Outline

Partial Differential Equations

- *Partial differential equations* (PDEs) involve partial derivatives with respect to more than one independent variable
- Independent variables typically include one or more space dimensions and possibly time dimension as well
- More dimensions complicate problem formulation: we can have pure initial value problem, pure boundary value problem, or mixture of both
- Equation and boundary data may be defined over irregular domain

Partial Differential Equations, continued

- For simplicity, we will deal only with single PDEs (as opposed to systems of several PDEs) with only two independent variables, either
  - two space variables, denoted by x and y, or
  - one space variable denoted by x and one time variable denoted by t
- Partial derivatives with respect to independent variables are denoted by subscripts, for example
  - $u_t = \partial u / \partial t$
  - $u_{xy} = \partial^2 u / \partial x \partial y$

Classification of PDEs

- *Order* of PDE is order of highest-order partial derivative appearing in equation
- For example, advection equation is first order $u_t = -c\,u_x$
- Important second-order PDEs include
  - *Heat equation*: $u_t = u_{xx}$
  - *Wave equation*: $u_{tt} = u_{xx}$
  - *Laplace equation*: $u_{xx} + u_{yy} = 0$

Classification of PDEs, continued

- Second-order linear PDEs of general form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

  are classified by value of *discriminant* $b^2 - 4ac$

- $b^2 - 4ac > 0$: *hyperbolic* (e.g., wave equation)
- $b^2 - 4ac = 0$: *parabolic* (e.g., heat equation)
- $b^2 - 4ac < 0$: *elliptic* (e.g., Laplace equation)

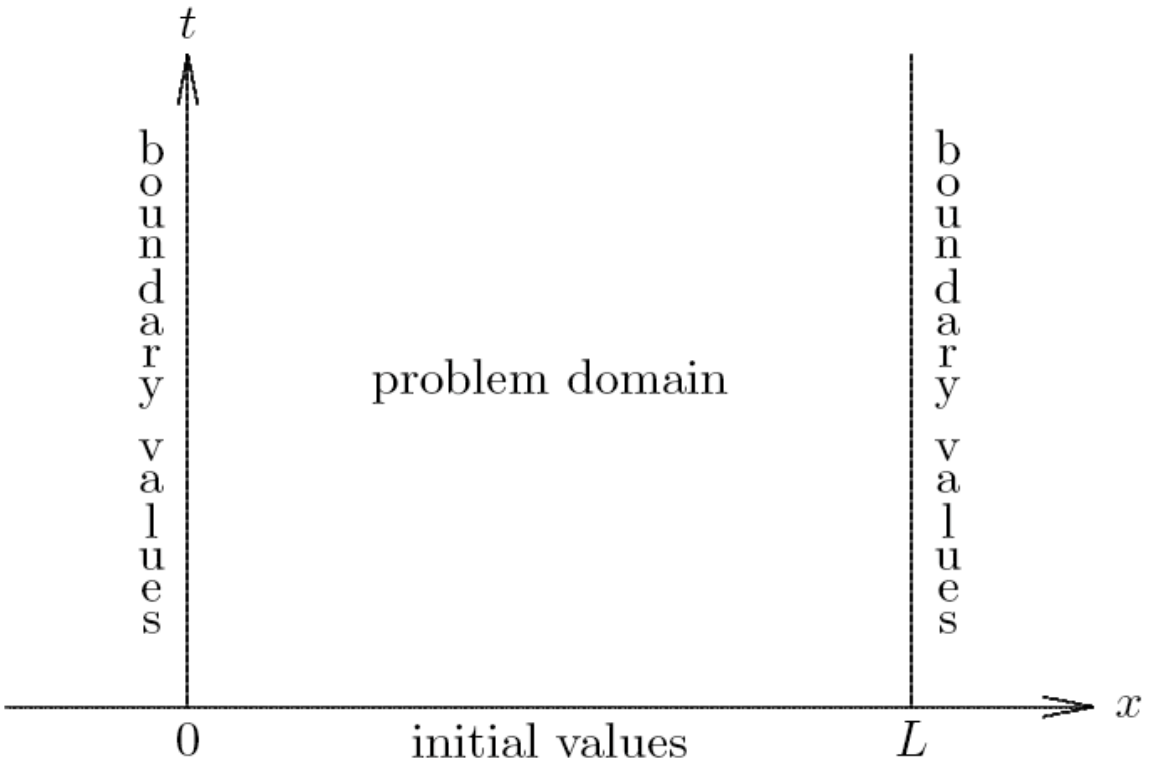
Classification of PDEs, continued

Classification of more general PDEs is not so clean and simple, but roughly speaking

- *Hyperbolic* PDEs describe time-dependent, conservative physical processes, such as convection, that *are not* evolving toward steady state
- *Parabolic* PDEs describe time-dependent, dissipative physical processes, such as diffusion, that *are* evolving toward steady state
- *Elliptic* PDEs describe processes that have already reached steady state, and hence are time-independent

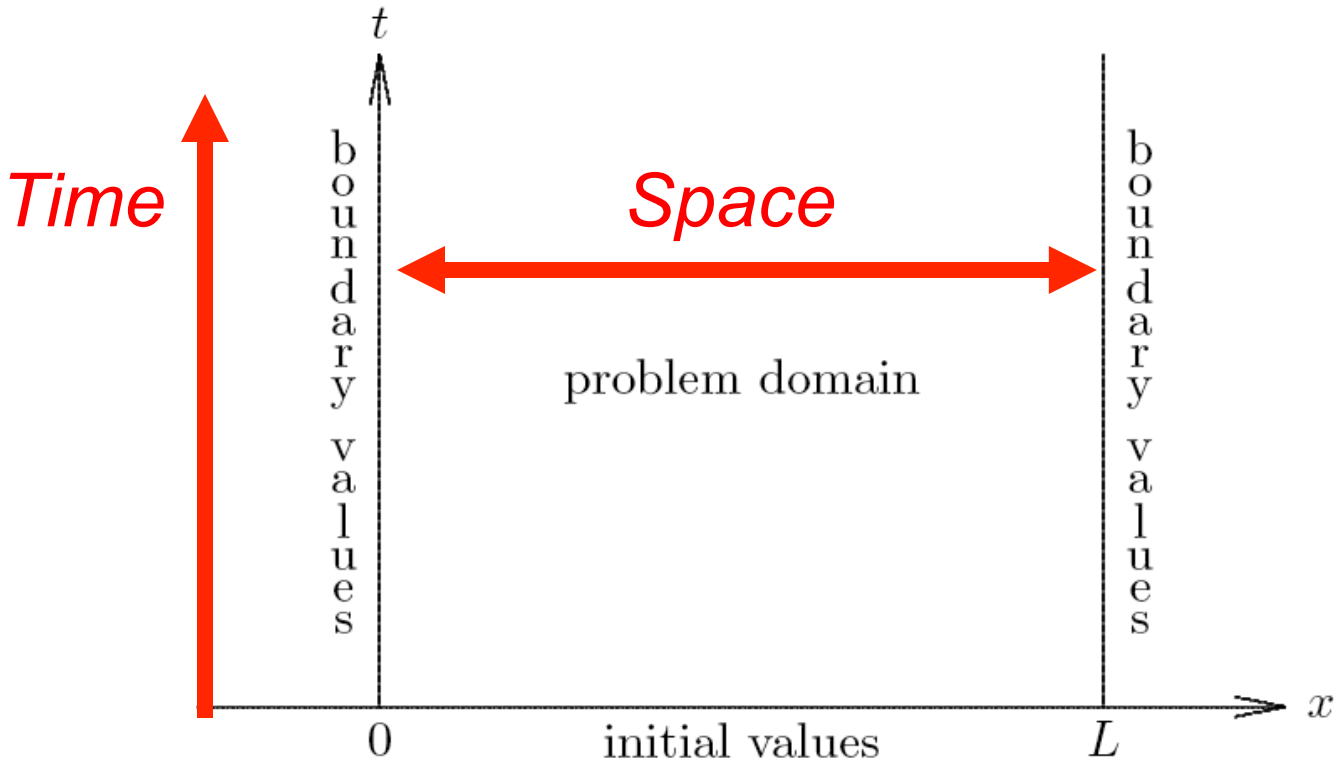
Time-Dependent Problems

- Time-dependent PDEs usually involve both initial values and boundary values

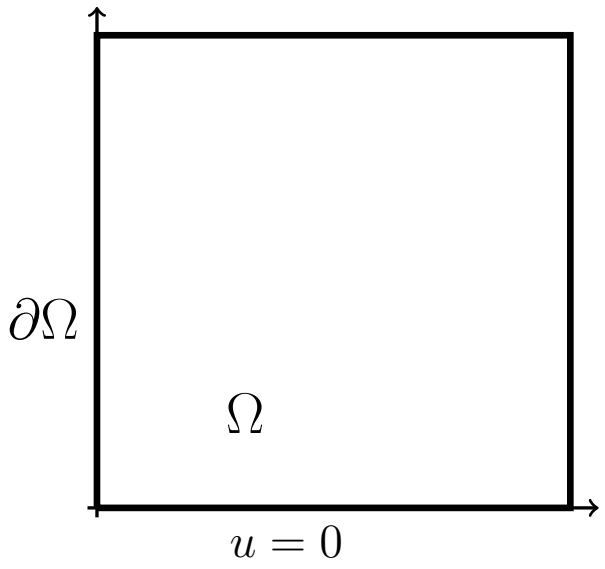


Time-Dependent Problems

- Time-dependent PDEs usually involve both initial values and boundary values



Example: Poisson Equation in 2D



$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y) \text{ in } \Omega$$

$$u = 0 \text{ on } \partial\Omega$$

- Ex 1: If $f(x,y) = \sin \pi x \ \sin \pi y$,

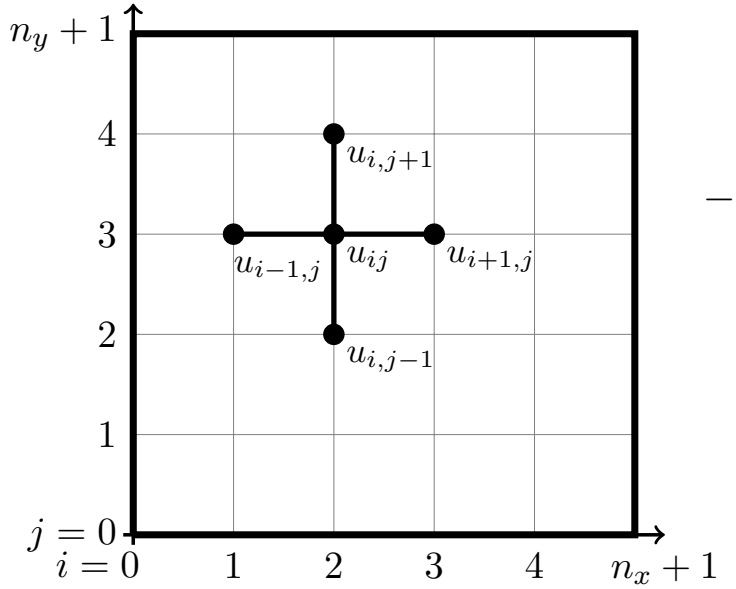
$$u(x,y) = \frac{1}{2\pi^2} \sin \pi x \ \sin \pi y$$

- Ex 2: If $f(x,y) = 1$,

$$u(x,y) = \sum_{k,l \text{ odd}}^{\infty,\infty} \frac{16}{\pi^2 kl(k^2 + l^2)} \sin k\pi x \ \sin l\pi y.$$

  - Q: How large must k and l be for "exact" solution to be correct to ϵ_M ?
  - Spectral collocation would yield $u = u_{\text{exact}} \pm \epsilon_M$ by $N \approx 15$.

Numerical Solution: Finite Differences



"5-point finite-difference stencil"

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \approx \left(\frac{u_{i+1,j} - 2u_{ij} - u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{ij} - u_{i,j-1}}{\Delta y^2}\right) = f_{ij}$$

$$i = 1 \ldots n_x$$

$$j = 1 \ldots n_y$$

- Here, the unknowns are $\mathbf{u} = [u_{11}, u_{21}, \ldots, u_{n_x,n_y}]^T$.
- This particular (so-called natural or lexicographical) ordering gives rise to a banded system matrix for $\mathbf{u}$.
- As in the 1D case, the error is $O(\Delta x^2) + O(\Delta y^2) = O(h^2)$ if we take $\Delta x = \Delta y =: h$.
- Assuming for simplicity that $N = n_x = n_y$, we have $n = N^2$ unknowns.

- For $i, j \in [1, \ldots, N]^2$, the governing finite difference equations are

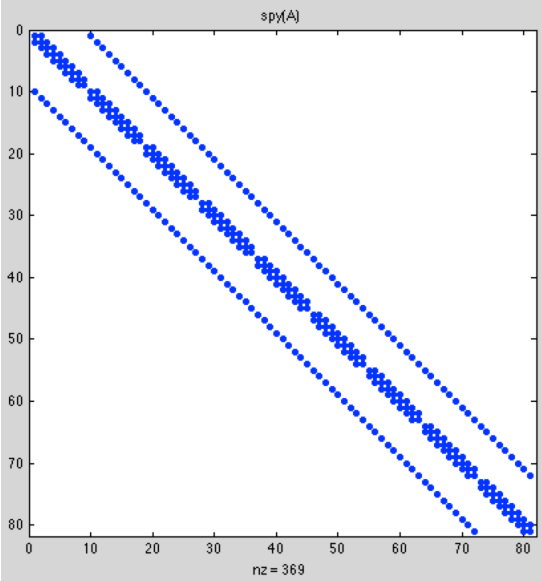
$$-\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2}\right) \quad = \quad f_{ij}.$$

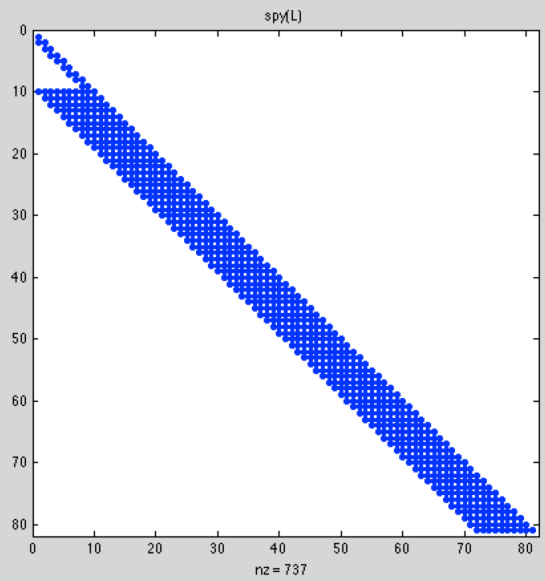
- Assuming a *lexicographical ordering* in which the i- (x-) index advances fastest, the system matrix has the form

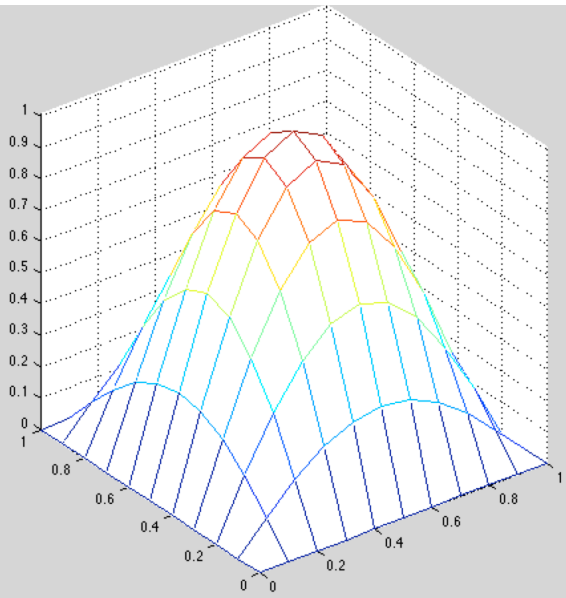
$$\frac{1}{h^2}\underbrace{\left(\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
 & -1 & \ddots & \ddots & & & \ddots & & & & & & & & & \\
 & & \ddots & \ddots & -1 & & & \ddots & & & & & & & & \\
 & & & -1 & 4 & & & & -1 & & & & & & & \\
\hline
-1 & & & & 4 & -1 & & & \ddots & & & & & & & \\
 & -1 & & & -1 & 4 & -1 & & & \ddots & & & & & & \\
 & & \ddots & & & -1 & \ddots & \ddots & & & \ddots & & & & & \\
 & & & \ddots & & & \ddots & \ddots & -1 & & & \ddots & & & & \\
 & & & & -1 & & & -1 & 4 & & & & \ddots & & & \\
\hline
 & & & & \ddots & & & & \ddots & & & & -1 & & & \\
 & & & & & \ddots & & & & \ddots & & & & -1 & & \\
 & & & & & & \ddots & & & & \ddots & & & & \ddots & \\
 & & & & & & & \ddots & & & & \ddots & & & & -1 \\
 & & & & & & & & \ddots & & & & \ddots & & & \\
\hline
 & & & & & & & & -1 & & & & 4 & -1 & & \\
 & & & & & & & & & -1 & & & -1 & 4 & \ddots & \\
 & & & & & & & & & & \ddots & & & \ddots & \ddots & \ddots \\
 & & & & & & & & & & & -1 & & & \ddots & -1 \\
 & & & & & & & & & & & & & & -1 & 4
\end{array}\right)}_{A}
\underbrace{\left(\begin{array}{c}
u_{11} \\ u_{21} \\ \vdots \\ \vdots \\ u_{N1} \\ \hline u_{12} \\ u_{22} \\ \vdots \\ \vdots \\ u_{N2} \\ \hline \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \hline u_{1N} \\ u_{2N} \\ \vdots \\ \vdots \\ u_{NN}
\end{array}\right)}_{\mathbf{u}}
=
\underbrace{\left(\begin{array}{c}
f_{11} \\ f_{21} \\ \vdots \\ \vdots \\ f_{N1} \\ \hline f_{12} \\ f_{22} \\ \vdots \\ \vdots \\ f_{N2} \\ \hline \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \hline f_{1N} \\ f_{2N} \\ \vdots \\ \vdots \\ f_{NN}
\end{array}\right)}_{\mathbf{f}}$$

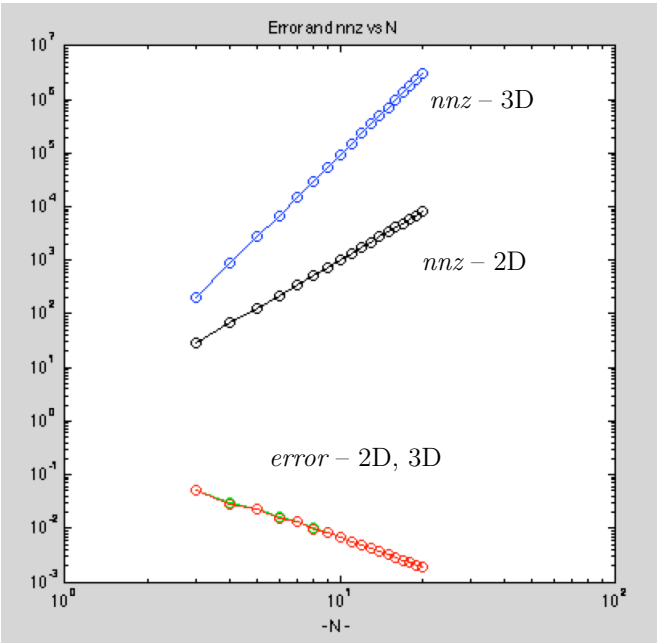
- The system matrix A is
  - *sparse*, with 5 nonzeros per row (good)
  - and has a bandwith N (bad).
- The difficulty is that solving $A\mathbf{u} = \mathbf{f}$ using Gaussian elimination results in signifcant *fill*— each of the factors L and U have $N^3 = n^{3/2}$ nonzeros.
- Worse, for 3D problems with N^3 unknowns, $\mathbf{u} = [u_{111}, u_{211}, \ldots, u_{n_x,n_y,n_z}]^T$, A is
  - *sparse*, with 7 nonzeros per row (good)
  - and has a bandwith N^2 (awful).
- In 3D, LU decomposition yields N^5=$n^{5/3}$ nonzeros in L and U.
- The situation can be rescued in 2D with a reordering of the unknowns (e.g., via nested-dissection) to yield $O(n \log n)$ nonzeros in L and U.
- In 3D, nested-dissection yields $O(n^{3/2})$ nonzeros in the factors. Direct solution is not scalable for more than two space dimensions.
- The following Matlab examples illustrate the issue of fill:
  - fd_poisson_2d.m
  - fd_poisson_3d.m

Matrix-Fill for 2D and 3D Poisson, Lexicographical Ordering



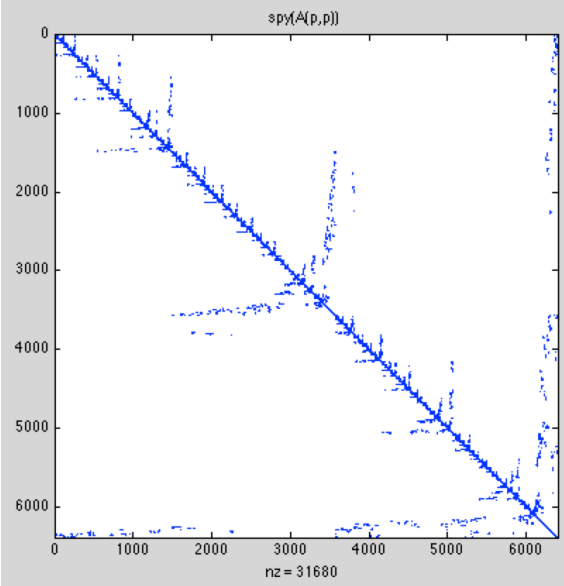


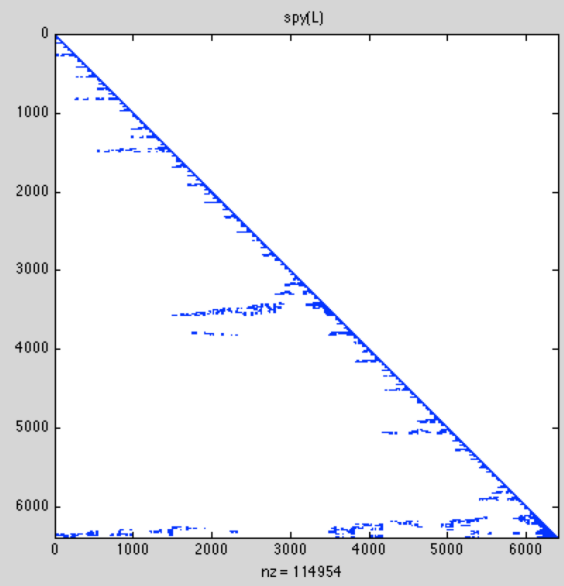


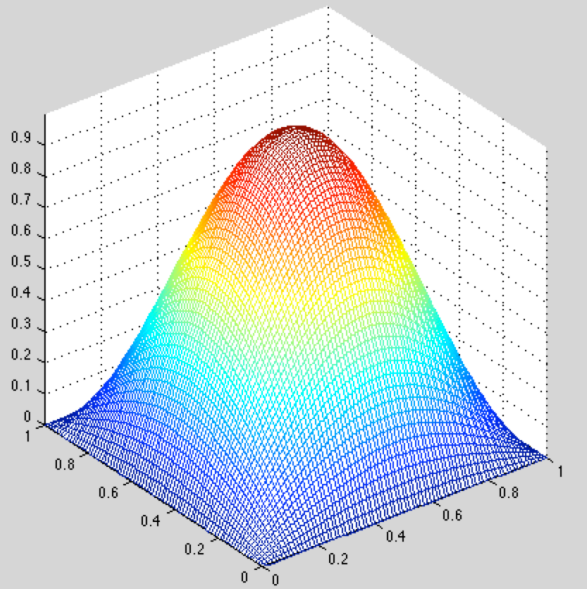


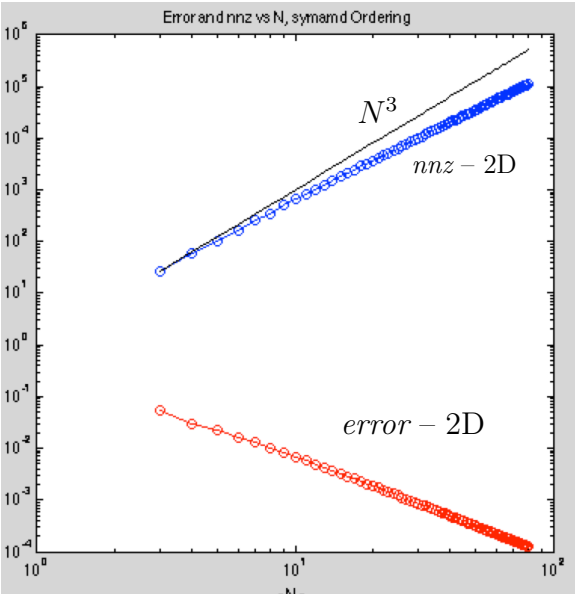
- As expected, the error scales like $h^2 \sim 1/N^2$ in both 2D and 3D.
- The resepctive storage costs (and work per rhs) are $\sim N^3$ and N^5.
- Alternative orderings are asymptotically better, but the constants tend to be large.

Matrix-Fill for 2D Poisson, `symamd` Ordering









- We see for $N = 80$ ($n = 6400$) a $5\times$ reduction in number of nonzeros by reording with matlab's `symamd` function.
- The requirements for indirect addressing to access elements of the complacty-stored matrix further adds to overhead.
- Gains tend to be realized only for very large N and are even less beneficial in 3D.
- Despite this, it's still a reasonable idea to reorder in matlab because it's available and easy to use.

