COMP52215

Continuous Systems

Prof. Anthony Yeates, Durham University, Mathematical Sciences

February 12, 2021

Contents

1 Introduction

This is a crash course in the numerical solution of partial differential equations. We will use Python to illustrate and experiment with different techniques.

The quantities we are interested in calculating are continuous functions of more than one variable, for example $u(x, y)$ or $u(x, t)$.

If we differentiate such a function with respect to one variable at a time, keeping the others constant, it is called a *partial derivative* – e.g.

$$\frac{\partial u(x,y)}{\partial x} = \lim_{h\to 0} \frac{u(x+h,y) - u(x,y)}{h}, \qquad \frac{\partial u(x,y)}{\partial y} = \lim_{h\to 0} \frac{u(x,y+h) - u(x,y)}{h}. \tag{1.0.1}$$

It is convenient to use subscript notation to indicate partial derivatives, thus

$$u_x \equiv \frac{\partial u}{\partial x}, \qquad u_y \equiv \frac{\partial u}{\partial y}. \tag{1.0.2}$$

Higher derivatives have more than one subscript:

$$u_{xx} \equiv \frac{\partial^2 u}{\partial x^2}, \qquad u_{xt} \equiv \frac{\partial}{\partial t}\left(\frac{\partial u}{\partial x}\right) \equiv \frac{\partial^2 u}{\partial t \partial x}. \tag{1.0.3}$$

A *partial differential equation* (*PDE*) is an equation involving a function u of two or more variables and its derivatives. The *order* of a PDE is the highest derivative occurring. A PDE is *linear* if u and all of its derivatives appear linearly (once per term in the equation).

1.1 Course overview

There are so many examples of PDEs arising in the physical sciences that it is impossible to give a comprehensive list. This is not our focus in this course. Rather, our aims are to:

1. Introduce the basic numerical methods for solving PDEs through three examples:
   (a) The *heat equation* $u_t = u_{xx}$ (a *parabolic* equation).
   (b) The *advection equation* $u_t + cu_x = 0$ (a *hyperbolic* equation).
   (c) The *Poisson equation* $u_{xx} + u_{yy} = f(x, y)$ (an *elliptic* equation).
   You will see some other PDEs in the tutorials.
2. Equip you to be able to solve other PDEs (or systems of PDEs) that you may encounter, or at least know where to start!

We won't worry about the technical definition of "parabolic", "hyperbolic" or "elliptic", but rather will see how these types of equation behave in practice. We will see that their solutions, and hence the numerical methods needed to solve them, are different for each type of equation.

The central idea behind the numerical solution of PDEs is very simple: *turn the continuous differential equation into a discrete (finite) set of algebraic equations, which we can solve.* Different methods just correspond to different ways of doing this, but some work better than others.

Throughout the course we will use Python, including standard packages such as `numpy`. You should be able to run the examples and solve the tutorial problems in either Python 2 or Python 3, although you will also find a pencil and paper useful for the tutorials.

There will be a single summative assignment set in week 18.

2 Parabolic equations

Loosely speaking, parabolic equations are equations where there is a "one-way" time evolution, so that information is "lost" over time. The simplest example of a parabolic equation is the heat equation

$$u_t = u_{xx}. \tag{2.0.1}$$

In this section we will study finite difference methods for parabolic equations like the heat equation.

2.1 Forward-Euler method for the heat equation

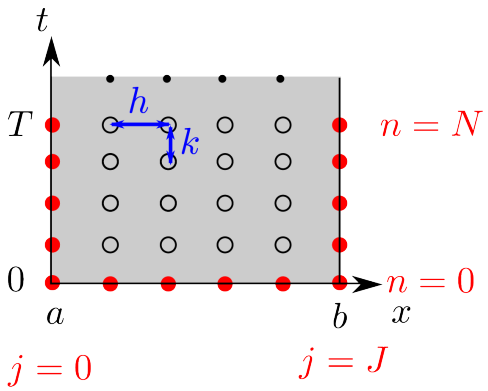
Suppose u is a function of x and t, and that we are given initial conditions $u(x,0) = u_0(x)$ in some finite domain $x \in [a,b]$. We need to find $u(x,t)$ satisfying this initial condition as well as the PDE – for illustration, we will take the heat equation

$$u_t = u_{xx}. \tag{2.1.1}$$

Step 1. Define the mesh. First, we need to define a discrete grid, or *mesh*, for our solution. For simplicity, let us take equally-spaced points

$$x_j = a + jh, \quad j = 0, 1, \ldots, J, \tag{2.1.2}$$
$$t_n = nk, \quad n = 0, 1, \ldots, N. \tag{2.1.3}$$

The constants $h = \dfrac{b-a}{J}$ and $k = \dfrac{T}{N}$ are the *grid spacing* in x and t (up to end time $t = T$).



We will use a capital U to denote the numerical solution. Since this is only known at mesh points, we use indices to denote the discrete set of values, thus

$$U_j^n \equiv U(x_j, t_n). \tag{2.1.4}$$

Step 2. Discretize the PDE. This is the tricky part: we need to "impose" the PDE using only the values of U_j^n at the discrete mesh points. In *finite-difference* methods, we approximate the partial derivatives u_t and u_{xx} by finite differences, just as we did for ODEs last term.

For example, we could use a forward difference for u_t, so that

$$u_t(x_j, t_n) \approx \frac{U_j^{n+1} - U_j^n}{k}. \tag{2.1.5}$$

For u_{xx}, we could use central differences twice, so

$$u_{xx}(x_j, t_n) \approx \frac{u_x(x_j + \frac{1}{2}h, t_n) - u_x(x_j - \frac{1}{2}h, t_n)}{h} \tag{2.1.6}$$

$$\approx \frac{1}{h}\left(\frac{U_{j+1}^n - U_j^n}{h} - \frac{U_j^n - U_{j-1}^n}{h}\right) \tag{2.1.7}$$

$$= \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}. \tag{2.1.8}$$

Putting (2.1.5) and (2.1.8) together, we try approximating the heat equation as

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \tag{2.1.9}$$

$$\iff \quad U_j^{n+1} = U_j^n + \mu\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right), \tag{2.1.10}$$

where $\mu \equiv \frac{k}{h^2}$. This is called the *forward Euler method*. If we specify boundary conditions U_0^n and U_J^n, then the interior points satisfy the following system of linear equations:

$$U_1^{n+1} = U_1^n + \mu\left(U_2^n - 2U_1^n + U_0^n\right), \tag{2.1.11}$$

$$U_2^{n+1} = U_2^n + \mu\left(U_3^n - 2U_2^n + U_1^n\right), \tag{2.1.12}$$

$$\vdots \qquad \vdots \qquad \vdots$$

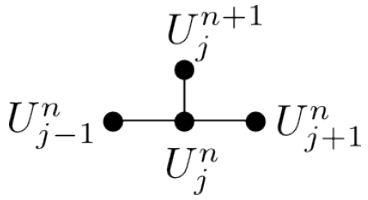
$$U_{J-2}^{n+1} = U_{J-2}^n + \mu\left(U_{J-1}^n - 2U_{J-2}^n + U_{J-3}^n\right), \tag{2.1.13}$$

$$U_{J-1}^{n+1} = U_{J-1}^n + \mu\left(U_J^n - 2U_{J-1}^n + U_{J-2}^n\right). \tag{2.1.14}$$

Notice how we have reduced the PDE to a finite-dimensional algebraic problem.

Step 3. Solve the discrete system. Notice that our method is *explicit*, meaning that if we know all of the U_j^n at time level n, we can calculate one-by-one those at level $n+1$. In other words, we can march forward from the initial data $U_j^0 = u_0(x_j)$.

It is useful to think of the formula (2.1.10) in terms of a *stencil* that we apply at each point on the grid:



This is coded up in Python in `heat_fwdeul.py`. Most of the code is concerned with plotting the solution, and the main calculation loop is very simple:

```
for n in range(N):
    U[1:-1] += mu*(U[:-2] - 2*U[1:-1] + U[2:])
```

A couple of points on the implementation:

1. For an explicit method like this, we only need to store the current $\boldsymbol{U}^n$, so I am using a one-dimensional array `U` and updating it "in place" as we step forward in time.
2. In this example, we are imposing the fixed boundary conditions

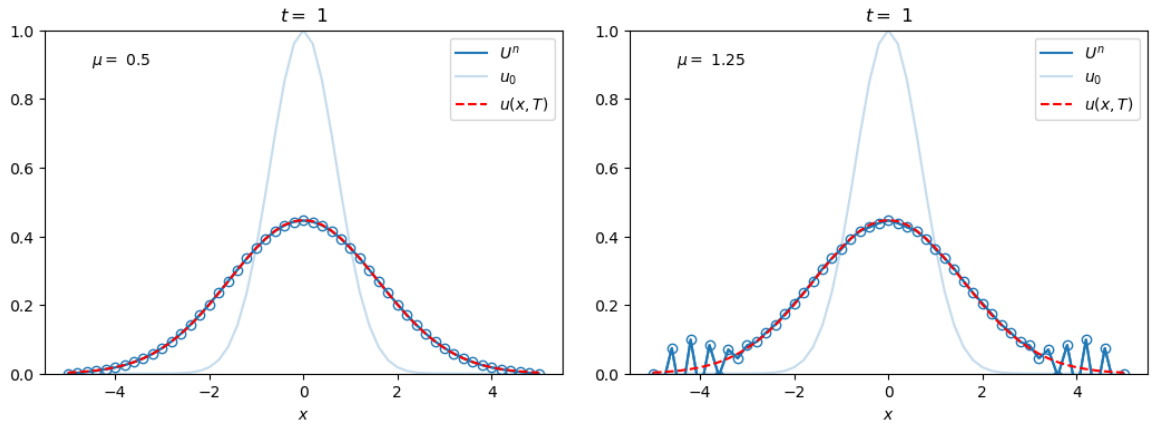
$$U_0^n = u_0(x_0), \qquad U_J^n = u_0(x_J),$$

   so we only need to update the *interior* values `U[1:-1]`.

Starting with a Gaussian "hump" $u_0(x) = \mathrm{e}^{-x^2}$, the effect of the heat equation is to spread it out. In the plots below, the initial condition is shown in light blue, and the forward-Euler solution at time $T = 1$ in dark blue. The dashed red line is the exact solution, which is known to be

$$u(x,t) = \frac{1}{\sqrt{4t+1}} \exp\left(-\frac{x^2}{(4t+1)}\right). \tag{2.1.15}$$

I took $J = 50$ in both cases, but on the left I had $N = 50$ and on the right $N = 20$.



Notice how it worked for $N = 50$ but not $N = 20$! Really, it is the value of μ that matters. If μ is too large, the forward-Euler method is *unstable*, meaning that errors grow in time. It is a general feature of explicit methods that they are only stable if the time step satisfies some restriction.

Important: it is our numerical solution that is wrong here – the real solution of the PDE does not behave like this!

What about the accuracy of the solution? In other words, how closely does $\boldsymbol{U}^N$ approximate $u(x, T)$? We measure this with the *global truncation error*, defined at step n as

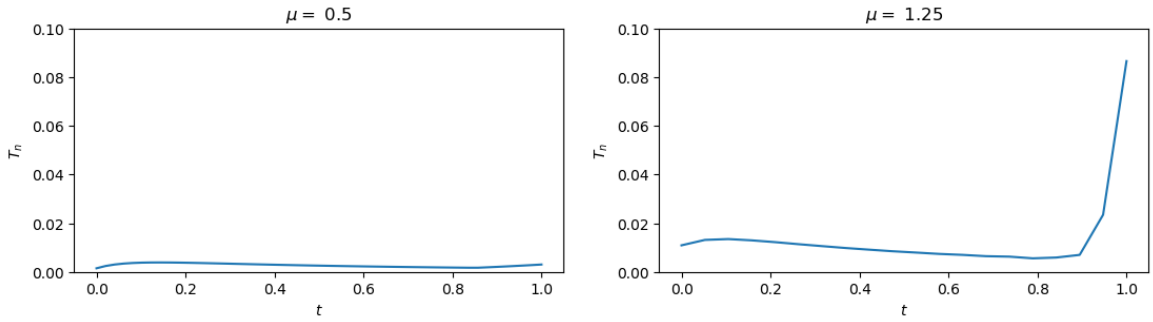
$$T_n = \|\boldsymbol{U}^n - \boldsymbol{u}(x_j, t_n)\|_\infty := \max_{j\in[0,J]} |U_j^n - u(x_j, t_n)|. \tag{2.1.16}$$

► For the mathematically inclined, this is called the ℓ_∞-norm and is the discrete version of the L_∞-norm for continuous functions.

