neumann condition (or Robin condition in case of cooling or heating).

The finite difference approximation at $r = 0$ of the spatial derivative term is the only new challenge in this problem. Let us in case of constant α expand the derivative to

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}.$$

The last term faces a difficulty at $r = 0$ since it becomes a $0/0$ expression because of the symmetry condition. L'Hosptial's rule can be used:

$$\lim_{r\to 0}\frac{1}{r}\frac{\partial u}{\partial r}=\lim_{r\to 0}\frac{\partial^2 u}{\partial r^2}\,.$$

The PDE at $r=0$ therefore becomes

$$\frac{\partial u}{\partial t}=2\alpha\frac{\partial^2 u}{\partial r^2}+f(t)\,. \tag{76}$$

For a variable coefficient $\alpha(r)$ the expanded derivative reads

$$\alpha(r)\frac{\partial^2 u}{\partial r^2}+\frac{1}{r}(\alpha(r)+r\alpha'(r))\frac{\partial u}{\partial r}\,.$$

We have that the limit of a product[6] is

$$\lim_{r\to 0}\frac{1}{r}(\alpha(r)+r\alpha'(r))\frac{\partial u}{\partial r}=\lim_{r\to 0}(\alpha(r)+r\alpha'(r))\ \lim_{x\to c}\frac{1}{r}\frac{\partial u}{\partial r}\,.$$

The second limit becomes as above, so the PDE at $r=0$, assuming $(\alpha(0)+r\alpha'(0))\neq 0$, looks like

$$\frac{\partial u}{\partial t}=(2\alpha+r\alpha')\frac{\partial^2 u}{\partial r^2}+f(t)\,. \tag{77}$$

The second-order derivative is discretized in the usual way. Consider first constant α:

$$2\alpha\frac{\partial^2}{\partial r^2}u(r_0,t_n)\approx[2\alpha 2D_rD_ru]_0^n=2\alpha\frac{u_1^n-2u_0^n+u_{-1}^n}{\Delta r^2}\,.$$

The fictitious value u_{-1}^n can be eliminated using the discrete symmetry condition

$$[D_{2r}u=0]_0^n\quad\Rightarrow\quad u_{-1}^n=u_1^n,$$

which then gives the modified approximation to the second-order derivative of u in r at $r=0$:

$$4\alpha\frac{u_1^n-u_0^n}{\Delta r^2}\,. \tag{78}$$

With variable α we simply get

$$(2\alpha+r\alpha')2D_rD_ru]_0^n=(2\alpha(0)+r\alpha'(0))\frac{u_1^n-2u_0^n+u_{-1}^n}{\Delta r^2}\,.$$

The discretization of the second-order derivative in r at another internal mesh point is straightforward:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\alpha\frac{\partial u}{\partial r}\right)\bigg|_{r=r_i}^{t=t_n}\approx[r^{-1}D_r(r\alpha D_ru)]_i^n=\frac{1}{\Delta r^2}\left(r_{i+\frac{1}{2}}\alpha_{i+\frac{1}{2}}(u_{i+1}^n-u_i^n)-r_{i-\frac{1}{2}}\alpha_{i-\frac{1}{2}}(u_i^n-u_{i-1}^n)\right)\,.$$

θ-rule in time...

[6] https://en.wikibooks.org/wiki/Calculus/Proofs_of_Some_Basic_Limit_Rules

4.6 Diffusion equation in spherically-symmetric geometries

Discretization in spherical coordinates. Let us now pose the problem from Section 4.5 in spherical coordinates, where u only depends on the radial coordinate r and time t. That is, we have spherical symmetry. For simplicity we restrict the diffusion coefficient α to be a constant. The PDE reads

$$\frac{\partial u}{\partial t} = \frac{\alpha}{r^\gamma}\frac{\partial}{\partial r}\left(r^\gamma \frac{\partial u}{\partial r}\right) + f(t), \tag{79}$$

for $r \in (0, R)$ and $t \in (0, T]$. The parameter γ is 2 for spherically-symmetric problems and 1 for axi-symmetric problems. The boundary and initial conditions have the same mathematical form as in (72)-(75).