Iterative Solvers

- The *curse of dimensionality* for $d > 2$ resulted in a move towards iterative (rather than direct-, LU-based) linear solvers once computers became fast enough to tackle 3D problems in the mid-80s.
- With iterative solvers, factorization

$$A\mathbf{u} \;=\; \mathbf{f} \;\;\Longrightarrow\;\; \mathbf{u} \;=\; A^{-1}\mathbf{f} \;=\; U^{-1}L^{-1}\mathbf{f}$$

  is replaced by, say,

$$\mathbf{u}_{k+1} \;\;=\;\; \mathbf{u}_k \;+\; M^{-1}\left(\mathbf{f} \;-\; A\mathbf{u}_k\right),$$

  which only requires matrix-vector products.
- With $\mathbf{e}_k \;:=\; \mathbf{u} \;-\; \mathbf{u}_k$, we have

$$\mathbf{e}_{k+1} \;\;=\;\; \left(I \;-\; M^{-1}A\right)\mathbf{e}_k, \qquad \text{(as we've seen before).}$$

- This is known as Richardson iteration.
- For the particular case $M = D = \mathrm{diag}(A)$, it is Jacobi iteration.
- We can derive Jacobi iteration (and multigrid by looking at a *parabolic* PDE, known as the (unsteady) heat equation. (The Poisson equation is sometimes referred to as the steady-state heat equation.)

- The intrinsic advantage of iterative solvers is that there is no *fill* associated with matrix factorization.
- Often one does not even construct the matrix. Rather, we simply evaluate the residual $\mathbf{r}_k := \mathbf{f} - A\mathbf{u}_k$ and set $\mathbf{u}_{k+1} = \mathbf{u}_k + M^{-1}\mathbf{r}_k$.
- For a *sparse matrix* A, the operation count is $O(n)$ per iteration.
- Assuming the preconditioner cost is also sparse, the overall cost is $O(n\,k_{\max})$, where $k_{\max}$ is the number of iterations required to reach a desired tolerance.
- The choice of iteration (Richardson, conjugate gradient, GMRES) can greatly influence $k_{\max}$.
- Even more significant is the choice of M.
- Usually, one seeks an M such that the cost of solving $M\mathbf{z} = \mathbf{r}$ is $O(n)$ and that $k_{\max} = O(1)$. That is, the iteration count is bounded, independent of n.
- The overall algorithm is therefore $O(n)$, which is optimal.

Iterative Solvers - Linear Elliptic Problems

- PDEs give rise to large sparse linear systems of the form

$$A\mathbf{u} = \mathbf{f}.$$

  Here, we'll take A to be the (SPD) matrix arising from finite differences applied to the Poisson equation

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y) \qquad x, y \in [0,1]^2, \qquad u = 0 \text{ on } \partial\Omega$$

$$-\left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right)_{ij} \approx f|_{ij}\,,$$

- Assuming uniform spacing in x and y we have

$$\frac{\delta^2 u}{\delta x^2} := \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} \quad \text{and} \quad \frac{\delta^2 u}{\delta y^2} := \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h^2}$$

- Our finite difference formula is thus,

$$\frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} - 4u_{ij} + u_{i,j+1} + u_{i,j-1}\right) = f_{ij}.$$

- Rearranging, we can solve for u_{ij}:

$$\frac{4}{h^2}u_{ij} = f_{ij} + \frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

$$u_{ij} = \frac{h^2}{4}f_{ij} + +\frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

- Jacobi iteration uses the preceding expression as a fixed-point iteration:

$$u_{ij}^{k+1} = \frac{h^2}{4} f_{ij} + \frac{1}{4}\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)$$

$$= \frac{h^2}{4} f_{ij} + \text{average of current neighbor values}$$

- Note that this is analogous to

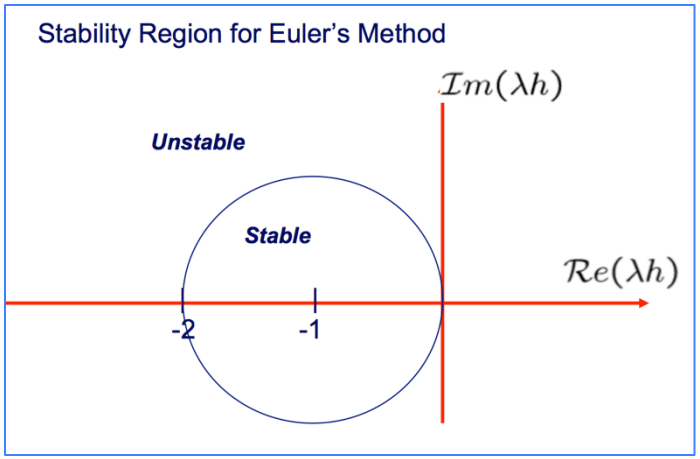
$$u_{ij}^{k+1} = u_{ij}^k + \frac{h^2}{4}\left[f_{ij} + \frac{1}{h^2}\left(u_{i+1,j}^k + u_{i-1,j}^k - 4u_{ij}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)\right]$$

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \Delta t\left(\mathbf{f} - A\mathbf{u}_k\right), \quad \boxed{\Delta t := \frac{h^2}{4},}$$

which is Euler forward applied to

$$\frac{d\mathbf{u}}{dt} = -A\mathbf{u} + \mathbf{f}.$$

- We note that we have stability if $|\lambda \Delta t| < 2$



- Recall that the eigenvalues for the 1D diffusion operator are

$$\lambda_j \;=\; \frac{2}{h^2}\left(1-\cos j\pi\Delta x\right) < \frac{4}{h^2}$$

- In 2D, we pick up contributions from both $\frac{\delta^2 u}{\delta x^2}$ and $\frac{\delta^2 u}{\delta y^2}$, so

$$\max|\lambda| \;<\; \frac{8}{h^2}$$

  and we have stability since

$$\max|\lambda\Delta t| \;<\; \frac{8}{h^2}\frac{h^2}{4} = 2$$

- So, Jacobi iteration is equivalent to solving $A\mathbf{u} = \mathbf{f}$ by time marching $\frac{d\mathbf{u}}{dt} \;=\; -A\mathbf{u} + \mathbf{f}$ using EF with maximum allowable timestep size,

$$\Delta t \;=\; \frac{h^2}{4}.$$

Jacobi Iteration in Matrix Form

- Our unsteady heat equation has the matrix form

$$\mathbf{u}_{k+1} \;=\; \mathbf{u}_k \,+\, \Delta t\,(\mathbf{f} - A\mathbf{u}_k)$$

- For variable diagonal entries, Richardson iteration is

$$\mathbf{u}_{k+1} \;=\; \mathbf{u}_k \,+\, \sigma M^{-1}\,(\mathbf{f} - A\mathbf{u}_k)$$

- If $\sigma = 1$ and $M = D^{-1}$ =diag(A) $[d_{ii} = 1/a_{ii},\, d_{ij} = 0,\, i \neq j]$, we have standard Jacobi iteration.
- If $\sigma < 1$ we have *damped Jacobi*.
- M is generally known as a smoother or a preconditioner, depending on context.

Rate of Convergence for Jacobi Iteration

- Let $\mathbf{e}_k := \mathbf{u} - \mathbf{u}_k$.
- Since $A\mathbf{u} = \mathbf{f}$, we have

$$\begin{aligned}
\mathbf{u}_{k+1} &= \mathbf{u}_k + \Delta t\,(A\mathbf{u} - A\mathbf{u}_k) \\
-\mathbf{u} &= -\mathbf{u} \\
--- &- ---------- \\
-\mathbf{e}_{k+1} &= -\mathbf{e}_k - \sigma \Delta t A \mathbf{e}_k \\
-\mathbf{e}_{k+1} &= -\left(I - \sigma \Delta t A\right) \mathbf{e}_k \\
\mathbf{e}_k &= \left(I - \sigma \Delta t A\right)^k \mathbf{e}_0 \\
&= \left(I - \sigma \Delta t A\right)^k \mathbf{u} \qquad \text{if } \mathbf{u}_0 = 0.
\end{aligned}$$

- If $\sigma < 1$, then the high wavenumber error components will decay because $\lambda \Delta t$ will be well within the stability region for EF.

- The low-wavenumber components of the solution (and error) evolve like $e^{-\lambda\sigma\Delta t k}$, because these will be well-resolved in time by Euler forward.
- Thus, we can anticipate

$$||\mathbf{e}_k|| \;\approx\; ||\mathbf{u}||e^{-\lambda_{\min}\sigma\Delta t k}$$

  with $\lambda_{\min} \approx 2\pi^2$ (for 2D).
- If $\sigma \approx 1$, we have

$$||\mathbf{e}_k|| \;\approx\; ||\mathbf{u}||e^{-2\pi^2(h^2/4)k} \;\leq\; \text{tol}$$

- Example, find the number of iterations when tol=10^{-12}.

$$e^{-(\pi^2h^2/4)k} \;\approx\; 10^{-12}$$

$$-(\pi^2h^2/4)k \;\approx\; \ln 10^{-12} \;\approx\; 24 \;\; (27.6...)$$

$$k \;\approx\; \frac{28\cdot 2}{\pi^2h^2} \;\approx\; 6N^2$$

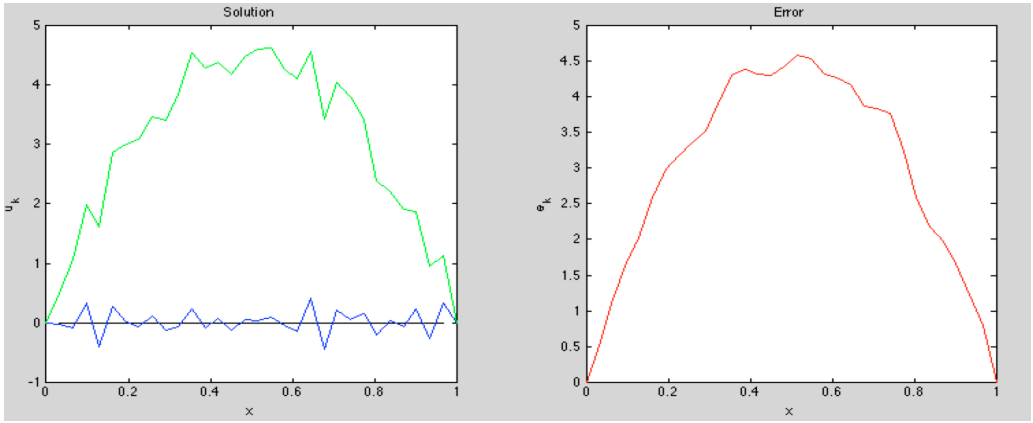
Here, N=number of points in each direction.

Recap

- Low-wavenumber components decay at a fixed rate: $e^{-\lambda_{\min}\Delta t k}$.
- Stability mandates $\Delta t < h^2/4 = 1/4(N+1)^{-2}$.
- Number of steps scales like N^2.
- Note, if $\sigma = 1$, then *highest* and *lowest* wavenumber components decay at *same* rate.
- If $\frac{1}{2} < \sigma < 1$, high wavenumber components of error decay very fast. We say that damped Jacobi iteration is a *smoother*.

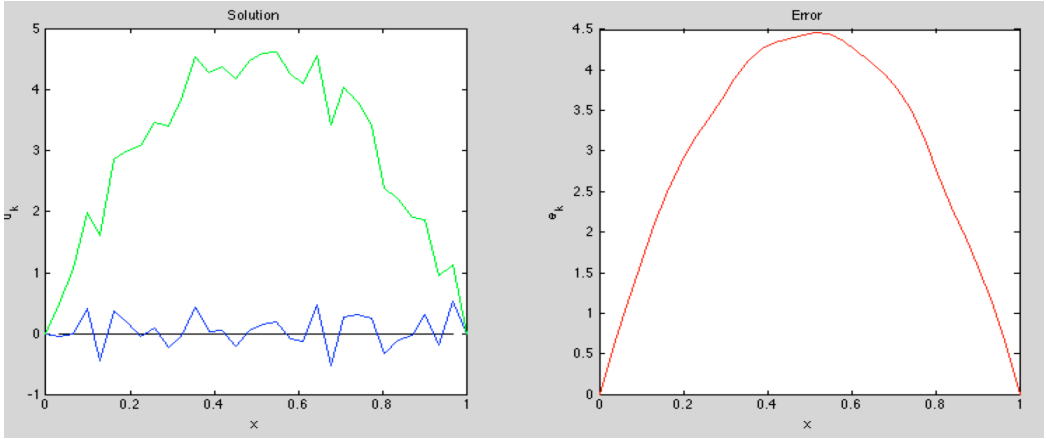
Example: 1D Jacobi Iteration

Solution after 1 iteration



Error after 1 iteration

Solution after 5 itierations



Error after 5 itierations

Observations:

- Error, $\mathbf{e}_k$ is smooth after just a few iterations:
  - Error components are $\approx \hat{u}_j e^{-j^2 k h^2 \pi^2/4} \sin k\pi x_j$, and components for $j > 1$ rapidly go to zero.
- Exact solution is $\mathbf{u} = \mathbf{u}_k + \mathbf{e}_k$ ($\mathbf{e}_k$ unknown, but smooth).
- Error satisfies, and can be computed from,

$$A\mathbf{e}_k = \mathbf{r}_k \qquad (:= \mathbf{f} - A\mathbf{u}_k = A\mathbf{u} - A\mathbf{u}_k = A\mathbf{e}_k)\,.$$

- These observations suggest that the *error* can be well approximated on a coarser grid and added back to $\mathbf{u}_k$ to improve the current guess.
- The two steps, *smooth* and *coarse-grid correction* are at the heart of one of the fastest iteration strategies, known as **multigrid**.

Multigrid:

- Solve $A\mathbf{e}_k = \mathbf{r}_k$ approximately on a coarse grid and set $\tilde{\mathbf{u}}_k = \mathbf{u}_k + \tilde{\mathbf{e}}_k$.
- Approximation strategy is similar to least squares. Let

$$\begin{aligned} \tilde{\mathbf{e}}_k &= V\mathbf{e}_c, \qquad \text{and} \\ AV\mathbf{e}_c &\approx \mathbf{r}, \end{aligned}$$

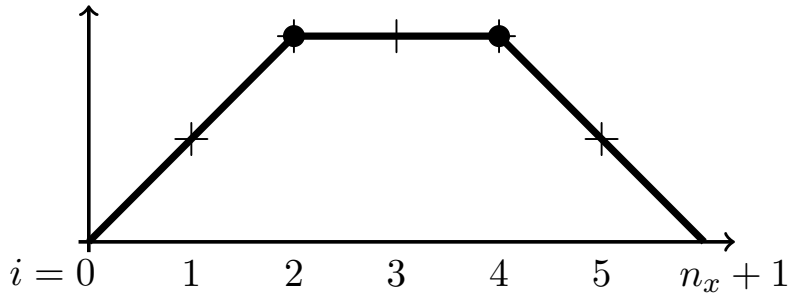
  where V is an $n \times n_c$ matrix with $n_c \approx n/2$.
- Typically, columns of V interpolate coarse point values to their midpoints.
- Most common approach (for A SPD) is to require $\mathbf{e}_c$ to solve

$$\begin{aligned} & V^T\left[AV\mathbf{e}_c - \mathbf{r}\right] = 0 \\ \Longrightarrow\ & \tilde{\mathbf{e}}_k = V\left(V^TAV\right)^{-1}V^T\mathbf{r} = V\left(V^TAV\right)^{-1}V^TA\,\mathbf{e}_k. \end{aligned}$$

- For A SPD, $\tilde{\mathbf{e}}_k$ is the A-orthogonal projection of $\mathbf{e}_k$ onto $\mathcal{R}(V)$.

An example of V for $n = 5$ and n_c=2 is

$$V = \begin{bmatrix} \frac{1}{2} & \\ 1 & \\ \frac{1}{2} & \frac{1}{2} \\ & 1 \\ & \frac{1}{2} \end{bmatrix}$$



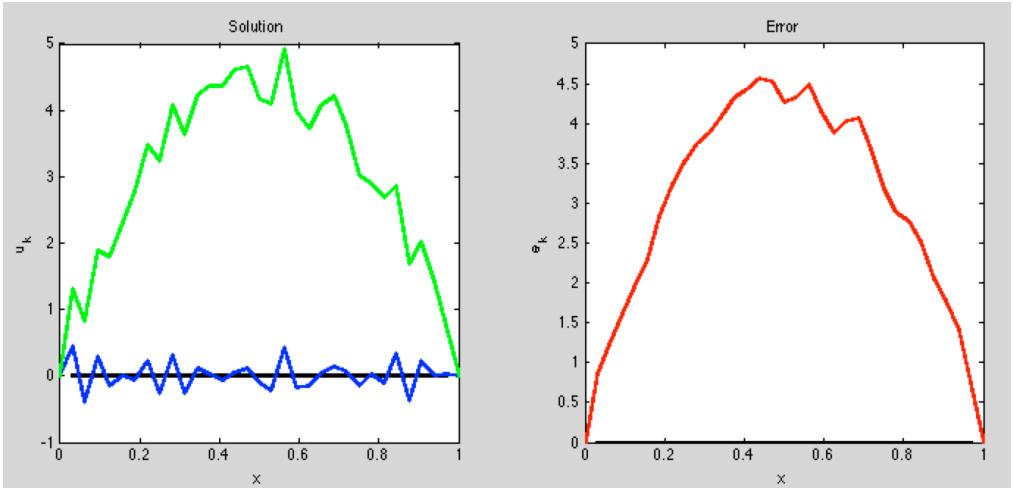
Coarse-to-fine interpolation

```
% Multigrid stuff % n must be odd!

nc = (n-1)/2; V=spalloc(n,nc,n*nc); i=1;
for j=1:nc;
   V(i,j)=1/2; V(i+1,j)=1; V(i+2,j)=1/2; i=i+2;
end;
Ac = V'*A*V;

% A Simple Two-Level MG iteration:

for k=1:5000

    r = f-A*u;            % Smoothing step
    u = u + d*r;

    r  = f-A*u;           % Coarse-grid correction
    rc = V'*r;
    ec = V*( Ac \ rc );
    u = u+ec;

end;
```

poisson_mg.m demo

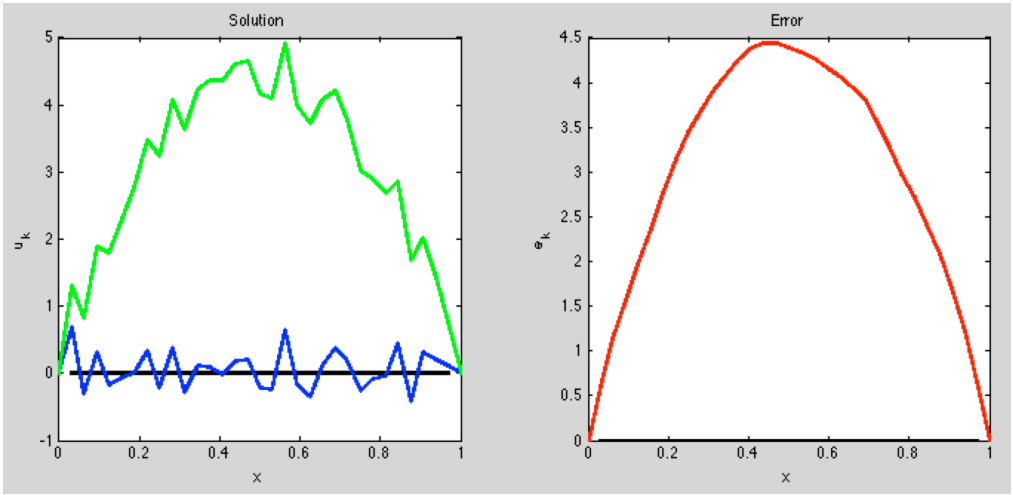
Example: Damped Jacobi (Richardson) Iteration

Solution after 1 iteration



Error after 1 iteration

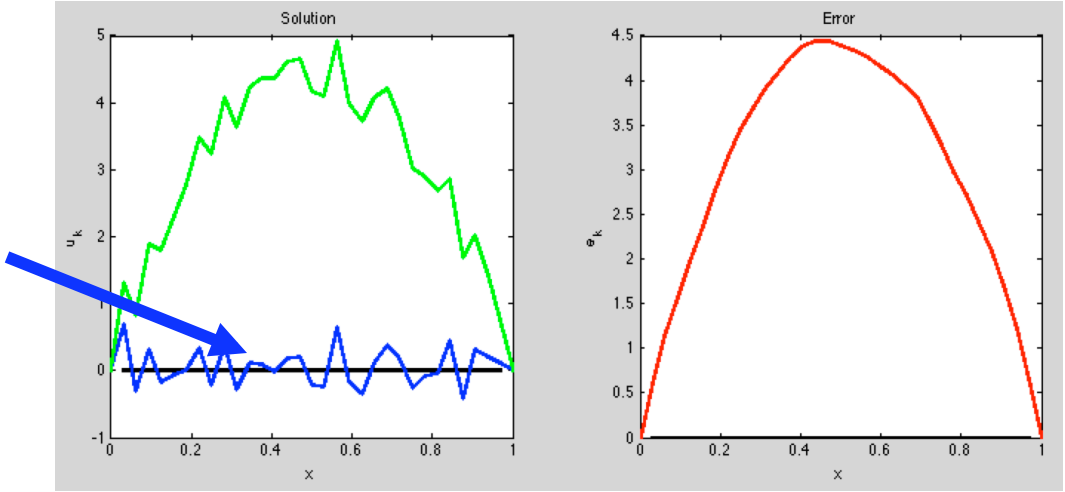
Solution after 5 itierations



Error after 5 itierations

Multigrid Summary – Main Ideas

Solution after 5 iterations



Error after 5 iterations

- Take a few damped-Jacobi steps (smoothing the *error*), to get $\mathbf{u}_k$.
- Approximate this *smooth error*, $\mathbf{e}_k := \mathbf{u} - \mathbf{u}_k$, on a coarser grid.
- Exact error satisfies

$$A\mathbf{e}_k \;=\; A\mathbf{u} - A\mathbf{u}_k \;=\; \mathbf{f} - A\mathbf{u} \;=:\; \mathbf{r}_k.$$

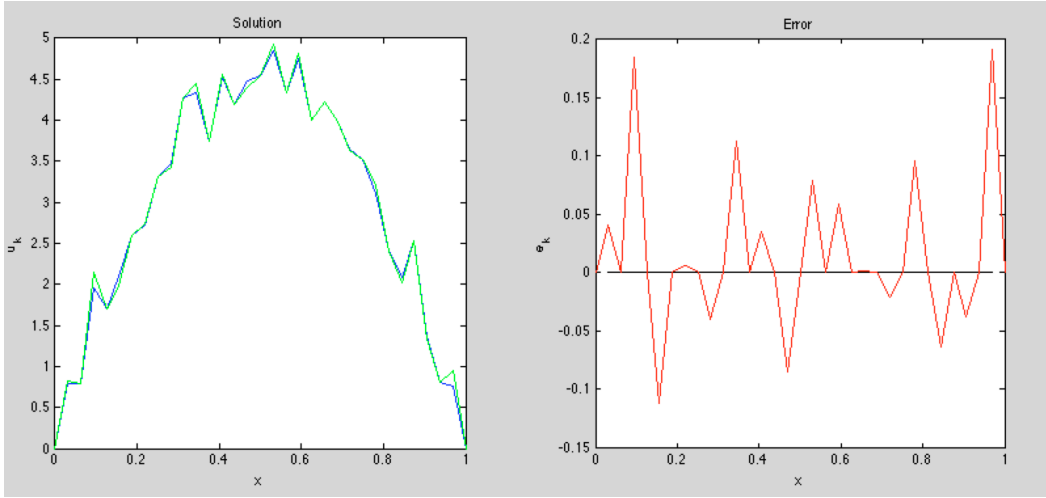
- Let $\mathbf{e}_f := V\mathbf{e}_c$ be the *interpolant* of $\mathbf{e}_c$, the coarse-grid approximation to $\mathbf{e}_k$.
- $\mathbf{e}_f$ is *closest element* in $\mathcal{R}(V)$ to $\mathbf{e}_k$ (in the A-norm), given by the **projection**:

$$\mathbf{e}_f \;=\; V\left(V^TAV\right)^{-1}V^TA\mathbf{e}_k \;=\; V\left(A_c\right)^{-1}V^T\mathbf{r}_k.$$

- **Update** $\mathbf{u}_k$ with the coarse-grid correction: $\mathbf{u}_k \longleftarrow \mathbf{u}_k + \mathbf{e}_f.$
- Smooth again and repeat.

