Finite differences:
Boundary value problems and PDEs
Here’s an example of a linear algebra problem (with some ODE context)...

Suppose we want to solve, for $y(x)$, the **boundary value problem**

$$y'' - y = x, \quad y(0) = 1, \quad y(1) = e - 1$$

which has the solution $y(x) = e^x - x$.

Unlike an initial value problem, we can’t just ‘start’ at an endpoint!

An approach is to approximate the function at mesh points $x_j$...

$$y''(x) \approx \frac{y(x + h) - 2y(x) + y(x - h)}{h^2}$$

for the second derivative.
**Finite differences**

\[ y'' - y = x, \]
\[ y(0) = 1, \quad y(1) = e - 1 \]

Let \( x_j = jh \) be the mesh points \((h = 1/N)\). Then, at \( x_j \),

\[ \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} - y_j \approx x_j \]

The formula for our approximation \( u_j \) is then

\[ u_{j+1} - 2u_j + u_{j-1} - h^2 u_j = h^2 x_j, \quad j = 1, \ldots, N - 1 \]

for the ‘interior’ points.

At the endpoints, we impose **boundary conditions**

\[ u_0 = 1, \quad u_N = e - 1 \]
Finite differences

To summarize, we have the problem/approximation

\[ y'' - y = x, \]
\[ y(0) = 1, \quad y(1) = e - 1 \]

For \( j = 1, 2, \cdots, N - 1, \)
\[ u_{j+1} - (2 + h^2)u_j + u_{j-1} = h^2 x_j \]
\[ u_0 = 1, \quad u_N = e - 1 \]

**Example:** With five mesh points 0, 0.25, \( \cdots, \) 1 we have \( h = 0.25 \) and

\[ u_2 - (2 + h^2)u_1 + 1 = h^2 x_1 \]
\[ u_3 - (2 + h^2)u_2 + u_1 = h^2 x_2 \]
\[ e - 1 - (2 + h^2)u_3 + u_2 = h^2 x_3 \]

which is the linear system

\[
\begin{bmatrix}
2 + h^2 & -1 & 0 \\
-1 & 2 + h^2 & -1 \\
0 & -1 & 2 + h^2 \\
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
u_3 \\
\end{bmatrix}
= -h^2
\begin{bmatrix}
0.25 \\
0.5 \\
0.75 \\
\end{bmatrix}
- \begin{bmatrix}
1 \\
0 \\
(e - 1) \\
\end{bmatrix}
\]
\[
\begin{align*}
    u_{j+1} - (2 + h^2)u_j + u_{j-1} &= h^2 x_j, \quad j = 1, \ldots, N-1 \\
    u_0 = u_N &= 0.
\end{align*}
\]

In general, the system to solve has the form
\[
\begin{bmatrix}
    2 + h^2 & -1 & 0 & \cdots & 0 \\
    -1 & 2 + h^2 & -1 & \ddots & \vdots \\
    0 & -1 & \ddots & \ddots & 0 \\
    \vdots & \ddots & \ddots & 2 + h^2 & -1 \\
    0 & \cdots & 0 & -1 & 2 + h^2
\end{bmatrix}
\begin{bmatrix}
    u_1 \\
    u_2 \\
    \vdots \\
    u_{N-2} \\
    u_{N-1}
\end{bmatrix}
= -h^2
\begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_{N-2} \\
    x_{N-1}
\end{bmatrix}
- 
\begin{bmatrix}
    u_0 \\
    0 \\
    \vdots \\
    0 \\
    u_N
\end{bmatrix}
\]

- The matrix has three diagonals (around the center), called **tri-diagonal**
- Matrices like this how up often when data relates only to adjacent data
- We can solve using Gaussian elimination!

But GE takes \(O(n^3)\) work... but only \(\approx 3n\) non-zeros - can we do better?
The answer is yes - we can get $O(n)$ time - extremely fast!

Now forget about the ODE context and just consider trying to solve

$$Ax = b,$$

where

$$A = \begin{bmatrix}
q_1 & r_1 & 0 & \cdots & 0 \\
p_2 & q_2 & r_2 & \ddots & \\
0 & p_3 & \ddots & \ddots & 0 \\
\vdots & \vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\
0 & \cdots & 0 & p_n & q_n
\end{bmatrix}$$

Let’s first look at an example, where we use GE to reduce

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

and get the LU factorization ($A = LU$).
Finite differences

Here entries of $L$ are noted in red (in the zeroed entries).