In our Gaussian example, we can compare the error for the two meshes using the Python code

```
error[n] = np.max(np.abs(U - uexact(x, t)))
```

which gives



Notice that (i) the error is lower when μ is smaller, and (ii) the instability for $\mu = 1.25$ eventually causes the error to grow. We will see how to analyse the error as a function of k and h below.

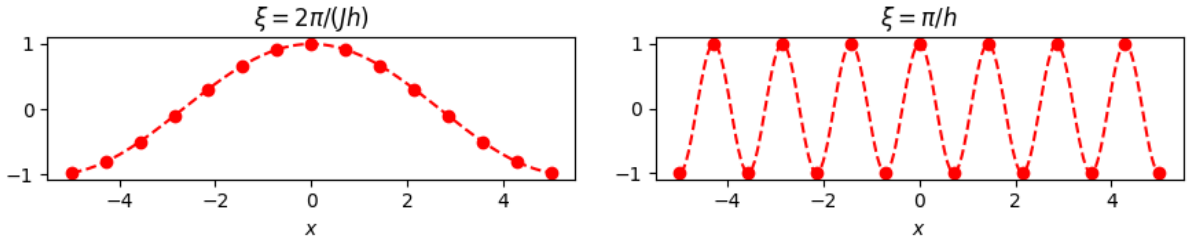
2.2 Stability

For linear PDEs on regular grids we can assess the stability of a finite-difference method using *von Neumann analysis*. The idea is to consider the behaviour of a single discrete Fourier mode, in other words a sine wave of the form

$$U_j^n = \mathrm{e}^{i\xi x_j} = \mathrm{e}^{i\xi jh}, \tag{2.2.1}$$

for some wavenumber ξ (where $i \equiv \sqrt{-1}$) and we have set $a = x_0 = 0$ (without loss of generality). It is conventional (and much easier) to use this complex form, with the implicit understanding that U_j^n is only the real part.

The following plot shows $\cos(\xi jh)$ for two different modes, where $J = 15$. The mode on the right has the highest wavenumber that can be resolved on this mesh (with two grid points per wavelength):



Because the PDE (or its discrete approximation) is linear, any solution can be written as a superposition of these Fourier modes, so if we could determine the stability of each Fourier mode we would know about the stability of any possible solution.

Let us illustrate the analysis using our forward-Euler method for the heat equation. If we have a Fourier mode (2.2.1), then substituting it into the formula (2.1.10) gives

$$\begin{aligned}
U_j^{n+1} &= \mathrm{e}^{i\xi jh} + \mu\left(\mathrm{e}^{i\xi(j+1)h} - 2\mathrm{e}^{i\xi jh} + \mathrm{e}^{i\xi(j-1)h}\right) && (2.2.2)\\
&= \mathrm{e}^{i\xi jh} + \mu\left(\mathrm{e}^{i\xi h} - 2 + \mathrm{e}^{-i\xi h}\right)\mathrm{e}^{i\xi jh} && (2.2.3)\\
&= \left[1 - 2\mu + \mu(\mathrm{e}^{i\xi h} + \mathrm{e}^{-i\xi h})\right] U_j^n && (2.2.4)\\
&= \left[1 - 2\mu + 2\mu\cos(\xi h)\right] U_j^n && (2.2.5)\\
&= \left[1 - 4\mu\sin^2(\xi h/2)\right] U_j^n. && (2.2.6)
\end{aligned}$$

So under the forward-Euler scheme our Fourier mode evolves by changing its amplitude, with so-called *amplification factor*

$$g(\xi) = 1 - 4\mu \sin^2\left(\frac{\xi h}{2}\right). \tag{2.2.7}$$

It is clear that stability will require $|g(\xi)| \leq 1$ for all ξ. Since

$$\max_{\xi} |g(\xi)| = \max\{1, |1 - 4\mu|\}, \tag{2.2.8}$$

we see that stability requires

$$|1 - 4\mu| \leq 1 \quad \Longleftrightarrow \quad -1 \leq 1 - 4\mu \leq 1 \quad \Longleftrightarrow \quad 0 \leq \mu \leq \tfrac{1}{2}. \tag{2.2.9}$$

In other words, stability of forward-Euler imposes the timestep restriction $k \leq \frac{1}{2}h^2$, as we saw in our numerical experiments.

The von Neumann analysis also tells us which Fourier mode is most unstable. This will be the wavenumber ξ with largest amplification factor. In this case, it will be when $\sin(\xi h/2) = \pm 1$, or in other words $\xi = \pm\pi/h$. These are the modes

$$U_j^n = g^n e^{\pm i\pi j} \tag{2.2.10}$$

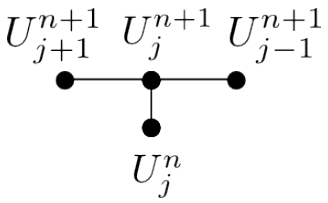
which have wavelength $2h$. This explains the wavelength of the oscillations seen earlier.

2.3 Implicit methods

To avoid timestep restrictions, we can use an *implicit* method. We will illustrate this for the heat equation $u_t = u_{xx}$ and the so-called *backward-Euler method*. This is the same as forward-Euler, except that we discretize the spatial derivatives at time t_{n+1}, so

$$U_j^{n+1} = U_j^n + \mu\left(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}\right). \tag{2.3.1}$$

The corresponding stencil is:



Now the solution is more difficult because there are unknowns on both sides. We can rewrite the formula with all of the unknowns on the left:

$$-\mu U_{j+1}^{n+1} + (1 + 2\mu)U_j^{n+1} - \mu U_{j-1}^{n+1} = U_j^n. \tag{2.3.2}$$

Now the linear system becomes

$$-\mu U_2^{n+1} + (1 + 2\mu)U_1^{n+1} - \mu U_0^{n+1} = U_1^n, \tag{2.3.3}$$
$$-\mu U_3^{n+1} + (1 + 2\mu)U_2^{n+1} - \mu U_1^{n+1} = U_2^n, \tag{2.3.4}$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$
$$-\mu U_{J-1}^{n+1} + (1 + 2\mu)U_{J-2}^{n+1} - \mu U_{J-3}^{n+1} = U_{J-2}^n, \tag{2.3.5}$$
$$-\mu U_J^{n+1} + (1 + 2\mu)U_{J-1}^{n+1} - \mu U_{J-2}^{n+1} = U_{J-1}^n. \tag{2.3.6}$$

In matrix form,

$$\begin{bmatrix} 1+2\mu & -\mu & - & \dots & 0 \\ -\mu & 1+2\mu & -\mu & \ddots & \vdots \\ 0 & -\mu & 1+2\mu & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -\mu \\ 0 & \dots & 0 & -\mu & 1+2\mu \end{bmatrix} \begin{bmatrix} U_1^{n+1} \\ U_2^{n+1} \\ U_3^{n+1} \\ \vdots \\ U_{J-1}^{n+1} \end{bmatrix} = \begin{bmatrix} U_1^n \\ U_2^n \\ U_3^n \\ \vdots \\ U_{J-1}^n \end{bmatrix} + \begin{bmatrix} \mu U_0^{n+1} \\ 0 \\ \vdots \\ 0 \\ \mu U_J^{n+1} \end{bmatrix}. \tag{2.3.7}$$

So we have to solve this linear system of size $(J-1) \times (J-1)$ at every timestep. Notice how the boundary conditions can be written in a separate vector.

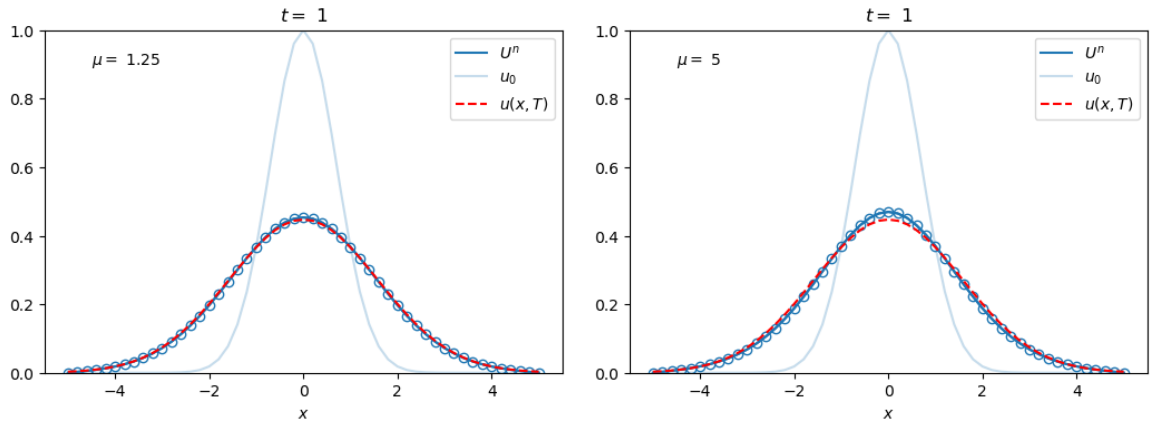
To implement this, let's use our previous example of $u_0(x) = e^{-x^2}$. We just need to store the three non-zero diagonals of the matrix:

```
A = np.zeros((3, J-1))
A[0,1:] = -mu  # upper diagonal
A[1,:] = 1 + 2*mu  # main diagonal
A[2,:-1] = -mu  # lower diagonal
```

Then we can take advantage of a specialised solver to solve the banded linear system:

```
U[1:-1] = scipy.linalg.solve_banded((1,1), A, U[1:-1] + mu*Ub)
```

The vector `Ub` contains the boundary conditions (zero in this example). Here are the resulting solutions for $N = 20$ and $N = 5$ (both with $J = 50$):



Notice how it remains stable even for only 5 timesteps, although the accuracy does suffer.

To explain the improved stability, we can use von Neumann analysis again. Again we start with a Fourier mode $U_j^n = e^{i\xi jh}$, and we assume that $U_j^{n+1} = g(\xi)U_j^n$ for some amplification factor $g(\xi)$ that

we need to determine. Substituting into (2.3.1) gives

$$ge^{i\xi jh} = e^{i\xi jh} + \mu\left(ge^{i\xi(j+1)h} - 2ge^{i\xi jh} + ge^{i\xi(j-1)h}\right), \tag{2.3.8}$$

$$\implies \quad g = 1 + \mu g\left(e^{i\xi h} - 2 + e^{-i\xi h}\right), \tag{2.3.9}$$

$$= 1 - 2\mu g\left[1 - \cos(\xi h)\right], \tag{2.3.10}$$

$$= 1 - 4\mu g \sin^2\left(\frac{\xi h}{2}\right), \tag{2.3.11}$$

$$\implies \quad g(\xi) = \frac{1}{1 + 4\mu \sin^2(\xi h/2)}. \tag{2.3.12}$$

It follows that $0 < g \leq 1$ for all μ, meaning that backward-Euler for the heat equation is *unconditionally stable*.

► Implicit methods are mainly needed for parabolic equations like the heat equation. These are equations where the solution $u(x, t + k)$ depends on the solution $u(x, t)$ at all values of x, so that "information travels infinitely fast". It intuitively makes sense to couple together all x values in the numerical solution. For "hyperbolic" equations like the wave equation, the solution has only a finite domain of dependence, so implicit schemes are not required.

2.4 Accuracy and convergence

There is a general rule with finite difference methods that *convergence to the solution (as the grid gets finer) requires both accuracy and stability*.

We have already seen stability, but what about accuracy? This is the (fairly obvious) requirement that the discrete formula should tend to the original PDE in the limit of small h and k. In other words, the method is "solving the correct equation at each time step". Such a method is called *consistent*.

To verify this, we define the *truncation error* to be the extent to which the exact solution fails to satisfy the finite-difference formula, when written in the same dimensional form as the original PDE. For example, consider again the forward-Euler method for the heat equation. Let u_j^n be the exact solution on the mesh at $t = t_n$, so $u_j^n \equiv u(x_j, t_n)$. Then the truncation error is defined to be

$$\varepsilon_j^n = \frac{u_j^{n+1} - u_j^n}{k} - \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}. \tag{2.4.1}$$

► Notice that it is conventional here to write the forward-Euler formula in the original form (2.1.9), rather than the rearranged form (2.1.10).