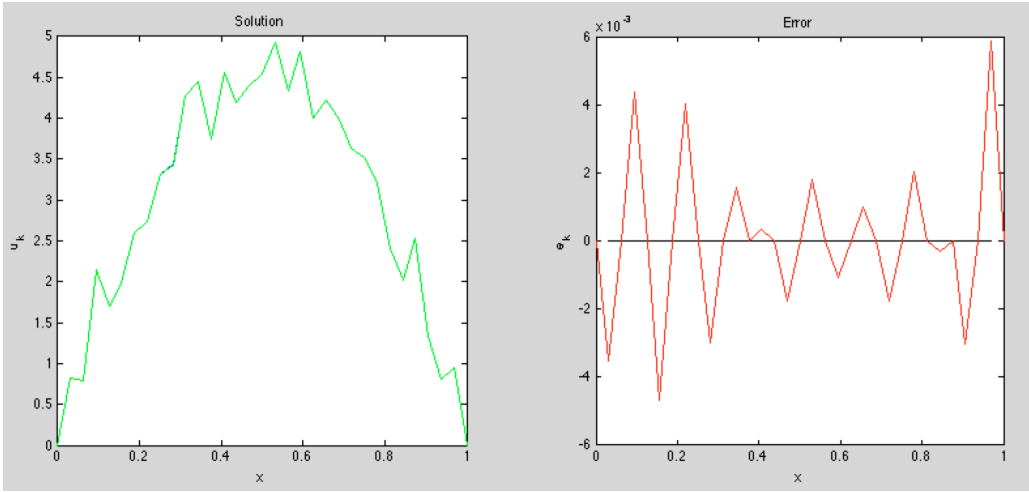
Example: Two-Level Multigrid

Solution after 1 iteration



Error after 1 iteration

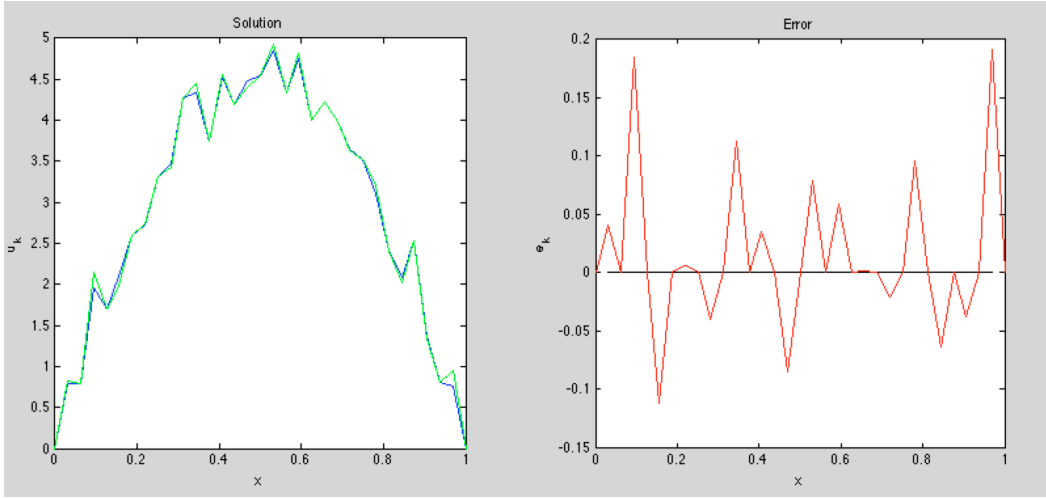
Solution after 5 itierations



Error after 5 itierations

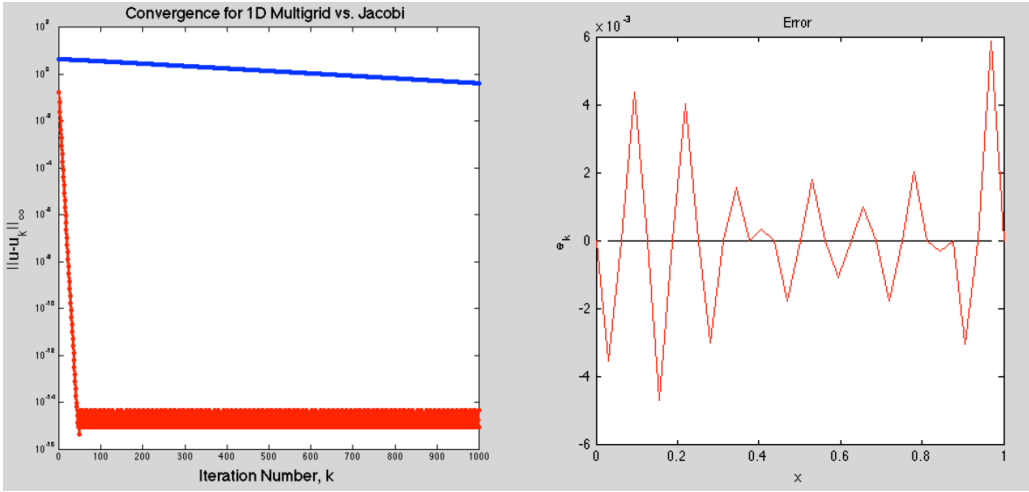
Example: Two-Level Multigrid

Solution after 1 iteration



Error after 1 iteration

Iteration History

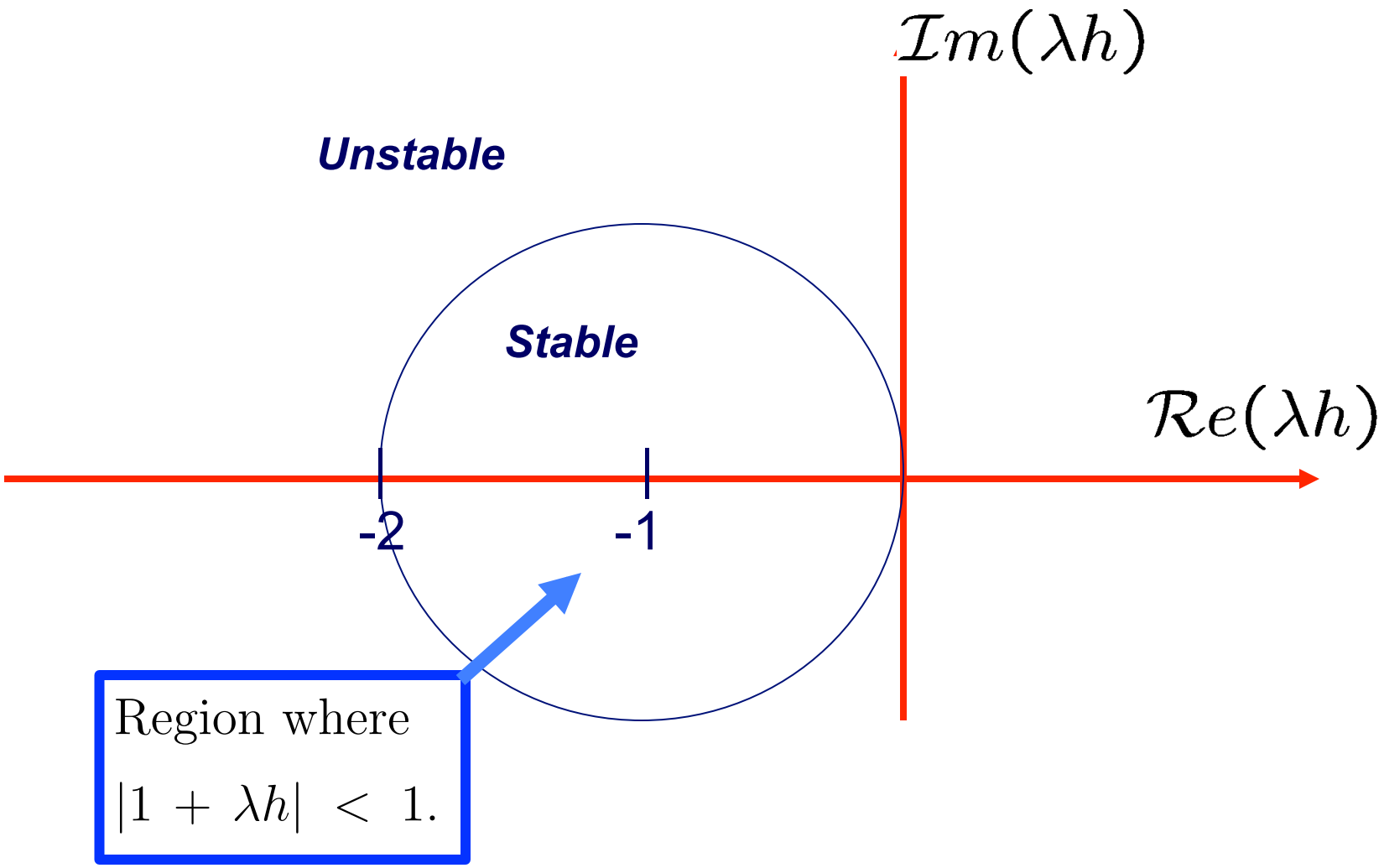


Error after 5 itierations

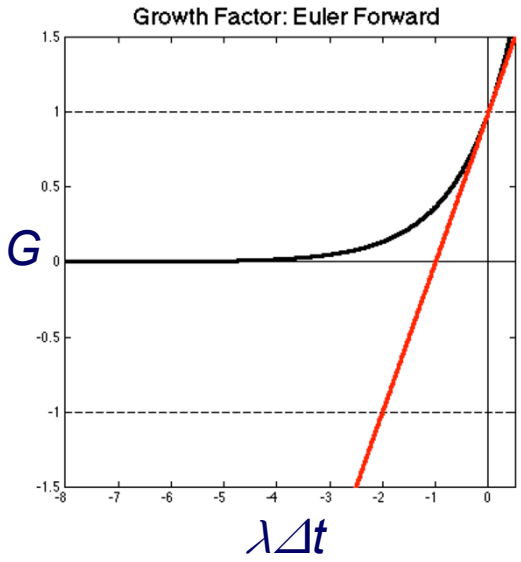
Multigrid Comments

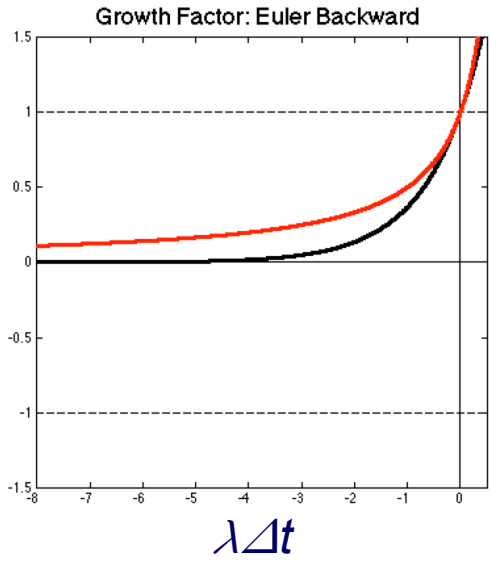
- Smoothing can be improved using under-relaxation (σ = 2/3 is optimal for 1D case).
  - Basically – want more of the high-end error spectrum to be damped.
- System in A_c is less expensive to solve, but is typically best solved by repeating the smooth/ coarse-grid correct pair on yet another level down.
- Can recur until $n_c \sim 1$, at which point system is easy to solve.
- Typical MG complexity is O(n) or O(n log n), with very good constants in higher space dimensions ($N_c = N/2 \rightarrow n_c = n/8$ in 3D).
- For high aspect-ratio cells, variable coefficients, etc., smoothing and coarsening strategies require more care, so this continues to be an active research area.

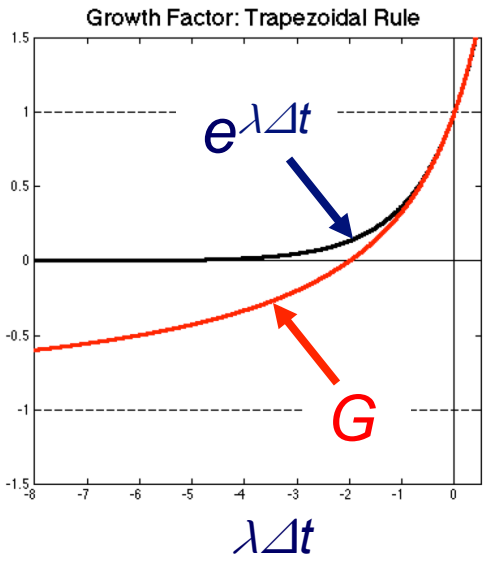
Stability Region for Euler's Method



Growth Factors for Real λ







- Each growth factor approximates $e^{\lambda \Delta t}$ for $\lambda \Delta t \rightarrow 0$
- For EF, |G| is not bounded by 1
- For Trapezoidal Rule, local (small$\lambda \Delta t$) approximation is $O(\lambda \Delta t^2)$, but $|G| \rightarrow -1$ as $\lambda \Delta t \rightarrow -\infty$. [Trapezoid method is not ***L-stable.***]
- BDF2 will give 2nd-order accuracy, stability, and $|G| \rightarrow 0$ as $\lambda \Delta t \rightarrow -\infty$.

Fast Solvers: Iterative Methods

- Consider solution of $A\underline{x} = \underline{b}$, with A and $n \times n$ matrix.
- The key motivations/requirements for choosing an iterative approach over a direct solver are
  - Direct solver is taking a long time (cost scales as n^3
  - Matrix-vector products, $\underline{w} = A\underline{x}$, are inexpensive ($\ll n^2$).
    - Typically, A is *sparse* or has a fast factorization.
  - A is well-conditioned or there exists a good preconditioner, $M \sim A$ such that $\kappa(M^{-1}A)$ is relatively small.
  - Example: If A is symmetric positive definite (SPD), $\kappa(A) = \lambda_n/\lambda_1$.
    - For $A =$ 1D finite-difference matrix ($n = N - 1$),

$$\left.\begin{array}{rcl} \lambda_n & \sim & 4N^2 \\ \lambda_1 & \sim & \pi^2 \end{array}\right\} \kappa \sim \frac{4N^2}{\pi^2}.$$

    - For $A =$ *spectral* (Nth-order WRT), $\kappa(A) \sim O(N^3)$.

Iterative Method I: $A\underline{x} = \underline{b}$

Richardson Iteration (*Preconditioner, M*):

$$\begin{aligned} \underline{x}_0 &= 0 \\ \underline{x}_k &= \underline{x}_{k-1} + M^{-1}\left(\underline{b} - A\underline{x}_{k-1}\right). \end{aligned}$$

- Example: $M = \Delta t I$, $\Delta t =$ *a number.*

$$\underline{x}_k = \underline{x}_{k-1} + \Delta t\left(\underline{b} - A\underline{x}_{k-1}\right)$$

$$\frac{\underline{x}_k - \underline{x}_{k-1}}{\Delta t} = -A\underline{x}_{k-1} + \underline{b} \quad \ldots \textit{ Euler Forward.}$$

Maximal Δt determined by max (magnitude) eigenvalue of A.

Jacobi Iteration:

- Jacobi = Richardson with $M = \sigma D$,

  $D := diag(A)$
  $\sigma = 1$: standard Jacobi
  $\sigma < 1$: *damped Jacobi* (useful for multigrid)

- Example:

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{bmatrix}$$

- $M^{-1} = \frac{h^2}{2} I =$ "$\Delta t I$".

$$\underline{x}_k = \underline{x}_{k-1} + \Delta t \left(\underline{b} - A\underline{x}_{k-1}\right)$$

$$\Delta t = \frac{h^2}{2}.$$

- Recall, max $\lambda(A) = 4N^2 \approx \frac{4}{h^2}$

$$\lambda \Delta t = \frac{h^2}{2} \cdot \frac{4}{h^2} = 2.$$

- **Largest possible Δt!**

What About Costs??

- Cost = (number of iterations) $\times$ (cost-per-iteration).

- Number of iterations (Jacobi):

  - Rel. Error $\sim e^{-\lambda_1 T}, \quad \lambda_1 \sim \pi^2.$

    $\sim e^{-\pi^2 T} \approx 10^{-10} \quad (say)$

  - $-\pi^2 T \approx -10 \ln 10$

    $T \approx \dfrac{10}{\pi^2} \ln 10 \approx 2 \quad \left(\ln e^2 = 2\right).$

  - # iterations $\approx \dfrac{T}{\Delta t} = \dfrac{2}{h^2/2} \approx \dfrac{1}{h^2} \approx N^2.$

Consider 3D –

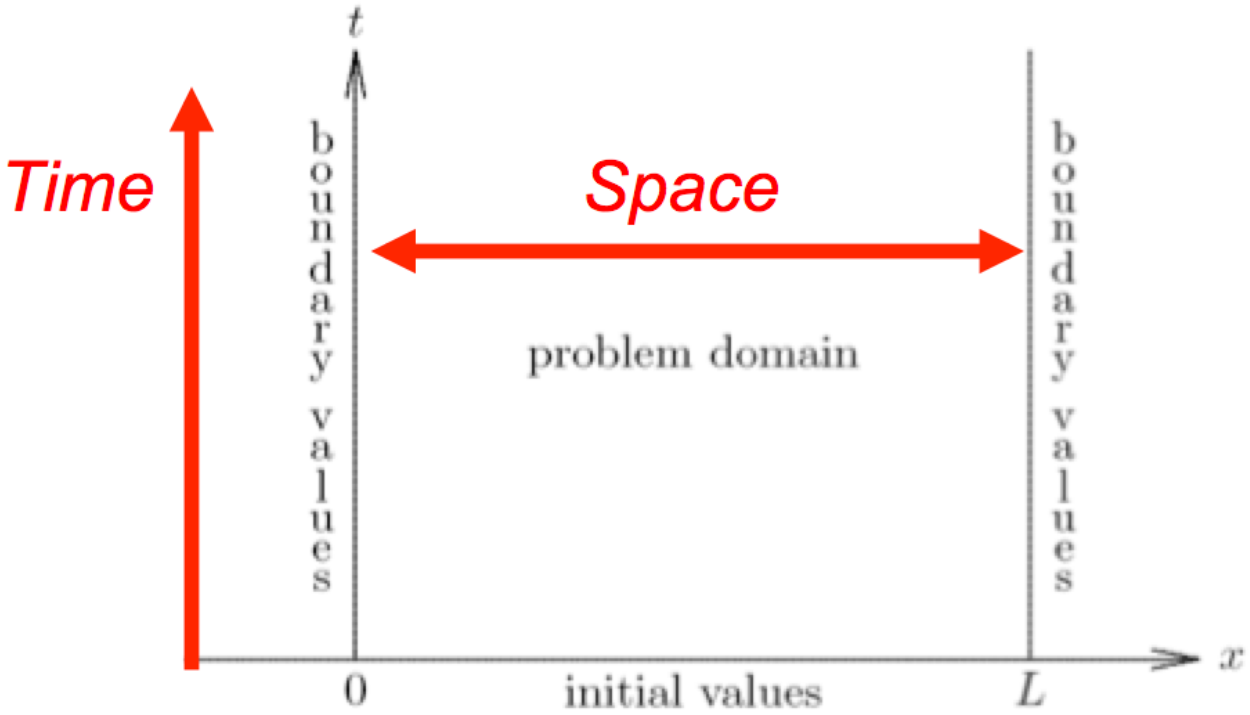
- Similar analysis yields same conclusion: # iter $\sim N^2 \sim \frac{1}{h^2}$.
- Matrix size: $n = N^3$ unknowns ($N \times N \times N$ grid in 3D).
- 7 nonzeros per row –
  - $\underline{r} = \underline{b} - A\underline{x}$: $14n = 14N^3$ ops.
  - Total cost $\approx 14N^5$.
  - Versus matrix factorization $\approx N^6$ or N^7.

Time Dependent Problems

- We'll consider two examples: diffusion (heat equation) and advection.

$$\textbf{heat equation:} \quad \frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \text{BCs and IC}$$

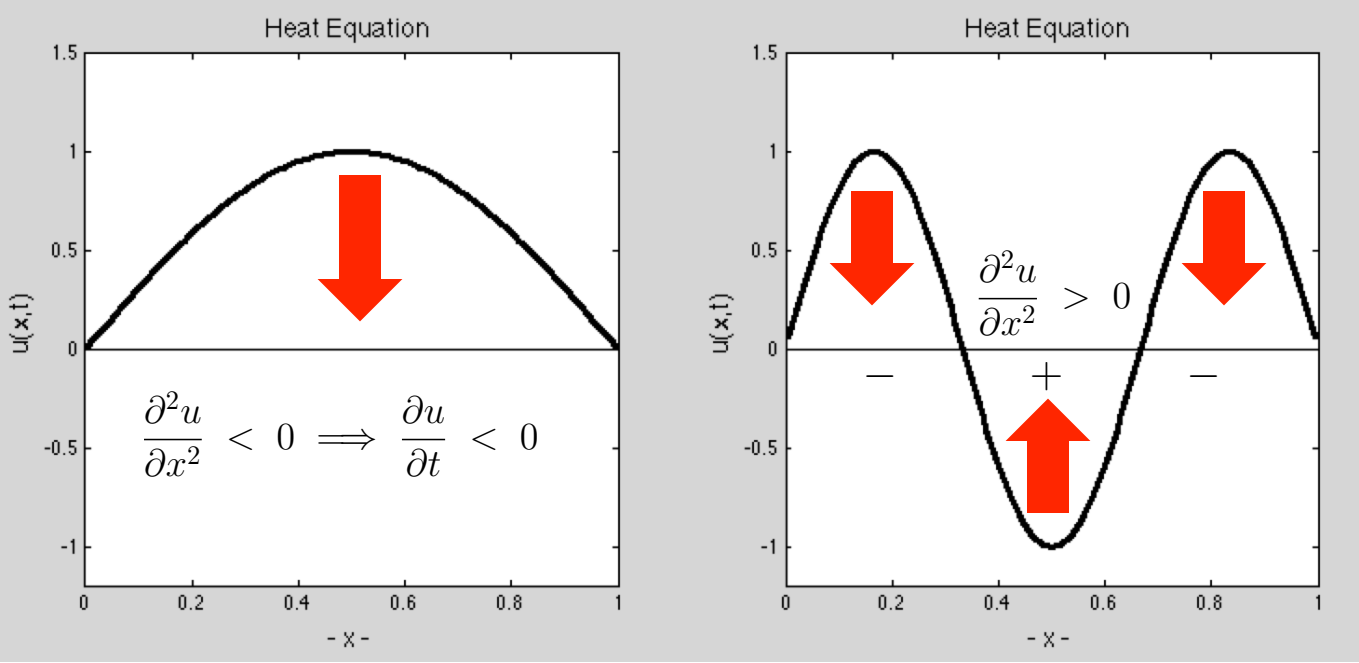
$$\textbf{advection:} \quad \frac{\partial u}{\partial t} = -c \frac{\partial u}{\partial x} + \text{BCs and IC}$$



Heat Equation:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}, \qquad \nu > 0$$

- For the heat equation, the solution evolves in the direction of local curvature.
  - If the the solution is locally concave down, u decreases there.
  - If the the solution is concave up, u increases.

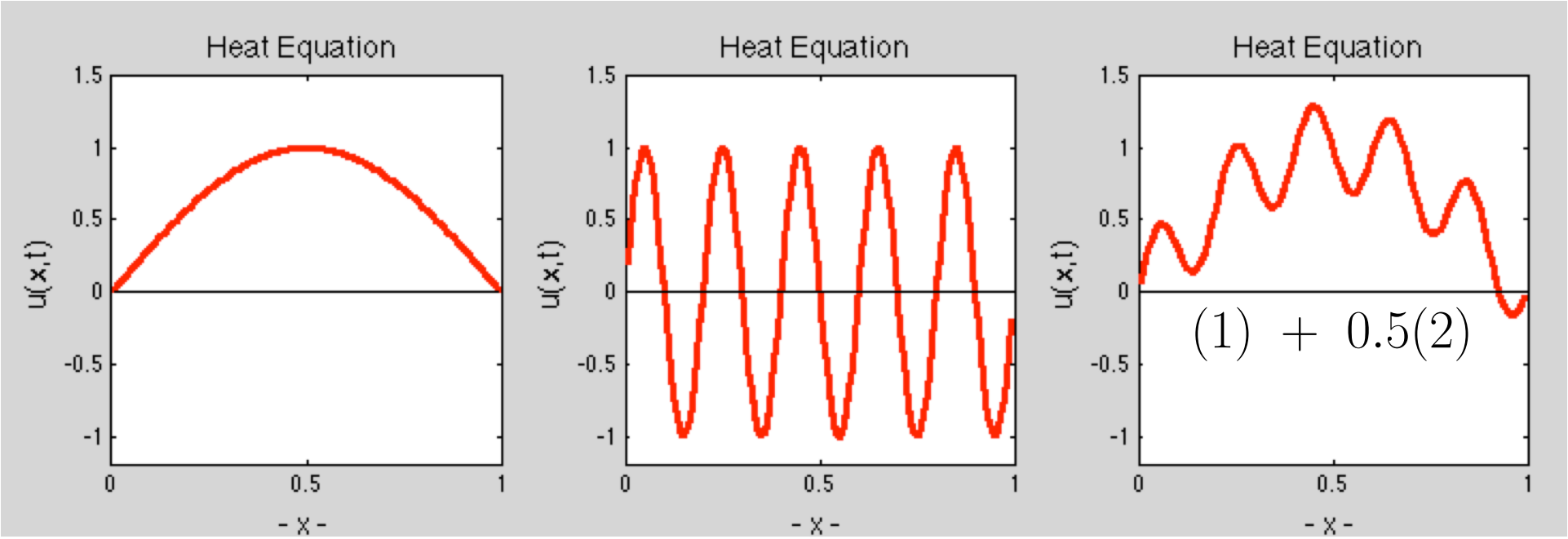


Example Solutions (eigenfunctions): $u_t = \nu u_{xx},\ u(0) = u(1) = 0$

$$
\begin{aligned}
u(x,t) &= \hat{u}(t) \sin \pi x \\
\frac{\partial u}{\partial t} &= \frac{d\hat{u}}{dt} \sin \pi x = -\nu \pi^2 \hat{u} \sin \pi x \\
\frac{d\hat{u}}{dt} &= -\nu \pi^2 \hat{u} \\
\hat{u} &= e^{-\nu \pi^2 t}\, \hat{u}(0)
\end{aligned}
$$

$$
\begin{aligned}
u(x,t) &= \hat{u}(t) \sin 10\pi x \\
\frac{\partial u}{\partial t} &= \frac{d\hat{u}}{dt} \sin \pi x = -\nu 100 \pi^2 \hat{u} \sin \pi x \\
\frac{d\hat{u}}{dt} &= -\nu 100 \pi^2 \hat{u} \\
\hat{u} &= e^{-\nu 100 \pi^2 t}\, \hat{u}(0)
\end{aligned}
$$

$\longrightarrow$ *Very rapid decay.*



Time Stepping for Diffusion Equation:

- Recall, with boundary conditions $u(0) = u(1) = 0$, the finite difference operator

$$A\mathbf{u} \;=\; -\frac{\nu}{h^2}\left[u_{j+1} \;-\; u_j \;-\; u_{j-1}\right]$$

  with $h := 1/(n+1)$ has eigenvalues in the interval $[0, M]$ with

$$M \;=\; \max_k \lambda_k \;=\; \max_k \frac{2\nu}{h^2}\left[1 \;-\; \cos k\pi h\right] \;\sim\; \frac{4}{h^2}$$

- Our ODE is $\mathbf{u}_t = -A\mathbf{u}$, so we are concerned with $-\lambda_k$.
- With Euler Forward, we require $|\lambda \Delta t| < 2$ for stability,
  - $\longrightarrow\; \Delta t < \frac{h^2}{2}$
  - *no matter how smooth the initial condition.*
- This intrinsic *stiffness* motivates the use of implicit methods for the heat equation (BDF2 is a good one).

heat1d_ef.m and heat1d_eb.m

Better Iterative Methods

- Jacobi Iteration is the simplest but certainly not the best iterative strategy.

