INF5620: Numerical Methods for Partial Differential Equations

Hans Petter Langtangen

Simula Research Laboratory, and

Dept. of Informatics, Univ. of Oslo

Last update: November, 2012

Nonlinear PDEs

Examples

- Some nonlinear model problems to be treated next:

$$
\begin{aligned}
-u''(x) &= f(u), \quad u(0) = u_L,\ u(1) = u_R, \\
-(\alpha(u)u')' &= 0, \quad u(0) = u_L,\ u(1) = u_R \\
-\nabla \cdot [\alpha(u)\nabla u] &= g(\boldsymbol{x}), \quad \text{with } u \text{ or } -\alpha\frac{\partial u}{\partial n} \text{ B.C.}
\end{aligned}
$$

- Discretization methods:
  - standard finite difference methods
  - standard finite element methods
  - the group finite element method
- We get *nonlinear* algebraic equations
- Solution method: iterate over linear equations

Nonlinear discrete equations; FDM

- Finite differences for $-u'' = f(u)$:

$$-\frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = f(u_i)$$

$\Rightarrow$ nonlinear system of algebraic equations

$$\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}, \text{ or } \boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{u}), \ \boldsymbol{u} = (u_0, \ldots, u_N)^T$$

- Finite differences for $(\alpha(u)u')' = 0$:

$$\frac{1}{h^2}(\alpha(u_{i+1/2})(u_{i+1} - u_i) - \alpha(u_{i-1/2})(u_i - u_{i-1})) = 0$$

$$\frac{1}{2h^2}([\alpha(u_{i+1}) + \alpha(u_i)](u_{i+1} - u_i) - [\alpha(u_i) + \alpha(u_{i-1})](u_i - u_{i-1})) = 0$$

$\Rightarrow$ nonlinear system of algebraic equations

$$\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0} \quad \text{or} \quad \boldsymbol{A}(\boldsymbol{u})\boldsymbol{u} = \boldsymbol{b}$$

Nonlinear discrete equations; FEM

- Finite elements for $-u'' = f(u)$ with $u(0) = u(1) = 0$
- Galerkin approach: find $u = \sum_{k=1}^{n} u_k \varphi_k(x) \in V$ such that

$$\int_0^1 u'v'dx = \int_0^1 f(u)vdx \quad \forall v \in V$$

- Left-hand side is easy to assemble: $v = \varphi_i$

$$-\frac{1}{h}\left(u_{i-1} - 2u_i + u_{i+1}\right) = \int_0^1 f(\sum_k u_k \varphi_k(x))\varphi_i dx$$

- We write $u = \sum_k u_k \varphi_k$ instead of $u = \sum_k c_k \varphi_k$ since $c_k = u(x_k) = u_k$ and u_k is used in the finite difference schemes

Nonlinearities in the FEM

- Note that

$$f(\sum_k \varphi_k(x) u_k)$$

  is a complicated function of $u_0, \ldots, u_N$

- F.ex.: $f(u) = u^2$

$$\int_0^1 \left(\sum_k \varphi_k u_k \right)^2 \varphi_i dx$$

  gives rise to a difference representation

$$\frac{h}{12} \left(u_{i-1}^2 + 2u_i(u_{i-1} + u_{i+1}) + 6u_i^2 + u_{i+1}^2 \right)$$

  (compare with $f(u_i) = u_i^2$ in FDM!)

- Must use numerical integration in general to evaluate $\int f(u)\varphi_i dx$

The group finite element method

- The *group* finite element method:

$$f(u) = f(\sum_j u_j \varphi_j(x)) \approx \sum_{j=1}^{n} f(u_j)\varphi_j$$

- Resulting term: $\int_0^1 f(u)\varphi_i dx = \int_0^1 \sum_j \varphi_i \varphi_j f(u_j)$ gives

$$\sum_j \left(\int_0^1 \varphi_k \varphi_i dx \right) f(u_j)$$

- This integral and formulation also arise from approxmating some function by $u = \sum_j u_j \varphi_j$
- We can write the term as $\boldsymbol{M}\boldsymbol{f}(\boldsymbol{u})$, where $\boldsymbol{M}$ has rows consisting of $h/6(1, 4, 1)$, and row i in $\boldsymbol{M}\boldsymbol{f}(\boldsymbol{u})$ becomes

$$\frac{h}{6}(f(u_{i-1}) + 4f(u_i) + f(u_{i+1}))$$

FEM for a nonlinear coefficient

- We now look at

$$(\alpha(u)u')' = 0, \quad u(0) = u_L,\ u(1) = u_R$$

- Using a finite element method (fine exercise!) results in an integral

$$\int_0^1 \alpha(\sum_k u_k \varphi_k)\varphi_i'\varphi_j'\,dx$$

$\Rightarrow$ complicated to integrate by hand or symbolically

- Linear P1 elements and trapezoidal rule (do it!):

$$\frac{1}{2}(\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - \frac{1}{2}(\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1}) = 0$$

$\Rightarrow$ same as FDM with arithmetic mean for $\alpha(u_{i+1/2})$

Nonlinear algebraic equations

- FEM/FDM for nonlinear PDEs gives nonlinear algebraic equations:

$$(\alpha(u)u')' = 0 \quad \Rightarrow \quad \boldsymbol{A}(\boldsymbol{u})\boldsymbol{u} = \boldsymbol{b}$$
$$-u'' = f(u) \quad \Rightarrow \quad \boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{u})$$

- In general a nonlinear PDE gives

$$\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$$

or

$$\begin{aligned} F_0(u_0, \ldots, u_N) &= 0 \\ &\ldots \\ F_N(u_0, \ldots, u_N) &= 0 \end{aligned}$$

Solving nonlinear algebraic eqs.

- Have
$$\boldsymbol{A}(\boldsymbol{u})\boldsymbol{u} - \boldsymbol{b} = \boldsymbol{0}, \quad \boldsymbol{A}\boldsymbol{u} - \boldsymbol{b}(\boldsymbol{u}) = \boldsymbol{0}, \quad \boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$$
- Idea: solve nonlinear problem as a sequence of linear subproblems
- Must perform some kind of linearization
- Iterative method: guess $\boldsymbol{u}^0$, solve linear problems for $\boldsymbol{u}^1$, $\boldsymbol{u}^2$, ... and hope that
$$\lim_{q\to\infty} \boldsymbol{u}^q = \boldsymbol{u}$$
i.e. the iteration converges

Picard iteration (1)

- Model problem: $\boldsymbol{A}(\boldsymbol{u})\boldsymbol{u} = \boldsymbol{b}$
- Simple iteration scheme:

$$\boldsymbol{A}(\boldsymbol{u}^q)\boldsymbol{u}^{q+1} = \boldsymbol{b}, \quad q = 0, 1, \ldots$$

- Must provide (good) guess $\boldsymbol{u}^0$
- Termination:

$$||\boldsymbol{u}^{q+1} - \boldsymbol{u}^q|| \leq \epsilon_u$$

or using the residual (expensive, req. new $\boldsymbol{A}(\boldsymbol{u}^{q+1})$!)

$$||\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{u}^{q+1})\boldsymbol{u}^{q+1}|| \leq \epsilon_r$$

- Relative criteria:

$$||\boldsymbol{u}^{q+1} - \boldsymbol{u}^q|| \leq \epsilon_u ||\boldsymbol{u}^q||$$

or (more expensive)

$$||\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{u}^{q+1})\boldsymbol{u}^{q+1}|| \leq \epsilon_r ||\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{u}^0)\boldsymbol{u}^0||$$

Picard iteration (2)

- Model problem: $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{u})$
- Simple iteration scheme:

$$\boldsymbol{A}\boldsymbol{u}^{q+1} = \boldsymbol{b}(\boldsymbol{u}^q), \quad q = 0, 1, \ldots$$

- Relaxation:

$$\boldsymbol{A}\boldsymbol{u}^* = \boldsymbol{b}(\boldsymbol{u}^q), \quad \boldsymbol{u}^{q+1} = \omega\boldsymbol{u}^* + (1-\omega)\boldsymbol{u}^q$$

  (may improve convergence, avoids too large steps)
- This method is also called *Successive substitutions*

Newton's method for scalar equations

- The Newton method for $f(x) = 0$, $x \in \mathbb{R}$
- Given an approximation x^q
- Approximate f by a linear function at x^q:

$$f(x) \approx M(x; x^q) = f(x^q) + f'(x^q)(x - x^q)$$

- Find new x^{q+1} such that

$$M(x^{q+1}; x^q) = 0 \Rightarrow x^{q+1} = x^q - \frac{f(x^q)}{f'(x^q)}$$

Newton's method for systems of equations

- Systems of nonlinear equations:

$$\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}, \quad \boldsymbol{F}(\boldsymbol{u}) \approx \boldsymbol{M}(\boldsymbol{u}; \boldsymbol{u}^q)$$

- Multi-dimensional Taylor-series expansion:

$$\boldsymbol{M}(\boldsymbol{u}; \boldsymbol{u}^q) = \boldsymbol{F}(\boldsymbol{u}^q) + \boldsymbol{J}(\boldsymbol{u} - \boldsymbol{u}^q), \quad \boldsymbol{J} \equiv \nabla \boldsymbol{F}$$

$$J_{i,j} = \frac{\partial F_i}{\partial u_j}$$

- Iteration no. q:
  - solve linear system $\boldsymbol{J}(\boldsymbol{u}^q)(\delta \boldsymbol{u})^{q+1} = -\boldsymbol{F}(\boldsymbol{u}^q)$
  - update: $\boldsymbol{u}^{q+1} = \boldsymbol{u}^q + (\delta \boldsymbol{u})^{q+1}$
- Can use relaxation: $\boldsymbol{u}^{q+1} = \boldsymbol{u}^q + \omega(\delta \boldsymbol{u})^{q+1}$