Consistency requires $\varepsilon_j^n \to 0$ as $h, k \to 0$. The standard way to verify this is to expand each term in ε_j^n in a Taylor series about the common point (x_j, t_n). We have

$$u_{j+1}^n \equiv u(x_j + h, t_n) = u_j^n + h(u_x)_j^n + \frac{h^2}{2}(u_{xx})_j^n + \frac{h^3}{6}(u_{xxx})_j^n + \frac{h^4}{24}(u_{xxxx})_j^n + O(h^5), \tag{2.4.2}$$

$$u_{j-1}^n \equiv u(x_j - h, t_n) = u_j^n - h(u_x)_j^n + \frac{h^2}{2}(u_{xx})_j^n - \frac{h^3}{6}(u_{xxx})_j^n + \frac{h^4}{24}(u_{xxxx})_j^n + O(h^5), \tag{2.4.3}$$

$$u_j^{n+1} \equiv u(x_k, t_n + k) = u_j^n + k(u_t)_j^n + \frac{k^2}{2}(u_{tt})_j^n + O(k^3). \tag{2.4.4}$$

So substituting in (2.4.1) gives

$$\varepsilon_j^n = \left(u_t\right)_j^n + \frac{k}{2}\left(u_{tt}\right)_j^n + O(k^2) - \left(u_{xx}\right)_j^n - \frac{h^2}{12}\left(u_{xxxx}\right)_j^n + O(h^4), \tag{2.4.5}$$

$$= \frac{k}{2}\left(u_{tt}\right)_j^n - \frac{h^2}{12}\left(u_{xxxx}\right)_j^n + O(k^2 + h^4). \tag{2.4.6}$$

In the last step, we used the fact that u_j^n is the exact solution of the PDE $u_t = u_{xx}$. From this expression, we see that $\varepsilon_j^n \to 0$ as $k \to 0$ and $h \to 0$. In other words, the forward-Euler method is consistent with the heat equation.

► An inconsistent method would have some term in ε_j^n that does not tend to zero as h and k tend to zero.

The remaining powers of h and k tell you how rapidly $\varepsilon_j^n \to 0$. This is what we call the *order of accuracy*. For this method, we will have asymptotically that $\varepsilon_j^n = O(k) + O(h^2)$, so we say that the method is *first-order* in time and *second-order* in space. A higher order of accuracy means that you don't need such small h or k to get a more accurate solution.

3 Hyperbolic equations

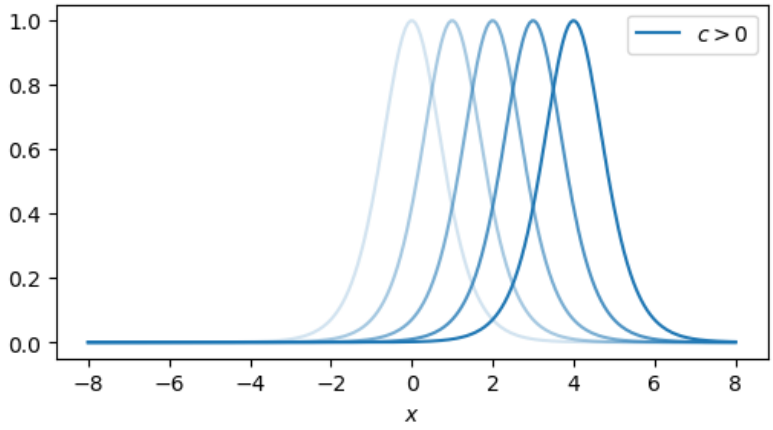
Hyperbolic equations are those where information propagates in time at a finite speed. The simplest example is the advection equation

$$u_t + cu_x = 0, \tag{3.0.1}$$

where we will assume that $c > 0$. Solutions are functions of the form

$$u(x,t) = f(x - ct), \tag{3.0.2}$$

namely functions moving to the right at constant speed $\dfrac{\mathrm{d}x}{\mathrm{d}t} = c$.



3.1 Upwind method

The simplest discretization of (3.0.1) would be forward differences in both derivatives. Then

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_j^n}{h} = 0, \tag{3.1.1}$$

$$\iff U_j^{n+1} = (1+\lambda)U_j^n - \lambda U_{j+1}^n, \tag{3.1.2}$$

where $\lambda = ck/h$. Again, we can check the method for consistency with the PDE by looking at the local truncation error:

$$\varepsilon_j^n = \frac{u_j^{n+1} - u_j^n}{k} + c\frac{u_{j+1}^n - u_j^n}{h} \tag{3.1.3}$$

$$= \frac{1}{k}\left[k(u_t)_j^n + \frac{k^2}{2}(u_{tt})_j^n + O(k^3)\right] + \frac{c}{h}\left[h(u_x)_j^n + \frac{h^2}{2}(u_{xx})_j^n + O(h^3)\right] \tag{3.1.4}$$

$$= (u_t)_j^n + \frac{k}{2}(u_{tt})_j^n + O(k^2) + c(u_x)_j^n + \frac{ch}{2}(u_{xx})_j^n + O(h^2) \tag{3.1.5}$$

$$= \frac{k}{2}(u_{tt})_j^n + \frac{ch}{2}(u_{xx})_j^n + O(k^2 + h^2). \tag{3.1.6}$$

Since this tends to zero as h and k tend to zero, the method is consistent, with order of accuracy $O(h+k)$.

To check stability, we apply the von Neumann analysis, which gives

$$U_j^{n+1} = \left(1 + \lambda - \lambda e^{i\xi h}\right) U_j^n \tag{3.1.7}$$

$$\implies \quad g(\xi) = 1 + \lambda\left(1 - e^{i\xi h}\right). \tag{3.1.8}$$

Stability requires $|g(\xi)| \leq 1$ for all ξ. We have

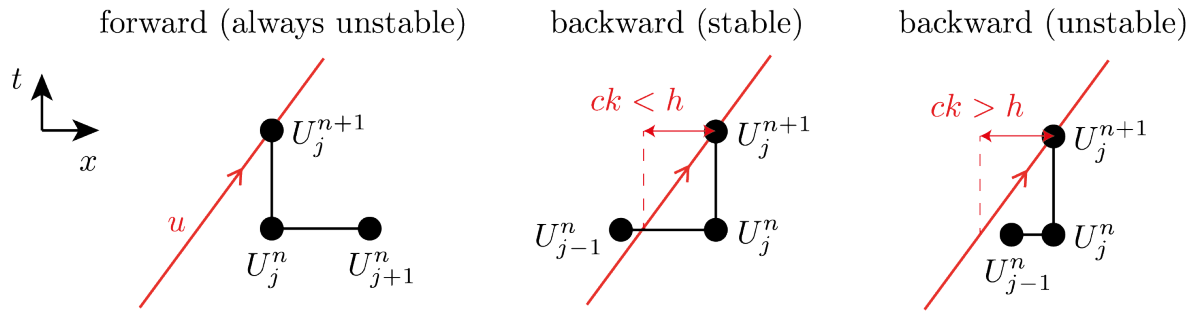
$$|g(\xi)|^2 = \Big[1 + \lambda - \lambda\cos(\xi h)\Big]^2 + \lambda^2 \sin^2(\xi h) \tag{3.1.9}$$

$$= 1 + 2\lambda(1 + \lambda)\Big[1 - \cos(\xi h)\Big]. \tag{3.1.10}$$

So for stability we would need

$$2\lambda(1 + \lambda) \leq 0 \qquad \Longleftrightarrow \qquad -1 \leq \lambda \leq 0. \tag{3.1.11}$$

But $\lambda = ck/h > 0$, so the scheme is unstable! We can see why by thinking about the domain of dependence of the true solution, which moves at speed $c > 0$. The true solution at t_{n+1} is determined by values at t_n that lie to the left of x_j. But for the forward-difference in space, these are outside the numerical stencil:



To rectify this, we need to replace the forward spatial difference with a backward difference. This is called the *upwind method*:

$$U_j^{n+1} = (1 - \lambda)U_j^n + \lambda U_{j-1}^n. \tag{3.1.12}$$

For stability, the above picture suggests that the distance travelled by the true solution in one time step should be no bigger than the mesh spacing in x, meaning $ck \leq h$ or in other words $\lambda \leq 1$. This is supported by the von Neumann analysis, which now gives

$$|g(\xi)|^2 = 1 - 2\lambda(1 - \lambda)\Big[1 - \cos(\xi h)\Big], \tag{3.1.13}$$

so that stability now requires

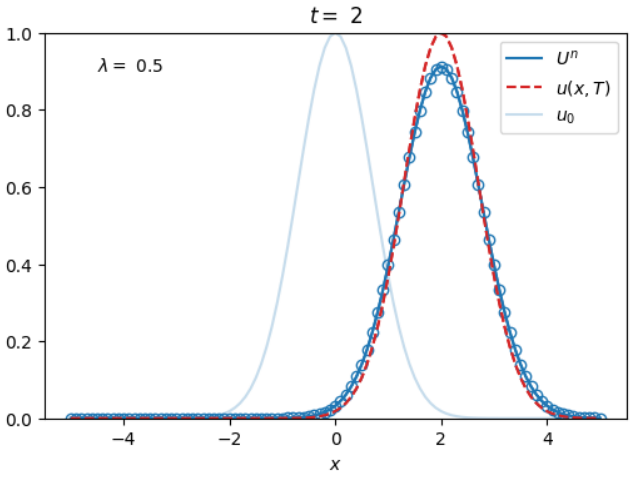
$$2\lambda(1 - \lambda) \geq 0 \qquad \Longleftrightarrow \qquad 0 \leq \lambda \leq 1. \tag{3.1.14}$$

This is called the *Courant-Friedrichs-Lewy*, or *CFL condition*.

To illustrate the upwind scheme, let $c = 1$ and consider the exact solution of a Gaussian pulse

$$u(x, t) = e^{-(x-t)^2}. \tag{3.1.15}$$

This is implemented in `adv_upwind.py`. The following plot shows the solution at $t = 2$ for $J = 100$, $N = 40$, giving $\lambda = \frac{1}{2}$.



The solution is stable, as predicted by the CFL condition. But, although it has been advected at the correct speed, the shape of the function has changed – it seems to be smoothed out.

To understand this, let us examine the local truncation error. Using $\lambda = ck/h$ and $u_{tt} = c^2 u_{xx}$,

$$\varepsilon_j^n = \frac{u_j^{n+1} - u_j^n}{k} + c\frac{u_j^n - u_{j-1}^n}{h} \tag{3.1.16}$$

$$= \frac{1}{k}\left[k(u_t)_j^n + \frac{k^2}{2}(u_{tt})_j^n + O(k^3)\right] + \frac{c}{h}\left[h(u_x)_j^n - \frac{h^2}{2}(u_{xx})_j^n + O(h^3)\right] \tag{3.1.17}$$

$$= (u_t)_j^n + c(u_x)_j^n + \frac{k}{2}(u_{tt})_j^n - \frac{ch}{2}(u_{xx})_j^n + O(k^2 + h^2) \tag{3.1.18}$$

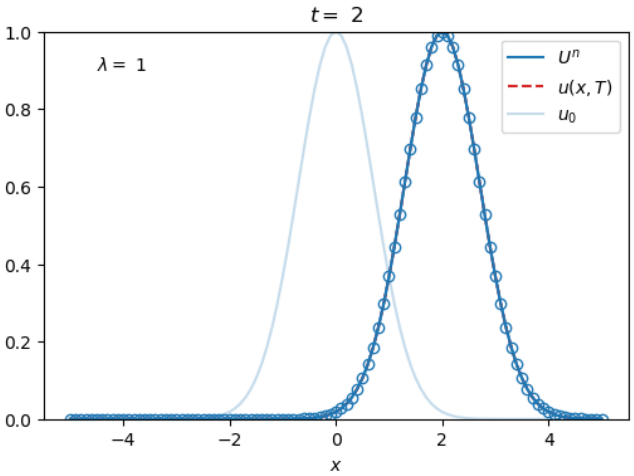
$$= (u_t)_j^n + c(u_x)_j^n + \left(\frac{c^2 k}{2} - \frac{ch}{2}\right)(u_{xx})_j^n + O(k^2 + h^2) \tag{3.1.19}$$

$$= (u_t)_j^n + c(u_x)_j^n - \frac{ch}{2}(1 - \lambda)(u_{xx})_j^n + O(k^2 + h^2). \tag{3.1.20}$$

So the numerical method behaves as though it is solving the PDE

$$u_t + cu_x - \frac{ch}{2}(1 - \lambda)u_{xx} = 0, \tag{3.1.21}$$

which is an *advection-diffusion* equation. The extra diffusive term explains the smoothing of the profile that we observed for $\lambda = \frac{1}{2}$. Our analysis predicts that this term should go away for $\lambda = 1$. We can verify this in the code by setting $N = 20$:



► Usually in practice we don't have the luxury of taking $\lambda = 1$. For example, there may be another term in the PDE that puts a further restriction on N, or the advection speed c may be spatially non-uniform so that we can't have the same value of λ everywhere.