- Can improve in two ways:
  - Choose better preconditioner, M (e.g., multigrid)
  - Compute *projection* of $\underline{x}$ onto approximation space
    - Best fit approximation.
    - Conjugate gradient (CG) iteration
    - GMRES

- Can always combine preconditioning and projection.

- We'll look briefly at preconditioned CG.

Preconditioned Projection Methods

- **Note:** Richardson Iteration does the following:

$$\begin{aligned}
\underline{x}_k &= \underline{x}_{k-1} + M^{-1}\left(\underline{b} - A\underline{x}_{k-1}\right) \\
&= \left(I - M^{-1}A\right)\underline{x}_k + M^{-1}\underline{b}
\end{aligned}$$

$$\begin{aligned}
\underline{x}_0 &= 0 \\
\underline{x}_1 &= M^{-1}\underline{b} \\
\underline{x}_2 &= \left(I - M^{-1}A\right)M^{-1}\underline{b} + M^{-1}\underline{b} && \in \quad \mathbb{P}_1(M^{-1}A)\,M^{-1}\underline{b} \\
\underline{x}_3 &= \left[\left(I - M^{-1}A\right)^2 + \left(I - M^{-1}A\right) + I\right]M^{-1}\underline{b} && \in \quad \mathbb{P}_2(M^{-1}A)\,M^{-1}\underline{b} \\
\underline{x}_k &\in K_k(M^{-1}A,\, M^{-1}\underline{b}) && := \mathbb{P}_{k-1}(M^{-1}A)\,M^{-1}\underline{b} \\
&\in \textit{Krylov subspace w.r.t. } M^{-1}A \textit{ and } M^{-1}\underline{b}.
\end{aligned}$$

- That is, $\underline{x}_k$ is a *linear combination* of the vectors in $K_k(M^{-1}A,\, M^{-1}\underline{b})$.

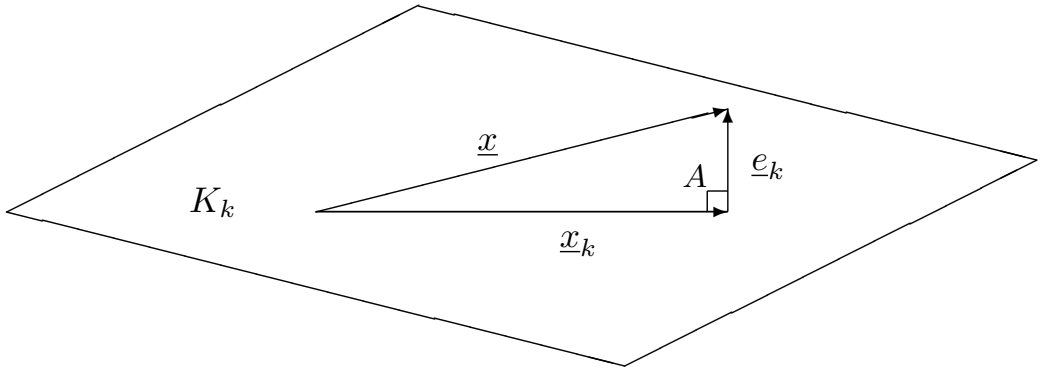
- For any matrix Q and vector $\underline{v}$, we define the Krylov subspace,

$$K_k(Q,\, \underline{v}) \;:=\; span\left\{\underline{v},\, Q\underline{v},\, Q^2\underline{v}, \ldots, Q^{k-1}\underline{v}\right\} \;\equiv\; \mathbb{P}_{k-1}(Q)\underline{v},$$

  which is the space of polynomials of degree $\leq (k-1)$ in the matrix Q times $\underline{v}$.

Conjugate Gradient Iteration

- *CG* or *PCG* seeks to find $\underline{x}_k \in K_k$ that is the closest (*best fit*) approximation to $\underline{x}$.
- The error is A-orthogonal to K_k, i.e., $\underline{x}_k$ is the *projection* of $\underline{x}$ onto K_k.



- This works if A (and M) are SPD.
- Otherwise, use (say) GMRES, which finds the best fit in $A^T A$-norm,

$$||\underline{v}|| \;:=\; \left(\underline{v}^T A^T A \underline{v}\right)^{\frac{1}{2}}.$$

One way (familiar, but not best) to compute the projection:

- Let $V_k = [\underline{v}_1\, \underline{v}_2\, \ldots\, \underline{v}_k]$ be the matrix whose columns span K_k.
- Note, the *range* of the matrix V_k, $\mathcal{R}(V_k) \equiv K_k$.
- Set $$\underline{x}_k \;=\; \sum_{j=1}^{k} \beta_j\, \underline{v}_j \;=\; V_k\, \underline{\beta}.$$
- Orthogonality condition, error $\perp_A\ \mathcal{R}(V_k)$,

$$\underline{v}_i^T\ (A\underline{x}_k \;-\; A\underline{x}) \;=\; 0, \quad i = 1, \ldots, k$$

$$V_k^T A V_k \underline{\beta} \;=\; V_k^T A \underline{x} \;=\; V_k^T \underline{b}.$$

- Define $A_k := V_k^T A V_k$ – *SPD.*

$$\left.\begin{array}{rcc} \underline{\beta} & = & A_k^{-1} V_k^T \underline{b} \\ \underline{x}_k & = & V_k \underline{\beta} \end{array}\right\} \textit{computable,}$$

$$= V_k A_k^{-1} V_k^T \underline{b}$$

$$= V_k \left(V_k^T A V_k\right)^{-1} V_k^T \underline{b}$$

$$= V_k \left(V_k^T A V_k\right)^{-1} V_k^T A \underline{x}.$$

- *Projection* of $\underline{x}$ onto $\mathcal{R}(V_k)$ in the A-norm.

- Requires:
  - k matrix vector products in $(M^{-1}A)$
  - Usually: k matvecs in A, k "solves" of $M\underline{z} = \underline{r}$.
  - Solution of *small* $k \times k$ (e.g., 30×30) linear system.
  - Recombination of basis vectors.

Benefits:

- # iters scales as $\sqrt{\kappa(M^{-1}A)}$.
- In our earlier examples with $M = I$,

$$\kappa(A) \;=\; \frac{\lambda_{\max}}{\lambda_{\min}} \;=\; \frac{4N^2}{\pi^2}.$$

- Now, # iters $\sim\ N$, *not* N^2 !
- **Cost** – 3D Poisson example (still no preconditioning): $N^4 \ll N^5$.

Conjugate Gradient Algorithm:

Solve $A\underline{x} = \underline{b}$ with $\underline{x}_0 = 0$.

- Initialize $\underline{x} = 0$, $\underline{r} = \underline{b}$, $\rho_0 = 1$ $\underline{p} = 0$.
- *for* $k = 1, \ldots,$

  Solve $M\underline{z} = \underline{r}$

$$\begin{aligned} \rho_1 &= \rho_0 \\ \rho_0 &= \underline{r}^T \underline{z} \\ \beta &= \rho_0/\rho_1 \\ \underline{p} &= \underline{z} + \beta\underline{p} \\ \underline{w} &= A\underline{p} \\ \gamma &= \underline{w}^T \underline{p} \\ \alpha &= \rho_0/\gamma \\ \underline{x} &= \underline{x} + \alpha\underline{p} \\ \underline{r} &= \underline{r} - \alpha\underline{w} \end{aligned}$$

  Stop when $||\underline{r}|| < tol.$

  end;

Time-Dependent Problems

Example: Advection Equation

- *Advection equation*

$$u_t = -c\,u_x$$

  where c is nonzero constant

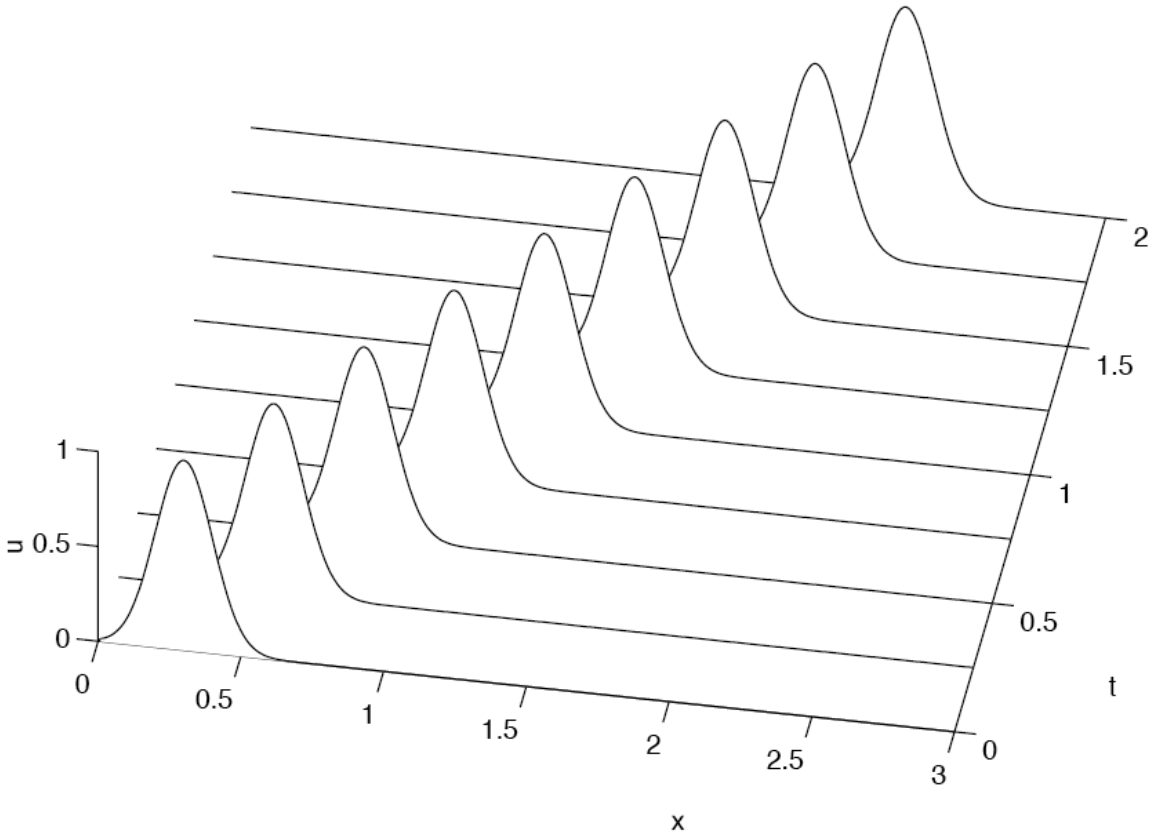
- Unique solution is determined by initial condition

$$u(0, x) = u_0(x), \qquad -\infty < x < \infty$$

  where u_0 is given function defined on $\mathbb{R}$

- We seek solution $u(t, x)$ for $t \geq 0$ and all $x \in \mathbb{R}$
- From chain rule, solution is given by $u(t, x) = u_0(x - c\,t)$
- Solution is initial function u_0 shifted by $c\,t$ to right if $c > 0$, or to left if $c < 0$

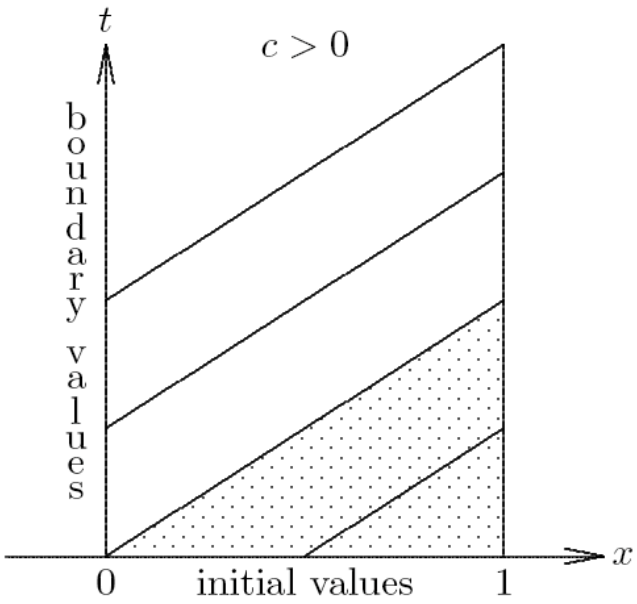
Example, continued

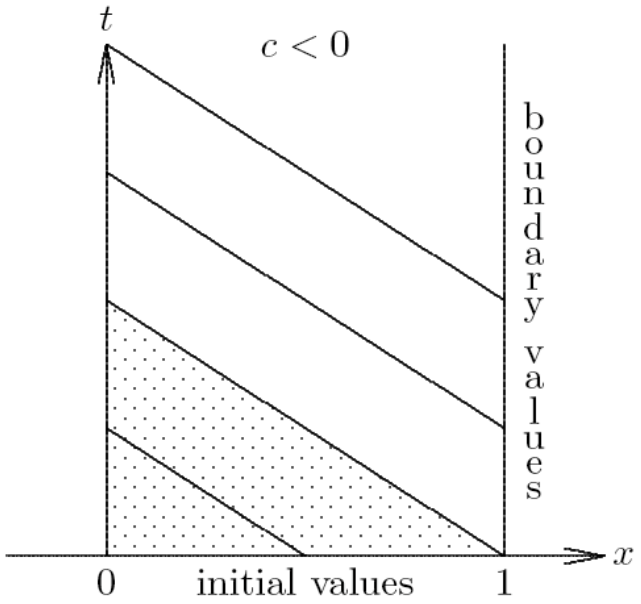


Typical solution of advection equation, with initial function "advected" (shifted) over time < interactive example >

Characteristics

- *Characteristics* for PDE are level curves of solution
- For advection equation $u_t = -c\,u_x$, characteristics are straight lines of slope $1/c$



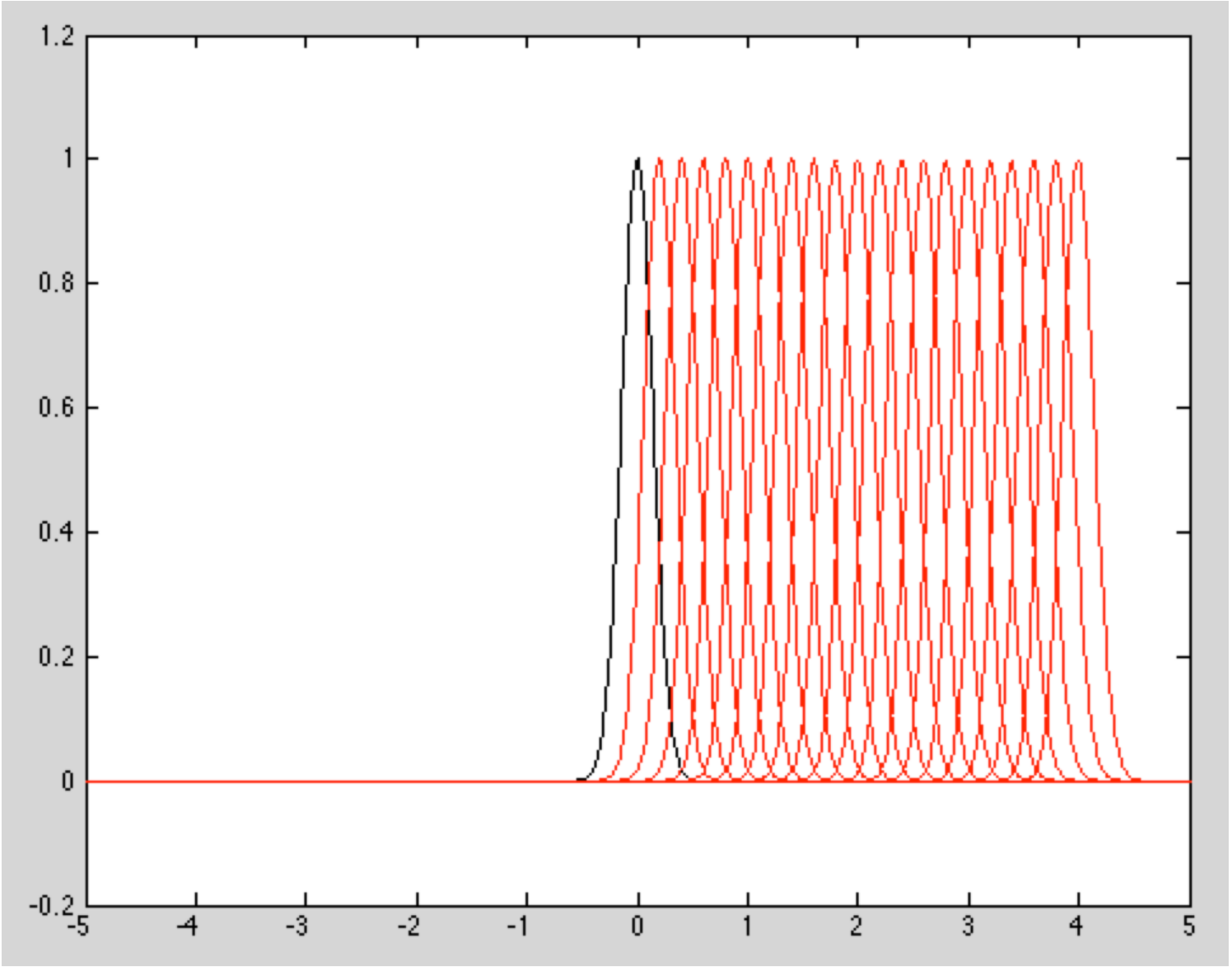


- Characteristics determine where boundary conditions can or must be imposed for problem to be well-posed

Matlab Demo: Convection

```
c=1; Tf = 4;  % Final time
x0=-5; xn=5;

dx = .01; x=x0:dx:xn; x=x'; n=length(x);

a = -1; b=0; c=1; e = ones(n,1);
C = spdiags([a*e b*e c*e],-1:1, n,n); C = C/(2*dx);

C(n,n)=-C(n,n-1); C(1,1)=C(1,2); % To drain energy at bdry

CFL = 0.50; dt  = CFL*dx/abs(c); nsteps = Tf/dt;

u=exp(-x.*x/.04); hold off; plot(x,u,'k-'); hold on;
f=0*u;f1=0*u;
io=floor(nsteps/20); kk=0; t=0;
for k=1:nsteps; t=t+dt;

    if k==1; c0=1; c1=0; c2=0;                    end;
    if k==2; c0=3/2; c1=-1/2; c2=0;               end;
    if k==3; c0=23/12; c1=-16/12; c2=5/12;        end;

    f2=f1; f1=f; f= -C*u;

    rhs = c0*f + c1*f1 + c2*f2;
    u   = u+dt*rhs;

    if mod(k,io)==0; plot(x,u,'r-'); pause(.2); end;
end;
```

Matlab Demo: Convection



Time Stepping for Advection Equation: $\frac{\partial u}{\partial t} = -c\frac{\partial u}{\partial x}$

- Unlike the diffusion equation, which smears out the initial condition (with high wavenumber components decaying particularly fast), the advection equation simply moves things around, with no decay.
- This property is evidenced by the spatial operator having purely (or close to purely) imaginary eigenvalues.
- Preserving high-wavenumber content (in space) for all time makes this problem particularly challenging.
  - There is always some spatial discretization error.
  - Its effects accumulate over time (with no decay of the error).
  - For sufficiently large final time T any fixed grid (i.e., fixed n) simulation for general problems will eventually have too much error.
  - Long time-integrations, therefore, typically require relatively fine meshes and/or high-order spatial discretizations.

CFL, Eigenvalues, and Stability: Fourier Analysis

- Consider: $u_t = -cu_x, \quad u(0) = u(1)$ (periodic BCs)
- Centered difference formula in space:

$$\frac{du_j}{dt} = -\frac{c}{2\Delta x}(u_{j+1} - u_{j-1}) = C\,\underline{u}|_j$$

$$C = -\frac{1}{2\Delta x}\underbrace{\begin{bmatrix} 0 & 1 & & & -1 \\ -1 & 0 & 1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ 1 & & & -1 & 0 \end{bmatrix}}_{\textit{Periodic Matrix}}$$

CFL, Eigenvalues, and Stability: Fourier Analysis

- Consider: $u_t = -cu_x, \quad u(0) = u(1)$ (periodic BCs)
- Centered difference formula in space:

$$\frac{du_j}{dt} = -\frac{c}{2\Delta x}(u_{j+1} - u_{j-1}) = C\,\underline{u}|_j$$

- Eigenvector: $u_j = e^{i2\pi k x_j}$.
- Eigenvalue:

$$\begin{aligned} C\,\underline{u}|_j &= -\frac{c}{2\Delta x}\left(e^{i2\pi k\Delta x} - e^{-i2\pi k\Delta x}\right) e^{i2\pi k x_j} \\ &= -\frac{2ic}{2\Delta x}\,\frac{\left(e^{i2\pi k\Delta x} - e^{-i2\pi k\Delta x}\right)}{2i}\,u_j \\ &= \lambda_k\, u_j \end{aligned}$$

$$\lambda_k = \frac{-ic}{\Delta x}\sin(2\pi k\Delta x)$$

CFL, Eigenvalues, and Stability: Fourier Analysis

- Eigenvalue:

$$
\begin{aligned}
C\,\underline{u}|_j &= -\frac{c}{2\Delta x}\left(e^{i2\pi k\Delta x} - e^{-i2\pi k\Delta x}\right) e^{i2\pi k x_j} \\
&= -\frac{2ic}{2\Delta x}\,\frac{\left(e^{i2\pi k\Delta x} - e^{-i2\pi k\Delta x}\right)}{2i}\,u_j \\
&= \lambda_k\, u_j \\
\\
\lambda_k &= \frac{-ic}{\Delta x}\sin(2\pi k\Delta x)
\end{aligned}
$$

- Eigenvalues are purely imaginary, max modulus is

$$\max_k |\lambda_k| = \frac{|c|}{\Delta x}$$

- For constant c and Δx, we define the CFL for the advection equation as

$$\mathrm{CFL} = \frac{\Delta t|c|}{\Delta x}.$$

Courant Number

Courant Number, Eigenvalues, and Stability: Fourier Analysis

- For constant c and Δx, we define the CFL for the advection equation as

$$\text{CFL} = \frac{\Delta t |c|}{\Delta x}.$$

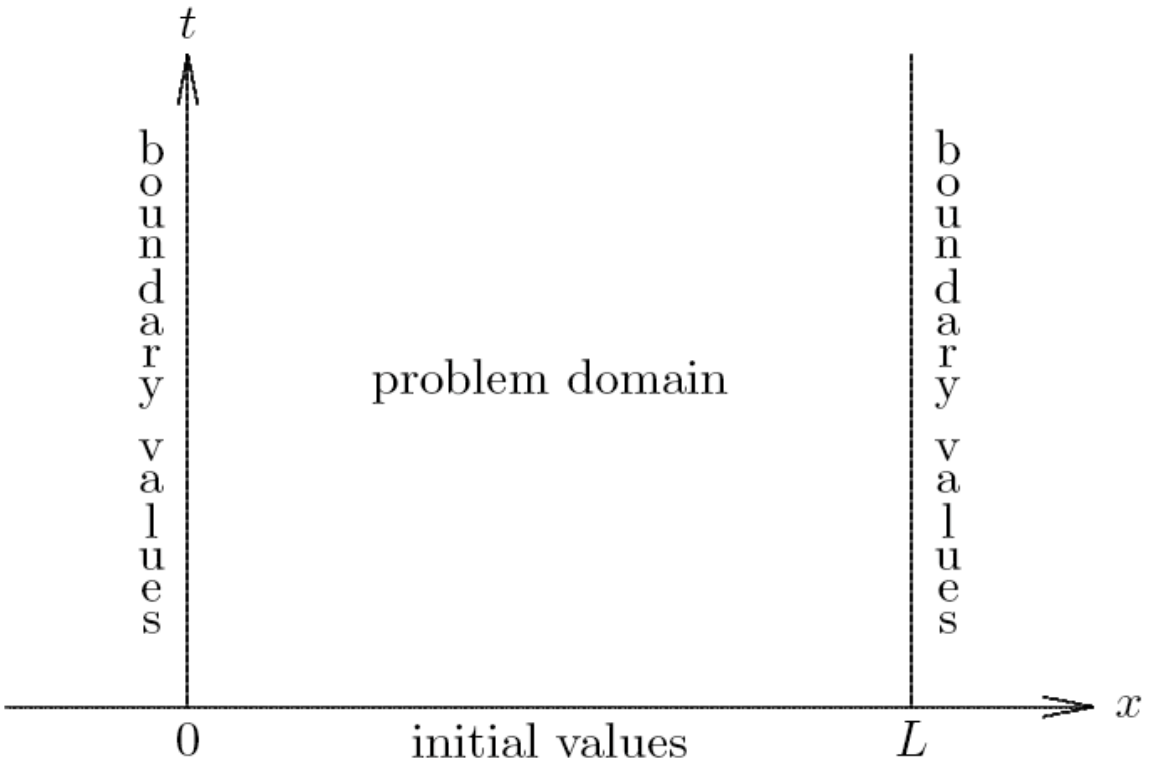
- CFL=1 would correspond to a timestep size where a particle moving at speed c would move one grid spacing in a single timestep.
- For centered finite differences in space, CFL=1 also corresponds $\lambda \Delta t = 1$.
- From our IVP stability analysis, we know that we need $|\lambda \Delta t| < .7236$ for AB3 and < 2.828 for RK4.
- This would correspond to CFL $< .7236$ and 2.828, respectively.