The Jacobian matrix; FDM (1)

- Model equation: $u'' = -f(u)$
- Scheme:

$$F_i \equiv \frac{1}{h^2}(u_{i-1} - 2u_i + u_{i+1}) + f(u_i) = 0$$

- Jacobian matrix term (FDM):

$$J_{i,j} = \frac{\partial F_i}{\partial u_j}$$

- F_i contains only $u_i, u_{i\pm1}$
- Only

$$J_{i,i-1} = \frac{\partial F_i}{\partial u_{i-1}}, \quad J_{i,i} = \frac{\partial F_i}{\partial u_i}, \quad J_{i,i+1} = \frac{\partial F_i}{\partial u_{i+1}} \neq 0$$

$\Rightarrow$ Jacobian is tridiagonal

The Jacobian matrix; FDM (2)

$$F_i \equiv \frac{1}{h^2}(u_{i-1} - 2u_i + u_{i+1}) - f(u_i) = 0$$

- Derivation:

$$J_{i,i-1} = \frac{\partial F_i}{\partial u_{i-1}} = \frac{1}{h^2}$$

$$J_{i,i+1} = \frac{\partial F_i}{\partial u_{i+1}} = \frac{1}{h^2}$$

$$J_{i,i} = \frac{\partial F_i}{\partial u_i} = -\frac{2}{h^2} + f'(u_i)$$

- Must form the Jacobian matrix $\boldsymbol{J}$ in each iteration and solve

$$\boldsymbol{J}\delta\boldsymbol{u}^{q+1} = -\boldsymbol{F}(\boldsymbol{u}^q)$$

and then update

$$\boldsymbol{u}^{q+1} = \boldsymbol{u}^q + \omega\delta\boldsymbol{u}^{q+1}$$

The Jacobian matrix; FEM

- $-u'' = f(u)$ on $\Omega = (0,1)$ with $u(0) = u(1) = 0$ and FEM

$$F_i \equiv \int_0^1 \left[\sum_k \varphi_i' \varphi_k' u_k - f(\sum_s u_s \varphi_s) \varphi_i \right] dx = 0$$

- First term of the Jacobian $J_{i,j} = \frac{\partial F_i}{\partial u_j}$:

$$\frac{\partial}{\partial u_j} \int_0^1 \sum_k \varphi_i' \varphi_k' u_k \, dx = \int_0^1 \frac{\partial}{\partial u_j} \sum_k \varphi_i' \varphi_k' u_k \, dx = \int_0^1 \varphi_i' \varphi_j' \, dx$$

- Second term:

$$-\frac{\partial}{\partial u_j} \int_0^1 f(\sum_s u_s \varphi_s) \varphi_i \, dx = -\int_0^1 f'(\sum_s u_s \varphi_s) \varphi_j \varphi_i \, dx$$

because when $u = \sum_s u_s \varphi_s$,
$\frac{\partial}{\partial u_j} f(u) = f'(u) \frac{\partial u}{\partial u_j} = f'(u) \frac{\partial}{\partial u_j} \sum_s u_s \varphi_s = f'(u) \varphi_j$

A 2D/3D transient nonlinear PDE (1)

- PDE for heat conduction in a solid where the conduction depends on the temperature u:

$$\varrho C\frac{\partial u}{\partial t} = \nabla\cdot[\kappa(u)\nabla u]$$

(f.ex. $u = g$ on the boundary and $u = I$ at $t = 0$)

- Stable Backward Euler FDM in time:

$$\frac{u^n - u^{n-1}}{\Delta t} = \nabla\cdot[\alpha(u^n)\nabla u^n]$$

with $\alpha = \kappa/(\varrho C)$

- Next step: Galerkin formulation, where $u^n = \sum_j u_j^n\varphi_j$ is the unknown and u^{n-1} is just a known function

A 2D/3D transient nonlinear PDE (2)

- FEM gives nonlinear algebraic equations:

$$F_i(u_0^n, \ldots, u_N^n) = 0, \quad i = 0, \ldots, N$$

where

$$F_i \equiv \int_\Omega \left[\left(u^n - u^{n-1}\right) v + \Delta t \alpha(u^n) \nabla u^n \cdot \nabla v \right] \, d\Omega$$

A 2D/3D transient nonlinear PDE (3)

- Picard iteration:
  Use "old" $u^{n,q}$ in $\alpha(u^n)$ term, solve linear problem for $u^{n,q+1}$,
  $q = 0, 1, \ldots$

$$A_{i,j} = \int_\Omega (\varphi_i \varphi_j + \Delta t \alpha(u^{n,q}) \nabla \varphi_i \cdot \nabla \varphi_j)\, d\Omega$$

$$b_i = \int_\Omega u^{n-1} \varphi_i d\Omega$$

- Newton's method: need Jacobian,

$$J_{i,j} = \frac{\partial F_i}{\partial u_j^n}$$

$$J_{i,j} = \int_\Omega (\varphi_i \varphi_j + \Delta t(\alpha'(u^{n,q})\varphi_j \nabla u^{n,q} \cdot \nabla \varphi_i + \alpha(u^{n,q}) \nabla \varphi_i \cdot \nabla \varphi_j))\, d\Omega$$

Iteration methods at the PDE level

- Consider $-u'' = f(u)$
- Could introduce Picard iteration at the PDE level:

$$-\frac{d^2}{dx^2}u^{q+1} = f(u^q), \quad q = 0, 1, \ldots$$

  $\Rightarrow$ linear problem for u^{q+1}
- A PDE-level Newton method can also be formulated (see the HPL book for details)
- We get identical results for our model problem
- Time-dependent problems: first use finite differences in time, then use an iteration method (Picard or Newton) at the time-discrete PDE level

Continuation methods

- Challenging nonlinear PDE:

$$\nabla \cdot (||\nabla u||^q \nabla u) = 0$$

- For $q = 0$ this problem is simple
- Idea: solve a sequence of problems, starting with $q = 0$, and increase q towards a target value
- Sequence of PDEs:

$$\nabla \cdot (||\nabla u^r||^{q_r} \nabla u^r) = 0, \quad r = 0, 1, 2, \ldots$$

  with $= 0 < q_0 < q_1 < q_2 < \cdots < q_m = q$

- Start guess for u^r is u^{r-1}
  (the solution of a "simpler" problem)
- CFD: The Reynolds number is often the continuation parameter q

Exercise 1

- Derive the nonlinear algebraic equations for the problem

$$\frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) = 0 \text{ on } (0,1), \quad u(0) = 0,\ u(1) = 1,$$

using a finite difference method and the Galerkin method with P1 finite elements and the Trapezoidal rule for approximating integrals.

Exercise 2

- For the problem in Exercise 1, use the group finite element method with P1 elements and the Trapezoidal rule for integrals and show that the resulting equations coincide with those obtained in Exercise 1.

Exercise 3

- For the problem in Exercises 1 and 2, identify the $\boldsymbol{F}$ vector in the system $\boldsymbol{F} = \boldsymbol{0}$ of nonlinear equations. Derive the Jacobian $\boldsymbol{J}$ for a general $\alpha(u)$. Then write the expressions for the Jacobian when $\alpha(u) = \alpha_0 + \alpha_1 u + \alpha_2 u^2$.

Exercise 4

- Explain why discretization of nonlinear differential equations by finite difference and finite element methods normally leads to a Jacobian with the same sparsity pattern as one would encounter in an associated linear problem. Hint: Which unknowns will enter equation number i?

Exercise 5

- Show that if $\boldsymbol{F}(\boldsymbol{u}) = \boldsymbol{0}$ is a *linear system* of equations, $\boldsymbol{F} = \boldsymbol{A}\boldsymbol{u} - \boldsymbol{b}$, for a constant matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$, then Newton's method (with $\omega = 1$) finds the correct solution in the first iteration.

Exercise 6

- The operator $\nabla \cdot (\alpha \nabla u)$, with $\alpha = ||\nabla u||^q$, $q \in \mathbb{R}$, and $||\cdot||$ being the Eucledian norm, appears in several physical problems, especially flow of non-Newtonian fluids. The quantity $\partial\alpha/\partial u_j$ is central when formulating a Newton method, where u_j is the coefficient in the finite element approximation $u = \sum_j u_j \varphi_j$. Show that

$$\frac{\partial}{\partial u_j}||\nabla u||^q = q||\nabla u||^{q-2}\nabla u \cdot \nabla \varphi_j \, .$$

Exercise 7

- Consider the PDE

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(u)\nabla u)$$

  discretized by a Forward Euler difference in time. Explain why this nonlinear PDE gives rise to a linear problem (and hence no need for Newton or Picard iteration) at each time level.
- Discretize the PDE by a Backward Euler difference in time and realize that there is a need for solving nonlinear algebraic equations. Formulate a Picard iteration method for the spatial PDE to be solved at each time level. Formulate a Galerkin method for discretizing the spatial problem at each time level. Choose some appropriate boundary conditions.
- Explain how to incorporate an initial condition.

Exercise 8

- Repeat Exercise 7 for the PDE

$$\varrho(u)\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(u)\nabla u)$$

Exercise 9

- For the problem in Exercise 8, assume that a nonlinear Newton cooling law applies at the whole boundary:

$$-\alpha(u)\frac{\partial u}{\partial n} = H(u)(u - u_S),$$

where $H(u)$ is a nonlinear heat transfer coefficient and u_S is the temperature of the surroundings (and u is the temperature). Use a Backward Euler scheme in time and a Galerkin method in space. Identify the nonlinear algebraic equations to be solved at each time level. Derive the corresponding Jacobian.
The PDE problem in this exercise is highly relevant when the temperature variations are large. Then the density times the heat capacity (ϱ), the heat conduction coefficient (α) and the heat transfer coefficient (H) normally vary with the temperature (u).