3.2 Lax-Wendroff method

The upwind method is only first-order, since the global error is $O(h + k)$. To derive a second-order method, start with the Taylor series in time,

$$u_j^{n+1} = u_j^n + k\left(u_t\right)_j^n + \frac{k^2}{2}\left(u_{tt}\right)_j^n + O(k^2). \tag{3.2.1}$$

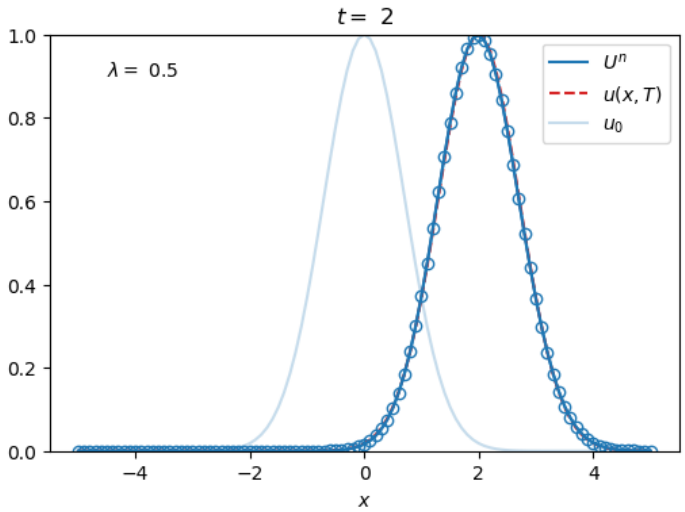
Now replace the time derivatives by spatial derivatives, using the fact that $u_t = -cu_x$ and $u_{tt} = c^2 u_{xx}$. Thus

$$u_j^{n+1} = u_j^n - ck\left(u_x\right)_j^n + \frac{c^2k^2}{2}\left(u_{xx}\right)_j^n + O(k^2). \tag{3.2.2}$$

Then approximate the spatial derivatives by central differences, leading to

$$U_j^{n+1} = U_j^n - \frac{\lambda}{2}\left(U_{j+1}^n - U_{j-1}^n\right) + \frac{\lambda^2}{2}\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right), \tag{3.2.3}$$

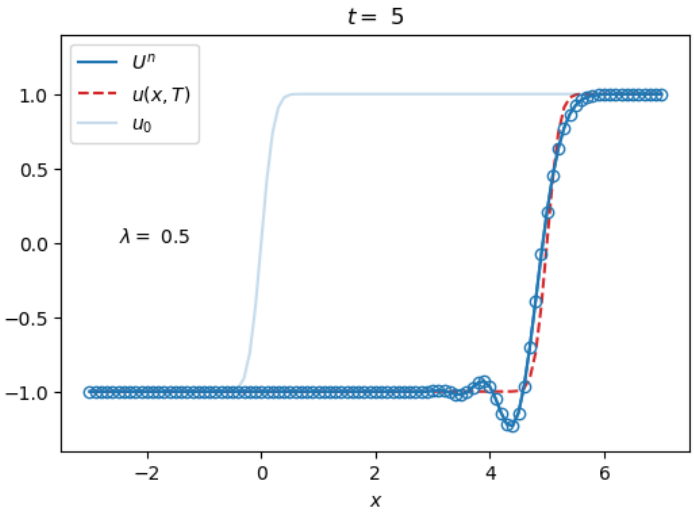
called the *Lax-Wendroff method*. You can show [exercise] that this method has global error $O(h^2 + k^2)$, and that the stability condition is still $\lambda \leq 1$. The method is implemented in `adv_laxwendroff.py`. For $\lambda = \frac{1}{2}$ the solution is more accurate than the upwind method:



To see the limitations of the Lax-Wendroff method, the code `adv_laxwendroff_erf.py` considers the advection of a steeper function,

$$u(x, t) = \operatorname{erf}\Big[4(x - t)\Big]. \tag{3.2.4}$$

With Lax-Wendroff we find:



The numerical method has introduced spurious oscillations, called numerical *dispersion*.

3.3 Dispersion error

To investigate the dispersion error, we can again use Fourier analysis. Unlike in von Neumann analysis, consider a travelling wave mode, $u(x,t) = e^{i(\xi x - \omega t)}$. If we substitute this into (3.0.1), we get

$$-i\omega e^{i(\xi x-\omega t)} + i\xi c e^{i(\xi x-\omega t)} = 0 \qquad \Longleftrightarrow \qquad \omega = c\xi. \tag{3.3.1}$$

This is called the *dispersion relation* for the advection equation – it says that modes of all wavenumbers travel at the same speed $\omega/\xi = c$.

Now suppose we compute the dispersion relation for the numerical method. We do this by substituting $U_j^n = e^{i(\xi x_j - \omega t_n)} = e^{i(\xi jh - \omega nk)}$ into the formula (3.2.3) for Lax-Wendroff, giving

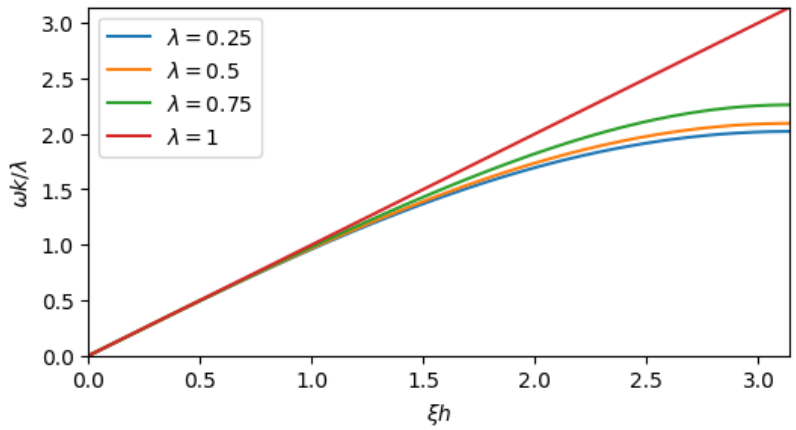
$$e^{-i\omega k} = 1 - \frac{\lambda}{2}\left(e^{i\xi h} - e^{-i\xi h}\right) + \frac{\lambda^2}{2}\left(e^{i\xi h} + e^{-i\xi h} - 2\right) \tag{3.3.2}$$

$$= 1 + \lambda^2\Big(\cos(\xi h) - 1\Big) - i\lambda\sin(\xi h). \tag{3.3.3}$$

Notice that for $\lambda = 1$, this reduces to $e^{-i\omega k} = e^{-i\xi h}$, so we recover $\omega = \xi h/k = (c/\lambda)\xi = c\xi$, which matches the true dispersion relation for the PDE. For $|\lambda| \leq 1$, the real part gives

$$\cos(\omega k) = 1 + \lambda^2\Big(\cos(\xi h) - 1\Big). \tag{3.3.4}$$

The following plot shows this function for different values of λ:



It is evident for $0 < \lambda < 1$ that modes with larger ξ have $\omega k/\lambda < \xi h$, or in other words they travel at speed $\omega/\xi < c$. This explains why the higher wavenumber modes were seen to "lag behind" in the error function example.

► In fact, it is impossible to find a linear second-order (or higher-order) method that does not produce unwanted oscillations for the advection equation. This is known as *Godunov's Theorem*. To get around this, fancier advection methods are nonlinear (*i.e.*, the coefficients of the finite differences depend themselves on U).

3.4 Finite volume methods

PDEs of the form

$$u_t + \big[f(u)\big]_x = 0 \tag{3.4.1}$$

are called *conservation laws* because

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_a^b u\,\mathrm{d}x = \int_a^b u_t\,\mathrm{d}x = -\int_a^b \big[f(u)\big]_x\,\mathrm{d}x = f\big[u(a,t)\big] - f\big[u(b,t)\big]. \tag{3.4.2}$$

In other words, the integral of u over an interval $a < x < b$ can change only by a net flux in or out of the boundaries $x = a, b$. Since this is often an underlying physical principle from which a PDE is derived, it is desirable to preserve this in a numerical scheme.

The idea of a finite-volume method is to approximate (3.4.2) over an individual cell of the mesh. We have exactly that

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{x_j}^{x_{j+1}} u\,\mathrm{d}x = f\big[u(x_j,t)\big] - f\big[u(x_{j+1},t)\big] \tag{3.4.3}$$

Integrating this from t_n to t_{n+1} gives

$$\int_{x_j}^{x_{j+1}} u(x,t_{n+1})\,\mathrm{d}x - \int_{x_j}^{x_{j+1}} u(x,t_n)\,\mathrm{d}x = \int_{t_n}^{t_{n+1}} f\big[u(x_j,t)\big]\,\mathrm{d}t - \int_{t_n}^{t_{n+1}} f\big[u(x_{j+1},t)\big]\,\mathrm{d}t. \tag{3.4.4}$$

Writing in terms of spatial or temporal averages, we have

$$h\Big(U^{n+1}_{j+\frac{1}{2}} - U^{n}_{j+\frac{1}{2}}\Big) = k\Big(F_j^{n+\frac{1}{2}} - F_{j+1}^{n+\frac{1}{2}}\Big), \tag{3.4.5}$$

$$\iff \qquad U^{n+1}_{j+\frac{1}{2}} = U^{n}_{j+\frac{1}{2}} - \frac{k}{h}\Big(F_{j+1}^{n+\frac{1}{2}} - F_j^{n+\frac{1}{2}}\Big). \tag{3.4.6}$$

In *finite volume methods*, we use this formula to solve directly for the spatial cell-averages $U^n_{j+\frac{1}{2}}$. Different methods correspond to different ways of approximating the time-averaged fluxes $F_j^{n+\frac{1}{2}}$ in terms of the $U^n_{j+\frac{1}{2}}$.

Notice that (3.4.5) mirrors the conservation property (3.4.2) of the original PDE, even when summed over a region of the mesh.

3.4.1 Advection equation

Let us apply the finite volume method to our usual advection equation $u_t + cu_x = 0$ with $c > 0$. In this case the flux is simply $f(u) = cu$, so we want to approximate

$$F_j^{n+\frac{1}{2}} \approx \frac{1}{k}\int_{t_n}^{t_{n+1}} cu(x_j, t)\,\mathrm{d}t. \tag{3.4.7}$$

Remember that the advection equation moves u at constant velocity c, without changing its shape. So

$$\frac{1}{k}\int_{t_n}^{t_{n+1}} cu(x_j, t)\,\mathrm{d}t = \frac{1}{k}\int_{t_n}^{t_{n+1}} cu\big[x_j - c(t-t_n), t_n\big]\,\mathrm{d}t. \tag{3.4.8}$$

We can therefore derive an approximation by choosing a *subgrid model* for u at $t = t_n$, and integrating this. The subgrid model has to be based on the known cell averages, $U^n_{j+\frac{1}{2}}$.

It is typical to assume a linear subgrid model, where u within the cell $x_{j-1} < x < x_j$ is given by

$$u(x, t_n) = U^n_{j-\frac{1}{2}} + \sigma^n_{j-\frac{1}{2}}(x - x_{j-\frac{1}{2}}), \tag{3.4.9}$$

for some slope $\sigma^n_{j-\frac{1}{2}}$ of the straight line. Notice that the subgrid model is defined so that

$$\frac{1}{h}\int_{x_{j-1}}^{x_j} u(x, t_n)\,\mathrm{d}x = U^n_{j-\frac{1}{2}} \tag{3.4.10}$$

for any slope $\sigma^n_{j-\frac{1}{2}}$. Different choices of slope give different finite-volume methods.