CFL, Eigenvalues, and Stability: Fourier Analysis

- MATLAB EXAMPLE: conv_ab3.m

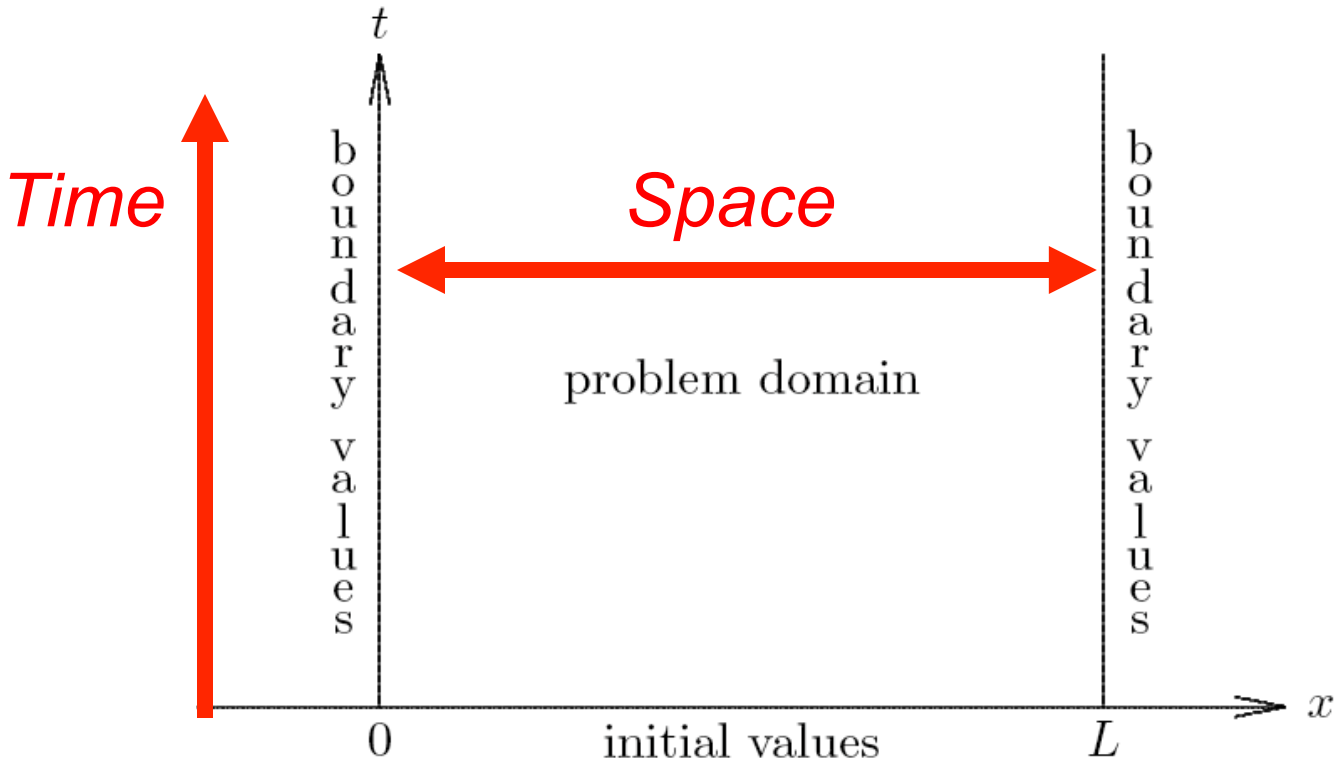
Time-Dependent Problems

- Time-dependent PDEs usually involve both initial values and boundary values



Time-Dependent Problems

- Time-dependent PDEs usually involve both initial values and boundary values



Semidiscrete Methods

- One way to solve time-dependent PDE numerically is to discretize in space but leave time variable continuous
- Result is system of ODEs that can then be solved by methods previously discussed
- For example, consider heat equation

$$u_t = c\,u_{xx}, \qquad 0 \le x \le 1, \qquad t \ge 0$$

with initial condition

$$u(0, x) = f(x), \qquad 0 \le x \le 1$$

and boundary conditions

$$u(t, 0) = 0, \qquad u(t, 1) = 0, \qquad t \ge 0$$

Semidiscrete Finite Difference Method

- Define spatial mesh points $x_i = i\Delta x$, $i = 0, \ldots, n+1$, where $\Delta x = 1/(n+1)$
- Replace derivative u_{xx} by finite difference approximation

$$u_{xx}(t, x_i) \approx \frac{u(t, x_{i+1}) - 2u(t, x_i) + u(t, x_{i-1})}{(\Delta x)^2}$$

- Result is system of ODEs

$$y_i'(t) = \frac{c}{(\Delta x)^2}\left(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)\right), \quad i = 1, \ldots, n$$

  where $y_i(t) \approx u(t, x_i)$

- From boundary conditions, $y_0(t)$ and $y_{n+1}(t)$ are identically zero, and from initial conditions, $y_i(0) = f(x_i)$, $i = 1, \ldots, n$
- We can therefore use ODE method to solve IVP for this system — this approach is called *Method of Lines*

Semidiscrete Finite Difference Method

- Define spatial mesh points $x_i = i\Delta x$, $i = 0, \ldots, n+1$, where $\Delta x = 1/(n+1)$
- Replace derivative u_{xx} by finite difference approximation

$$u_{xx}(t, x_i) \approx \frac{u(t, x_{i+1}) - 2u(t, x_i) + u(t, x_{i-1})}{(\Delta x)^2}$$

- Result is system of ODEs

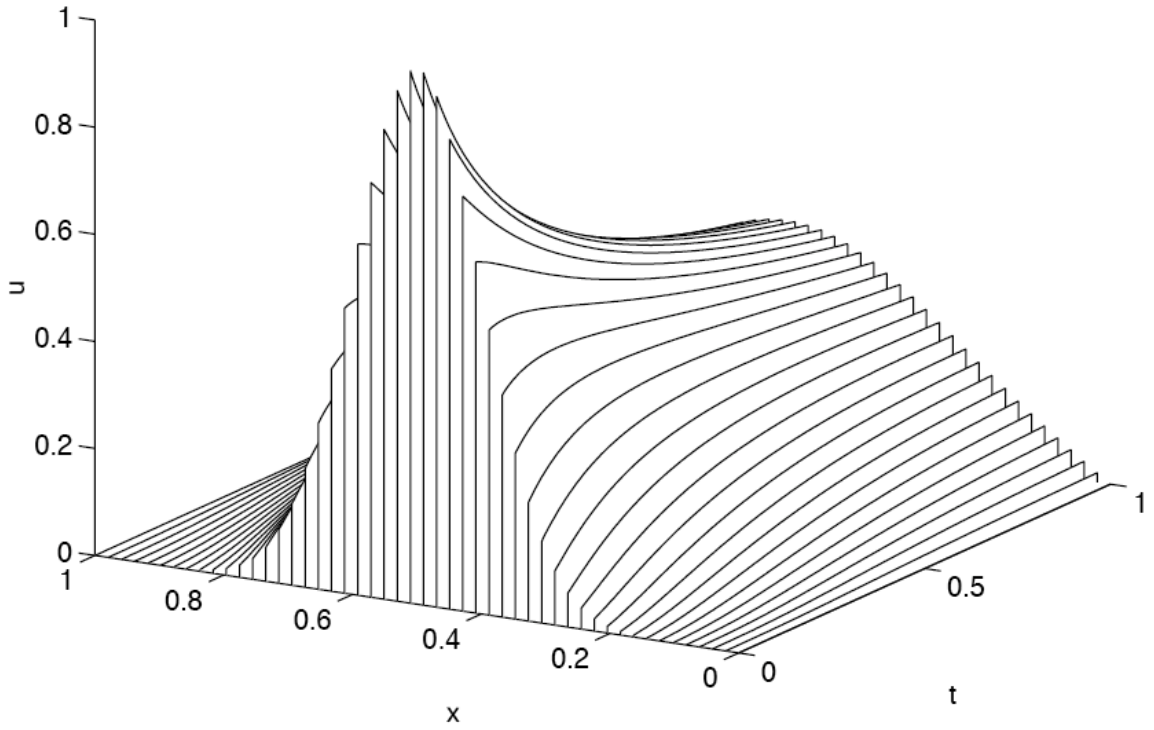
$$y_i'(t) = \frac{c}{(\Delta x)^2}\left(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)\right), \quad i = 1, \ldots, n$$

where $y_i(t) \approx u(t, x_i)$

- From boundary conditions, $y_0(t)$ and $y_{n+1}(t)$ are identically zero, and from initial conditions, $y_i(0) = f(x_i)$, $i = 1, \ldots, n$
- We can therefore use ODE method to solve IVP for this system — this approach is called *Method of Lines*

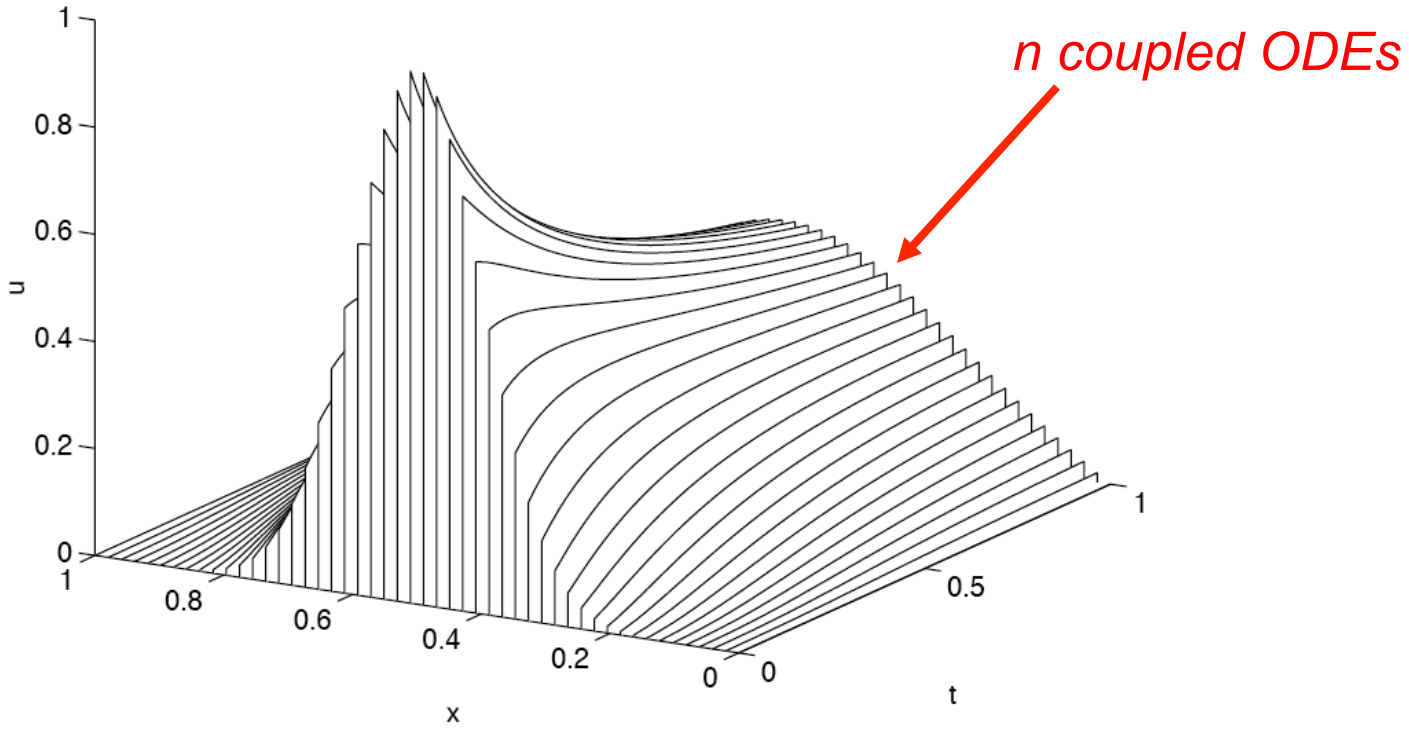
Method of Lines

- *Method of lines* uses ODE solver to compute cross-sections of solution surface over space-time plane along series of lines, each parallel to time axis and corresponding to discrete spatial mesh point



Method of Lines

- *Method of lines* uses ODE solver to compute cross-sections of solution surface over space-time plane along series of lines, each parallel to time axis and corresponding to discrete spatial mesh point



Stiffness

- Semidiscrete system of ODEs just derived can be written in matrix form

$$\boldsymbol{y}' = \frac{c}{(\Delta x)^2} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \boldsymbol{y} = \boldsymbol{A}\boldsymbol{y}$$

- Jacobian matrix $\boldsymbol{A}$ of this system has eigenvalues between $-4c/(\Delta x)^2$ and 0, which makes ODE very *stiff* as spatial mesh size Δx becomes small
- This stiffness, which is typical of ODEs derived from PDEs, must be taken into account in choosing ODE method for solving semidiscrete system

Semidiscrete Collocation Method

- Spatial discretization to convert PDE into system of ODEs can also be done by spectral or finite element approach
- Approximate solution is expressed as linear combination of basis functions, but with time dependent coefficients
- Thus, we seek solution of form

$$u(t,x) \approx v(t,x,\boldsymbol{\alpha}(t)) = \sum_{j=1}^{n} \alpha_j(t)\phi_j(x)$$

  where $\phi_j(x)$ are suitably chosen basis functions

- If we use collocation, then we substitute this approximation into PDE and require that equation be satisfied exactly at discrete set of points x_i

Semidiscrete Collocation, continued

- For heat equation, this yields system of ODEs

$$\sum_{j=1}^{n} \alpha_j'(t)\phi_j(x_i) = c\sum_{j=1}^{n} \alpha_j(t)\phi_j''(x_i)$$

  whose solution is set of coefficient functions $\alpha_i(t)$ that determine approximate solution to PDE

- Implicit form of this system is not explicit form required by standard ODE methods, so we define $n \times n$ matrices $\boldsymbol{M}$ and $\boldsymbol{N}$ by

$$m_{ij} = \phi_j(x_i), \qquad n_{ij} = \phi_j''(x_i)$$

Semidiscrete Collocation, continued

- Assuming $\boldsymbol{M}$ is nonsingular, we then obtain system of ODEs

$$\boldsymbol{\alpha}'(t) = c\,\boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{\alpha}(t)$$

  which is in form suitable for solution with standard ODE software
  - As usual, $\boldsymbol{M}$ need not be inverted explicitly, but merely used to solve linear systems
- Initial condition for ODE can be obtained by requiring solution to satisfy given initial condition for PDE at mesh points x_i
- Matrices involved in this method will be sparse if basis functions are “local,” such as B-splines

Semidiscrete Collocation, continued

- Unlike finite difference method, spectral or finite element method does not produce approximate values of solution u directly, but rather it generates representation of approximate solution as linear combination of basis functions
- Basis functions depend only on spatial variable, but coefficients of linear combination (given by solution to system of ODEs) are time dependent
- Thus, for any given time t, corresponding linear combination of basis functions generates cross section of solution *surface* parallel to spatial axis
- As with finite difference methods, systems of ODEs arising from semidiscretization of PDE by spectral or finite element methods tend to be stiff

Fully Discrete Methods

- *Fully discrete methods* for PDEs discretize in both time and space dimensions
- In fully discrete finite difference method, we
  - replace continuous domain of equation by discrete mesh of points
  - replace derivatives in PDE by finite difference approximations
  - seek numerical solution as table of approximate values at selected points in space and time

Fully Discrete Methods, continued

- In two dimensions (one space and one time), resulting approximate solution values represent points on solution *surface* over problem domain in space-time plane
- Accuracy of approximate solution depends on step sizes in both space and time
- Replacement of all partial derivatives by finite differences results in system of algebraic equations for unknown solution at discrete set of sample points
- Discrete system may be linear or nonlinear, depending on underlying PDE

Fully Discrete Methods, continued

- With initial-value problem, solution is obtained by starting with initial values along boundary of problem domain and marching forward in time step by step, generating successive rows in solution table
- Time-stepping procedure may be explicit or implicit, depending on whether formula for solution values at next time step involves only past information
- We might expect to obtain arbitrarily good accuracy by taking sufficiently small step sizes in time and space
- Time and space step sizes cannot always be chosen independently of each other, however

Example: Heat Equation

- Consider heat equation

$$u_t = c\,u_{xx}, \qquad 0 \le x \le 1, \qquad t \ge 0$$

with initial and boundary conditions

$$u(0,x) = f(x), \qquad u(t,0) = \alpha, \qquad u(t,1) = \beta$$

- Define spatial mesh points $x_i = i\Delta x$, $i = 0, 1, \ldots, n+1$, where $\Delta x = 1/(n+1)$, and temporal mesh points $t_k = k\Delta t$, for suitably chosen Δt
- Let u_i^k denote approximate solution at (t_k, x_i)

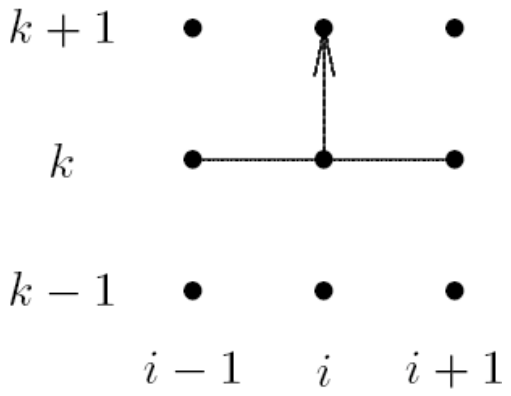
Heat Equation, continued

- Replacing u_t by forward difference in time and u_{xx} by centered difference in space, we obtain

$$\frac{u_i^{k+1} - u_i^k}{\Delta t} = c\,\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{(\Delta x)^2}, \quad \text{or}$$

$$u_i^{k+1} = u_i^k + c\,\frac{\Delta t}{(\Delta x)^2}\left(u_{i+1}^k - 2u_i^k + u_{i-1}^k\right), \quad i = 1, \ldots, n$$

- *Stencil*: pattern of mesh points involved at each level



Heat Equation, continued

- Boundary conditions give us $u_0^k = \alpha$ and $u_{n+1}^k = \beta$ for all k, and initial conditions provide starting values $u_i^0 = f(x_i)$, $i = 1, \ldots, n$
- So we can march numerical solution forward in time using this *explicit* difference scheme
- Local truncation error is $\mathcal{O}(\Delta t) + \mathcal{O}((\Delta x)^2)$, so scheme is first-order accurate in time and second-order accurate in space

< interactive example >

Stability

- Unlike Method of Lines, where time step is chosen automatically by ODE solver, user must choose time step Δt in fully discrete method, taking into account both accuracy and stability requirements
- For example, fully discrete scheme for heat equation is simply Euler's method applied to semidiscrete system of ODEs for heat equation given previously
- We saw that Jacobian matrix of semidiscrete system has eigenvalues between $-4c/(\Delta x)^2$ and 0, so stability region for Euler's method requires time step to satisfy

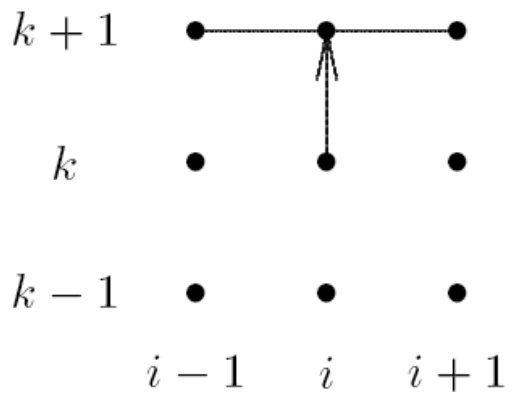
$$\Delta t \leq \frac{(\Delta x)^2}{2\,c}$$

- Severe restriction on time step can make explicit methods relatively inefficient < interactive example >

Implicit Finite Difference Methods

- For ODEs we saw that implicit methods are stable for much greater range of step sizes, and same is true of implicit methods for PDEs
- Applying *backward Euler method* to semidiscrete system for heat equation gives *implicit* finite difference scheme

$$u_i^{k+1} = u_i^k + c\,\frac{\Delta t}{(\Delta x)^2}\left(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}\right), \quad i = 1, \ldots, n$$



Implicit Finite Difference Methods, continued

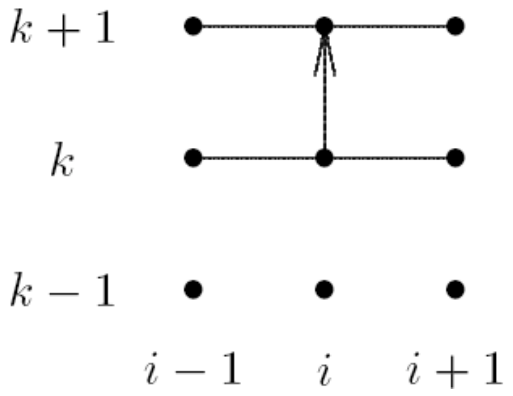
- This scheme inherits unconditional stability of backward Euler method, which means there is no stability restriction on relative sizes of Δt and Δx
- But first-order accuracy in time still severely limits time step

< interactive example >

Crank-Nicolson Method

- Applying *trapezoid method* to semidiscrete system of ODEs for heat equation yields implicit *Crank-Nicolson* method

$$u_i^{k+1} = u_i^k + c\,\frac{\Delta t}{2(\Delta x)^2}\left(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1} + u_{i+1}^k - 2u_i^k + u_{i-1}^k\right)$$



- This method is unconditionally stable and second-order accurate in time < interactive example >

Implicit Finite Difference Methods, continued

- Much greater stability of implicit finite difference methods enables them to take much larger time steps than explicit methods, but they require more work per step, since system of equations must be solved at each step
- For both backward Euler and Crank-Nicolson methods for heat equation in one space dimension, this linear system is tridiagonal, and thus both work and storage required are modest
- In higher dimensions, matrix of linear system does not have such simple form, but it is still very sparse, with nonzeros in regular pattern

Convergence

- In order for approximate solution to converge to true solution of PDE as step sizes in time and space jointly go to zero, following two conditions must hold
  - *Consistency*: local truncation error must go to zero
  - *Stability*: approximate solution at any fixed time t must remain bounded
- *Lax Equivalence Theorem* says that for well-posed linear PDE, consistency and stability are together *necessary and sufficient* for convergence
- Neither consistency nor stability alone is sufficient to guarantee convergence

Stability

- Consistency is usually fairly easy to verify using Taylor series expansion
- Analyzing stability is more challenging, and several methods are available
  - *Matrix method*, based on location of eigenvalues of matrix representation of difference scheme, as we saw with Euler's method for heat equation
  - *Fourier method*, in which complex exponential representation of solution error is substituted into difference equation and analyzed for growth or decay
  - *Domains of dependence*, in which domains of dependence of PDE and difference scheme are compared

CFL Condition

- *Domain of dependence of PDE* is portion of problem domain that influences solution at given point, which depends on characteristics of PDE
- *Domain of dependence of difference scheme* is set of all other mesh points that affect approximate solution at given mesh point
- *CFL Condition*: necessary condition for explicit finite difference scheme for hyperbolic PDE to be stable is that for each mesh point domain of dependence of PDE must lie *within* domain of dependence of finite difference scheme

- Observations:
  - Error is smooth after just

Outline

Partial Differential Equations

- *Partial differential equations* (PDEs) involve partial derivatives with respect to more than one independent variable
- Independent variables typically include one or more space dimensions and possibly time dimension as well
- More dimensions complicate problem formulation: we can have pure initial value problem, pure boundary value problem, or mixture of both
- Equation and boundary data may be defined over irregular domain

Partial Differential Equations, continued

- For simplicity, we will deal only with single PDEs (as opposed to systems of several PDEs) with only two independent variables, either
  - two space variables, denoted by x and y, or
  - one space variable denoted by x and one time variable denoted by t
- Partial derivatives with respect to independent variables are denoted by subscripts, for example
  - $u_t = \partial u / \partial t$
  - $u_{xy} = \partial^2 u / \partial x \partial y$

Classification of PDEs

- *Order* of PDE is order of highest-order partial derivative appearing in equation
- For example, advection equation is first order
- Important second-order PDEs include
  - *Heat equation*: $u_t = u_{xx}$
  - *Wave equation*: $u_{tt} = u_{xx}$
  - *Laplace equation*: $u_{xx} + u_{yy} = 0$

Classification of PDEs, continued

- Second-order linear PDEs of general form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

  are classified by value of *discriminant* $b^2 - 4ac$

- $b^2 - 4ac > 0$: *hyperbolic* (e.g., wave equation)
- $b^2 - 4ac = 0$: *parabolic* (e.g., heat equation)
- $b^2 - 4ac < 0$: *elliptic* (e.g., Laplace equation)

Classification of PDEs, continued

Classification of more general PDEs is not so clean and simple, but roughly speaking