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(Zero out $(2, 1)$ entry using $R_2 \leftarrow R_2 + 0.5R_1$).
From here, we use ‘lazy’ notation: $X$ denotes a value we could compute.

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 1.5 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 1.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(Zero out $(3, 2)$ entry using $R_3 \leftarrow R_3 + (1/2.5)R_2$.

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 1.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -0.5 & 1.5 & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & X & X \end{bmatrix}$$

Done! Notice the mostly-zero structure has greatly simplified things...
Thus we have found that the result looks like ($X$ being some numbers)

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \implies \begin{bmatrix} 2 & -1 & 0 & 0 \\ X & X & -1 & 0 \\ 0 & X & X & -1 \\ 0 & 0 & X & X \end{bmatrix}$$

$\implies A = LU$ where $L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ X & 1 & 0 & 0 \\ 0 & X & 1 & 0 \\ 0 & 0 & X & 1 \end{bmatrix}$, $U = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & X & -1 & 0 \\ 0 & 0 & X & -1 \\ 0 & 0 & 0 & X \end{bmatrix}$

This process generalizes to the $N \times n$ tri-diagonal matrix, where:

- We only need to zero out one entry below the diagonal for each column
- The upper-diagonal never changes
- Both $L$ and $U$ have one diagonal other than the center (‘bi-diagonal’)

Now let’s derive an efficient Gaussian elimination for a tridiagonal matrix:

\[
\begin{bmatrix}
q_1 & r_1 & 0 & \cdots & 0 \\
p_2 & q_2 & r_2 & \cdots & \vdots \\
0 & p_3 & \ddots & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & q_{n-1} & r_{n-1} \\
0 & \cdots & 0 & p_n & q_n
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
d_1 & r_1 & 0 & \cdots & 0 \\
\ell_2 & d_2 & r_2 & \cdots & \vdots \\
0 & \ell_3 & \ddots & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ell_{n-1} & r_{n-1} \\
0 & \cdots & 0 & \ell_n & d_n
\end{bmatrix}
\]

We want to find the \(\ell\)'s and \(d\)'s.
First, \(d_1 = q_1\) trivially. Then the first step of GE gives

\[
\ell_2 = \frac{p_2}{d_1}, \quad d_2 = q_2 - \ell_2 r_1, \quad \text{ (multiplier: } \ell_2) \]

Then for the next step after that (and so on),

\[
\ell_3 = \frac{p_3}{d_2}, \quad d_3 = q_3 - \ell_3 r_2,
\]

\[
\ell_j = \frac{p_j}{d_{j-1}}, \quad d_j = q_j - \ell_j r_{j-1}, \quad j = 2, 3 \cdots, n.
\]

Thus we can solve for variables in the order

\[
\ell_2 \rightarrow d_2 \rightarrow \ell_3 \rightarrow d_3 \rightarrow \cdots \ell_n \rightarrow d_n.
\]
Finally, to solve $Ax = b$ we solve

$$Ly = b, \quad Ux = y.$$  

Both solves are quite fast - forward/back substitution also simplify!

**Forward solve:** we have

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ \ell_2 & 1 & \cdots & \vdots \\ \vdots & \vdots & \ddots & 1 \\ 0 & \cdots & \ell_n & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \implies y_j + \ell_j y_{j-1} = b_j$$

so $y$ is given by

$$y_1 = b_1, \quad y_j = b_j - \ell_j y_{j-1}, \quad j = 2, \ldots, n.$$
Finally, to solve $Ax = b$ we solve

$$Ly = b, \quad Ux = y.$$  

Both solves are quite fast - forward/back substitution also simplify!