The resulting approximation of the flux is

$$F_j^{n+\frac{1}{2}} = \frac{1}{k}\int_{t_n}^{t_{n+1}} cu[x_j - c(t-t_n), t_n]\,\mathrm{d}t. \tag{3.4.11}$$

Assuming the CFL condition, we know that $x_j - c(t-t_n)$ lies in the cell $x_{j-1} < x < x_j$, so

$$F_j^{n+\frac{1}{2}} = \frac{1}{k}\int_{t_n}^{t_{n+1}} c\left(U^n_{j-\frac{1}{2}} + \sigma^n_{j-\frac{1}{2}}\left[x_j - c(t-t_n) - x_{j-\frac{1}{2}}\right]\right)\mathrm{d}t \tag{3.4.12}$$

$$= cU^n_{j-\frac{1}{2}} + \frac{c}{k}\sigma^n_{j-\frac{1}{2}}\int_{t_n}^{t_{n+1}}\left(\frac{h}{2} - c(t-t_n)\right)\mathrm{d}t \tag{3.4.13}$$

$$= cU^n_{j-\frac{1}{2}} + \frac{c}{k}\sigma^n_{j-\frac{1}{2}}\int_0^{k}\left(\frac{h}{2} - ct'\right)\mathrm{d}t' \tag{3.4.14}$$

$$= cU^n_{j-\frac{1}{2}} + \frac{c}{k}\sigma^n_{j-\frac{1}{2}}\left(\frac{hk}{2} - \frac{ck^2}{2}\right) \tag{3.4.15}$$

$$= cU^n_{j-\frac{1}{2}} + \frac{c}{2}\sigma^n_{j-\frac{1}{2}}\Big(h - ck\Big). \tag{3.4.16}$$

So the method is

$$U_{j+\frac{1}{2}}^{n+1} = U_{j+\frac{1}{2}}^{n} - \lambda\Big(U_{j+\frac{1}{2}}^{n} - U_{j-\frac{1}{2}}^{n}\Big) - \frac{\lambda(1-\lambda)h}{2}\Big(\sigma_{j+\frac{1}{2}}^{n} - \sigma_{j-\frac{1}{2}}^{n}\Big). \tag{3.4.17}$$

It remains to choose the slopes $\sigma_{j+\frac{1}{2}}^{n}$.

Donor cell method

If we choose $\sigma_{j+\frac{1}{2}}^{n} \equiv 0$ (zero slopes) we get the *donor cell* method

$$U_{j+\frac{1}{2}}^{n+1} = U_{j+\frac{1}{2}}^{n} - \lambda\Big(U_{j+\frac{1}{2}}^{n} - U_{j-\frac{1}{2}}^{n}\Big). \tag{3.4.18}$$

This is the same formula as the upwind method, except for an $h/2$ shift. So we expect it to behave similarly, smoothing out the solution.

Lax-Wendroff method

If we make a "downwind" approximation of the slopes, with

$$\sigma_{j+\frac{1}{2}}^{n} = \frac{U_{j+\frac{3}{2}}^{n} - U_{j+\frac{1}{2}}^{n}}{h}, \tag{3.4.19}$$

you can show after some algebra [exercise] that we recover the Lax-Wendroff method

$$U_{j+\frac{1}{2}}^{n+1} = U_{j+\frac{1}{2}}^{n} - \frac{\lambda}{2}\big(U_{j+\frac{3}{2}}^{n} - U_{j-\frac{1}{2}}^{n}\big) + \frac{\lambda^2}{2}\big(U_{j-\frac{1}{2}}^{n} - 2U_{j+\frac{1}{2}}^{n} + U_{j+\frac{3}{2}}^{n}\big), \tag{3.4.20}$$

again shifted by $h/2$.

Fromm's method

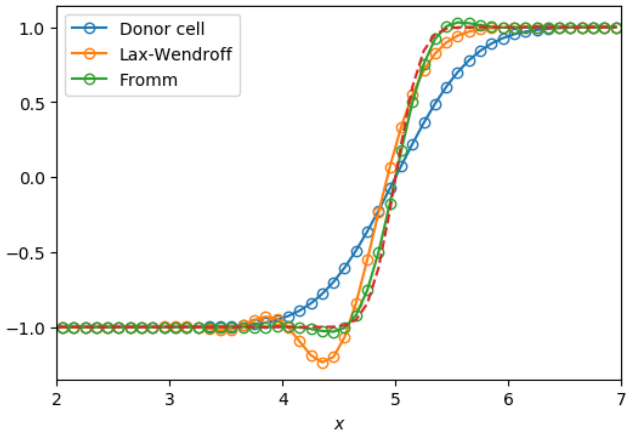
If we make a centred approximation of the slope,

$$\sigma_{j+\frac{1}{2}}^{n} = \frac{U_{j+\frac{3}{2}}^{n} - U_{j-\frac{1}{2}}^{n}}{2h}, \tag{3.4.21}$$

we obtain a new method called *Fromm's method*:

$$U_{j+\frac{1}{2}}^{n+1} = U_{j+\frac{1}{2}}^{n} - \lambda\Big(U_{j+\frac{1}{2}}^{n} - U_{j-\frac{1}{2}}^{n}\Big) - \frac{\lambda(1-\lambda)}{4}\Big(U_{j+\frac{3}{2}}^{n} - U_{j-\frac{1}{2}}^{n} - U_{j+\frac{1}{2}}^{n} + U_{j-\frac{3}{2}}^{n}\Big). \tag{3.4.22}$$

The Python code `adv_fv.py` compares these three methods for the test problem $u(x,t) = \operatorname{erf}\Big[4(x - t)\Big]$, with $J = N = 100$:



The donor cell and Lax-Wendroff methods behave the same as before (note that I stretched out the x-axis for clarity). The Fromm method is better but still shows some overshoot both ahead of the front and behind it.

3.4.2 Slope limiters

To avoid spurious oscillations, it is a good idea to choose a method that is *total variation diminishing* (*TVD*). This means that

$$\sum_{j=1}^{J-1} \left| U^{n+1}_{j+\frac{1}{2}} - U^{n+1}_{j-\frac{1}{2}} \right| \leq \sum_{j=1}^{J-1} \left| U^{n}_{j+\frac{1}{2}} - U^{n}_{j-\frac{1}{2}} \right|, \tag{3.4.23}$$

so that new oscillations cannot develop.

It can be shown mathematically that the donor cell method ($\sigma^n_{j+\frac{1}{2}} = 0$) is TVD, consistent with the behaviour that we have seen. On the other hand, Lax-Wendroff and the Fromm method are not.

It is possible to make a TVD scheme that is less dissipative than donor cell by allowing non-zero slopes subject to a *slope limiter*. The simplest example is the *minmod* limiter, whereby

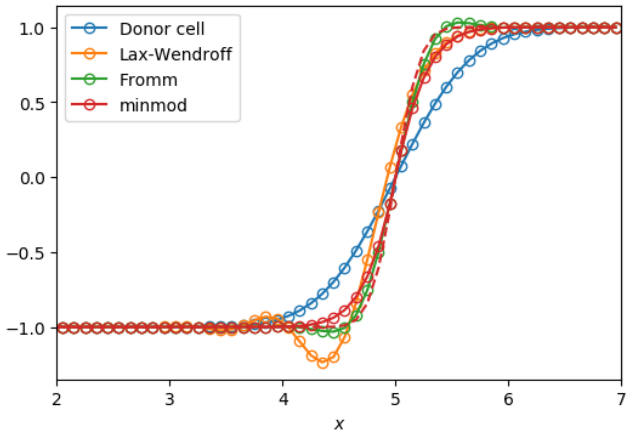
$$\sigma^n_{j+\frac{1}{2}} = \text{minmod}\left(\frac{U^n_{j+\frac{1}{2}} - U^n_{j-\frac{1}{2}}}{h}, \frac{U^n_{j+\frac{3}{2}} - U^n_{j+\frac{1}{2}}}{h} \right) \tag{3.4.24}$$

and the minmod function is defined by

$$\text{minmod}(a,b) = \begin{cases} a & \text{if } |a| < |b| \text{ and } ab > 0, \\ b & \text{if } |a| > |b| \text{ and } ab > 0, \\ 0 & \text{if } ab \leq 0. \end{cases} \tag{3.4.25}$$

This chooses the smallest of the two slopes, provided they both have the same sign, otherwise it sets the slope to zero. This prevents oscillations from developing.

The Python code `adv_fv_minmod.py` adds this to the previous plot:



► The minmod method is TVD but does still smooth sharp functions a little. A TVD method that sharpens discontinuities (sometimes excessively) is the *superbee*.

4 Elliptic equations

Intuitively, elliptic equations are those with no notion of "time dependence". The whole domain is coupled together, and the solution is determined solely by boundary conditions. The classic example is the Poisson equation

$$u_{xx} + u_{yy} = f(x, y). \tag{4.0.1}$$

We will look at both finite difference methods for regular grids and finite element methods for more general grids.

The fact that the whole domain is coupled together means that the algebra tends to be more expensive, and techniques are needed to simplify this.

4.1 Finite difference methods

We will discretize the Poisson equation (4.0.1) in a square domain $x \in [a, b]$, $y \in [a, b]$. Similar to before, we use an equally spaced mesh with

$$x_j = a + jh, \qquad j = 0, 1, \ldots, J, \tag{4.1.1}$$

$$y_l = a + lh, \qquad l = 0, 1, \ldots, J, \tag{4.1.2}$$

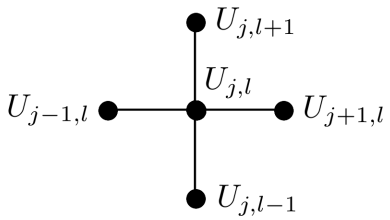
where the mesh spacing is $h = \dfrac{b-a}{J}$. We approximate $u(x_j, y_l)$ by the discrete solution $U_{j,l}$.

The simplest method is to use central differencing in both spatial dimensions, approximating the PDE (4.0.1) by

$$\frac{U_{j+1,l} - 2U_{j,l} + U_{j-1,l}}{h^2} + \frac{U_{j,l+1} - 2U_{j,l} + U_{j,l-1}}{h^2} = f_{j,l}, \tag{4.1.3}$$

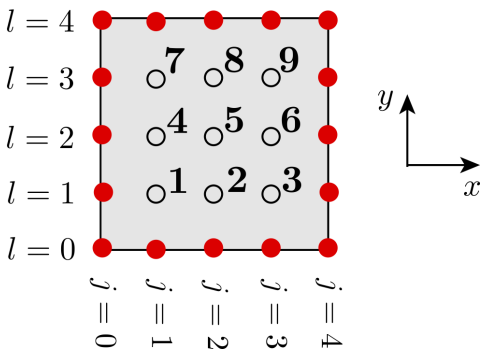
$$\iff \quad U_{j-1,l} + U_{j+1,l} - 4U_{j,l} + U_{j,l-1} + U_{j,l+1} = h^2 f_{j,l}, \tag{4.1.4}$$

where $f_{j,l} \equiv f(x_j, y_l)$. This is known as the five-point method, and has the following stencil:



This gives a system of $(J-1)^2$ simultaneous linear equations for the interior points, with boundary values needing to be supplied.