- *Hyperbolic* PDEs describe time-dependent, conservative physical processes, such as convection, that *are not* evolving toward steady state
- *Parabolic* PDEs describe time-dependent, dissipative physical processes, such as diffusion, that *are* evolving toward steady state
- *Elliptic* PDEs describe processes that have already reached steady state, and hence are time-independent

Time-Independent Problems

- We next consider time-independent, elliptic PDEs in two space dimensions, such as *Helmholtz equation*

$$u_{xx} + u_{yy} + \lambda u = f(x, y)$$

- Important special cases
  - *Poisson equation*: $\lambda = 0$
  - *Laplace equation*: $\lambda = 0$ and $f = 0$
- For simplicity, we will consider this equation on unit square
- Numerous possibilities for boundary conditions specified along each side of square
  - *Dirichlet*: u is specified
  - *Neumann*: u_x or u_y is specified
  - *Mixed*: combinations of these are specified

Finite Difference Methods

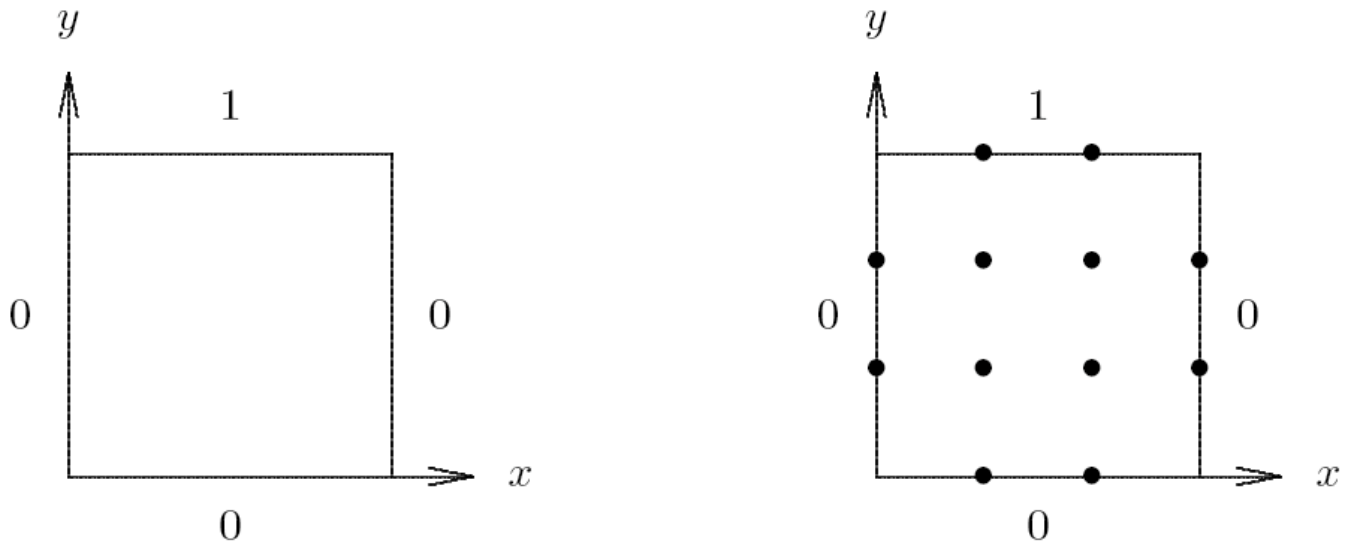
- Finite difference methods for such problems proceed as before
  - Define discrete mesh of points within domain of equation
  - Replace derivatives in PDE by finite difference approximations
  - Seek numerical solution at mesh points
- Unlike time-dependent problems, solution is not produced by marching forward step by step in time
- Approximate solution is determined at all mesh points simultaneously by solving single system of algebraic equations

Example: Laplace Equation

- Consider Laplace equation

$$u_{xx} + u_{yy} = 0$$

on unit square with boundary conditions shown below left



- Define discrete mesh in domain, including boundaries, as shown above right

Laplace Equation, continued

- Interior grid points where we will compute approximate solution are given by

$$(x_i, y_j) = (ih, jh), \qquad i, j = 1, \ldots, n$$

where in example $n = 2$ and $h = 1/(n+1) = 1/3$

- Next we replace derivatives by centered difference approximation at each interior mesh point to obtain finite difference equation

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0$$

where $u_{i,j}$ is approximation to true solution $u(x_i, y_j)$ for $i, j = 1, \ldots, n$, and represents one of given boundary values if i or j is 0 or $n+1$

Laplace Equation, continued

- Simplifying and writing out resulting four equations explicitly gives

$$4u_{1,1} - u_{0,1} - u_{2,1} - u_{1,0} - u_{1,2} = 0$$

$$4u_{2,1} - u_{1,1} - u_{3,1} - u_{2,0} - u_{2,2} = 0$$

$$4u_{1,2} - u_{0,2} - u_{2,2} - u_{1,1} - u_{1,3} = 0$$

$$4u_{2,2} - u_{1,2} - u_{3,2} - u_{2,1} - u_{2,3} = 0$$

Laplace Equation, continued

- Writing previous equations in matrix form gives

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \boldsymbol{b}$$

- System of equations can be solved for unknowns $u_{i,j}$ either by direct method based on factorization or by iterative method, yielding solution

$$\boldsymbol{x} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} 0.125 \\ 0.125 \\ 0.375 \\ 0.375 \end{bmatrix}$$

Laplace Equation, continued

- In practical problem, mesh size h would be much smaller, and resulting linear system would be much larger
- Matrix would be very sparse, however, since each equation would still involve only five variables, thereby saving substantially on work and storage

< interactive example >

Degrees of Freedom for 2D Poisson Equation

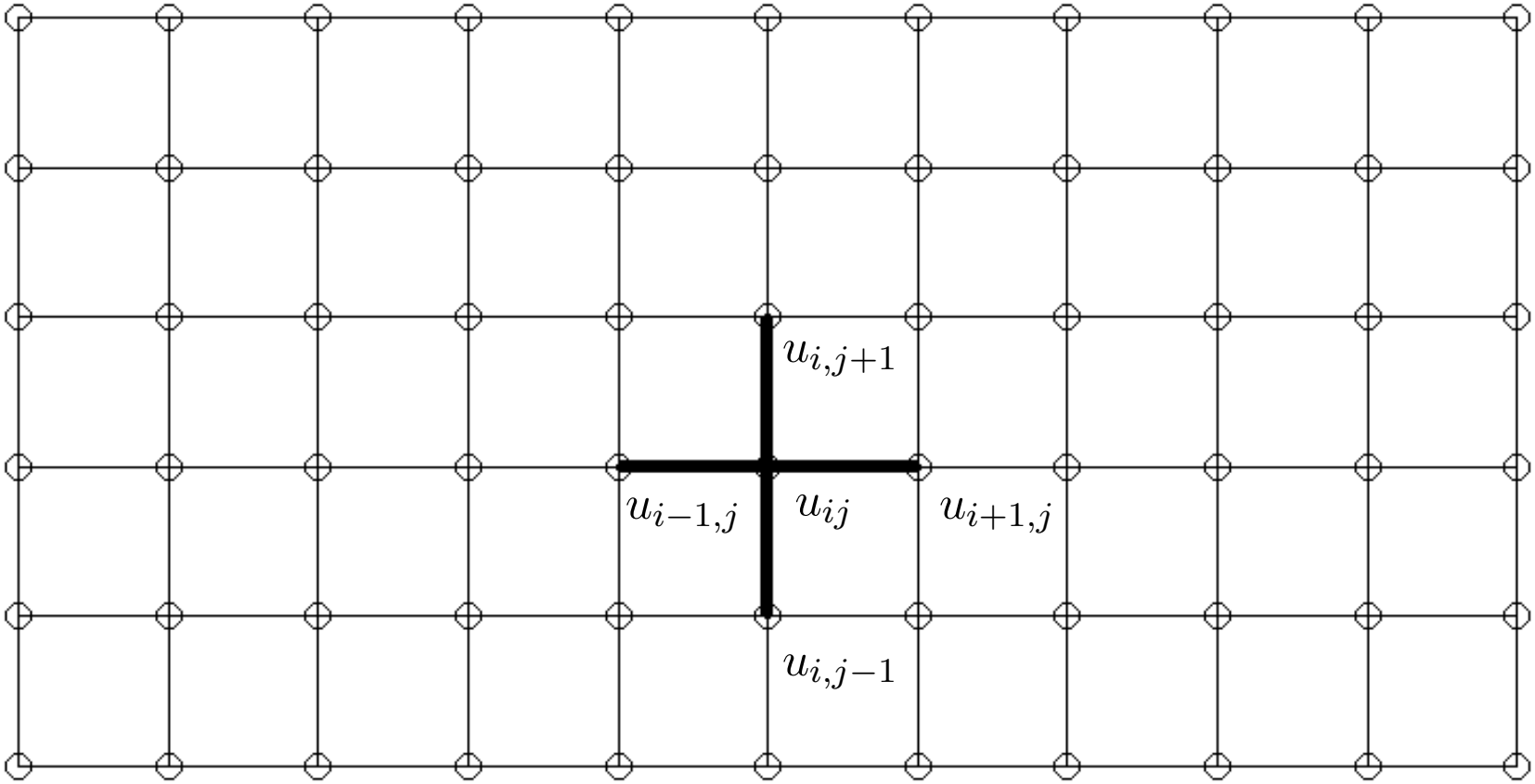
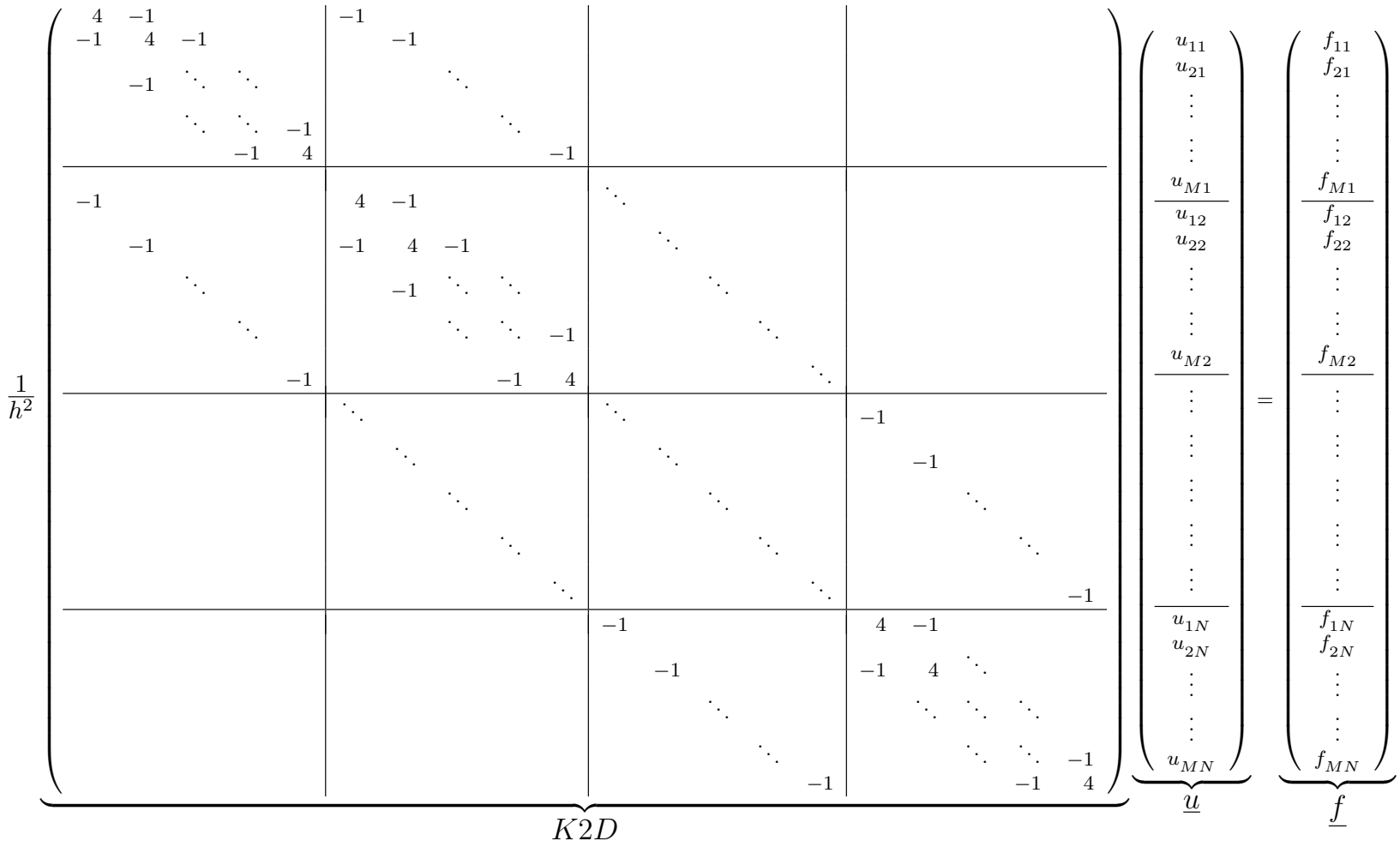


Figure 1: Grid notation for 2D problem showing 5-point finite difference stencil.

- It is convenient to represent the unknowns as a vector of the form $\underline{u} = (u_{11}\ u_{21}\ u_{31}\ \ldots\ u_{ij}\ \ldots\ u_{mn})^T$

System Matrix for 2D Poisson Problem



$$-\left(\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2}+\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{\Delta y^2}\right) \quad = \quad f_{ij}$$

System Matrix for 2D Poisson Problem

$$
K2D \quad = \quad \frac{1}{h^2}
\left(\begin{array}{cccc|cccc|cccc|cccc}
2 & -1 & & & & & & & & & & & & & & \\
-1 & 2 & -1 & & & & & & & & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & & & & & & & & \\
 & & -1 & 2 & & & & & & & & & & & & \\
\hline
 & & & & 2 & -1 & & & & & & & & & & \\
 & & & & -1 & 2 & -1 & & & & & & & & & \\
 & & & & & \ddots & \ddots & \ddots & & & & & & & & \\
 & & & & & & -1 & 2 & & & & & & & & \\
\hline
 & & & & & & & & \ddots & & & & & & & \\
 & & & & & & & & & \ddots & & & & & & \\
 & & & & & & & & & & \ddots & & & & & \\
 & & & & & & & & & & & \ddots & & & & \\
\hline
 & & & & & & & & & & & & 2 & -1 & & \\
 & & & & & & & & & & & & -1 & 2 & \ddots & \\
 & & & & & & & & & & & & & \ddots & \ddots & -1 \\
 & & & & & & & & & & & & & & -1 & 2
\end{array}\right)
\left(\begin{array}{c}
u_{11} \\ u_{21} \\ \vdots \\ \vdots \\ \vdots \\ u_{N1} \\ \hline
u_{12} \\ u_{22} \\ \vdots \\ \vdots \\ \vdots \\ u_{N2} \\ \hline
\vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \hline
u_{1N} \\ u_{2N} \\ \vdots \\ \vdots \\ \vdots \\ u_{NN}
\end{array}\right)
$$

System Matrix for 2D Poisson Problem

$$+ \quad \frac{1}{h^2} \left(\begin{array}{c|c|c|c} \begin{array}{ccccc} 2 & & & & \\ & 2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & 2 \end{array} & \begin{array}{ccccc} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & -1 \end{array} & & \\ \hline \begin{array}{ccccc} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & -1 \end{array} & \begin{array}{ccccc} 2 & & & & \\ & 2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & 2 \end{array} & \begin{array}{ccccc} \ddots & & & & \\ & \ddots & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \ddots \end{array} & \\ \hline & \begin{array}{ccccc} \ddots & & & & \\ & \ddots & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \ddots \end{array} & \begin{array}{ccccc} \ddots & & & & \\ & \ddots & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \ddots \end{array} & \begin{array}{ccccc} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & -1 \end{array} \\ \hline & & \begin{array}{ccccc} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & -1 \end{array} & \begin{array}{ccccc} 2 & & & & \\ & 2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & 2 \end{array} \end{array} \right) \begin{pmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{N1} \\ \hline u_{12} \\ u_{22} \\ \vdots \\ u_{N2} \\ \hline \vdots \\ \hline u_{1N} \\ u_{2N} \\ \vdots \\ u_{NN} \end{pmatrix}$$

System Matrix for 2D Poisson Problem

$$K2D = \begin{pmatrix} K_x & & & \\ & K_x & & \\ & & \ddots & \\ & & & K_x \end{pmatrix} + \frac{1}{h^2} \begin{pmatrix} 2I_x & -I_x & & \\ -I_x & 2I_x & \ddots & \\ & \ddots & \ddots & -I_x \\ & & -I_x & 2I_x \end{pmatrix}$$

$$-\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} \right) = f_{ij}$$

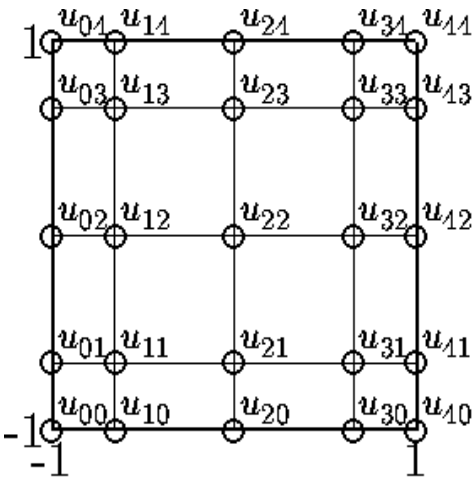
Finite Element Methods

- Finite element methods are also applicable to boundary value problems for PDEs
- Conceptually, there is no change in going from one dimension to two or three dimensions
  - Solution is represented as linear combination of basis functions
  - Some criterion (e.g., Galerkin) is applied to derive system of equations that determines coefficients of linear combination
- Main practical difference is that instead of subintervals in one dimension, elements usually become triangles or rectangles in two dimensions, or tetrahedra or hexahedra in three dimensions

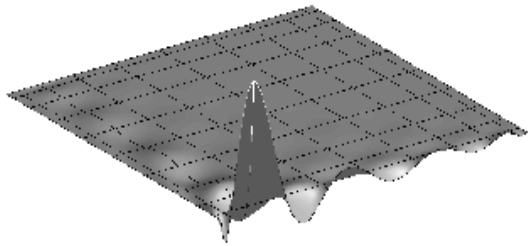
Finite Element Methods, continued

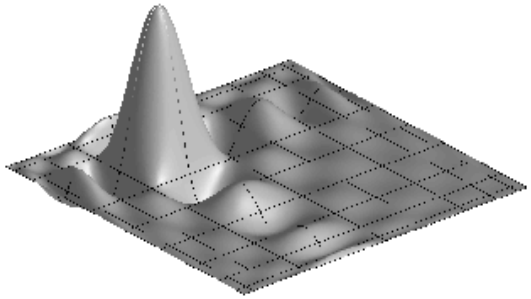
- Basis functions typically used are bilinear or bicubic functions in two dimensions or trilinear or tricubic functions in three dimensions, analogous to piecewise linear “hat” functions or piecewise cubics in one dimension
- Increase in dimensionality means that linear system to be solved is much larger, but it is still sparse due to local support of basis functions
- Finite element methods for PDEs are extremely flexible and powerful, but detailed treatment of them is beyond scope of this course

Example of high-order finite element (one element)

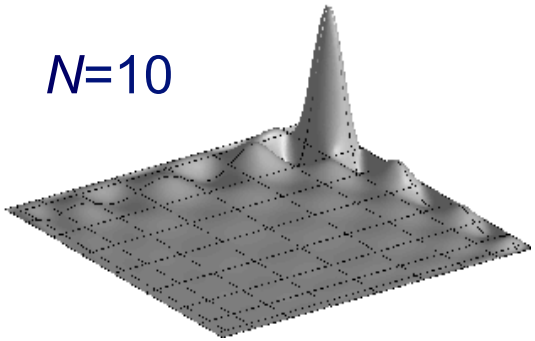


N=4





N=10



Sparse Linear Systems

- Boundary value problems and implicit methods for time-dependent PDEs yield systems of linear algebraic equations to solve
- Finite difference schemes involving only a few variables each, or localized basis functions in finite element approach, cause linear system to be *sparse*, with relatively few nonzero entries
- Sparsity can be exploited to use much less than $\mathcal{O}(n^2)$ storage and $\mathcal{O}(n^3)$ work required in standard approach to solving system with dense matrix

Sparse Factorization Methods

- Gaussian elimination and Cholesky factorization are applicable to large sparse systems, but care is required to achieve reasonable efficiency in solution time and storage requirements
- Key to efficiency is to store and operate on only nonzero entries of matrix
- Special data structures are required instead of simple 2-D arrays for storing dense matrices

Band Systems

- For 1-D problems, equations and unknowns can usually be ordered so that nonzeros are concentrated in narrow band, which can be stored efficiently in rectangular 2-D array by diagonals
- Bandwidth can often be reduced by reordering rows and columns of matrix
- For problems in two or more dimensions, even narrowest possible band often contains mostly zeros, so 2-D array storage is wasteful

General Sparse Data Structures

- In general, sparse systems require data structures that store only nonzero entries, along with indices to identify their locations in matrix
- Explicitly storing indices incurs additional storage overhead and makes arithmetic operations on nonzeros less efficient due to indirect addressing to access operands
- Data structure is worthwhile only if matrix is sufficiently sparse, which is often true for very large problems arising from PDEs and many other applications

Fill

- When applying LU or Cholesky factorization to sparse matrix, taking linear combinations of rows or columns to annihilate unwanted nonzero entries can introduce new nonzeros into matrix locations that were initially zero
- Such new nonzeros, called *fill*, must be stored and may themselves eventually need to be annihilated in order to obtain triangular factors
- Resulting triangular factors can be expected to contain at least as many nonzeros as original matrix and usually significant fill as well

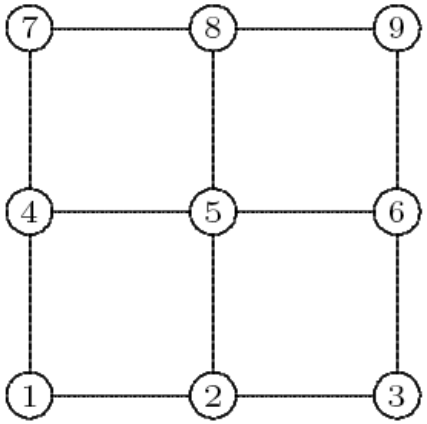
Reordering to Limit Fill

- Amount of fill is sensitive to order in which rows and columns of matrix are processed, so basic problem in sparse factorization is reordering matrix to limit fill during factorization
- Exact minimization of fill is hard combinatorial problem (NP-complete), but heuristic algorithms such as minimum degree and nested dissection limit fill well for many types of problems

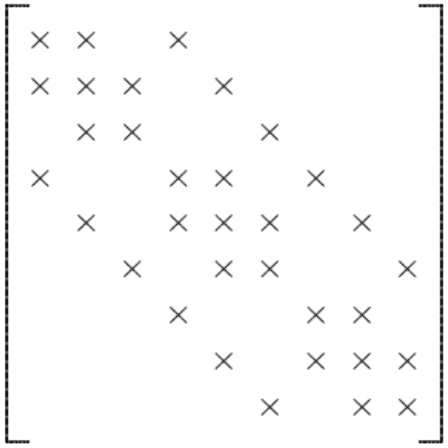
Example: Laplace Equation

- Discretization of Laplace equation on square by second-order finite difference approximation to second derivatives yields system of linear equations whose unknowns correspond to mesh points (nodes) in square grid
- Two nodes appearing in same equation of system are *neighbors* connected by *edge* in mesh or *graph*
- Diagonal entries of matrix correspond to nodes in graph, and nonzero off-diagonal entries correspond to edges in graph: $a_{ij} \neq 0 \Leftrightarrow$ nodes i and j are neighbors

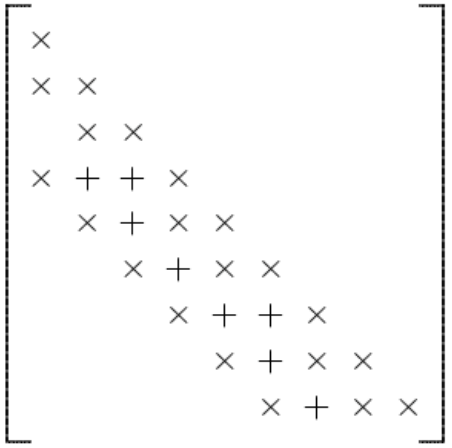
Natural, Row-Wise Ordering



mesh



A



L

- With nodes numbered row-wise (or column-wise), matrix is block tridiagonal, with each nonzero block either tridiagonal or diagonal
- Matrix is banded but has many zero entries inside band
- Cholesky factorization fills in band almost completely