Exercise 10

- In Exercise 8, restrict the problem to one space dimension, choose simple boundary conditions like $u = 0$, use the group finite element method for all nonlinear coefficients, apply P1 elements, use the Trapezoidal rule for all integrals, and derive the system of nonlinear algebraic equations that must be solved at each time level.
- Set up some finite difference method and compare the form of the nonlinear algebraic equations.

Exercise 11

- In Exercise 8, use the Picard iteration method with one iteration at each time level, and introduce this method at the PDE level. Realize the similarities with the resulting discretization and that of the corresponding linear diffusion problem.

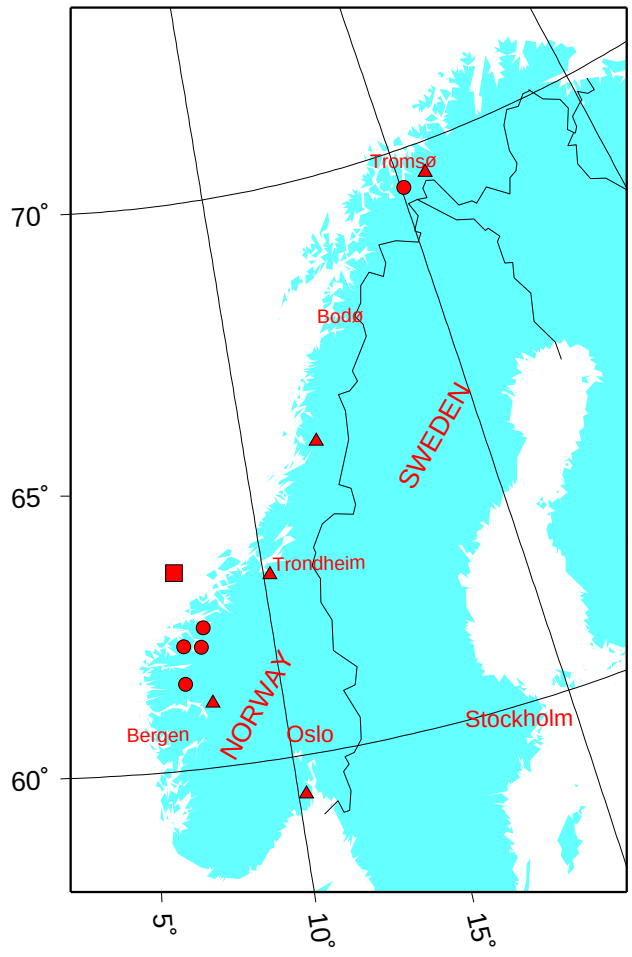
Shallow water waves

Tsunamis

- Waves in fjords, lakes, or oceans, generated by
  - slide
  - earthquake
  - subsea volcano
  - asteroid

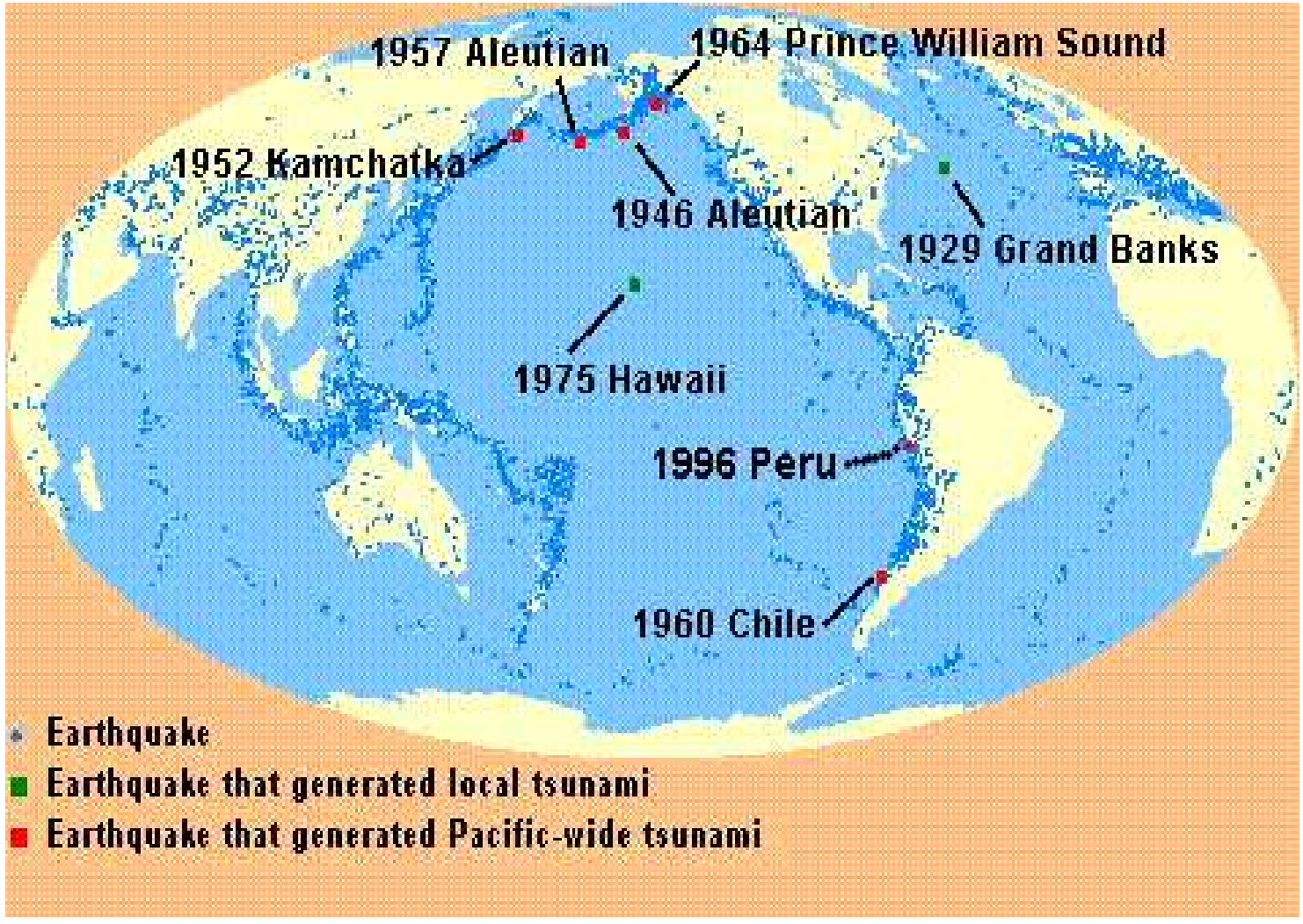
  human activity, like nuclear detonation, or slides generated by oil drilling, may generate tsunamis
- Propagation over large distances
- Hardly recognizable in the open ocean, but wave amplitude increases near shore
- Run-up at the coasts may result in severe damage
- Giant events: Dec 26 2004 (≈ 300000 killed), 1883 (similar to 2004), 65 My ago (extinction of the dinosaurs)

Norwegian tsunamis



Circules: Major incidents, > 10 killed; Triangles: Selected smaller incidents; Square: Storegga (5000 B.C.)

Tsunamis in the Pacific



Scenario: earthquake outside Chile, generates tsunami, propagating at 800 km/h accross the Pacific, run-up on densly populated coasts in Japan;

Selected events; slides

location	year	run-up	dead
Loen	1905	$40m$	61
Tafjord	1934	$62m$	41
Loen	1936	$74m$	73
Storegga	5000 B.C.	$10m$(?)	??
Vaiont, Italy	1963	$270m$	2600
Litua Bay, Alaska	1958	$520m$	2
Shimabara, Japan	1792	$10m$(?)	15000

Selected events; earthquakes etc.

location	year	strength	run-up	dead
Thera	1640 B.C.	volcano	?	?
Thera	1650	volcano	?	?
Lisboa	1755	M=9 ?	15(?)m	?000
Portugal	1969	M=7.9	1 m	
Amorgos	1956	M=7.4	5(?)m	1
Krakatao	1883	volcano	40 m	36 000
Flores	1992	M=7.5	25 m	1 000
Nicaragua	1992	M=7.2	10 m	168
Sumatra	2004	M=9	50 m	300 000