The matrix that we get depends on the numbering system for the points. The simplest is so-called *lexicographic ordering*: start at the bottom left and number consecutively from left to right, bottom to top. For example, with $J = 4$ we have the following situation:



The red points are boundary values and the bold numbers show the lexicographic ordering of the internal nodes. Centering the stencil at **1** gives

$$U_{0,1} + U_{2,1} - 4U_{1,1} + U_{1,0} + U_{1,2} = h^2 f_{1,1}. \tag{4.1.5}$$

Centering the stencil at **2** gives

$$U_{1,1} + U_{3,1} - 4U_{2,1} + U_{2,0} + U_{2,2} = h^2 f_{2,1}. \tag{4.1.6}$$

And so on. Writing out the equations in full, and moving the boundary terms to the right-hand side, we have

$$\underbrace{\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}}_{A} \begin{bmatrix} U_{1,1} \\ U_{2,1} \\ U_{3,1} \\ U_{1,2} \\ U_{2,2} \\ U_{3,2} \\ U_{1,3} \\ U_{2,3} \\ U_{3,3} \end{bmatrix} = h^2 \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ f_{3,1} \\ f_{1,2} \\ f_{2,2} \\ f_{3,2} \\ f_{1,3} \\ f_{2,3} \\ f_{3,3} \end{bmatrix} - \begin{bmatrix} U_{0,1} + U_{1,0} \\ U_{2,0} \\ U_{3,0} + U_{4,1} \\ U_{0,2} \\ 0 \\ U_{4,2} \\ U_{0,3} + U_{1,4} \\ U_{2,4} \\ U_{4,3} + U_{3,4} \end{bmatrix} \tag{4.1.7}$$

More succinctly, we can write the matrix A in *block form* as

$$A = \begin{bmatrix} C & \mathbb{I} & 0 \\ \mathbb{I} & C & \mathbb{I} \\ 0 & \mathbb{I} & C \end{bmatrix}, \quad \text{where} C = \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{bmatrix}. \tag{4.1.8}$$

There are 3 blocks in each row/column of A because the mesh has 3 *rows* of unknowns. Each block is 3×3 because the grid has 3 *columns* of unknowns.

In general, for a $J \times K$ grid, the matrix A will have $K - 1$ blocks in each direction, with each submatrix having dimension $(J - 1) \times (J - 1)$,

$$A = \begin{bmatrix} C & \mathbb{I} & 0 & \cdots & 0 \\ \mathbb{I} & C & \mathbb{I} & \ddots & \vdots \\ 0 & \mathbb{I} & C & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \mathbb{I} \\ 0 & \cdots & 0 & \mathbb{I} & C \end{bmatrix}, \quad \text{where} C = \begin{bmatrix} -4 & 1 & 0 & \cdots & 0 \\ 1 & -4 & 1 & \ddots & \vdots \\ 0 & 1 & -4 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -4 \end{bmatrix}. \tag{4.1.9}$$

As an example, consider the simple case of $f(x, y) = 0$ (the *Laplace equation*) on the unit square, with boundary conditions $u(x, 0) = \sin(\pi x)$, $u(x, 1) = \sin(\pi x)e^{-\pi}$ and $u(0, y) = u(1, y) = 0$. You can easily verify (by substituting into the PDE) that the exact solution to this problem is

$$u(x, y) = \sin(\pi x)e^{-\pi y}. \tag{4.1.10}$$

The 5-point finite difference solution is implemented in `laplace_5pt.py`.

A mathematical trick for constructing the matrix A in Python is to write it as a *Kronecker sum*

$$A = B \otimes \mathbb{I}_{J-1} + \mathbb{I}_{J-1} \otimes B, \tag{4.1.11}$$

where $\mathbb{I}_{J-1}$ is the $(J-1) \times (J-1)$ identity matrix and B is the $(J-1) \times (J-1)$ matrix

$$B = \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}. \tag{4.1.12}$$

The *Kronecker product* $M \otimes N$ of two $n \times n$ matrices M, N is the $n^2 \times n^2$ matrix obtained by replacing entry m_{ij} by the matrix $m_{ij}N$. For example,

$$\begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 \cdot 1 & -2 \cdot 0 & 1 \cdot 1 & 1 \cdot 0 \\ -2 \cdot 0 & -2 \cdot 1 & 1 \cdot 0 & 1 \cdot 1 \\ 1 \cdot 1 & 1 \cdot 0 & -2 \cdot 1 & -2 \cdot 0 \\ 1 \cdot 0 & 1 \cdot 1 & -2 \cdot 0 & -2 \cdot 1 \end{bmatrix} = \begin{bmatrix} -2 & 0 & 1 & 0 \\ 0 & -2 & 0 & 1 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -2 \end{bmatrix}. \tag{4.1.13}$$

You should check that

$$\begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix}. \tag{4.1.14}$$

The `scipy.sparse` library in Python has a built-in function for the Kronecker sum (4.1.11), so we can construct A with the following code:

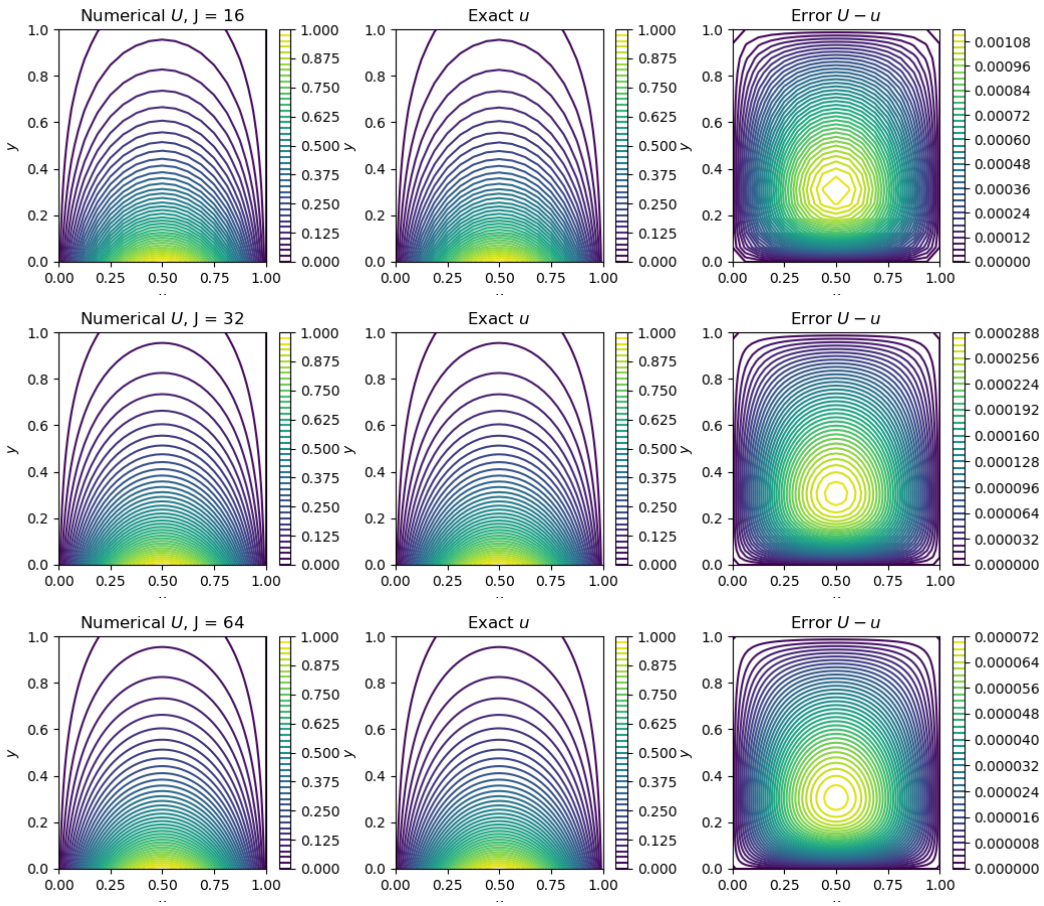
```
import scipy.sparse as sp
main = [-2 for i in range(J-1)]
upper = [1 for i in range(J-2)]
lower = upper
B = sp.diags([main, upper, lower], offsets=[0,1,-1])
A = sp.kronsum(B, B)
```

Notice that we are storing A as a *sparse* matrix, meaning that we don't waste memory storing all of the zeros. If you have a sparse matrix, you can turn it into a full matrix with

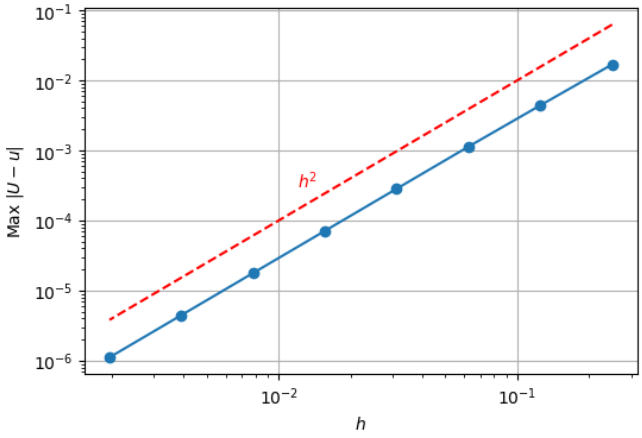
```
print(A.todense())
```

The tricky part of the Python implementation is to get the boundary conditions correct, which is an exercise in keeping track of the order of the elements.

Here is the result of our test problem (4.1.10), with $J = 16$ (top), $J = 32$ (middle) and $J = 64$ (bottom):



The following log-log plot shows how the maximum error (blue points) scales with h:



There is a clear slope of 2 on the log-log plot, indicating that the (global) error scales like $O(h^2)$ and this is therefore a second-order method.

► The "local truncation error" method no longer gives us an obvious way to bound the global error, but it may be proven mathematically that the global error is indeed $O(h^2)$.

4.2 Fast-Poisson solvers

If you try the `laplace_5pt.py` solution for larger grids, you will see that the computation – solving the linear system – becomes slower and slower. One way to get around this is to use an iterative method such as *multigrid* to solve the linear system. [This is beyond our scope.]

Here, I will discuss a simple but highly effective method for solving the Poisson equation on a regular grid, called a *fast-Poisson solver*. This is not really a different method – it is just a trick for solving the linear system that arises in the 5-point finite-difference scheme.

Consider again the 5-point method for the Poisson equation, namely

$$U_{j-1,l} + U_{j+1,l} - 4U_{j,l} + U_{j,l-1} + U_{j,l+1} = h^2 f_{j,l}. \tag{4.2.1}$$

If the solution $u(x, y)$ is periodic in x and y, then it makes sense to approximate $U_{j,l}$ as a *discrete Fourier transform* in both x and y, which means we write

$$U_{j,l} = \frac{1}{J^2} \sum_{m=0}^{J-1} \sum_{n=0}^{J-1} \hat{U}_{m,n} \mathrm{e}^{2\pi i j m/J} \mathrm{e}^{2\pi i l n/J}. \tag{4.2.2}$$

Here $\hat{U}_{m,n}$ is the *discrete Fourier transform* of $U_{j,l}$. Notice that we don't sum over the last point $m = J$ or $n = J$ because it is the same as the first point $m = 0$ or $n = 0$ by periodicity.

If we also do a similar transform to the right-hand side,

$$f_{j,l} = \frac{1}{J^2} \sum_{m=0}^{J-1} \sum_{n=0}^{J-1} \hat{f}_{m,n} \mathrm{e}^{2\pi i j m/J} \mathrm{e}^{2\pi i l n/J}, \tag{4.2.3}$$

then we can substitute these into the finite-difference formula (4.2.1) and cancel common factors to get

$$\hat{U}_{m,n} \left(\mathrm{e}^{-2\pi i m/J} + \mathrm{e}^{2\pi i m/J} - 4 + \mathrm{e}^{-2\pi i n/J} + \mathrm{e}^{2\pi i n/J} \right) = h^2 \hat{f}_{m,n}, \tag{4.2.4}$$

$$\iff \quad \hat{U}_{m,n} \left[2\cos\left(\frac{2\pi m}{J}\right) + 2\cos\left(\frac{2\pi n}{J}\right) - 4 \right] = h^2 \hat{f}_{m,n}, \tag{4.2.5}$$

$$\iff \quad \hat{U}_{m,n} = \frac{h^2 \hat{f}_{m,n}}{2\left[\cos\left(\frac{2\pi m}{J}\right) + \cos\left(\frac{2\pi n}{J}\right) - 2\right]}. \tag{4.2.6}$$

In other words, we have a simple formula for $\hat{U}_{m,n}$ in "Fourier space". We can then transform the solution back to real space by plugging these values into (4.2.2).

To implement the scheme, we need to be able to compute $\hat{f}_{j,l}$, which is given by forward discrete Fourier transform (4.2.3), or

$$\hat{f}_{m,n} = \sum_{j=0}^{J-1} \sum_{l=0}^{J-1} f_{j,l} \, \mathrm{e}^{-2\pi i m j/J} \mathrm{e}^{-2\pi i n l/J}. \tag{4.2.7}$$

Note the change of signs in the exponentials and also the difference in overall normalisation.

