The Method of Lines

Time Stepping the Heat Equation

CS 410 / 510 — Scientific Computing

Lecture: Numerical Methods for PDEs (Part 2)

Where we left off

Last time we took the heat equation

$$u_t = K\,u_{xx} + F(x,t), \quad u(x,0) = f(x), \quad u(0,t) = u(L,t) = 0$$

and replaced the spatial derivative with a central difference. That gave us a semi-discrete equation at every interior node:

$$\frac{du_j}{dt} = K\,\frac{u_{j+1}(t) - 2\,u_j(t) + u_{j-1}(t)}{\Delta x^2} + F(x_j,t)$$

for $j = 2, 3, \ldots, M$ (interior nodes, 1-indexed).

The time derivative is still continuous. We have $M-1$ coupled ODEs — one per interior node.

Today's plan

1. Package those $M-1$ ODEs into a single vector equation

$$\vec{Y}'(t) = A\,\vec{Y}(t) + \vec{b}(t), \qquad \vec{Y}(0) = \vec{Y}_0.$$

2. Identify exactly what A, $\vec{b}(t)$, and $\vec{Y}_0$ are.
3. March forward in time using Forward Euler.
4. See why we might prefer Backward Euler instead, and derive the linear solve it requires.

This whole approach — discretize space first, then treat time as an ODE — is called the **method of lines**.

From node equations to a vector ODE

Stacking the interior unknowns into one vector

Collect the values at all interior nodes into a single time-dependent vector:

$$\vec{Y}(t) = \begin{bmatrix} u_2(t) \\ u_3(t) \\ \vdots \\ u_M(t) \end{bmatrix} \in \mathbb{R}^{M-1}.$$

- $\vec{Y}(t)$ has $M-1$ entries — one per interior node.
- The boundary values $u_1(t)$ and $u_{M+1}(t)$ are both 0 (by the BCs) so they are *not* in $\vec{Y}$.
- If we know $\vec{Y}(t)$, we know the full temperature profile.

Goal: write a single equation for $\vec{Y}'(t) = \dfrac{d\vec{Y}}{dt}$.

Look carefully at each row

Write the semi-discrete equation at $j = 2$, $j = 3$, $\ldots$, $j = M$.

$$u_t|_{x_2} = \frac{K}{\Delta x^2}\left(u_3 - 2u_2 + u_1\right) + F(x_2, t) \xrightarrow{u_1=0} \frac{K}{\Delta x^2}(-2u_2 + u_3) + F(x_2, t)$$

$$u_t|_{x_3} = \frac{K}{\Delta x^2}\left(u_4 - 2u_3 + u_2\right) + F(x_3, t) \qquad \text{(all interior)}$$

$$\vdots$$

$$u_t|_{x_M} = \frac{K}{\Delta x^2}\left(u_{M+1} - 2u_M + u_{M-1}\right) + F(x_M, t) \xrightarrow{u_{M+1}=0} \frac{K}{\Delta x^2}(u_{M-1} - 2u_M) + F(x_M, t)$$

- The BCs kill the off-grid terms u_1 and u_{M+1}.
- Every row is a **linear combination** of (at most) three entries of $\vec{Y}$.
- That linear combination $\Rightarrow$ matrix-vector product.

The matrix A

$$A = \frac{K}{\Delta x^2} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \in \mathbb{R}^{(M-1)\times(M-1)}$$

- **Tridiagonal:** nonzero only on the diagonal and the two adjacent diagonals.
- **Symmetric.**
- The top-left -2 and bottom-right -2 look just like the interior ones because the BCs are zero — the missing neighbors contribute nothing.
- Sparse and structured: very cheap to store and to multiply.

The source vector $\vec{b}(t)$ and the initial condition

The source term $F(x,t)$ contributes a (known) right-hand side at each row:

$$\vec{b}(t) = \begin{bmatrix} F(x_2,t) \\ F(x_3,t) \\ \vdots \\ F(x_M,t) \end{bmatrix}.$$

The initial condition $u(x,0) = f(x)$, evaluated at each interior node, gives us $\vec{Y}_0$:

$$\vec{Y}_0 = \vec{Y}(0) = \begin{bmatrix} f(x_2) \\ f(x_3) \\ \vdots \\ f(x_M) \end{bmatrix}.$$

The full ODE system

Putting everything together:

$$\vec{Y}'(t) = A\vec{Y}(t) + \vec{b}(t), \qquad \vec{Y}(0) = \vec{Y}_0$$

- This is a **linear system of ODEs** in $\mathbb{R}^{M-1}$.
- It is called the semi-discrete system: space is discrete, time is still continuous.
- Every ODE technique you already know can be applied here — Forward Euler, Backward Euler, Runge-Kutta, ...

That is the method of lines.

Forward Euler in time

Forward Euler: quick reminder

For a generic ODE $y'(t) = g(t, y)$ with $y(t_n) = y_n$, Forward Euler approximates the slope at t_n:

$$y_{n+1} = y_n + \Delta t \cdot g(t_n, y_n).$$

It is explicit: the update formula gives y_{n+1} directly.

It is first-order accurate in Δt: local truncation error is $O(\Delta t^2)$, global error is $O(\Delta t)$.

Forward Euler on our system

Apply it row-by-row to $\vec{Y}'(t) = A\vec{Y}(t) + \vec{b}(t)$:

$$\vec{Y}_{n+1} \;=\; \vec{Y}_n \;+\; \Delta t\,\big(\,A\,\vec{Y}_n + \vec{b}(t_n)\,\big), \qquad n = 0, 1, \ldots, N-1$$

with $\vec{Y}_0 = \big[\, f(x_2),\, \ldots,\, f(x_M)\,\big]^T$.