**Backward solve:** Similarly,

\[
\begin{bmatrix}
  d_1 & r_1 & \cdots & 0 \\
  0 & d_2 & \ddots & \vdots \\
  \vdots & \ddots & \ddots & r_{n-1} \\
  0 & \cdots & 0 & d_n \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix}
= 
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix}
\implies d_j x_j + r_j x_{j+1} = y_j
\]

so we can solve for $x$ by

$$x_n = y_n / d_n, \quad x_j = \frac{y_j - r_j x_{j+1}}{d_j}, \quad j = n - 1, n - 2, \cdots, 1$$
In summary, we have an efficient Gaussian elimination for solving $Ax = b$ where

\[
A = \begin{bmatrix}
q_1 & r_1 & 0 & \cdots & 0 \\
p_2 & q_2 & r_2 & \cdots & 0 \\
0 & p_3 & \cdots & \cdots & 0 \\
& \ddots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & p_n & q_n
\end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix}
d_1 & r_1 & 0 & \cdots & 0 \\
\ell_2 & d_2 & r_2 & \cdots & 0 \\
0 & \ell_3 & \cdots & \cdots & 0 \\
& \ddots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \ell_n & d_n
\end{bmatrix}
\]

This method is sometimes called the **Thomas algorithm**.

- *(initialize)* Set $d_1 = q_1$ and $y_1 = b_1$.
- *(LU and fwd. solve)* Then for $j = 2, \cdots, n$:

  \[
  \ell_j = p_j / d_{j-1}, \quad d_j = q_j - \ell_j r_{j-1} \\
y_j = b_j - \ell_j y_{j-1}.
  \]

- *(Back solve)* Finally set $x_n = y_n / d_n$ and for $j = n-1, n-2, \cdots, 1$:

  \[
x_j = (y_j - r_j x_{j+1}) / d_j.
  \]

Note that you can do the $Ux = y$ solve in parallel with the $LU$. 


A tridiagonal matrix should be stored in banded form:

\[
A = \begin{bmatrix}
q_1 & r_1 & 0 & \cdots & 0 \\
p_2 & q_2 & r_2 & \ddots & \\
0 & p_3 & \ddots & \ddots & 0 \\
\vdots & \ddots & \ddots & q_{n-1} & r_{n-1} \\
0 & \cdots & 0 & p_n & q_n
\end{bmatrix}
\]

is stored as

\[
\begin{bmatrix}
0 & q_1 & r_1 \\
p_2 & q_2 & r_2 \\
\vdots & \ddots & \ddots \\
p_{N-1} & q_{N-1} & r_{N-1} \\
p_{N-1} & q_{N-1} & 0
\end{bmatrix}
\]

Pay attention to:

- The zeros - not part of the data (correct code should never read them!)
- Conventions may differ on the unused zeros (‘padding’)
- Indexing (easy to be off by one!). Here:

\[
\text{row } k \text{ of the array } \iff \text{row } k \text{ of the matrix}
\]

We store only 3n numbers - much more feasible than \(n^2\).
See example code for the finite difference method. We solve

\[ y'' - y = x, \quad y(0) = y_a, \quad y(b) = y_b \]

by solving the linear system In general, the system to solve has the form

\[
\begin{bmatrix}
2 + h^2 & -1 & 0 & \cdots & 0 \\
-1 & 2 + h^2 & -1 & \ddots & \vdots \\
0 & -1 & \ddots & \ddots & 0 \\
\vdots & \ddots & \ddots & 2 + h^2 & -1 \\
0 & \cdots & 0 & -1 & 2 + h^2
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
\vdots \\
u_{N-2} \\
u_{N-1}
\end{bmatrix}
= -h^2
\begin{bmatrix}
h^2x_1 - y_a \\
h^2x_2 \\
\vdots \\
h^2x_{N-2} \\
h^2x_{N-1} - y_b
\end{bmatrix}
\] (FD)

breaking up into the following functions:

a) `build_fd` that creates \( A \) (as an array `bands`) and `rhs` as in (FD)

b) `trisolve(bands, rhs)`: solves \( Ax = rhs \), with \( A \) tri-diagonal

- A solve ‘main’ function that:
  - gets the \( Ax = rhs \) system from (a)...
  - then solves it using (b).
From ODEs to partial differential equations...
Suppose I am in a (cold) 1d room of length $L$. I open windows at both ends to the outside.

Let the temperature in the room be $u(x, t)$. Over time, $u$ will equalize with the outside.