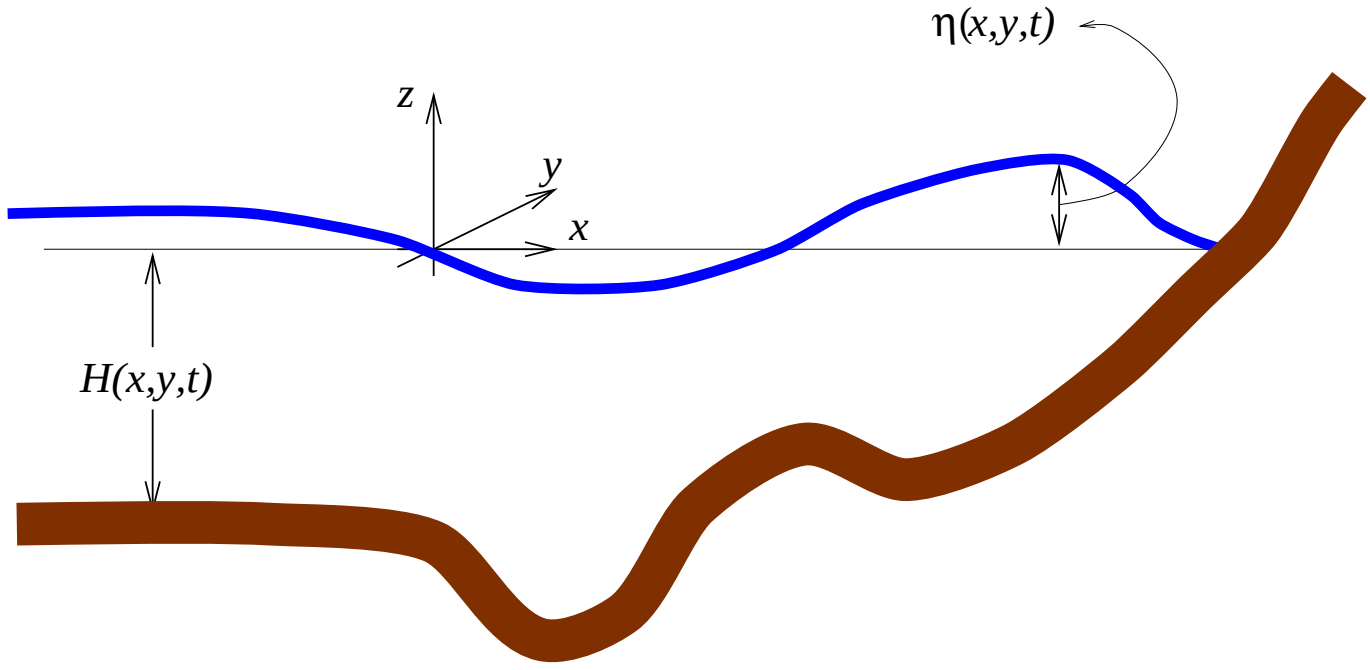
The selection is biased wrt. European events; 150 catastrophic tsunami events have been recorded along along the Japanese coast in modern times.

Tsunamis: no. 5 killer among natural hazards

Why simulation?

- Increase the understanding of tsunamis
- Assist warning systems
- Assist building of harbor protection (break waters)
- Recognize critical coastal areas (e.g. move population)
- Hindcast historical tsunamis (assist geologists/biologists)

Problem sketch



- Assume wavelength $\gg$ depth (long waves)
- Assume small amplitudes relative to depth
- Appropriate approx. for many ocean wave phenomena

Mathematical model

- PDEs:

$$\begin{aligned}
\frac{\partial \eta}{\partial t} &= -\frac{\partial}{\partial x}(uH) - \frac{\partial}{\partial y}(vH) \quad \left(-\frac{\partial H}{\partial t}\right) \\
\frac{\partial u}{\partial t} &= -\frac{\partial \eta}{\partial x}, \quad \boldsymbol{x} \in \Omega,\ t > 0 \\
\frac{\partial v}{\partial t} &= -\frac{\partial \eta}{\partial y}, \quad \boldsymbol{x} \in \Omega,\ t > 0
\end{aligned}$$

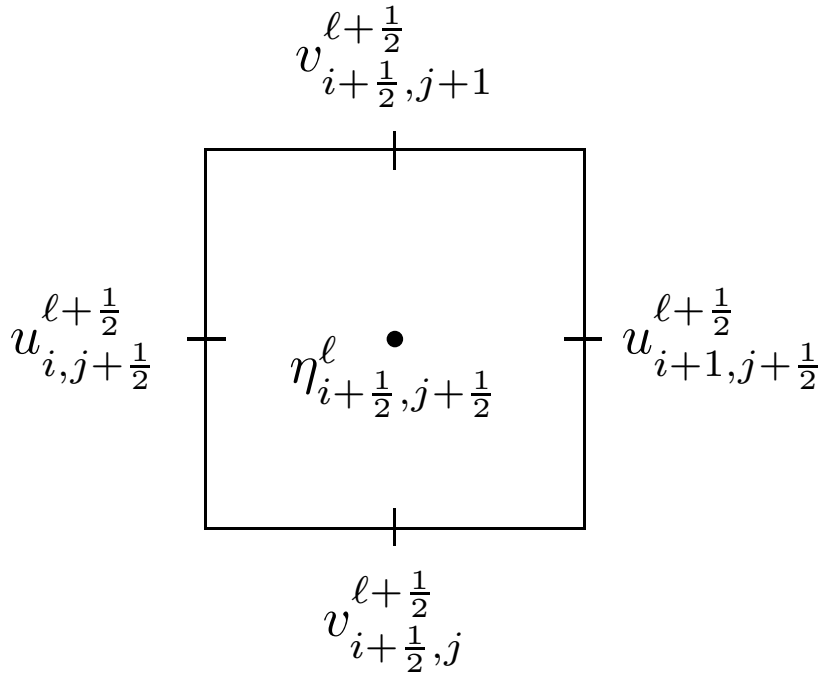
- $\eta(x, y, t)$: surface elevation
- $u(x, y, t)$ and $v(x, y, t)$: horizontal (depth averaged) velocities
- $H(x, y)$: stillwater depth (given)
- Boundary conditions: either η, u or v given at each point
- Initial conditions: all of η, u and v given

Primary unknowns

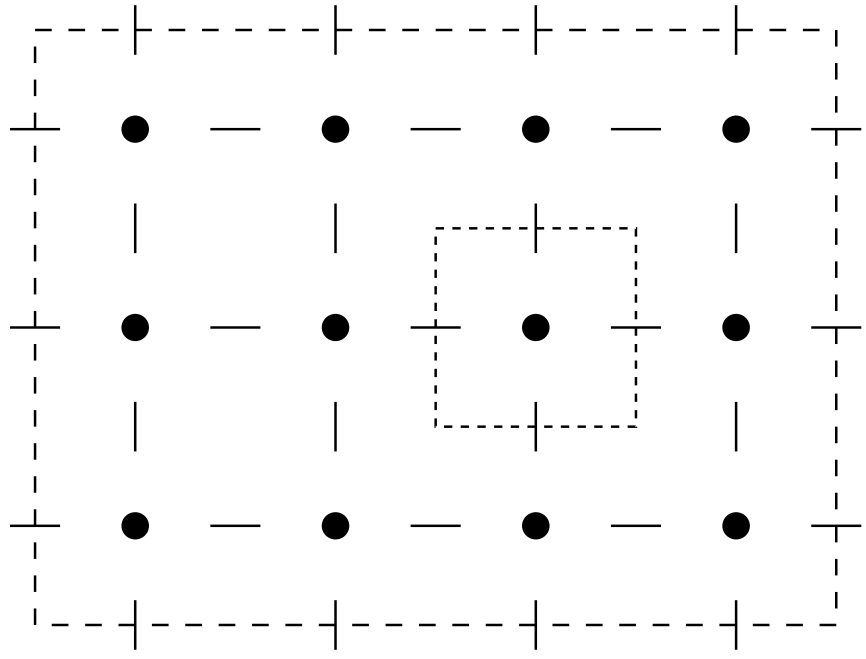
- Discretization: finite differences
- Staggered mesh in time and space

$\Rightarrow$ η, u, and v unknown at different points:

$$\eta^{\ell}_{i+\frac{1}{2},j+\frac{1}{2}}, \quad u^{\ell+\frac{1}{2}}_{i,j+\frac{1}{2}}, \quad v^{\ell+\frac{1}{2}}_{i+\frac{1}{2},j+1}$$



A global staggered mesh



- Widely used mesh in computational fluid dynamics (CFD)
- Important for Navier-Stokes solvers
- Basic idea:
  centered differences in time and space

Discrete equations; η

$$\begin{aligned}
\frac{\partial \eta}{\partial t} &= -\frac{\partial}{\partial x}(uH) - \frac{\partial}{\partial y}(vH) \\
&\quad \text{at } (i+\frac{1}{2}, j+\frac{1}{2}, \ell-\frac{1}{2}) \\
[D_t \eta &= -D_x(uH) - D_y(vH)]^{\ell-\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2}} \\
\frac{1}{\Delta t}\left[\eta^{\ell}_{i+\frac{1}{2},j+\frac{1}{2}} - \eta^{\ell-1}_{i+\frac{1}{2},j+\frac{1}{2}}\right] &= -\frac{1}{\Delta x}\left[(Hu)^{\ell-\frac{1}{2}}_{i+1,j+\frac{1}{2}} - (Hu)^{\ell-\frac{1}{2}}_{i,j+\frac{1}{2}}\right] \\
&\quad -\frac{1}{\Delta y}\left[(Hv)^{\ell-\frac{1}{2}}_{i+\frac{1}{2},j+1} - (Hv)^{\ell-\frac{1}{2}}_{i+\frac{1}{2},j}\right]
\end{aligned}$$

Discrete equations; u

$$\frac{\partial u}{\partial t} = -\frac{\partial \eta}{\partial x} \quad \text{at} \quad (i, j+\frac{1}{2}, \ell)$$

$$[D_t u = -D_x \eta]^{\ell}_{i,j+\frac{1}{2}}$$

$$\frac{1}{\Delta t}\left[u^{\ell+\frac{1}{2}}_{i,j+\frac{1}{2}} - u^{\ell-\frac{1}{2}}_{i,j+\frac{1}{2}}\right] = -\frac{1}{\Delta x}\left[\eta^{\ell}_{i+\frac{1}{2},j+\frac{1}{2}} - \eta^{\ell}_{i-\frac{1}{2},j+\frac{1}{2}}\right]$$