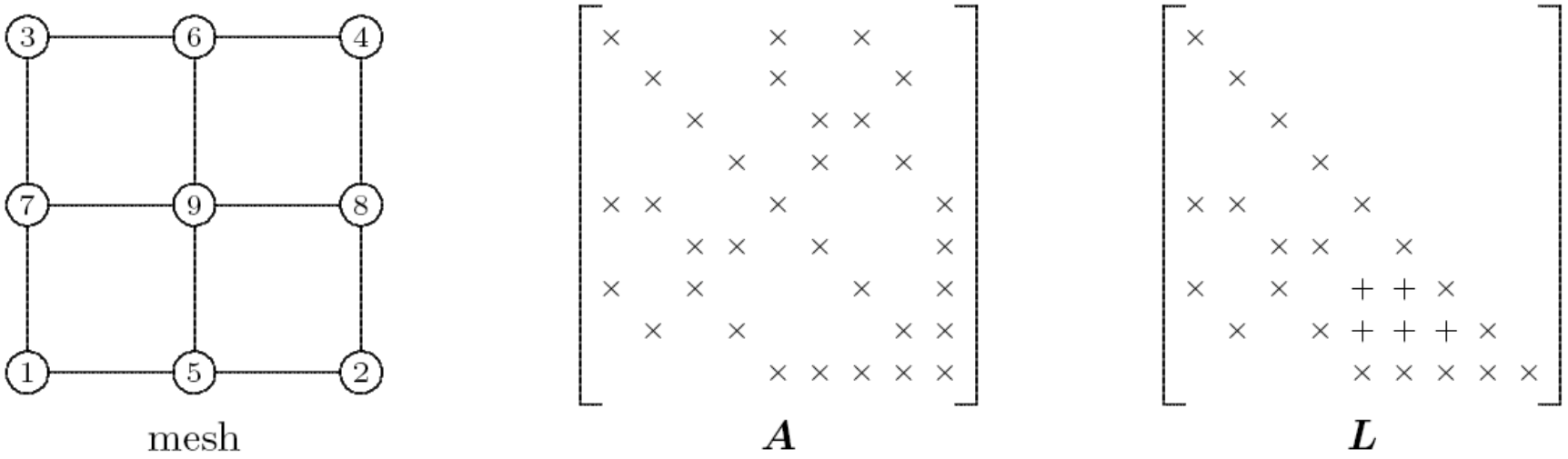
Graph Model of Elimination

- Each step of factorization process corresponds to elimination of one node from graph
- Eliminating node causes its neighboring nodes to become connected to each other
- If any such neighbors were not already connected, then *fill* results (new edges in graph and new nonzeros in matrix)

Minimum Degree Ordering

- Good heuristic for limiting fill is to eliminate first those nodes having fewest neighbors
- Number of neighbors is called *degree* of node, so heuristic is known as *minimum degree*
- At each step, select node of smallest degree for elimination, breaking ties arbitrarily
- After node has been eliminated, its neighbors become connected to each other, so degrees of some nodes may change
- Process is then repeated, with new node of minimum degree eliminated next, and so on until all nodes have been eliminated

Minimum Degree Ordering, continued

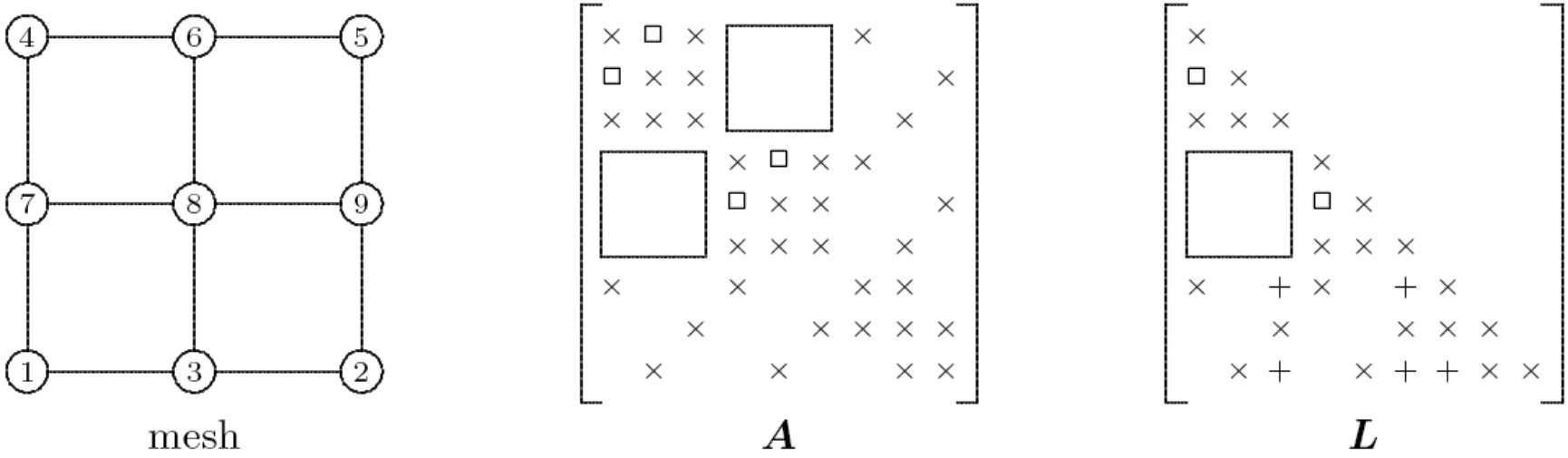


- Cholesky factor suffers much less fill than with original ordering, and advantage grows with problem size
- Sophisticated versions of minimum degree are among most effective general-purpose orderings known

Nested Dissection Ordering

- *Nested dissection* is based on divide-and-conquer
- First, small set of nodes is selected whose removal splits graph into two pieces of roughly equal size
- No node in either piece is connected to any node in other, so no fill occurs in either piece due to elimination of any node in the other
- *Separator* nodes are numbered *last*, then process is repeated recursively on each remaining piece of graph until all nodes have been numbered

Nested Dissection Ordering, continued



- Dissection induces blocks of zeros in matrix that are automatically preserved during factorization
- Recursive nature of algorithm can be seen in hierarchical block structure of matrix, which would involve many more levels in larger problems
- Again, Cholesky factor suffers much less fill than with original ordering, and advantage grows with problem size

Sparse Factorization Methods

- Sparse factorization methods are accurate, reliable, and robust
- They are methods of choice for 1-D problems and are usually competitive for 2-D problems, but they can be prohibitively expensive in both work and storage for very large 3-D problems
- Iterative methods often provide viable alternative in these cases

The qualitative difference between 2D and 3D led to a burst of research in iterative methods in the 80s, when computers finally became powerful enough to make simulation of 3D problems tractable.

Developments were prominent in multigrid and Krylov subspace projection methods.

Fast Direct Methods

- For certain elliptic boundary value problems, such as Poisson equation on rectangular domain, fast Fourier transform can be used to compute solution to discrete system very efficiently
- For problem with n mesh points, solution can be computed in $\mathcal{O}(n \log n)$ operations
- This technique is basis for several "fast Poisson solver" software packages
- *Cyclic reduction* can achieve similar efficiency, and is somewhat more general
- *FACR* method combines Fourier analysis and cyclic reduction to produce even faster method with $\mathcal{O}(n \log \log n)$ complexity

Iterative Methods for Linear Systems

- Iterative methods for solving linear systems begin with initial guess for solution and successively improve it until solution is as accurate as desired
- In theory, infinite number of iterations might be required to converge to exact solution
- In practice, iteration terminates when residual $\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|$, or some other measure of error, is as small as desired

Stationary Iterative Methods

- Simplest type of iterative method for solving $\boldsymbol{Ax} = \boldsymbol{b}$ has form

$$\boldsymbol{x}_{k+1} = \boldsymbol{G}\boldsymbol{x}_k + \boldsymbol{c}$$

  where matrix $\boldsymbol{G}$ and vector $\boldsymbol{c}$ are chosen so that *fixed point* of function $\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{G}\boldsymbol{x} + \boldsymbol{c}$ is solution to $\boldsymbol{Ax} = \boldsymbol{b}$

- Method is *stationary* if $\boldsymbol{G}$ and $\boldsymbol{c}$ are fixed over all iterations

- $\boldsymbol{G}$ is Jacobian matrix of fixed-point function $\boldsymbol{g}$, so stationary iterative method converges if

$$\rho(\boldsymbol{G}) < 1$$

  and smaller spectral radius yields faster convergence

Iterative Solvers - Linear Elliptic Problems

- PDEs give rise to large sparse linear systems of the form

$$A\underline{x} = \underline{b} \quad \text{or} \quad A\underline{u} = \underline{f}.$$

Here, we'll take A to be the (SPD) matrix arising from finite differences applied to the Poisson equation

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y) \qquad x, y \in [0,1]^2, \qquad u = 0 \text{ on } \partial\Omega$$

$$-\left(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}\right)_{ij} \approx f|_{ij}\,,$$

- Assuming uniform spacing in x and y we have

$$\frac{\delta^2 u}{\delta x^2} := \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} \quad \text{and} \quad \frac{\delta^2 u}{\delta y^2} := \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h^2}$$

- Assuming uniform spacing in x and y we have

$$\frac{\delta^2 u}{\delta x^2} := \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} \quad \text{and} \quad \frac{\delta^2 u}{\delta y^2} := \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h^2}$$

- Our finite difference formula is thus,

$$\frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} - 4u_{ij} + u_{i,j+1} + u_{i,j-1}\right) = f_{ij}.$$

- Rearranging, we can solve for u_{ij}:

$$\frac{4}{h^2}u_{ij} = f_{ij} + \frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

$$u_{ij} = \frac{h^2}{4}f_{ij} + +\frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

$$u_{ij} \;=\; \frac{h^2}{4} f_{ij} \;+\; + \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

- Jacobi iteration amounts to using the expression above as a fixed-point iteration scheme:

$$u_{ij}^{k+1} \;=\; \frac{h^2}{4} f_{ij} \;+\; + \frac{1}{4}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}\right)$$

$$= \; \frac{h^2}{4} f_{ij} \;+\; \text{average of current neighbor values}$$

$$u_{ij}^{k+1} \;=\; \frac{h^2}{4} f_{ij} \;+\; + \frac{1}{4}\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)$$

- Note that this is analogous to

$$u_{ij}^{k+1} \;=\; u_{ij}^k + \frac{h^2}{4}\left[f_{ij} + \frac{1}{h^2}\left(u_{i+1,j}^k + u_{i-1,j}^k - 4u_{ij}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)\right]$$

$$\underline{u}^{k+1} \;=\; \underline{u}^k \;+\; \Delta t\left(\underline{f} - A\underline{u}^k\right), \quad \Delta t := \frac{h^2}{4},$$

  which is Euler forward applied to $\underline{u}' = \underline{f} - A\underline{u}$.

- We note that we have stability if $|\lambda \Delta t| < 2$ (because this system, $-A$, has real negative eigenvalues).

- Recall that the eigenvalues for the 1D diffusion operator are

$$\lambda_i \;=\; \frac{2}{h^2}\left(1-\cos i\pi\Delta x\right) \le \frac{4}{h^2} = 4(n+1)^2$$

- In 2D, we pick up contributions from both $\frac{\delta^2 u}{\delta x^2}$ and $\frac{\delta^2 u}{\delta y^2}$, so

$$\max|\lambda| \;<\; \sim \frac{8}{h^2}$$

  and we have stability since

$$\max|\lambda\Delta t| \;<\; \frac{8}{h^2}\frac{h^2}{4} = 2$$

- So, Jacobi iteration is equivalent to solving $A\underline{u} = \underline{f}$ by time marching $\underline{u}' = \underline{f} - A\underline{u}$ using EF

$$\underline{u}^{k+1} \;=\; \underline{u}^k \;+\; \Delta t\left(\underline{f} - A\underline{u}^k\right)$$

  with maximum allowable timestep size

$$\Delta t \;=\; \frac{h^2}{4}$$

Jacobi Iteration in Matrix Form

- Jacobi iteration has the matrix form

$$\underline{u}^{k+1} \;=\; \underline{u}^{k} \,+\, \Delta t \left(\underline{f} - A\underline{u}^{k}\right)$$

- More generally, we have Richardson iteration

$$\underline{u}^{k+1} \;=\; \underline{u}^{k} \,+\, \sigma D \left(\underline{f} - A\underline{u}^{k}\right)$$

- If $\sigma = 1$ and D^{-1} =diag(A) $[d_{ii} = 1/a_{ii},\, d_{ij} = 0,\, i \neq j]$, we have standard Jacobi iteration.
- If $\sigma < 1$ we have *damped Jacobi*.
- D is generally known as a smoother or a preconditioner, depending on context.

Rate of Convergence for Jacobi Iteration

- Let $\underline{e}^k := \underline{u} - \underline{u}^k$.
- Since $A\underline{u} = \underline{f}$, we have

$$\begin{aligned}
\underline{u}^{k+1} &= \underline{u}^k + \Delta t \left(A\underline{u} - A\underline{u}^k \right) \\
-\underline{u} &= -\underline{u} \\
--- &- ---------- \\
-\underline{e}^{k+1} &= -\underline{e}^k - \sigma \Delta t A \underline{e}^k \\
-\underline{e}^{k+1} &= -\left(I - \sigma \Delta t A \right) \underline{e}^k \\
\underline{e}^k &= \left(I - \sigma \Delta t A \right)^k \underline{e}^0 \\
&= \left(I - \sigma \Delta t A \right)^k \underline{u} \qquad \text{if } \underline{u}^0 = 0.
\end{aligned}$$

- If $\sigma < 1$, then the high wavenumber error components will decay because $\lambda \Delta t$ will be well within the stability region for EF.
- The low-wavenumber components of the solution (and error) evolve like $e^{-\lambda t}$, because these will be well-resolved in time by Euler forward.
- Thus, we can anticipate

$$||\underline{e}^k|| \;\approx\; ||\underline{u}||e^{-\lambda_{\min}\sigma\Delta t k}$$

  with $\lambda_{\min} \approx 2\pi^2$ (for 2D).
- If $\sigma \approx 1$, we have

$$||\underline{e}^k|| \;\approx\; ||\underline{u}||e^{-2\pi^2 t} \;=\; ||\underline{u}||e^{-2\pi^2(kh^2/4)} \;\leq\; \text{tol}$$

- Thus, we can anticipate

$$||\underline{e}^k|| \;\approx\; ||\underline{u}||e^{-\lambda_{\min}\sigma\Delta t k}$$

  with $\lambda_{\min} \approx 2\pi^2$ (for 2D).

- If $\sigma \approx 1$, we have

$$||\underline{e}^k|| \;\approx\; ||\underline{u}||e^{-2\pi^2 t} \;=\; ||\underline{u}||e^{-2\pi^2(kh^2/4)} \;\leq\; \text{tol}$$

- Example, find the number of iterations when tol=10^{-12}.

$$e^{-(\pi^2h^2/4)k} \;\approx\; 10^{-12}$$

$$-(\pi^2h^2/4)k \;\approx\; \ln 10^{-12} \;\approx\; 24 \;\; (27.6...)$$

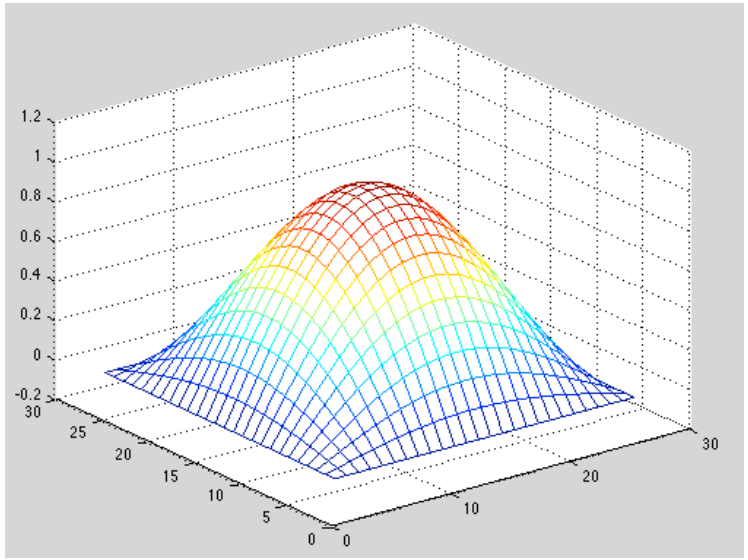
$$k \;\approx\; \frac{28\cdot 2}{\pi^2h^2} \;\approx\; 6n^2$$

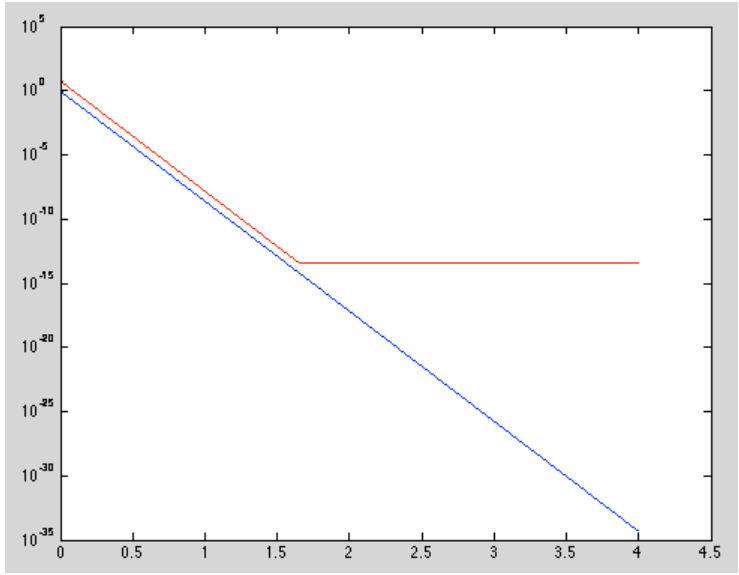
Recap

- Low-wavenumber components decay at a fixed rate: $e^{-\lambda_{\min}\Delta t k}$.
- Stability mandates $\Delta t <\approx h^2/4 = 1/4(n+1)^{-2}$.
- Number of steps scales like n^2.
- Note, if $\sigma = 1$, then *highest* and *lowest* wavenumber components decay at *same* rate.
- If $\frac{1}{2} < \sigma < 1$, high wavenumber components of error decay very fast. We say that damped Jacobi iteration is a *smoother*.

Matlab Example

```
Tf = 4;  % Final time

m=24; Lx=1; dx=Lx/(m+1); m2=m+2; x=0:(n+1); x=dx*x';
n=24; Ly=1; dy=Ly/(n+1); n2=n+2; y=0:(n+1); y=dy*y';

uh=rand(m,n)-.5; u=0*uh; uh(1,1)=1;
u=zeros(m2,n2);

for l=1:n for k=1:m
   sx = sin(k*pi*x); sy=sin(l*pi*y);
   un = (sx*sy')*uh(k,l)/(k^2+l^2);
   u = u + un;
end; end;
ue = u;  %  mesh (ue); pause

hm2 = 1./(dx*dx); f=0*u;
for j=2:n+1; for i=2:m+1;
   f(i,j)=-hm2*(u(i+1,j)+u(i,j+1)-4*u(i,j)+u(i-1,j)+u(i,j-1));
end; end;


u = 0*f; w=u;

lambda_max = (4/(dx^2) + 4/(dy^2)); dt = 0.8*(2/lambda_max);

nsteps = floor(Tf/dt); dt = Tf/nsteps; t=0;

io=floor(nsteps/30);

hm2 = 1./(dx*dx);

for k=1:nsteps; t=t+dt;
     for j=2:n+1; for i=2:m+1;
       w(i,j)=f(i,j)+hm2*( u(i+1,j)+u(i,j+1)-4*u(i,j)+u(i-1,j)+u(i,j-1));
     end; end;
     u = u+dt*w;
     ee = ue-u;
     if k<80; mesh(ee); pause(.1); k, end;          %% PLOT ERROR
     if mod(k,io)==0; mesh(ee); pause(.2); k, end;
%    if k<80; mesh(u); pause(.1); k, end;           %% PLOT SOLUTION
%    if mod(k,io)==0; mesh(u); pause(.2); k, end;
     nk(k)=norm(u-ue);
     tk(k)=t;
end;

semilogy(tk,nk,'r-',tk,exp(-2*pi*pi*tk),'b-')
```





Conjugate Gradient Iteration

- Recall that Jacobi iteration is of the form

$$\underline{u}^{k+1} \;=\; \underline{u}^{k} \,+\, \Delta t\left(\underline{f} - A\underline{u}^{k}\right)$$

and the error is

$$\underline{u} - \underline{u}^{k} = \underline{e}^{k} \;=\; (I \,-\, \Delta t A)^{k}\,\underline{e}^{0} \;=\; (I \,-\, \Delta t A)^{k}\,\underline{u} \qquad (\underline{e}^{0} = \underline{u} \text{ if } \underline{u}^{0} = 0)$$

$$= \;\left(I \,+\, a_1 A \,+\, a_2 A^2 \,+\, \ldots \,+\, a_k A^k\right)\underline{u}$$

- The solution is therefore of the form

$$\begin{aligned}
\underline{u}^k &= -\left(a_1 A + a_2 A^2 + \ldots + a_k A^k\right) \underline{u} \\
&= -\left(a_1 + a_2 A + \ldots + a_k A^{k-1}\right) A\underline{u} \\
&= -\left(a_1 + a_2 A + \ldots + a_k A^{k-1}\right) \underline{f} \\
&\in \mathbb{P}_{k-1}(A)\underline{f} \\
&\in \mathcal{K}_k(A, \underline{f}), \quad \text{which is the Krylov subspace associated with } A \text{ and } \underline{f}.
\end{aligned}$$

- Thus Jacobi iteration generates an element in $\mathcal{K}_k(A, \underline{f})$, but it is not the *best fit* in the space.
- Conjugate gradient iteration, on the other hand, finds $\underline{u}^k$ that satisfies

$$||\underline{u} - \underline{u}^k||_A \leq ||\underline{u} - \underline{v}||_A \; \forall \; \underline{v} \in \mathcal{K}_k$$

Conjugate Gradient Method

- If $\boldsymbol{A}$ is $n \times n$ symmetric positive definite matrix, then quadratic function

$$\phi(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}$$

  attains minimum precisely when $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

- Optimization methods have form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha\boldsymbol{s}_k$$

  where α is search parameter chosen to minimize objective function $\phi(\boldsymbol{x}_k + \alpha\boldsymbol{s}_k)$ along $\boldsymbol{s}_k$

- For quadratic problem, negative gradient is residual vector

$$-\nabla\phi(\boldsymbol{x}) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x} = \boldsymbol{r}$$

- Line search parameter can be determined analytically

$$\alpha = \boldsymbol{r}_k^T\boldsymbol{s}_k / \boldsymbol{s}_k^T\boldsymbol{A}\boldsymbol{s}_k$$

Conjugate Gradient Method, continued

- Using these properties, we obtain *conjugate gradient* (CG) method for linear systems

$\boldsymbol{x}_0 =$ initial guess
$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\boldsymbol{s}_0 = \boldsymbol{r}_0$
for $k = 0, 1, 2, \ldots$
$\alpha_k = \boldsymbol{r}_k^T \boldsymbol{r}_k / \boldsymbol{s}_k^T \boldsymbol{A}\boldsymbol{s}_k$
$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$
$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{A}\boldsymbol{s}_k$
$\beta_{k+1} = \boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1} / \boldsymbol{r}_k^T \boldsymbol{r}_k$
$\boldsymbol{s}_{k+1} = \boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{s}_k$
end

Conjugate Gradient Method, continued

- Key features that make CG method effective
  - Short recurrence determines search directions that are A-orthogonal (conjugate)
  - Error is minimal over space spanned by search directions generated so far
- Minimum error implies exact solution is reached in at most n steps, since n linearly independent search directions must span whole space
- In practice, loss of orthogonality due to rounding error spoils finite termination property, so method is used iteratively

< interactive example >

Preconditioning

- Convergence rate of CG can often be substantially accelerated by *preconditioning*
- Apply CG algorithm to $M^{-1}A$, where M is chosen so that $M^{-1}A$ is better conditioned and systems of form $Mz = y$ are easily solved
- Popular choices of preconditioners include
  - Diagonal or block-diagonal
  - SSOR
  - Incomplete factorization
  - Polynomial
  - Approximate inverse

Generalizations of CG Method

- CG is not directly applicable to nonsymmetric or indefinite systems
- CG cannot be generalized to nonsymmetric systems without sacrificing one of its two key properties (short recurrence and minimum error)
- Nevertheless, several generalizations have been developed for solving nonsymmetric systems, including GMRES, QMR, CGS, BiCG, and Bi-CGSTAB
- These tend to be less robust and require more storage than CG, but they can still be very useful for solving large nonsymmetric systems

Example: Iterative Methods

- We illustrate various iterative methods by using them to solve 4×4 linear system for Laplace equation example
- In each case we take $x_0 = 0$ as starting guess
- Jacobi method gives following iterates

k	x_1	x_2	x_3	x_4
0	0.000	0.000	0.000	0.000
1	0.000	0.000	0.250	0.250
2	0.062	0.062	0.312	0.312
3	0.094	0.094	0.344	0.344
4	0.109	0.109	0.359	0.359
5	0.117	0.117	0.367	0.367
6	0.121	0.121	0.371	0.371
7	0.123	0.123	0.373	0.373
8	0.124	0.124	0.374	0.374
9	0.125	0.125	0.375	0.375

Example, continued

- Gauss-Seidel method gives following iterates

k	x_1	x_2	x_3	x_4
0	0.000	0.000	0.000	0.000
1	0.000	0.000	0.250	0.312
2	0.062	0.094	0.344	0.359
3	0.109	0.117	0.367	0.371
4	0.121	0.123	0.373	0.374
5	0.124	0.125	0.375	0.375
6	0.125	0.125	0.375	0.375

Example, continued

- SOR method (with optimal $\omega = 1.072$ for this problem) gives following iterates

k	x_1	x_2	x_3	x_4
0	0.000	0.000	0.000	0.000
1	0.000	0.000	0.268	0.335
2	0.072	0.108	0.356	0.365
3	0.119	0.121	0.371	0.373
4	0.123	0.124	0.374	0.375
5	0.125	0.125	0.375	0.375

- CG method converges in only two iterations for this problem

k	x_1	x_2	x_3	x_4
0	0.000	0.000	0.000	0.000
1	0.000	0.000	0.333	0.333
2	0.125	0.125	0.375	0.375