This method is implemented in `poisson_5ptfast_periodic.py`. The discrete Fourier transform assumes periodic boundary conditions. So we try it with the test problem

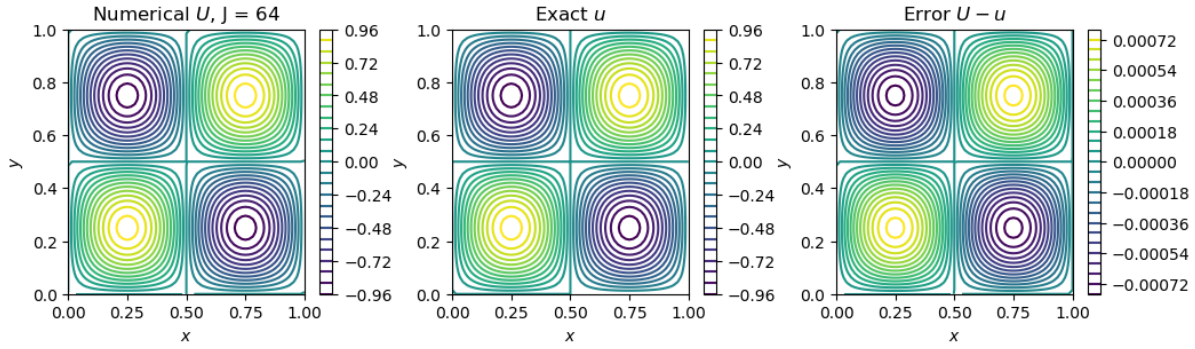
$$u(x, y) = \sin(2\pi x) \sin(2\pi y) \tag{4.2.8}$$

which solves the Poisson equation with right-hand side $f(x, y) = -8\pi^2 \sin(2\pi x) \sin(2\pi y)$. The fast-Poisson solver is simply coded as follows:

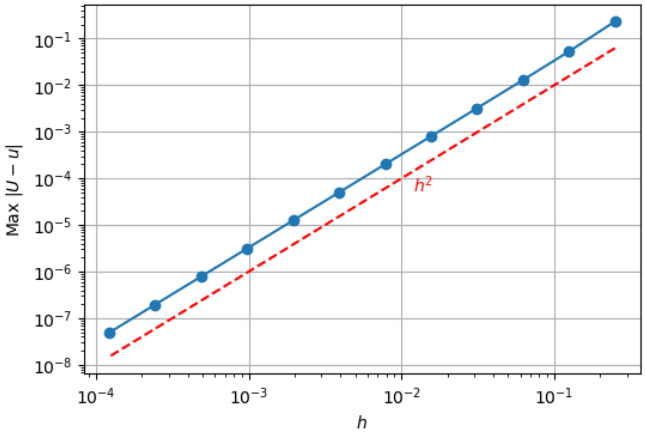
```
U = -8*np.pi**2*np.sin(2*np.pi*X)*np.sin(2*np.pi*Y)  # f
U = np.fft.fft2(U)  # f-hat
M, N = np.meshgrid(range(J), range(J))
U = 0.5*h**2*U/(np.cos(2*np.pi*M/J) + np.cos(2*np.pi*N/J) - 2) # U-hat
U[0,0] = 0  # otherwise we divide by zero!
U = np.real(np.fft.ifft2(U))  # U
```

The method works nicely because of the speed at which the discrete Fourier transform (and its inverse) can be computed (FFT stands for *fast Fourier transform*). Notice that I only need a single array `U`, which I first use to store the right-hand side, then overwrite several times.

Here is the result:



Again we find that the convergence is second-order, as we see from the following plot, where I was easily able to take $J = 8192$ on my desktop machine:



We can deal with non-periodic boundary conditions by restricting the Fourier transform. In particular

1. To specify $u \rightarrow$ use only sine functions (*discrete sine transform*).
2. To specify $\dfrac{\partial u}{\partial n} \rightarrow$ use only cosine functions (*discrete cosine transform*).

We will illustrate only the first case, to see how to apply the fast method to our Laplace problem for $u_{xx} + u_{yy} = 0$ on the unit square with boundary conditions $u(x, 0) = \sin(\pi x)$, $u(x, 1) = \sin(\pi x)e^{-\pi}$, $u(0, y) = u(1, y) = 0$.

For these boundary conditions, we write $U_{j,l}$ as a *discrete sine transform*

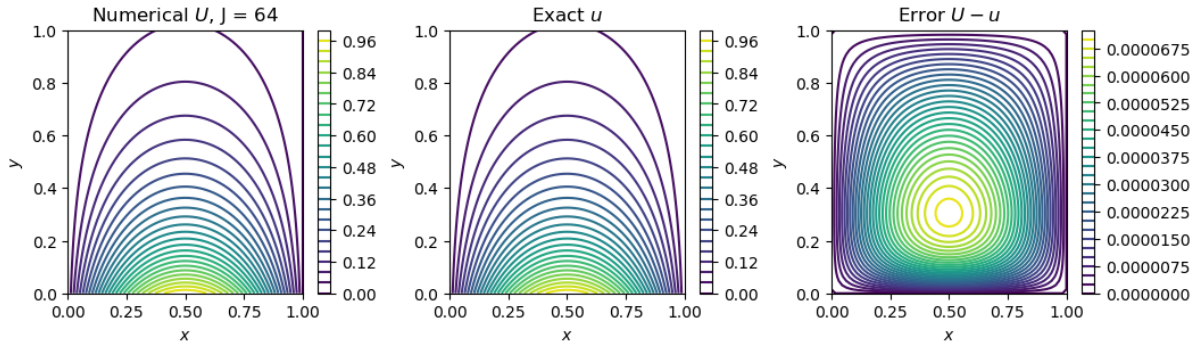
$$U_{j,l} = \frac{1}{4J^2} \sum_{m=1}^{J-1} \sum_{n=1}^{J-1} \hat{U}_{m,n} \sin\left(\frac{\pi jm}{J}\right) \sin\left(\frac{\pi ln}{J}\right). \tag{4.2.9}$$

This means that $U_{j,l}$ will automatically vanish all around the boundary, due to the symmetry of the sine functions (and hence we only need to solve for the interior values $1, \ldots, J-1$). As in equation (4.1.7), we need to modify the right-hand-side vector $f_{j,l}$ to include the required boundary conditions.

This scheme is implemented in `laplace_5ptfast.py`. Note that `numpy` doesn't have a routine for computing the DST, but `scipy` does. We have to use the "type I" DST.

```
import scipy.fftpack as fft
U = fft.dstn(U, type=1)
M, N = np.meshgrid(range(1,J), range(1,J))
U = 0.5*U/(np.cos(np.pi*M/J) + np.cos(np.pi*N/J) - 2)
U = fft.idstn(U, type=1)/(2*J)**2
```

Notice that the formula (4.2.6) is slightly modified when we use the discrete sine transform; you can verify that two factors of two are removed. The solution is close to our previous calculation, with a small difference due to rounding error:



4.3 Finite element methods

These differ from those we have seen so far because they do not approximate the PDE by finite differences on a regular mesh. Instead, they approximate what is called the *weak form* of the PDE. As we will see, this makes it easier to deal with irregular meshes.

4.3.1 1D Poisson equation

To illustrate the idea, we first consider the 1-dimensional Poisson problem,

$$u_{xx} = f(x) \quad \text{for } -1 < x < 1, \qquad u(-1) = u(1) = 0. \tag{4.3.1}$$

With $f(x) = 1$ the exact solution to this problem is $u(x) = \frac{1}{2}(x^2 - 1)$.

To apply finite elements we first need to derive the *weak form* of the PDE. To do this, multiply it by an arbitrary *test function* $v(x)$ then integrate to get

$$\int_{-1}^{1} u_{xx} v \, dx = \int_{-1}^{1} fv \, dx \quad \text{for all } v \in \mathcal{V}. \tag{4.3.2}$$

The set $\mathcal{V}$ of allowable test functions are those (i) with finite $\int_{-1}^{1} v_x^2 \, dx$, and (ii) which vanish at the boundary, $v(-1) = v(1) = 0$. We then integrate (4.3.2) by parts to find

$$\Big(u_x v\Big)_{-1}^{1} - \int_{-1}^{1} u_x v_x \, dx = \int_{-1}^{1} fv \, dx \tag{4.3.3}$$

$$\iff \quad - \int_{-1}^{1} u_x v_x \, dx = \int_{-1}^{1} fv \, dx \quad \text{for all } v \in \mathcal{V}. \tag{4.3.4}$$

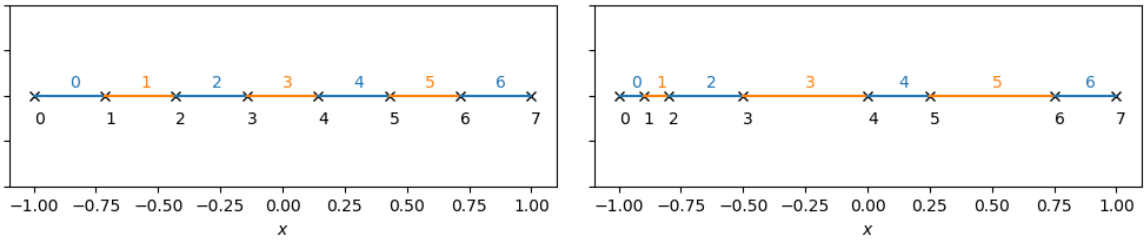
This integral equation is called the *weak form* of the PDE. It can be shown to have a unique solution, so if the original PDE has a solution we can find it by solving the weak form.

The idea of *finite element methods* is to discretize the weak form by approximating u and v with a finite-dimensional basis $\{\phi_0(x), \phi_1(x), \ldots, \phi_J(x)\}$. For example, the discrete approximation of $u(x)$ would be

$$U(x) = \sum_{j=0}^{J} U_j \phi_j(x), \tag{4.3.5}$$

which is described by a finite set of coefficients U_j. Unlike with finite-difference methods, the approximating function $U(x)$ is defined everywhere in $[-1, 1]$. The basis functions $\phi_j(x)$ are chosen for ease of computation on a given mesh. We will consider only a piecewise-linear basis, so that $U(x)$ gives a piecewise linear approximation to $u(x)$. The method is as follows.

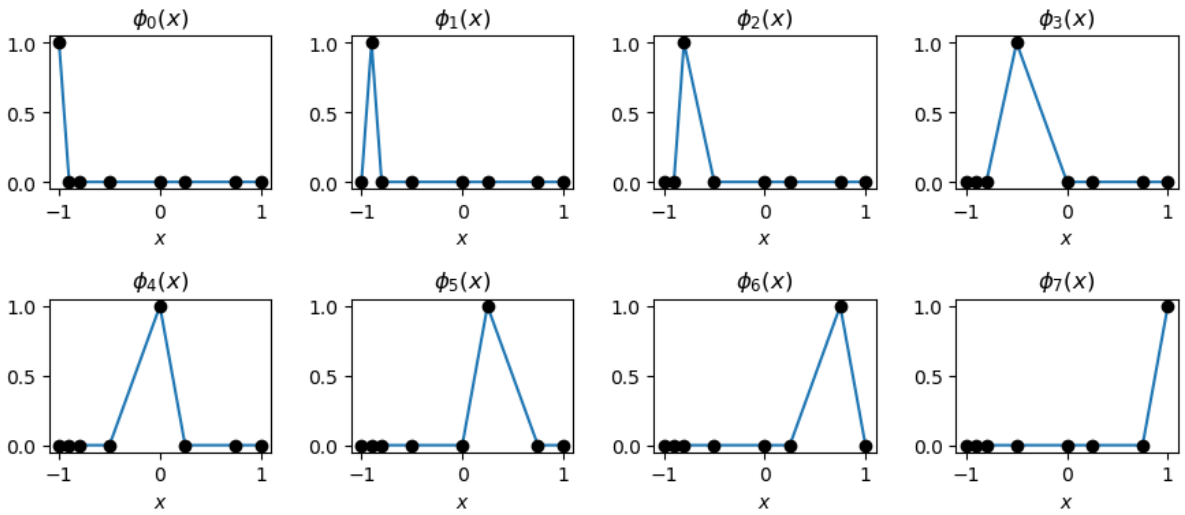
Step 1. Define a mesh. Just like with finite differences we divide the domain into a discrete mesh. The individual cells are called *elements*. For our 1D example, suppose we have an ordered set of nodes $\{x_0, x_1, \ldots, x_J\}$. Then the elements are line segments. They could be equal or unequal in length:



Step 2. Define the basis functions ϕ_j. We take each ϕ_j to be a "hat" function that is equal to 1 at node x_j, equal to 0 at all of the other nodes, and a straight line in between each node. Thus

$$\phi_j(x) = \begin{cases} \dfrac{x - x_{j-1}}{x_j - x_{j-1}} & x_{j-1} < x < x_j, \\ \dfrac{x_{j+1} - x}{x_{j+1} - x_j} & x_j < x < x_{j+1} \\ 0 & \text{elsewhere} \end{cases}. \tag{4.3.6}$$

For the right-hand mesh above, these look as follows:



Notice that there are exactly two non-zero basis functions on each element, and exactly one non-zero basis function at each node.