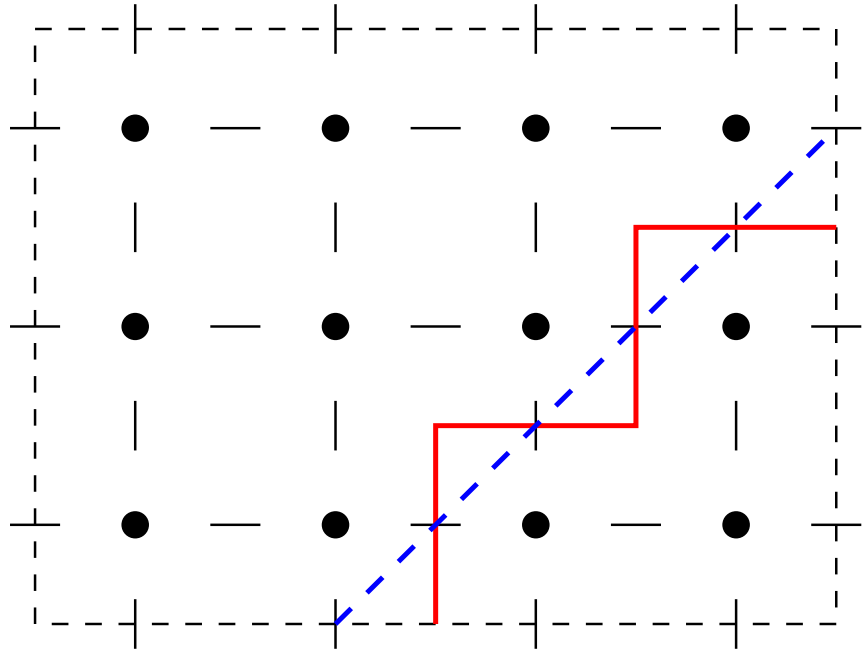
Discrete equations; v

$$\frac{\partial v}{\partial t} = -\frac{\partial \eta}{\partial y} \quad \text{at} \quad (i+\frac{1}{2}, j, \ell)$$

$$[D_t v = -D_y \eta]^{\ell}_{i+\frac{1}{2},j}$$

$$\frac{1}{\Delta t}\left[v^{\ell+\frac{1}{2}}_{i+\frac{1}{2},j} - v^{\ell-\frac{1}{2}}_{i+\frac{1}{2},j}\right] = \frac{1}{\Delta y}\left[\eta^{\ell}_{i+\frac{1}{2},j+\frac{1}{2}} - \eta^{\ell}_{i+\frac{1}{2},j-\frac{1}{2}}\right]$$

Complicated costline boundary



- Saw-tooth approximation to real boundary
- Successful method, widely used
- Warning: can lead to nonphysical waves

Relation to the wave equation

- Eliminate u and v (easy in the PDEs) and get

$$\frac{\partial^2 \eta}{\partial t^2} = \nabla \cdot [H(x,y)\nabla \eta]$$

- Eliminate discrete u and v

$\Rightarrow$ Standard 5-point explicit finite difference scheme for discrete η (quite some algebra needed, try 1D first)

Stability and accuracy

- Centered differences in time and space
- Truncation error, dispersion analysis: $\mathcal{O}(\Delta x^2, \Delta y^2, \Delta t^2)$
- Stability as for the std. wave equation in 2D:

$$\Delta t \leq H^{-\frac{1}{2}} \sqrt{\frac{1}{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}}}$$

(CFL condition)

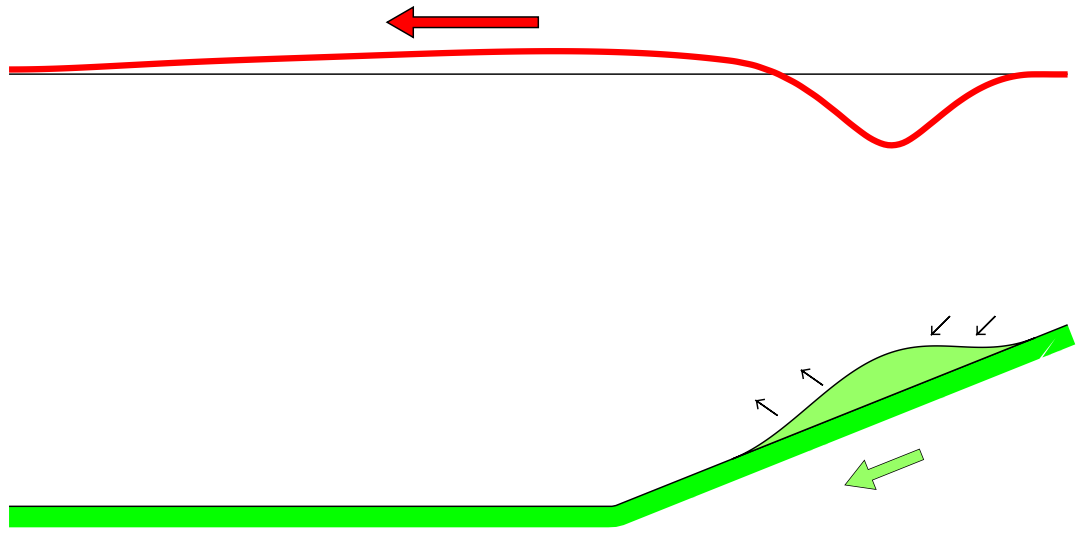
- If H const, exact numerical solution is possible for one-dimensional wave propagation

Verification of an implementation

How can we verify that the program works?

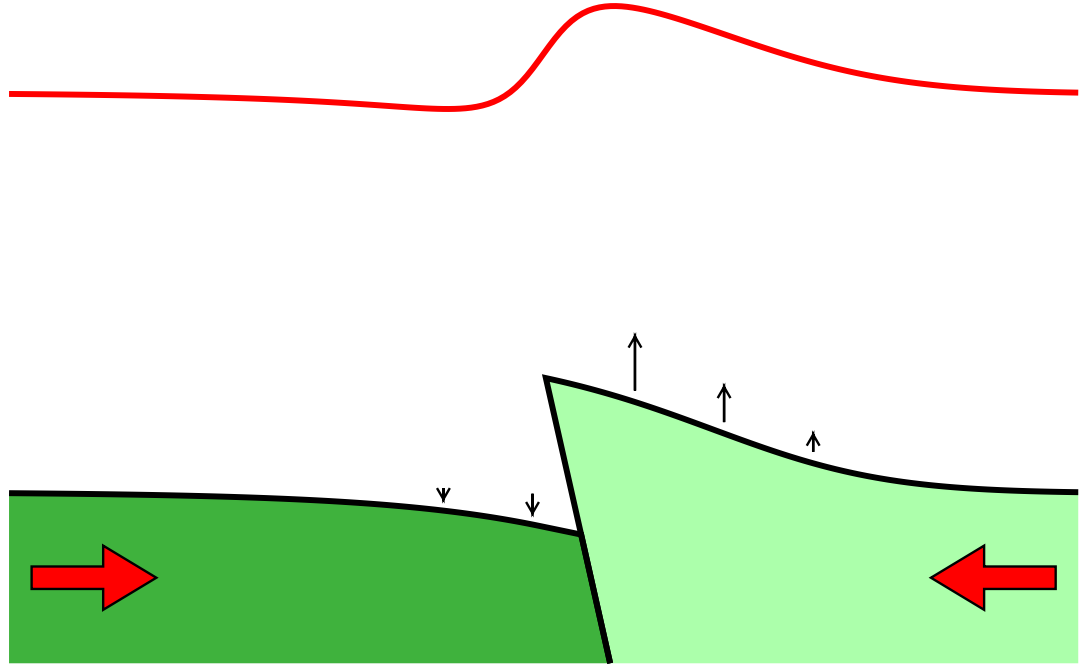
- Compare with an analytical solution
  (if possible)
- Check that basic physical mechanisms are reproduced in a qualitatively correct way by the program

Tsunami due to a slide



- Surface elevation ahead of the slide, dump behind
- Initially, negative dump propagates backwards
- The surface waves propagate faster than the slide moves

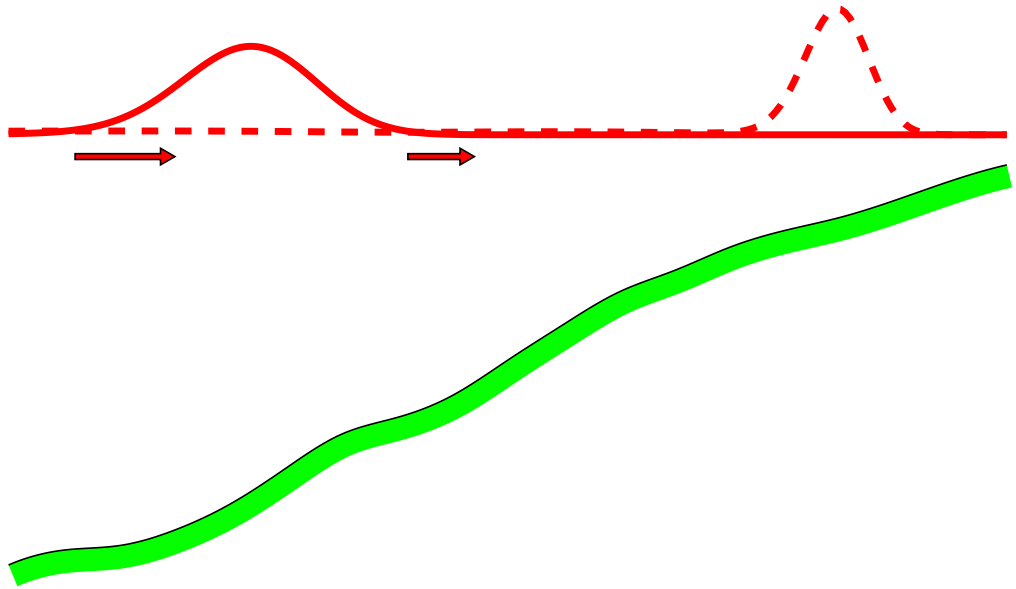
Tsunami due to faulting



- The sea surface deformation reflect the bottom deformation
- Velocity of surface waves ($H \sim 5$ km): 790 km/h
- Velocity of seismic waves in the bottom: 6000–25000 km/h

Tsunami approaching the shore

The velocity of a tsunami is $\sqrt{gH(x,y,t)}$.