This process is modeled by the **heat equation**

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

which is a **partial differential equation** (it has derivs. in $x$ **and** $t$).

Physical interpretation (‘Fourier’s law’):

heat flow per time $= -\beta \frac{\partial u}{\partial x}$.

That is, heat flows from higher to lower temperature, and faster if the difference is large.
The heat equation

Other heat-like equations (different fluxes):

- Fisher’s equation (genetics):

  \[ \frac{\partial f}{\partial t} = \frac{\partial^2}{\partial x^2}(x(1-x)f(x,t)) \]

  (describes distribution \( f(x,t) \) of a trait in a population - "genetic drift")

- Black-Scholes - derivatives in finance

- Height of a liquid droplet \( h(x,t) \) ("Porous medium equation")

  \[ \frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(h^3 \frac{\partial h}{\partial x}) \]

- "advection-diffusion":

  \[ \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(F(u)) = \nu \frac{\partial^2 u}{\partial x^2} \]

  e.g. transport of a chemical in a solution

- And much more!

Can be solved with similar methods!
\[ \frac{\partial u}{\partial t} = \beta \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), \quad t > 0. \]

To complete the problem, there are also **initial conditions**

\[ u(x, 0) = f(x) \] (initial distribution of heat)

and **boundary conditions** (where \( T_0 \) is the outside temp.)

\[ u(t, 0) = u(t, L) = T_0 \quad \text{for all } t. \]

The goal: reduce the problem into manageable pieces for computation. We’ll need: derivative approximations, an ODE solver, and a bit more.
Finite differences

First, let’s review some ways of approximating derivatives...

Forward difference: \( f'(x) \approx \frac{f(x + h) - f(x)}{h} \)

Central difference: \( f'(x) \approx \frac{f(x + h) - f(x - h)}{2h} \)

Central (2nd) difference: \( f''(x) \approx \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} \)

A ‘stencil’ diagram shows the points used in the approximation:
PDEs and the method of lines

We’ll solve the heat equation using the method of lines.

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

Step 1 (define a mesh in space): First define the points in space where the approximation is defined...

\[x_j = jh, \quad j = 0, 1, \ldots, N, \quad \text{where } h = L/N.\]

Now think of \(u\) at each fixed \(x\) as a function in \(t\):

\[u(x_j, t) := u_j(t) \text{ along the ‘line’ } x = x_j \quad t > 0.\]
\[ \frac{\partial u}{\partial t} = \beta \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), \quad t > 0. \]

**Step 2 (approximate x-derivatives):** Using the central difference in \( x \),

\[ \frac{\partial u}{\partial t} \approx \beta \frac{u(x + h, t) - 2u(x, t) + u(x - h, t)}{h^2}. \]

At \( x = x_j \):

\[ \frac{du_j}{dt} \approx \beta \frac{u_{j+1} - u_j + u_{j-1}}{h^2}, \quad j = 1, \cdots, N - 1 \]

which is a system of **ODEs** for the functions along each ‘line’.

The boundary conditions give the last two equations... \( (u_0 = u_N = T_0) \).
### PDEs and the method of lines

We have derived a system of \( N - 2 \) ODEs

\[
\frac{d u_j}{dt} \approx \beta \frac{u_{j+1} - u_j + u_{j-1}}{h^2}, \quad j = 1, \ldots, N - 1
\]

\[
u(t, 0) = u(t, L) = T_0 \implies u_0(t) = u_N(t) = T_0.
\]

The initial conditions come from the IC for the original problem:

\[
u(x, 0) = f(x) \implies u_j(0) = f(x_j).
\]

This system approximates the solution to the PDE (the **method of lines**).

As \( h \to 0 \) (i.e. as \( N \to \infty \)), it can be shown to converge.

(That is, a higher density of lines will give a better solution).
The ODE system can now be solved by any usual method. In summary:

\[
\frac{du_j}{dt} \approx \beta \frac{u_{j+1} - u_j + u_{j-1}}{h^2}, \quad j = 1, \ldots, N - 1
\]

\[
u(t, 0) = u(t, L) = T_0 \implies u_0(t) = u_N(t) = T_0.
\]

\[
u_j(0) = f(x_j).
\]

As an example, let’s see what **Euler’s method** looks like.