Step 3. Discretize the weak form. If we consider only piecewise-linear test functions on the same mesh, then it suffices to satisfy (4.3.4) for each of the basis functions ϕ_k. This gives us a system of $J+1$ equations

$$-\int_{-1}^{1} \frac{\partial}{\partial x}\left(\sum_{j=0}^{J} U_j \phi_j\right) \frac{\partial \phi_k}{\partial x}\,\mathrm{d}x = \int_{-1}^{1} f\phi_k\,\mathrm{d}x \qquad \text{for } k = 0,\dots,J, \tag{4.3.7}$$

which can be rewritten in the form

$$\sum_{j=0}^{J} U_j \underbrace{\int_{-1}^{1} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_k}{\partial x}\,\mathrm{d}x}_{a_{kj}} = \underbrace{-\int_{-1}^{1} f\phi_k\,\mathrm{d}x}_{f_k} \qquad \text{for } k = 0,\dots,J. \tag{4.3.8}$$

This is a linear system and the resulting matrix with entries a_{kj} is called the *stiffness matrix*. The right-hand-side vector f_k is called the *load vector*. Since the basis functions have compact support (being non-zero only on two elements), they only overlap with their immediate neighbours and the stiffness matrix for this 1d problem will be tridiagonal. The diagonal entries have the form

$$a_{jj} = \int_{-1}^{1} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_j}{\partial x}\,\mathrm{d}x = \int_{x_{j-1}}^{x_j} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_j}{\partial x}\,\mathrm{d}x + \int_{x_j}^{x_{j+1}} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_j}{\partial x}\,\mathrm{d}x \tag{4.3.9}$$

for $1 \le j \le J-1$, and include only a single integral at the boundary points $j=0$ and $j=J$. The off-diagonal entries have the form

$$a_{j,j+1} = \int_{-1}^{1} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_{j+1}}{\partial x}\,\mathrm{d}x = \int_{x_j}^{x_{j+1}} \frac{\partial \phi_j}{\partial x}\frac{\partial \phi_{j+1}}{\partial x}\,\mathrm{d}x. \tag{4.3.10}$$

Inserting the definition (4.3.6) gives, for internal points $1 \le j \le J-1$,

$$a_{jj} = \int_{x_{j-1}}^{x_j} \frac{1}{(x_j - x_{j-1})^2}\,\mathrm{d}x + \int_{x_j}^{x_{j+1}} \frac{1}{(x_{j+1} - x_j)^2}\,\mathrm{d}x \tag{4.3.11}$$

$$= \frac{1}{x_j - x_{j-1}} + \frac{1}{x_{j+1} - x_j}. \tag{4.3.12}$$

For the boundary points, we have

$$a_{00} = \frac{1}{x_1 - x_0}, \qquad a_{JJ} = \frac{1}{x_J - x_{J-1}}. \tag{4.3.13}$$

For the off-diagonal terms, we get

$$a_{j,j+1} = \int_{x_j}^{x_{j+1}} \left(\frac{-1}{x_{j+1} - x_j}\right)\left(\frac{1}{x_{j+1} - x_j}\right) \mathrm{d}x = \frac{-1}{x_{j+1} - x_j}. \tag{4.3.14}$$

This gives the stiffness matrix in terms of the element sizes.

Step 4. Solve the discrete system. We solve the system (4.3.8) using a standard linear solver, giving the coefficients U_j. The solution $U(x)$ can then be evaluated at any x using (4.3.5).

The Python code `poisson1d_fem.py` implements the method for the case $f(x) = 1$. The 1d array `nodes` is a list of the x-coordinates of the nodes, and the 2d array `elements` is a list of node pairs, defining the line segments. For example, with five equally-spaced nodes in left-to-right order we would have

```
nodes = [-1, -0.5, 0, 0.5, 1]
elements = [[0, 1], [1, 2], [2, 3], [3, 4]]
```

We construct the stiffness matrix `A` by looping through each element $[x_j, x_{j+1}]$ and adding the four contributions of $\pm(x_{j+1} - x_j)$ to different cells in the matrix. For the equally-spaced (left-hand) mesh above, the matrix is

$$A = \begin{bmatrix} \frac{7}{2} & -\frac{7}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{7}{2} & 7 & -\frac{7}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{7}{2} & 7 & -\frac{7}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{7}{2} & 7 & -\frac{7}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{7}{2} & 7 & -\frac{7}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{7}{2} & 7 & -\frac{7}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{7}{2} & 7 & -\frac{7}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{7}{2} & \frac{7}{2} \end{bmatrix}. \tag{4.3.15}$$

For the right-hand mesh, the matrix is still tri-diagonal, but the entries vary because of the non-uniform mesh spacing:

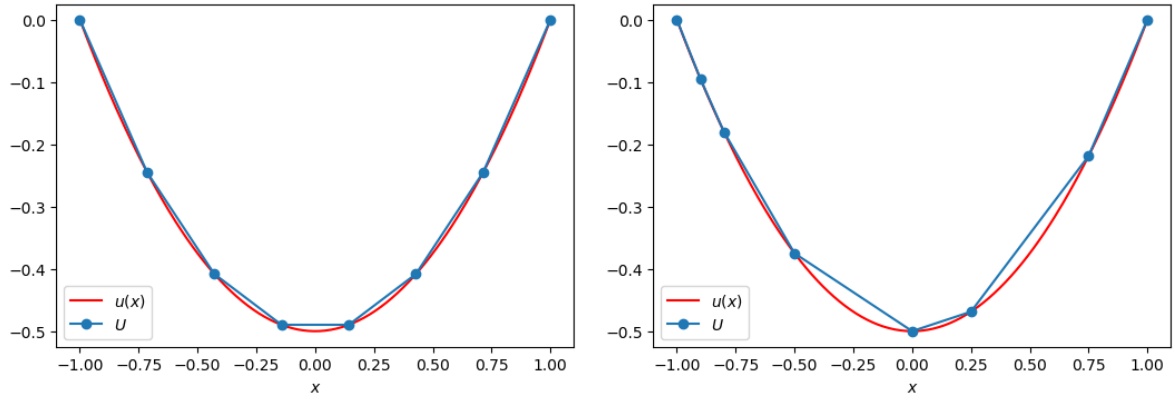
$$A = \begin{bmatrix} 10 & -10 & 0 & 0 & 0 & 0 & 0 & 0 \\ -10 & 20 & -10 & 0 & 0 & 0 & 0 & 0 \\ 0 & -10 & \frac{40}{3} & -\frac{10}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{10}{3} & \frac{16}{3} & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & 6 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 & 6 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 6 & -4 \\ 0 & 0 & 0 & 0 & 0 & 0 & -4 & 4 \end{bmatrix}. \tag{4.3.16}$$

We also need to construct the load vector with entries $f_k = -\int_{-1}^{1} f\phi_k \,\mathrm{d}x$. For our problem $f = 1$ we simply have

$$f_k = -\int_{-1}^{1} \phi_k \,\mathrm{d}x = -\int_{x_{k-1}}^{x_{k+1}} \phi_k \,\mathrm{d}x = -\frac{1}{2}(x_{k+1} - x_{k-1}). \tag{4.3.17}$$

▶ If the function $f(x)$ in the Poisson equation is more complicated, then the integrals f_k typically need to be computed numerically, for example with the trapezium rule.

The code then solves the linear system with `numpy.linalg.solve`. For our boundary conditions $u(-1) = u(1) = 0$, we know that $U_0 = U_J = 0$, so we only need to solve for the interior points $1 \leq j \leq J-1$. The plots below show the solutions on these two meshes – notice that the solution $U(x)$ is a piecewise-linear function and is defined not only at discrete points:



4.3.2 Accuracy

Error analysis is more complex than for finite difference methods, so here we only present the most important result, on optimality.

Recall that $\mathcal{V}$ is the space of functions with (i) finite $\int_{-1}^{1} v_x^2 \, \mathrm{d}x$ and (ii) $v(-1) = v(1) = 0$. To simplify notation, we write the inner product of two functions $f, g \in \mathcal{V}$ as

$$(f, g) := \int_{-1}^{1} fg \, \mathrm{d}x \tag{4.3.18}$$

and define the bilinear form

$$a(f, g) := \int_{-1}^{1} f_x g_x \, \mathrm{d}x. \tag{4.3.19}$$

Observe that $a(g, f) = a(f, g)$ – *i.e.*, this bilinear form is *self-adjoint*.

As usual, let u denote the exact solution to (4.3.1), and let U be our finite element approximation. In our shorthand notation, the weak form (4.3.4) of the PDE says that

$$a(u, v) = -(f, v) \qquad \text{for all } v \in \mathcal{V}. \tag{4.3.20}$$

The function u belongs to $\mathcal{V}$, but U belongs to a subset $\mathcal{V}_h \subset \mathcal{V}$, consisting of all piecewise-linear functions on a fixed mesh of elements. Because it imposes (4.3.8), it follows that U satisfies

$$a(U, V) = -(f, V) \qquad \text{for all } V \in \mathcal{V}_h. \tag{4.3.21}$$

1. Galerkin orthogonality. We will show that the error $u - U$ is *orthogonal* to the space $\mathcal{V}_h$. To see this, note that $V \in \mathcal{V}$ for any $V \in \mathcal{V}_h$, so by (4.3.20) we have

$$a(u, V) = -(f, V) \qquad \text{for all } V \in \mathcal{V}_h. \tag{4.3.22}$$

Subtracting (4.3.21) gives

$$a(u, V) - a(U, V) = -(f, V) + (f, V) \qquad \text{for all } V \in \mathcal{V}_h \tag{4.3.23}$$

$$\iff \quad a(u - U, V) = 0 \qquad \text{for all } V \in \mathcal{V}_h. \tag{4.3.24}$$

2. Céa's Lemma. This result says that U is the "best possible" approximation to u from the subspace $\mathcal{V}_h$, when measured by the *energy norm*

$$\|f\|_{\mathrm{E}}^2 := a(f, f) = \int_{-1}^{1} \left(f_x\right)^2 \mathrm{d}x. \tag{4.3.25}$$

To see this, note first that for any $V \in \mathcal{V}_h$,

$$a(u - U, u - U) = a(u - U, u - V + V - U) \tag{4.3.26}$$

$$= a(u - U, u - V) + a(u - U, V - U) \tag{4.3.27}$$

$$= a(u - U, u - V), \tag{4.3.28}$$

where we used Galerkin orthogonality given that $V - U \in \mathcal{V}_h$. So we have, for all $V \in \mathcal{V}_h$,

$$a^2(u - U, u - U) = a^2(u - U, u - V) \tag{4.3.29}$$

$$\iff a^2(u - U, u - U) \leq a(u - U, u - U)a(u - V, u - V) \tag{4.3.30}$$

$$\iff \quad a(u - U, u - U) \leq a(u - V, u - V). \tag{4.3.31}$$

Here we used the Cauchy-Schwartz inequality $a^2(f, g) \leq a(f, f)a(g, g)$. It follows that

$$\|u - U\|_{\mathrm{E}} \leq \|u - V\|_{\mathrm{E}} \qquad \text{for all } V \in \mathcal{V}_h. \tag{4.3.32}$$

In other words, no other piecewise-linear approximation on this mesh could have a lower error, at least not measured in this norm.

► Of course, we could probably find a more accurate approximation by using a different basis $\{\phi_k(x)\}$. For example, see the "periodic table of the finite elements" at `http://z.umn.edu/femtable`.