Since the PDE in spherical coordinates has the same form as the PDE in Section 4.5, just with the γ parameter being different, we can use the same discretization approach. At the origin $r = 0$ we get problems with the term

$$\frac{\gamma}{r}\frac{\partial u}{\partial t},$$

but L'Hosptial's rule shows that this term equals $\gamma\partial^2 u/\partial r^2$, and the PDE at $r = 0$ becomes

$$\frac{\partial u}{\partial t} = (\gamma + 1)\alpha\frac{\partial^2 u}{\partial r^2} + f(t)\,. \tag{80}$$

Same discretization, write up with γ.

Discretization in Cartesian coordinates. The spherically-symmetric spatial derivative can be transformed to the Cartesian counterpart by introducing

$$v(r, t) = ru(r, t)\,.$$

Inserting $u = v/r$ in the PDE yields

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(\alpha(r)r^2\frac{\partial u}{\partial t}\right),$$

and then

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + \alpha\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(\alpha\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one Cartesian coordinate.

5 Random walk

6 Exercises

Exercise 6: Stabilizing the Crank-Nicolson method by Rannacher time stepping

It is well known that the Crank-Nicolson method may give rise to non-physical oscillations in the solution of diffusion equations if the initial data exhibit jumps (see Section 3.7). Rannacher [2] suggested a stabilizing technique consisting of using the Backward Euler scheme for the first two time steps with step length $\frac{1}{2}\Delta t$. One can generalize this idea to taking $2m$ time steps of size $\frac{1}{2}\Delta t$ with the Backward Euler method and then continuing with the Crank-Nicolson method, which is of second-order in time. The idea is that the high frequencies of the initial solution are quickly damped out, and the Backward Euler scheme treats these high frequencies correctly. Thereafter, the high frequency content of the solution is gone and the Crank-Nicolson method will do well.

Test this idea for $m = 1, 2, 3$ on a diffusion problem with a discontinuous initial condition. Measure the convergence rate using the solution (45) with the boundary conditions (46)-(47) for t values such that the conditions are in the vicinity of ± 1. For example, $t < 5a1.6 \cdot 10^{-2}$ makes the solution diffusion from a step to almost a straight line. The program `diffu_erf_sol.py` shows how to compute the analytical solution.

Project 7: Energy estimates for diffusion problems

This project concerns so-called *energy estimates* for diffusion problems that can be used for qualitative analytical insight and for verification of implementations.

a) We start with a 1D homogeneous diffusion equation with zero Dirichlet conditions:

$$u_t = \alpha u_x x, \qquad x \in \Omega = (0, L),\ t \in (0, T], \tag{81}$$

$$u(0, t) = u(L, t) = 0, \qquad t \in (0, T], \tag{82}$$

$$u(x, 0) = I(x), \qquad x \in [0, L]\,. \tag{83}$$

The energy estimate for this problem reads

$$||u||_{L^2} \leq ||I||_{L^2}, \tag{84}$$

where the $||\cdot||_{L^2}$ norm is defined by

$$||g||_{L^2} = \sqrt{\int_0^L g^2 dx}\,. \tag{85}$$

The quantify $||u||_{L^2}$ or $\frac{1}{2}||u||_{L^2}$ is known as the *energy* of the solution, although it is not the physical energy of the system. A mathematical tradition has introduced the notion *energy* in this context.

The estimate (84) says that the "size of u" never exceeds that of the initial condition, or more equivalently, that the area under the u curve decreases with time.

To show (84), multiply the PDE by u and integrate from 0 to L. Use that uu_t can be expressed as the time derivative of u^2 and that u_xxu can integrated by parts to form an integrand u_x^2. Show that the time derivative of $||u||_{L^2}^2$ must be less than or equal to zero. Integrate this expression and derive (84).

b) Now we address a slightly different problem,

$$u_t = \alpha u_x x + f(x,t), \qquad x \in \Omega = (0,L),\ t \in (0,T], \tag{86}$$
$$u(0,t) = u(L,t) = 0, \qquad t \in (0,T], \tag{87}$$
$$u(x,0) = 0, \qquad x \in [0,L]\,. \tag{88}$$

The associated energy estimate is

$$||u||_{L^2} \leq ||f||_{L^2}\,. \tag{89}$$

(This result is more difficult to derive.)