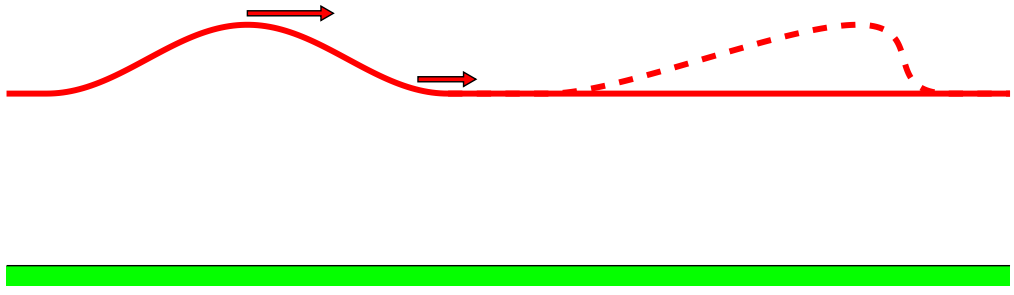


- The back part of the wave moves at higher speed $\Rightarrow$ the wave becomes more peak-formed
- Deep water ($H \sim 3$ km): wave length 40 km, height 1 m
- Shallow water ($H \sim 10$ m): wave length 2 km, height 4 m

Tsunamis experienced from shore

- As a fast tide, with strong currents in fjords
- A wall of water approaching the beach

Wave breaking: the top has larger effective depth and moves faster than the front part (requires a nonlinear PDE)



Convection-dominated flow

Typical transport PDE

- Transport of a scalar u (heat, pollution, ...) in fluid flow $\boldsymbol{v}$:

$$\frac{\partial u}{\partial t} + \boldsymbol{v} \cdot \nabla u = \alpha \nabla^2 u + f$$

- Convection (change of u due to the flow): $\boldsymbol{v} \cdot \nabla u$
- Diffusion (change of u due to molecular collisions): $\alpha \nabla^2 u$
- Common case: convection $\gg$ diffusion $\rightarrow$ numerical difficulties
- Important dimensionless number: Peclet number Pe

$$\text{Pe} = \frac{|\boldsymbol{v} \cdot \nabla u|}{|\alpha \nabla^2 u|} \sim \frac{VU/L}{\alpha U/L^2} = \frac{VL}{\alpha}$$

V: characteristic velocity $\boldsymbol{v}$, L: characteristic length scale, α: diffusion constant, U: characteristic size of u

The transport PDE for fluid flow

- The fluid flow itself is governed by Navier-Stokes equations:

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{v} = -\frac{1}{\varrho}\nabla p + \nu \nabla^2 \boldsymbol{v} + \boldsymbol{f}$$

$$\nabla \cdot \boldsymbol{v} = 0$$

- Important dimensionless number: Reynolds number Re

$$\text{Re} = \frac{\text{convection}}{\text{diffusion}} = \frac{|\boldsymbol{v} \cdot \nabla \boldsymbol{v}|}{|\nu \nabla^2 \boldsymbol{v}|} \sim \frac{V^2/L}{\alpha V/L^2} = \frac{VL}{\nu}$$

- Re $\gg 1$ and Pe $\gg 1$: numerical difficulties

A 1D stationary transport problem

- Assumption: no time, 1D, no source term

$$\boldsymbol{v} \cdot \nabla u = \alpha \nabla^2 u \quad \rightarrow \quad v u' = \alpha u'' \quad \rightarrow \quad u' = \epsilon u'', \quad \epsilon = \frac{\alpha}{v}$$

- Complete model problem:

$$u'(x) = \epsilon u''(x),\ x \in (0,1), \quad u(0) = 0,\ u(1) = 1$$

- ϵ small: boundary layer at $x = 1$
- Standard numerics (i.e. centered differences) will fail!
- Cure: upwind differences

Notation for difference equations (1)

- Define

$$[D_x u]^n_{i,j,k} \equiv \frac{u^n_{i+\frac{1}{2},j,k} - u^n_{i-\frac{1}{2},j,k}}{h}$$

  with similar definitions of D_y, D_z, and D_t

- Another difference:

$$[D_{2x} u]^n_{i,j,k} \equiv \frac{u^n_{i+1,j,k} - u^n_{i-1,j,k}}{2h}$$

- Compound difference:

$$[D_x D_x u]^n_i = \frac{1}{h^2}\left(u^n_{i-1} - 2u^n_i + u^n_{i+1}\right)$$

Notation for difference equations (1)

- One-sided forward difference:

$$[D_x^+ u]_i^n \equiv \frac{u_{i+1}^n - u_i^n}{h}$$

and the backward difference:

$$[D_x^- u]_i^n \equiv \frac{u_i^n - u_{i-1}^n}{h}$$

- Put the whole equation inside brackets:

$$[D_x D_x u = -f]_i$$

is a finite difference scheme for $u'' = -f$

Centered differences

$$u'(x) = \epsilon u''(x),\ x \in (0,1), \quad u(0) = 0,\ u(1) = 1$$

$$\frac{u_{i+1} - u_{i-1}}{2h} = \epsilon \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2},\ i = 2, \ldots, n-1$$

$$u_1 = 0, \quad u_n = 1$$

or

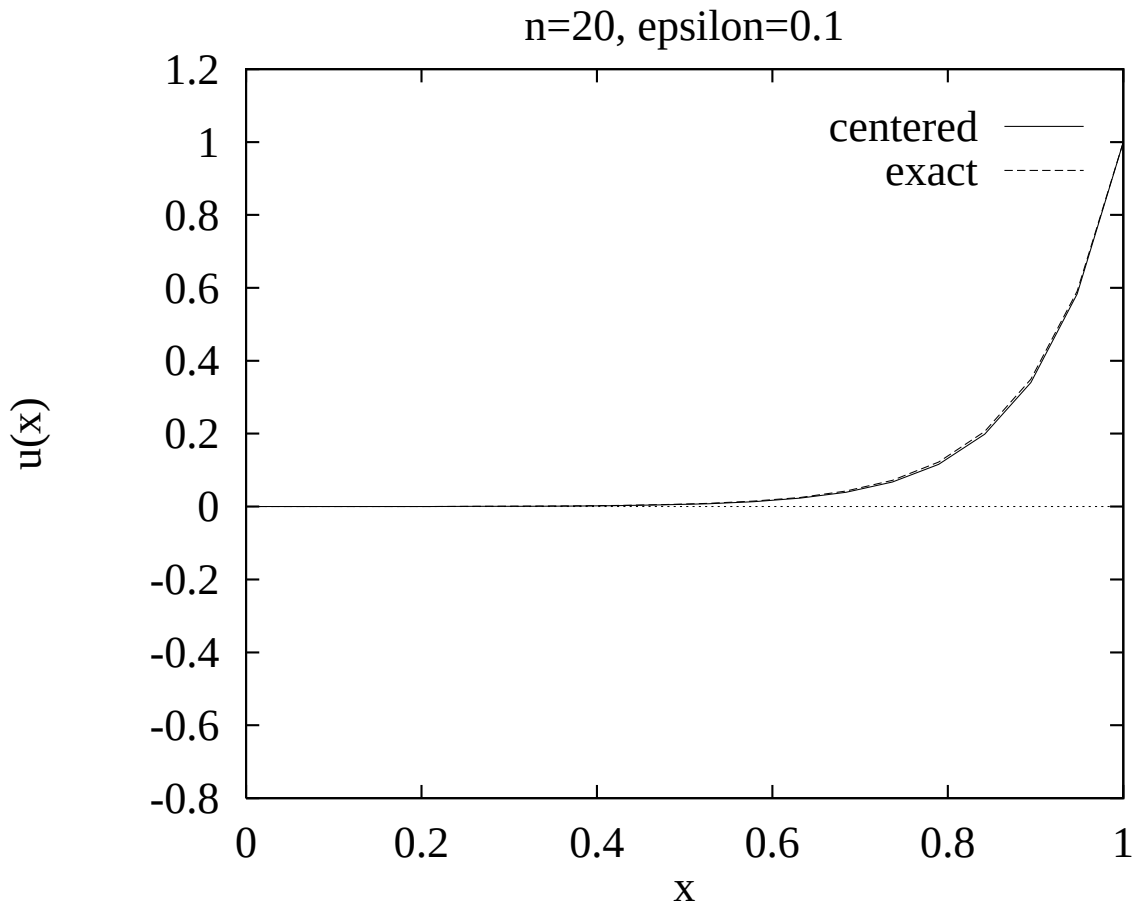
$$[D_{2x} u = \epsilon D_x D_x u]_i$$

Analytical solution:

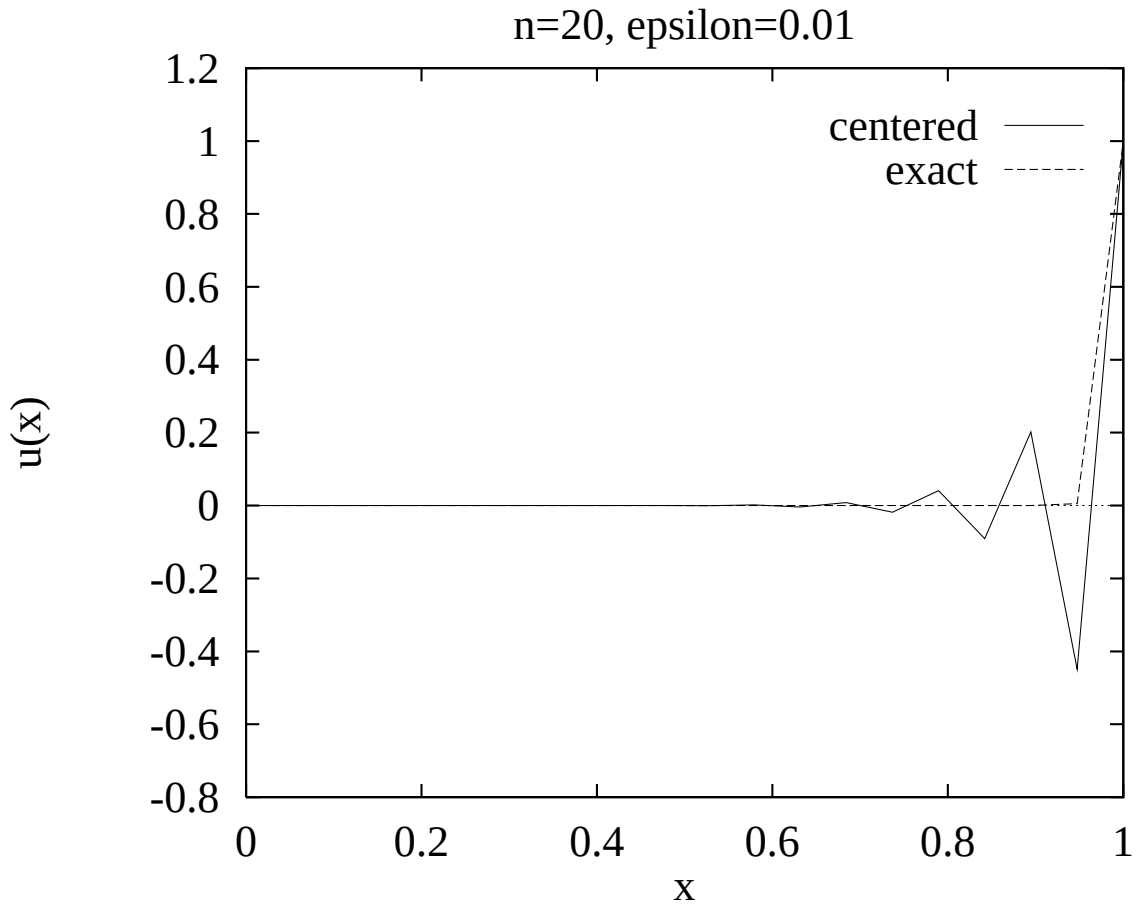
$$u(x) = \frac{1 - e^{x/\epsilon}}{1 - e^{1/\epsilon}}$$

$\Rightarrow$ $u'(x) > 0$, i.e., monotone function

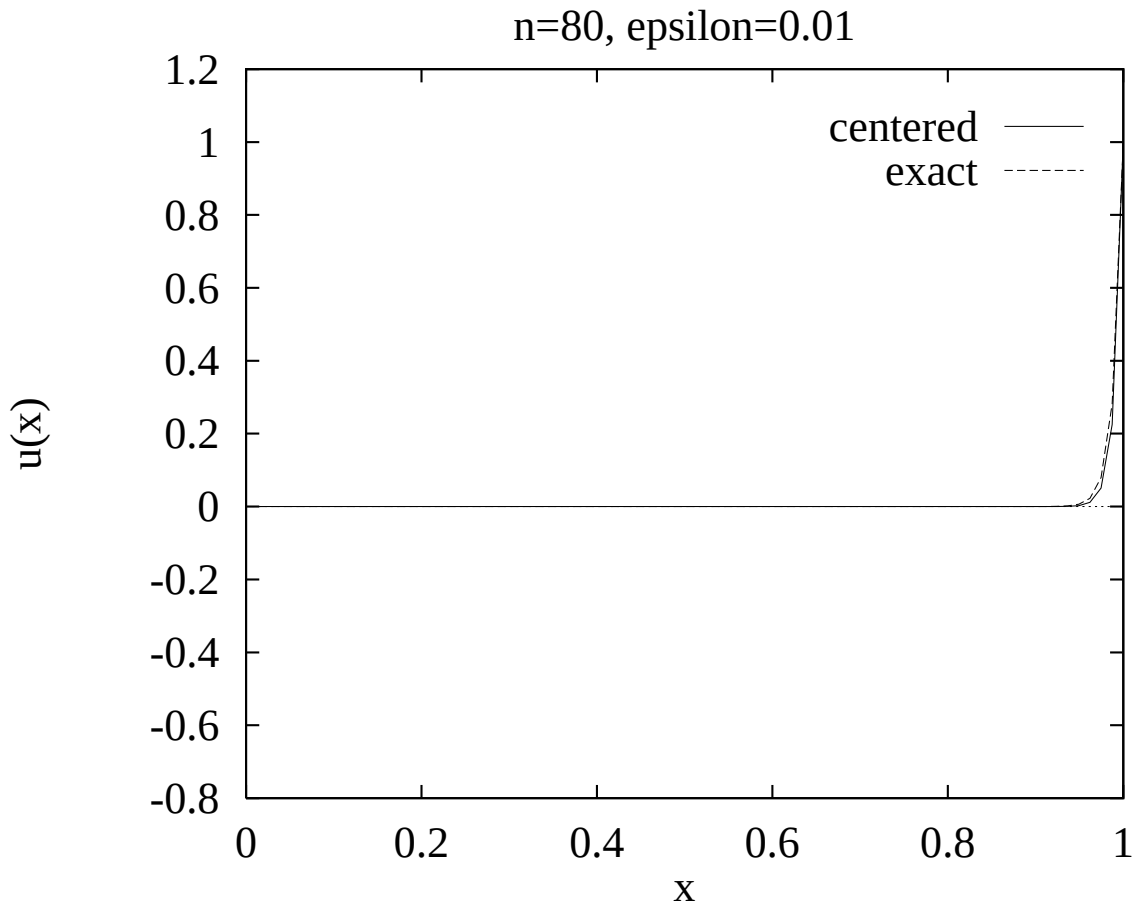
Numerical experiments (1)



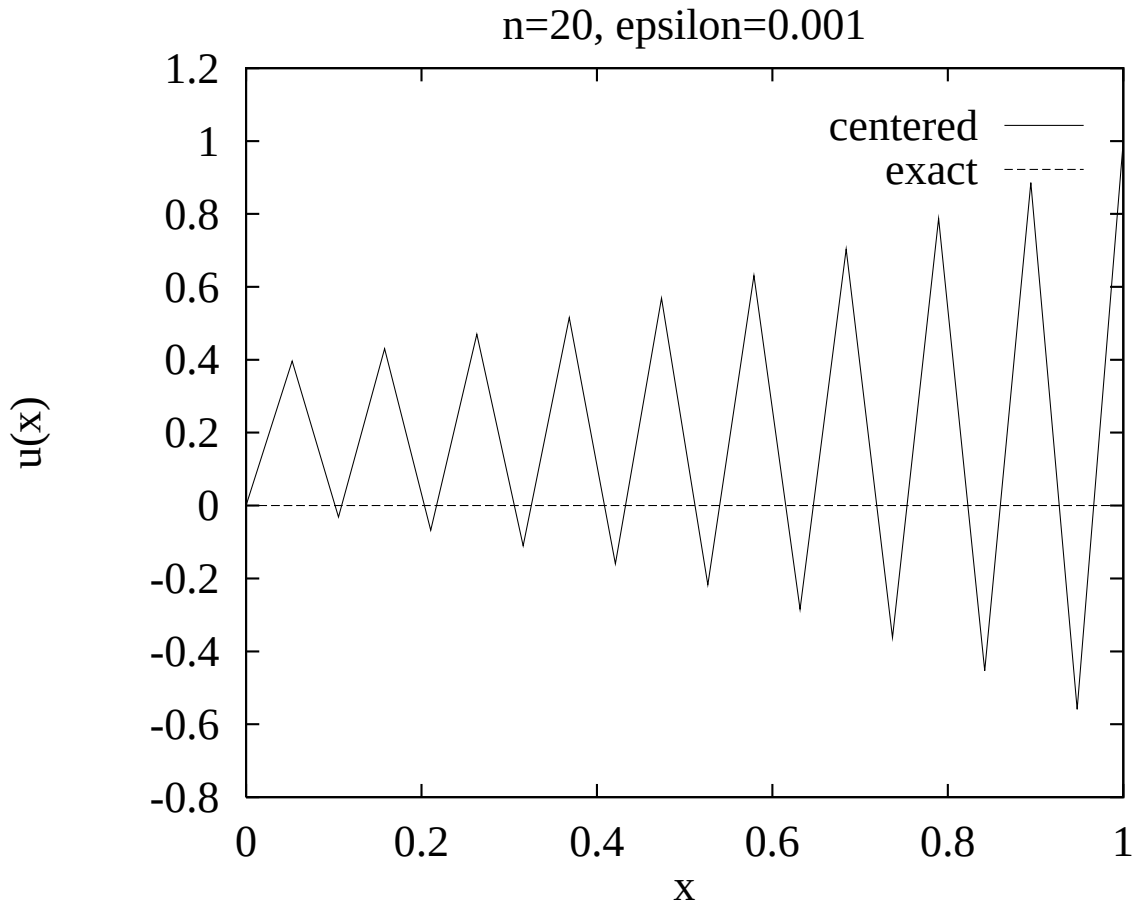
Numerical experiments (2)



Numerical experiments (3)



Numerical experiments (4)



Numerical experiments; summary

- The solution is not monotone if $h > 2\epsilon$
- The convergence rate is h^2
  (expected since both differences are of 2nd order)
  provided $h \leq 2\epsilon$
- Completely wrong qualitative behavior for $h \gg 2\epsilon$