Our system is already in ‘generic first order system form’

\[
u' = G(u), \quad u(0) = (f(x_1), \ldots, f(x_{N-1}))
\]

where \(u(t) = (u_1(t), \ldots, u_{N-1}(t))\) and \(G\) has components

\[
G_j(u) = \frac{\beta}{h^2} (u_{j+1} - 2u_j + u_{j-1}), \quad j = 1, \ldots, N - 1
\]

with \(u_0\) and \(u_N\) replaced by \(T_0\).
PDEs and the method of lines

For implementation, we just need to create the system of ODEs:

\[ u' = G(u), \quad u(0) = (f(x_1), \cdots, f(x_{N-1})) \]

for \( u = (u_1, \cdots, u_{N-1}) \) with

\[ G_j(u) = \frac{\beta}{h^2} (u_{j+1} - 2u_j + u_{j-1}), \quad j = 1, \cdots, N - 1, \]

\[ u_0(t) = u_N(t) = T_0. \]

Let's define \( c = \beta / h^2 \). A simple implementation:

```python
def odefunc(t, u, c, ul, ur):
    n = len(u) + 2  # u = (u_1, \ldots, u_{n-1})
    du = np.zeros(m)
    for j in range(1, n-2):  # interior points
    # boundary points
    du[0] = a*(u[1] - 2*u[0] + ul)  # x = 0
    du[n-2] = a*(ur - 2*u[n-2] + u[n-3])  # x = L
    return du
```

(Note: to improve this, have odefunc not create a new array each call).
PDEs and the method of lines

With \( u(t) = (u_1(t), \cdots, u_{N-1}(t)) \),

\[
\begin{align*}
\mathbf{u}' &= G(\mathbf{u}), & \mathbf{u}(0) &= (f(x_1), \cdots, f(x_{N-1})) \\
G_j(\mathbf{u}) &= \frac{\beta}{h^2}(u_{j+1} - 2u_j + u_{j-1}), & j &= 1, \cdots, N - 1
\end{align*}
\]

Let \( \mathbf{u}^{(k)} \) denote the solution vector at time \( t_k \).
We use a super-script for time, and sub-script for space here, so

\[
\mathbf{u}^{(k)} = (u_1(t_k), \cdots, u_{n-1}(t_k))
\]

Euler’s method approximates at times

\[
0 = t_0 < t_1 < \cdots
\]

where we assume that the \( t \)'s have an equal spacing \( \Delta t \). Then

\[
\mathbf{u}^{(k+1)} = \mathbf{u}^{k} + \Delta t G(\mathbf{u}^{k}), \quad k = 0, 1, \cdots.
\]
PDEs and the method of lines

Euler’s method:

\[ u^{(k+1)} = u^k + \Delta t G(u^k), \quad k = 0, 1, \cdots. \]

This is enough to write up the code, but it’s worth ‘plugging in’ \( G \),

\[ G_j(u) = \frac{\beta}{h^2}(u_{j+1} - 2u_j + u_{j-1}), \quad j = 1, \cdots, N - 1. \]

For the \( j \)-th component,

\[ u_j^{k+1} = u_j^k + \frac{\beta \Delta t}{h^2} (u_{j+1}^k - 2u_j^k + u_{j-1}^k) \]

Set \( a = \beta \Delta t / h^2 \). This equation is linear in \( u \). In matrix form...

\[
\begin{bmatrix}
  u_1 \\
  u_2 \\
  \vdots \\
  u_{N-2} \\
  u_{N-1}
\end{bmatrix}^{(k+1)} = 
\begin{bmatrix}
  1 - 2a & a & \cdots & 0 \\
  a & 1 - 2a & \cdots & \vdots \\
  \vdots & \vdots & \ddots & a \\
  0 & \cdots & a & 1 - 2a
\end{bmatrix}
\begin{bmatrix}
  u_1 \\
  u_2 \\
  \vdots \\
  u_{N-2} \\
  u_{N-1}
\end{bmatrix}^{(k)} + 
\begin{bmatrix}
  aT_0 \\
  0 \\
  \vdots \\
  0 \\
  aT_0
\end{bmatrix}
\]