Now consider the compound problem with an initial condition $I(x)$ and a right-hand side $f(x,t)$:

$$u_t = \alpha u_x x + f(x,t), \qquad x \in \Omega = (0,L),\ t \in (0,T], \tag{90}$$
$$u(0,t) = u(L,t) = 0, \qquad t \in (0,T], \tag{91}$$
$$u(x,0) = I(x), \qquad x \in [0,L]\,. \tag{92}$$

Show that if w_1 fulfills (81)-(83) and w_2 fulfills (86)-(88), then $u = w_1 + w_2$ is the solution of (90)-(92). Using the triangle inequality for norms,

$$||a+b|| \leq ||a|| + ||b||,$$

show that the energy estimate for (90)-(92) becomes

$$||u||_{L^2} \leq ||I||_{L^2} + ||f||_{L^2}\,. \tag{93}$$

c) One application of (93) is to prove uniqueness of the solution. Suppose u_1 and u_2 both fulfill (90)-(92). Show that $u = u_1 - u_2$ then fulfills (90)-(92) with $f = 0$ and $I = 0$. Use (93) to deduce that the energy must be zero for all times and therefore that $u_1 = u_2$, which proves that the solution is unique.

d) Generalize (93) to a 2D/3D diffusion equation $u_t = \nabla \cdot (\alpha \nabla u)$ for $x \in \Omega$.

Hint. Use integration by parts in multi dimensions:

$$\int_\Omega u\nabla\cdot(\alpha\nabla u)\,\mathrm{d}x = -\int_\Omega \alpha\nabla u\cdot\nabla u\,\mathrm{d}x + \int_{\partial\Omega} u\alpha\frac{\partial u}{\partial n},$$

where $\frac{\partial u}{\partial n} = \boldsymbol{n}\cdot\nabla u$, $\boldsymbol{n}$ being the outward unit normal to the boundary $\partial\Omega$ of the domain Ω.

e) Now we also consider the multi-dimensional PDE $u_t = \nabla\cdot(\alpha\nabla u)$. Integrate both sides over Ω and use Gauss' divergence theorem, $\int_\Omega \nabla\cdot\boldsymbol{q}\,\mathrm{d}x = \int_{\partial\Omega}\boldsymbol{q}\cdot\boldsymbol{n}\,\mathrm{d}s$ for a vector field $\boldsymbol{q}$. Show that if we have homogeneous Neumann conditions on the boundary, $\partial u/\partial n = 0$, area under the u surface remains constant in time and

$$\int_\Omega u\,\mathrm{d}x = \int_\Omega I\,\mathrm{d}x\,. \tag{94}$$

f) Establish a code in 1D, 2D, or 3D that can solve a diffusion equation with a source term f, initial condition I, and zero Dirichlet or Neumann conditions on the whole boundary.

We can use (93) and (94) as a partial verification of the code. Choose some functions f and I and check that (93) is obeyed at any time when zero Dirichlet conditions are used. Iterate over the same I functions and check that (94) is fulfilled when using zero Neumann conditions.

g) Make a list of some possible bugs in the code, such as indexing errors in arrays, failure to set the correct boundary conditions, evaluation of a term at a wrong time level, and similar. For each of the bugs, see if the verification tests from the previous subexercise pass or fail. This investigation shows how strong the energy estimates and the estimate (94) are for pointing out errors in the implementation.
Filename: `diffu_energy`.

References

[1] H. P. Langtangen. *Scaling of Differential Equations*. 2015. `http://tinyurl.com/qfjgxmf/web`.

[2] R. Rannacher. Finite element solution of diffusion problems with irregular data. *Numerische Mathematik*, 43:309–327, 1984.

Index