Rate of Convergence

- For more systematic comparison of methods, we compare them on $k \times k$ model grid problem for Laplace equation on unit square
- For this simple problem, spectral radius of iteration matrix for each method can be determined analytically, as well as optimal ω for SOR
- Gauss-Seidel is asymptotically twice as fast as Jacobi for this model problem, but for both methods, number of iterations per digit of accuracy gained is proportional to number of mesh points
- Optimal SOR is order of magnitude faster than either of them, and number of iterations per digit gained is proportional to number of mesh points along one side of grid

Rate of Convergence, continued

- For some specific values of k, values of spectral radius are shown below

k	Jacobi	Gauss-Seidel	Optimal SOR
10	0.9595	0.9206	0.5604
50	0.9981	0.9962	0.8840
100	0.9995	0.9990	0.9397
500	0.99998	0.99996	0.98754

- Spectral radius is extremely close to 1 for large values of k, so all three methods converge very slowly
- For $k = 10$ (linear system of order 100), to gain single decimal digit of accuracy Jacobi method requires more than 50 iterations, Gauss-Seidel more than 25, and optimal SOR about 4

Rate of Convergence, continued

- For $k = 100$ (linear system of order 10,000), to gain single decimal digit of accuracy Jacobi method requires about 5000 iterations, Gauss-Seidel about 2500, and optimal SOR about 37
- Thus, Jacobi and Gauss-Seidel methods are impractical for problem of this size, and optimal SOR, though perhaps reasonable for this problem, also becomes prohibitively slow for still larger problems
- Moreover, performance of SOR depends on knowledge of optimal value for relaxation parameter ω, which is known analytically for this simple model problem but is much harder to determine in general

Rate of Convergence, continued

- Convergence behavior of CG is more complicated, but error is reduced at each iteration by factor of roughly

$$\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$$

  on average, where

$$\kappa = \text{cond}(\boldsymbol{A}) = \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\| = \frac{\lambda_{\max}(\boldsymbol{A})}{\lambda_{\min}(\boldsymbol{A})}$$

- When matrix $\boldsymbol{A}$ is well-conditioned, convergence is rapid, but if $\boldsymbol{A}$ is ill-conditioned, convergence can be arbitrarily slow
- This is why preconditioner is usually used with CG method, so preconditioned matrix $\boldsymbol{M}^{-1}\boldsymbol{A}$ has much smaller condition number than $\boldsymbol{A}$

Rate of Convergence, continued

- This convergence estimate is conservative, however, and CG method may do much better
- If matrix $\boldsymbol{A}$ has only m *distinct* eigenvalues, then theoretically CG converges in at most m iterations
- Detailed convergence behavior depends on entire spectrum of $\boldsymbol{A}$, not just its extreme eigenvalues, and in practice convergence is often superlinear

Smoothers

- Disappointing convergence rates observed for stationary iterative methods are *asymptotic*
- Much better progress may be made initially before eventually settling into slow asymptotic phase
- Many stationary iterative methods tend to reduce high-frequency (i.e., oscillatory) components of error rapidly, but reduce low-frequency (i.e., smooth) components of error much more slowly, which produces poor asymptotic rate of convergence
- For this reason, such methods are sometimes called *smoothers*

< interactive example >

Multigrid Methods

- Smooth or oscillatory components of error are relative to mesh on which solution is defined
- Component that appears smooth on fine grid may appear oscillatory when sampled on coarser grid
- If we apply smoother on coarser grid, then we may make rapid progress in reducing this (now oscillatory) component of error
- After few iterations of smoother, results can then be interpolated back to fine grid to produce solution that has both higher-frequency and lower-frequency components of error reduced

Multigrid Methods, continued

- *Multigrid methods*: This idea can be extended to multiple levels of grids, so that error components of various frequencies can be reduced rapidly, each at appropriate level
- Transition from finer grid to coarser grid involves *restriction* or *injection*
- Transition from coarser grid to finer grid involves *interpolation* or *prolongation*

Residual Equation

- If $\hat{\boldsymbol{x}}$ is approximate solution to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, with residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}$, then error $\boldsymbol{e} = \boldsymbol{x} - \hat{\boldsymbol{x}}$ satisfies equation $\boldsymbol{A}\boldsymbol{e} = \boldsymbol{r}$
- Thus, in improving approximate solution we can work with just this *residual equation* involving error and residual, rather than solution and original right-hand side
- One advantage of residual equation is that zero is reasonable starting guess for its solution

Two-Grid Algorithm

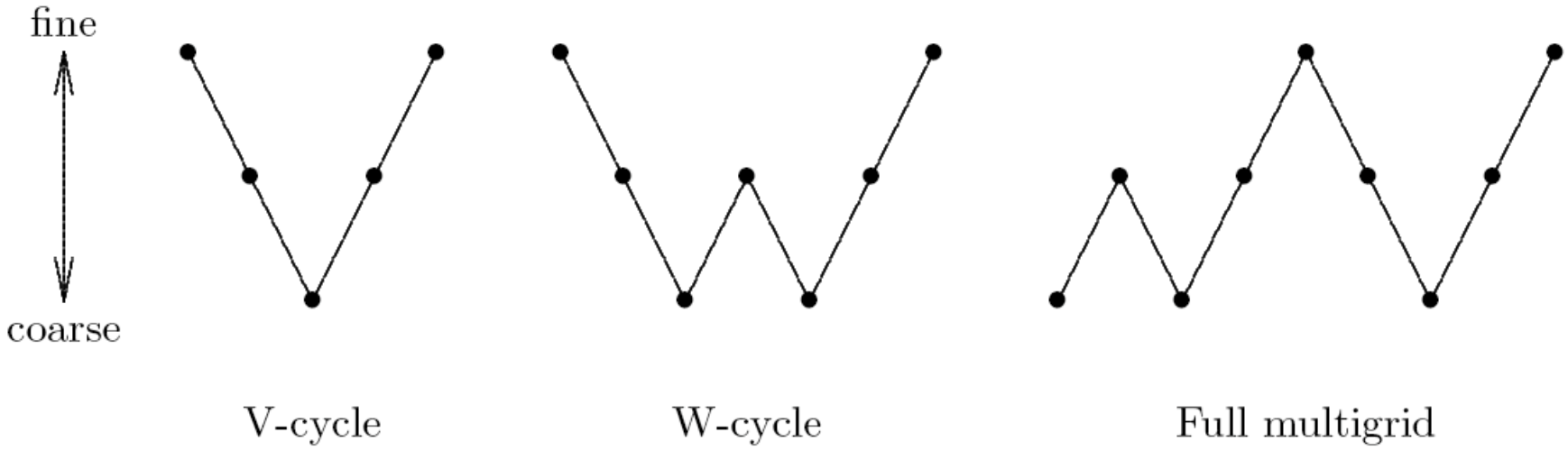
1. On fine grid, use few iterations of smoother to compute approximate solution $\hat{\boldsymbol{x}}$ for system $\boldsymbol{Ax} = \boldsymbol{b}$
2. Compute residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}$
3. Restrict residual to coarse grid
4. On coarse grid, use few iterations of smoother on residual equation to obtain coarse-grid approximation to error
5. Interpolate coarse-grid correction to fine grid to obtain improved approximate solution on fine grid
6. Apply few iterations of smoother to corrected solution on fine grid

Multigrid Methods, continued

- *Multigrid method* results from recursion in Step 4: coarse grid correction is itself improved by using still coarser grid, and so on down to some bottom level
- Computations become progressively cheaper on coarser and coarser grids because systems become successively smaller
- In particular, direct method may be feasible on coarsest grid if system is small enough

Cycling Strategies

Common strategies for cycling through grid levels



Cycling Strategies, continued

- *V-cycle* starts with finest grid and goes down through successive levels to coarsest grid and then back up again to finest grid
- *W-cycle* zig-zags among lower level grids before moving back up to finest grid, to get more benefit from coarser grids where computations are cheaper
- *Full multigrid* starts at coarsest level, where good initial solution is easier to come by (perhaps by direct method), then bootstraps this solution up through grid levels, ultimately reaching finest grid

Multigrid Methods, continued

- By exploiting strengths of underlying iterative smoothers and avoiding their weaknesses, multigrid methods are capable of extraordinarily good performance, linear in number of grid points in best case
- At each level, smoother reduces oscillatory component of error rapidly, at rate independent of mesh size h, since few iterations of smoother, often only one, are performed at each level
- Since all components of error appear oscillatory at some level, convergence rate of entire multigrid scheme should be rapid and independent of mesh size, in contrast to other iterative methods

Multigrid Methods, continued

- Moreover, cost of entire cycle of multigrid is only modest multiple of cost of single sweep on finest grid
- As result, multigrid methods are among most powerful methods available for solving sparse linear systems arising from PDEs
- They are capable of converging to within truncation error of discretization at cost comparable with fast direct methods, although latter are much less broadly applicable

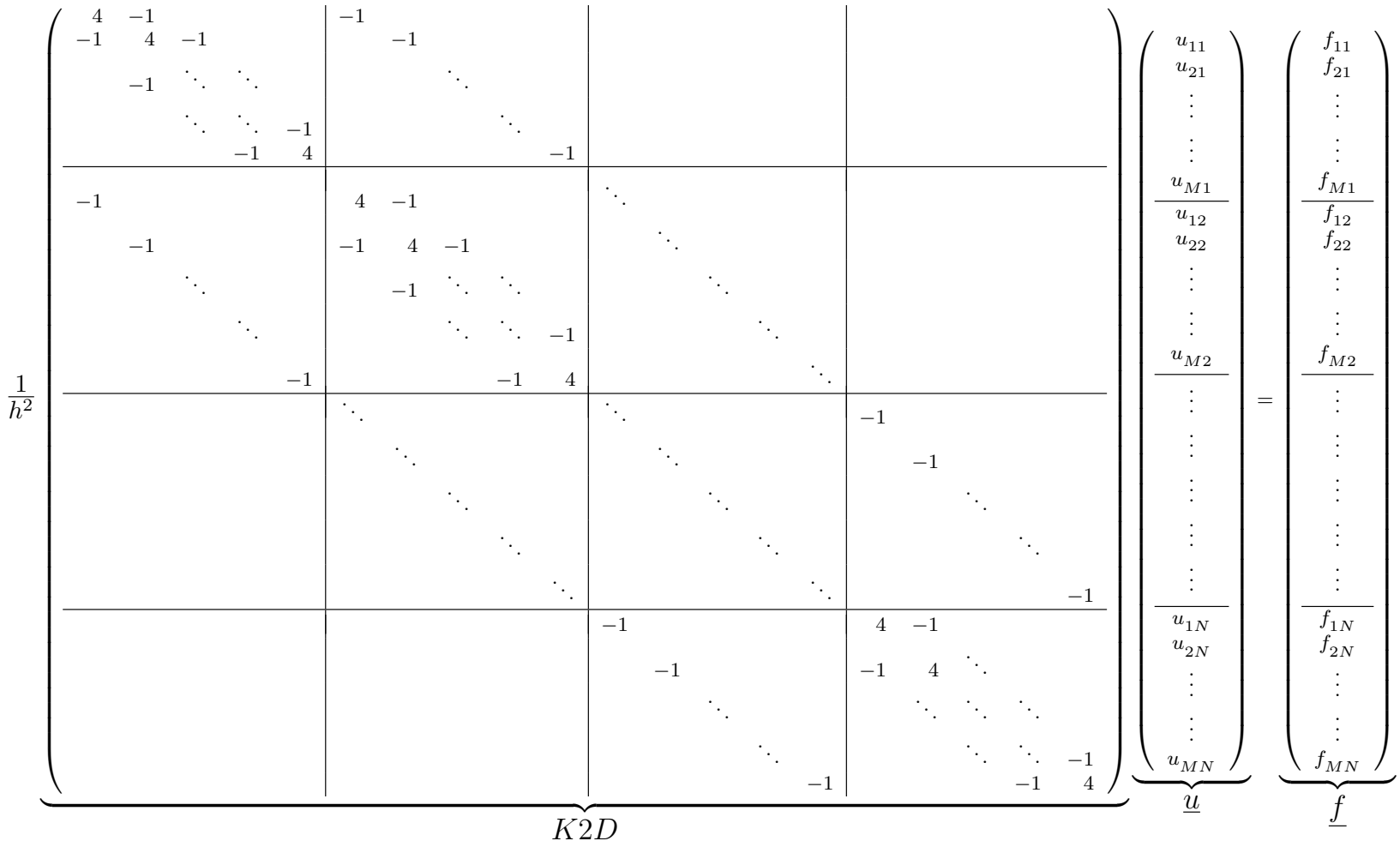
Direct vs. Iterative Methods

- Direct methods require no initial estimate for solution, but take no advantage if good estimate happens to be available
- Direct methods are good at producing high accuracy, but take no advantage if only low accuracy is needed
- Iterative methods are often dependent on special properties, such as matrix being symmetric positive definite, and are subject to very slow convergence for badly conditioned systems; direct methods are more robust in both senses

Direct vs. Iterative Methods

- Iterative methods usually require less work if convergence is rapid, but often require computation or estimation of various parameters or preconditioners to accelerate convergence
- Iterative methods do not require explicit storage of matrix entries, and hence are good when matrix can be produced easily on demand or is most easily implemented as linear operator
- Iterative methods are less readily embodied in standard software packages, since best representation of matrix is often problem dependent and “hard-coded” in application program, whereas direct methods employ more standard storage schemes

System Matrix for 2D Poisson Problem



$$-\left(\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2}+\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{\Delta y^2}\right) \quad = \quad f_{ij}$$

Computational Cost for $k \times k\,(\times k)$ Grid

method	2-D	3-D
Dense Cholesky	k^6	k^9
Jacobi	$k^4 \log k$	$k^5 \log k$
Gauss-Seidel	$k^4 \log k$	$k^5 \log k$
Band Cholesky	k^4	k^7
Optimal SOR	$k^3 \log k$	$k^4 \log k$
Sparse Cholesky	k^3	k^6
Conjugate Gradient	k^3	k^4
Optimal SSOR	$k^{2.5} \log k$	$k^{3.5} \log k$
Preconditioned CG	$k^{2.5}$	$k^{3.5}$
Optimal ADI	$k^2 \log^2 k$	$k^3 \log^2 k$
Cyclic Reduction	$k^2 \log k$	$k^3 \log k$
FFT	$k^2 \log k$	$k^3 \log k$
Multigrid V-cycle	$k^2 \log k$	$k^3 \log k$
FACR	$k^2 \log \log k$	$k^3 \log \log k$
Full Multigrid	k^2	k^3

Comparison of Methods

- For those methods that remain viable choices for finite element discretizations with less regular meshes, computational cost of solving elliptic boundary value problems is given below in terms of exponent of n (order of matrix) in dominant term of cost estimate

method	2-D	3-D
Dense Cholesky	3	3
Band Cholesky	2	2.33
Sparse Cholesky	1.5	2
Conjugate Gradient	1.5	1.33
Preconditioned CG	1.25	1.17
Multigrid	1	1

Comparison of Methods, continued

- Multigrid methods can be optimal, in sense that cost of computing solution is of same order as cost of reading input or writing output
- FACR method is also optimal for all practical purposes, since $\log \log k$ is effectively constant for any reasonable value of k
- Other fast direct methods are almost as effective in practice unless k is very large
- Despite their higher cost, factorization methods are still useful in some cases due to their greater robustness, especially for nonsymmetric matrices
- Methods akin to factorization are often used to compute effective preconditioners

Software for PDEs

- Methods for numerical solution of PDEs tend to be very problem dependent, so PDEs are usually solved by custom written software to take maximum advantage of particular features of given problem
- Nevertheless, some software does exist for some general classes of problems that occur often in practice
- In addition, several good software packages are available for solving sparse linear systems arising from PDEs as well as other sources

Gauss-Seidel Method

- One reason for slow convergence of Jacobi method is that it does not make use of latest information available
- Gauss-Seidel method remedies this by using each new component of solution as soon as it has been computed

$$x_i^{(k+1)} = \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right) / a_{ii}$$

- Using same notation as before, Gauss-Seidel method corresponds to splitting $\boldsymbol{M} = \boldsymbol{D} + \boldsymbol{L}$ and $\boldsymbol{N} = -\boldsymbol{U}$
- Written in matrix terms, this gives iteration scheme

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{D}^{-1} \left(\boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{(k+1)} - \boldsymbol{U}\boldsymbol{x}^{(k)} \right) \\ &= (\boldsymbol{D} + \boldsymbol{L})^{-1} \left(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)} \right) \end{aligned}$$

Gauss-Seidel Method, continued

- In addition to faster convergence, another benefit of Gauss-Seidel method is that duplicate storage is not needed for x, since new component values can overwrite old ones immediately
- On other hand, updating of unknowns must be done successively, in contrast to Jacobi method, in which unknowns can be updated in any order, or even simultaneously
- If we apply Gauss-Seidel method to solve system of finite difference equations for Laplace equation, we obtain

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right)$$

Gauss-Seidel Method, continued

- Thus, we again average solution values at four surrounding grid points, but always use new component values as soon as they become available, rather than waiting until current iteration has been completed
- Gauss-Seidel method does not always converge, but it is guaranteed to converge under conditions that are often satisfied in practice, and are somewhat weaker than those for Jacobi method (e.g., if matrix is symmetric and positive definite)
- Although Gauss-Seidel converges more rapidly than Jacobi, it is often still too slow to be practical

< interactive example >

Successive Over-Relaxation

- Convergence rate of Gauss-Seidel can be accelerated by *successive over-relaxation* (SOR), which in effect uses step to next Gauss-Seidel iterate as search direction, but with fixed search parameter denoted by ω
- Starting with $\boldsymbol{x}^{(k)}$, first compute next iterate that would be given by Gauss-Seidel, $\boldsymbol{x}_{GS}^{(k+1)}$, then instead take next iterate to be

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + \omega(\boldsymbol{x}_{GS}^{(k+1)} - \boldsymbol{x}^{(k)}) \\ &= (1-\omega)\boldsymbol{x}^{(k)} + \omega \boldsymbol{x}_{GS}^{(k+1)} \end{aligned}$$

  which is weighted average of current iterate and next Gauss-Seidel iterate

Successive Over-Relaxation, continued

- ω is fixed *relaxation* parameter chosen to accelerate convergence
- $\omega > 1$ gives *over-relaxation*, $\omega < 1$ gives *under-relaxation*, and $\omega = 1$ gives Gauss-Seidel method
- Method diverges unless $0 < \omega < 2$, but choosing optimal ω is difficult in general and is subject of elaborate theory for special classes of matrices

Successive Over-Relaxation, continued

- Using same notation as before, SOR method corresponds to splitting

$$\boldsymbol{M} = \frac{1}{\omega}\boldsymbol{D} + \boldsymbol{L}, \qquad \boldsymbol{N} = \left(\frac{1}{\omega} - 1\right)\boldsymbol{D} - \boldsymbol{U}$$

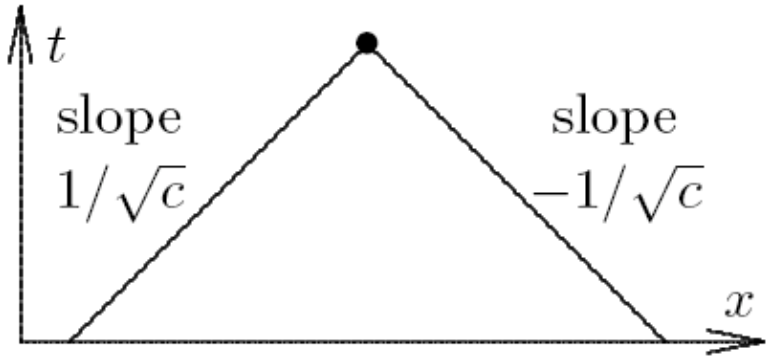
and can be written in matrix terms as

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + \omega\left(\boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{(k+1)} - \boldsymbol{U}\boldsymbol{x}^{(k)}) - \boldsymbol{x}^{(k)}\right) \\ &= (\boldsymbol{D} + \omega\boldsymbol{L})^{-1}\left((1-\omega)\boldsymbol{D} - \omega\boldsymbol{U}\right)\boldsymbol{x}^{(k)} + \omega\,(\boldsymbol{D} + \omega\boldsymbol{L})^{-1}\boldsymbol{b} \end{aligned}$$

< interactive example >

Example: Wave Equation

- Consider explicit finite difference scheme for wave equation given previously
- Characteristics of wave equation are straight lines in (t, x) plane along which either $x + \sqrt{c}\,t$ or $x - \sqrt{c}\,t$ is constant
- Domain of dependence for wave equation for given point is triangle with apex at given point and with sides of slope $1/\sqrt{c}$ and $-1/\sqrt{c}$

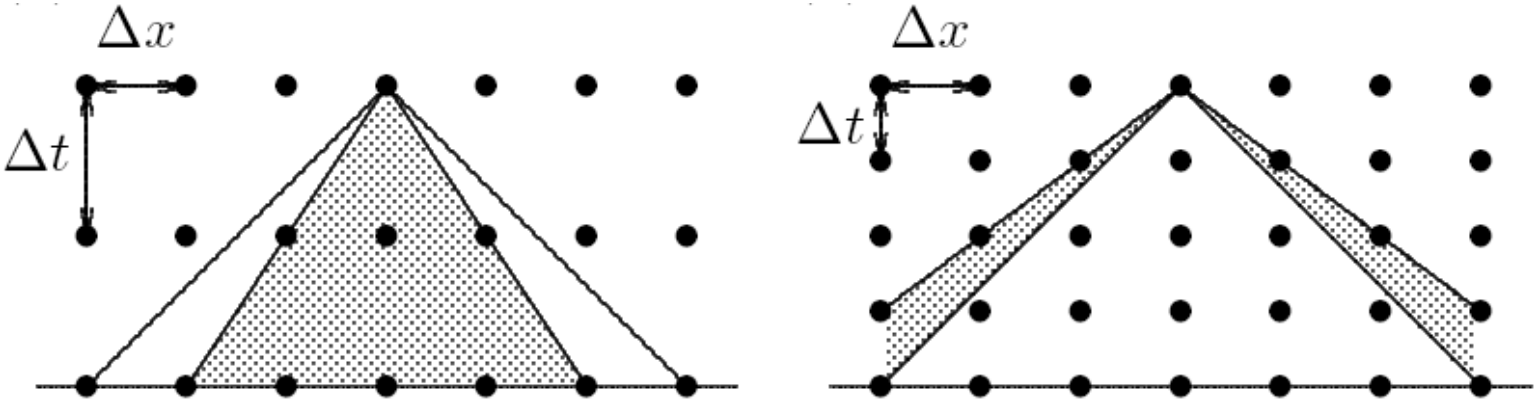


Example: Wave Equation

- CFL condition implies step sizes must satisfy

$$\Delta t \leq \frac{\Delta x}{\sqrt{c}}$$

for this particular finite difference scheme



unstable stable

< interactive example >

Example: Iterative Refinement

- Iterative refinement is example of stationary iterative method
- Forward and back substitution using LU factorization in effect provides approximation, call it $\boldsymbol{B}^{-1}$, to inverse of $\boldsymbol{A}$
- Iterative refinement has form

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \boldsymbol{B}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k) \\ &= (\boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A})\boldsymbol{x}_k + \boldsymbol{B}^{-1}\boldsymbol{b} \end{aligned}$$

- So it is stationary iterative method with

$$\boldsymbol{G} = \boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A}, \qquad \boldsymbol{c} = \boldsymbol{B}^{-1}\boldsymbol{b}$$

- It converges if

$$\rho(\boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A}) < 1$$

Splitting

- One way to obtain matrix G is by *splitting*

$$A = M - N$$

with M nonsingular

- Then take $G = M^{-1}N$, so iteration scheme is

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b$$

- which is implemented as

$$Mx_{k+1} = Nx_k + b$$

(i.e., we solve linear system with matrix M at each iteration)

Convergence

- Stationary iteration using splitting converges if

$$\rho(\boldsymbol{G}) = \rho(\boldsymbol{M}^{-1}\boldsymbol{N}) < 1$$

  and smaller spectral radius yields faster convergence

- For fewest iterations, should choose $\boldsymbol{M}$ and $\boldsymbol{N}$ so $\rho(\boldsymbol{M}^{-1}\boldsymbol{N})$ is as small as possible, but cost per iteration is determined by cost of solving linear system with matrix $\boldsymbol{M}$, so there is tradeoff

- In practice, $\boldsymbol{M}$ is chosen to approximate $\boldsymbol{A}$ in some sense, but is constrained to have simple form, such as diagonal or triangular, so linear system at each iteration is easy to solve

Jacobi Method

- In matrix splitting $A = M - N$, simplest choice for M is diagonal of A
- Let D be diagonal matrix with same diagonal entries as A, and let L and U be strict lower and upper triangular portions of A
- Then $M = D$ and $N = -(L + U)$ gives splitting of A
- Assuming A has no zero diagonal entries, so D is nonsingular, this gives *Jacobi method*

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

Jacobi Method, continued

- Rewriting this scheme componentwise, we see that Jacobi method computes next iterate by solving for each component of x in terms of others

$$x_i^{(k+1)} = \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) / a_{ii}, \quad i = 1, \ldots, n$$

- Jacobi method requires double storage for x, since old component values are needed throughout sweep, so new component values cannot overwrite them until sweep has been completed

Example: Jacobi Method

- If we apply Jacobi method to system of finite difference equations for Laplace equation, we obtain

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right)$$

  so new approximate solution at each grid point is average of previous solution at four surrounding grid points

- Jacobi method does not always converge, but it is guaranteed to converge under conditions that are often satisfied in practice (e.g., if matrix is strictly diagonally dominant)

- Unfortunately, convergence rate of Jacobi is usually unacceptably slow < interactive example >