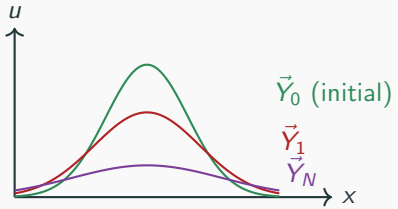
Per time step, the cost is essentially one sparse matrix-vector product $A\vec{Y}_n$ plus two vector additions. Very cheap.

Putting together the full space-time solution

Once we have computed $\vec{Y}_0, \vec{Y}_1, \ldots, \vec{Y}_N$ (each an $(M-1)$-vector), we reattach the zero boundary values to recover the full grid:

$$U_{\text{all}} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ \vec{Y}_0 & \vec{Y}_1 & \vec{Y}_2 & \cdots & \vec{Y}_N \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{(M+1)\times(N+1)}$$

Each column is a snapshot of the temperature at one time:



Forward Euler: a warning

Forward Euler is cheap, but for diffusion problems it is conditionally stable.

Without going into the derivation, one can show that for the heat equation Forward Euler requires

$$\Delta t \leq \frac{\Delta x^2}{2K}.$$

- Halving Δx means Δt must drop by a factor of **four**.
- Fine spatial grids $\Rightarrow$ many, many tiny time steps.
- This can be painfully slow.

Is there a scheme that lets us take larger time steps? Yes — Backward Euler.

Backward Euler and the linear solve

Backward Euler: same idea, different point

Forward Euler used the slope at the *old* time t_n.

Backward Euler uses the slope at the *new* time t_{n+1}:

$$y_{n+1} = y_n + \Delta t \cdot g(t_{n+1}, y_{n+1}).$$

- This is an implicit method: y_{n+1} appears on both sides.
- In general we need to solve an equation to get y_{n+1}.
- For our *linear* ODE system the equation is a linear system — nice.
- Big payoff: Backward Euler is unconditionally stable for the heat equation. Any Δt works.

Backward Euler on our ODE system

Apply the scheme to $\vec{Y}'(t) = A\vec{Y}(t) + \vec{b}(t)$:

$$\vec{Y}_{n+1} \;=\; \vec{Y}_n + \Delta t\,\big[A\,\vec{Y}_{n+1} + \vec{b}(t_{n+1})\big].$$

The unknown $\vec{Y}_{n+1}$ appears on both sides. Move it all to the left:

$$\vec{Y}_{n+1} - \Delta t\,A\,\vec{Y}_{n+1} \;=\; \vec{Y}_n + \Delta t\,\vec{b}(t_{n+1}).$$

Factor out $\vec{Y}_{n+1}$:

$$\big(\mathbf{I} - \Delta t\,A\big)\,\vec{Y}_{n+1} \;=\; \vec{Y}_n + \Delta t\,\vec{b}(t_{n+1}).$$

This is a linear system of the standard form $C\vec{x} = \vec{c}$ at every time step, with

$$C = \mathbf{I} - \Delta t\,A, \qquad \vec{c} = \vec{Y}_n + \Delta t\,\vec{b}(t_{n+1}), \qquad \vec{x} = \vec{Y}_{n+1}.$$

Solving the linear system

At each time step we need to solve

$$\left(\mathbf{I} - \Delta t\, A\right) \vec{Y}_{n+1} \;=\; \vec{Y}_n + \Delta t\, \vec{b}(t_{n+1}).$$

Conceptually:

$$\vec{Y}_{n+1} = (\mathbf{I} - \Delta t\, A)^{-1} \left[\vec{Y}_n + \Delta t\, \vec{b}(t_{n+1})\right].$$

In practice: never compute the inverse explicitly. Use a linear solver — in Julia or MATLAB, the backslash operator:

```
Y_new = C \ (Y + dt * b(t_new))
```

`C \ c` reads: "solve $C\vec{x} = \vec{c}$ for $\vec{x}$." It is much faster and more stable than computing C^{-1}.

HW 3 note
For HW 3 it is perfectly fine to just use "\". You do not have to code up your own linear solver.

Forward vs. Backward Euler at a glance

	Forward Euler	**Backward Euler**
Formula	$\vec{Y}_{n+1} = \vec{Y}_n + \Delta t(A\vec{Y}_n + \vec{b}_n)$	$(\mathbf{I} - \Delta t A)\vec{Y}_{n+1} = \vec{Y}_n + \Delta t\, \vec{b}_{n+1}$
Type	explicit	implicit
Per-step work	matrix-vector multiply	linear solve
Stability (heat eq.)	$\Delta t \leq \Delta x^2/(2K)$	unconditionally stable
Accuracy	$O(\Delta t)$	$O(\Delta t)$

Both first-order in time. Choice is usually driven by stability needs and size of time step.

Wrapping up

The full method of lines pipeline

Putting the two lectures together:

1. **Discretize in space** (central differences): get a system of ODEs $\vec{Y}' = A\vec{Y} + \vec{b}(t)$.
2. **Build** A, $\vec{b}(t)$, $\vec{Y}_0$ from the problem data.
3. **Discretize in time** (Forward or Backward Euler, or anything else):
   - Forward: one matrix-vector multiply per step.
   - Backward: one linear solve per step.
4. **Assemble** all time steps into the full space-time solution (re-attach the zero boundary rows).

You now have a complete numerical method for a class of PDEs that do not have closed-form solutions in general.

Questions?