Note that \( u_0(t) = u_N(t) = T_0 \) has been plugged in here.
Euler’s method:

\[ u^{(k+1)} = u + \Delta t G(u^k), \quad k = 0, 1, \ldots. \]

\[ u_j^{k+1} = u_j^k + \frac{\beta \Delta t}{h^2} (u_{j+1}^k - 2u_j^k + u_{j-1}^k) \]

We can simplify a bit by defining the ‘differentiation matrix’

\[
D = \frac{1}{h^2} \begin{bmatrix}
-2 & 1 & \cdots & 0 \\
1 & -2 & \ddots & \vdots \\
\vdots & \ddots & \ddots & 1 \\
0 & \cdots & 1 & -2
\end{bmatrix}, \quad (D \vec{f})_j \approx f''(x_j).
\]

Then, from the previous slide, Euler’s method becomes

\[ u^{(k+1)} = (I + \Delta t D) u^{(k)} + b \]

It’s worth noting that:

- This matrix form is nice for theory...
- ... but the ODE or difference formula are used in implementation
Euler’s method (directly)

\[ u_{j+1}^k = u_j^k + \frac{\beta \Delta t}{h^2} (u_{j+1}^k - 2u_j^k + u_{j-1}^k), \quad u_0^k = u_\ell, \quad u_N^k = u_r. \]

You can implement this directly with a for loop:

```python
# code sketch:
c = beta*dt/h**2
n = len(x) - 1  # x = (x_0, ... x_n)
u = f(x[1:n])  # u at interior points
while t < t_final:
    for j in range(1, n-2):
        unext[j] = u[j] + c*(u[j+1]-2*u[j]+u[j-1])
    # eqs. with boundary terms
    unext[0] = u[0] + c*(u[1] - 2*u[0] + ul)
    u[:] = unext[:]
    t += dt
```

- **u**: \( u \) at current \( t \)
- **unext**: space for \( u \) at next \( t \)
- **n**: grid points \( x_0, \ldots, x_n \)
- **x_points**: The array of \( x_j \)’s
Euler’s method (directly)

\[
  u_{j}^{k+1} = u_{j}^{k} + \frac{\beta \Delta t}{h^2} (u_{j+1}^{k} - 2u_{j}^{k} + u_{j-1}^{k}), \quad u_{0}^{k} = u_{\ell}, \quad u_{N}^{k} = u_{r}.
\]

This can be simplified by keeping the boundary points in; we compute

\[
  u = (u_0, \cdots, u_N)
\]

but update only \( u_1, \cdots, u_{N-1} \). The formula then holds for all (relevant) \( j \).

---

# code sketch:

```python
# c = beta*dt/h**2
n = len(x) - 1  # x = (x_0, ..., x_n)
u = f(x)  # u at *all* points
while t < t_final:
    for j in range(1, n+1):
        unext[j] = u[j] + c*(u[j+1]-2*u[j]+u[j-1])
    u[:] = unext[:]
    t += dt
```

- Technique can be extended...
- Use fictional ‘ghost points’ to make the formula always work (\( u_{-1}, \cdots \))
- Simplifies loops (no special cases)
Implicit methods...

A problem: there is a **stability constraint**

\[ \Delta t < C \Delta x^2 \]

or else numerical solutions grow exponentially!
The fix: use a (good) **implicit** method.

Forward Euler (bad stability):

\[
\begin{align*}
    u_j^{k+1} &= u_j^k + \frac{\beta \Delta t}{h^2} (u_{j+1}^k - 2u_j^k + u_{j-1}^k) \\
\end{align*}
\]

matrix form: \( \mathbf{u}^{(k+1)} = (I + \Delta t \mathbf{D})\mathbf{u}^{(k)} + \mathbf{b} \)

Backward Euler (always stable!):

\[
\begin{align*}
    u_j^{k+1} &= u_j^k + \frac{\beta \Delta t}{h^2} (u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}) \\
\end{align*}
\]

matrix form: \( (I - \Delta t \mathbf{D})\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \mathbf{b} \)

Implicit - at each step, we must solve a **tridiagonal** linear system!