Analysis

- Can find an analytical solution of the discrete problem (!)
- Method: insert $u_i \sim \beta^i$ and solve for β

$$\beta_1 = 1, \quad \beta_2 = \frac{1 + h/(2\epsilon)}{1 - h/(2\epsilon)}$$

- Complete solution:

$$u_i = C_1 \beta_1^i + C_2 \beta_2^i$$

- Determine C_1 and C_2 from boundary conditions

$$u_i = \frac{\beta_2^i - \beta_2}{\beta_2^n - \beta_2}$$

Important result

- Observe: u_i oscillates if $\beta_2 < 0$

$$\Rightarrow \quad \frac{1 + h/(2\epsilon)}{1 - h/(2\epsilon)} < 0 \quad \Rightarrow \quad h > 2\epsilon$$

- Must require $h \leq 2\epsilon$ for u_i to have the same qualitative property as $u(x)$
- This explains why we observed oscillations in the numerical solution

Upwind differences

- Problem:

$$u'(x) = \epsilon u''(x),\ x \in (0,1), \quad u(0) = 0,\ u(1) = 1$$

- Use a backward difference, called upwind difference, for the u' term:

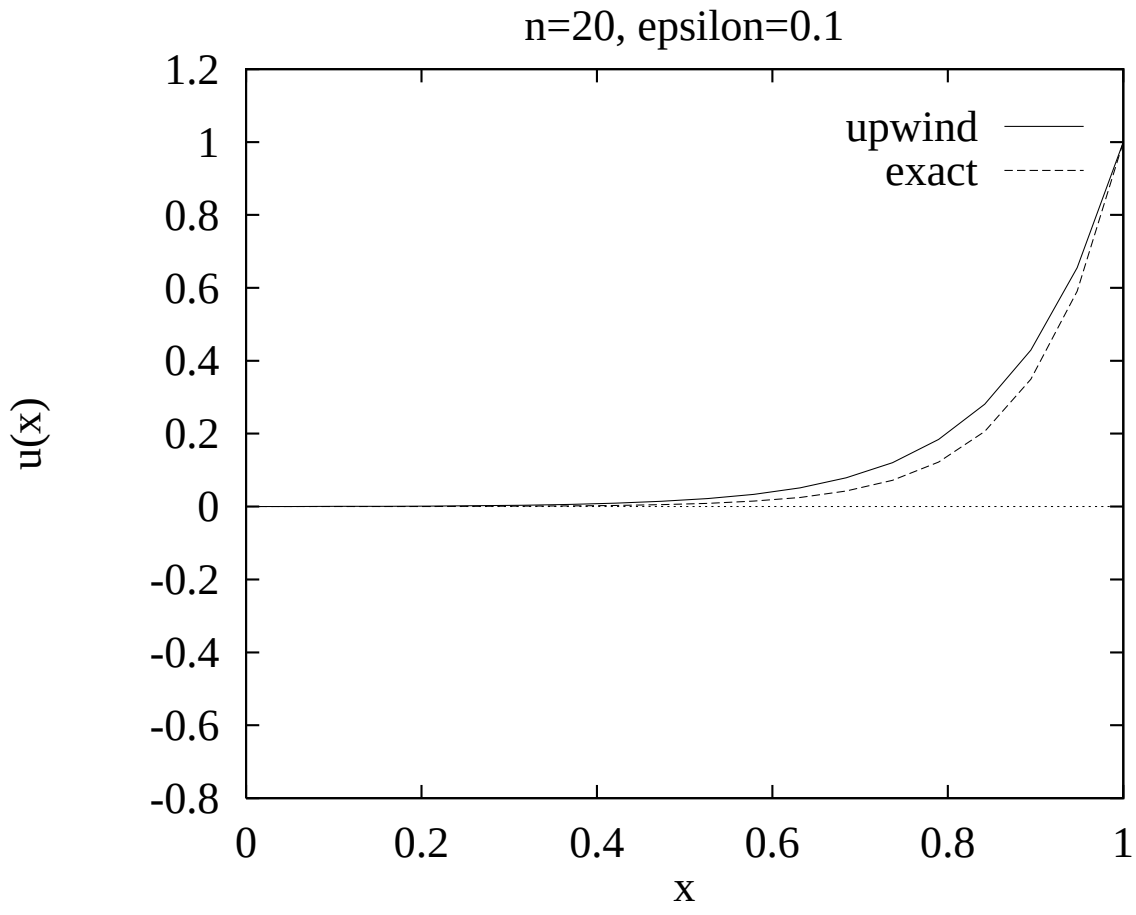
$$\frac{u_i - u_{i-1}}{h} = \epsilon \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2},\ i = 2, \ldots, n-1$$

$$u_1 = 0, \quad u_n = 1$$

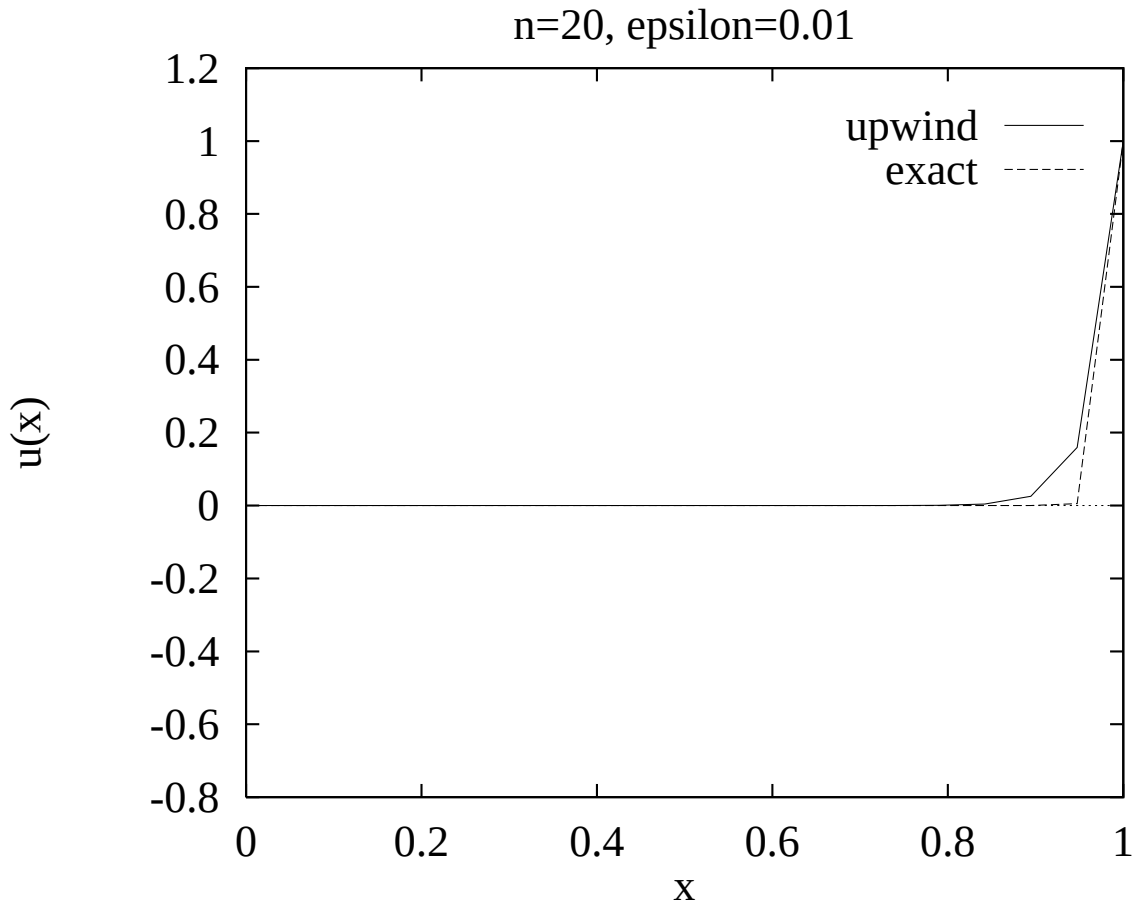
- The scheme can be written as

$$[D_x^- u = \epsilon D_x D_x u]_i$$

Numerical experiments (1)



Numerical experiments (2)



Numerical experiments; summary

- The solution is always monotone, i.e., always qualitatively correct
- The boundary layer is too thick
- The convergence rate is h
  (in agreement with truncation error analysis)

Analysis

- Analytical solution of the discrete equations:

$$u_i = \beta^i \quad \Rightarrow \quad \beta_1 = 1,\ \beta_2 = 1 + h/\epsilon$$

$$u_i = C_1 + C_2 \beta_2^i$$

Using boundary conditions:

$$u_i = \frac{\beta_2^i - \beta_2}{\beta_2^n - \beta_2}$$

- Since $\beta_2 > 0$ (actually $\beta_2 > 1$), β_2^i does not oscillate

Centered vs. upwind scheme

- Truncation error:
  centered is more accurate than upwind
- Exact analysis:
  centered is more accurate than upwind when centered is stable
  (i.e. monotone u_i), but otherwise useless
- $\epsilon = 10^{-6} \Rightarrow$ 500 000 grid points to make $h \leq 2\epsilon$
- Upwind gives best reliability, at a cost of a too thick boundary layer

An interpretation of the upwind scheme

- The upwind scheme

$$\frac{u_i - u_{i-1}}{h} = \epsilon \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

or

$$[D_x^- u = \epsilon D_x D_x u]_i$$

can be rewritten as

$$\frac{u_{i+1} - u_{i-1}}{2h} = (\epsilon + \frac{h}{2}) \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

or

$$[D_{2x} u = (\epsilon + \frac{h}{2}) D_x D_x u]_i$$

- Upwind = centered + artificial diffusion ($h/2$)

Finite elements for the model problem

- Galerkin formulation of

$$u'(x) = \epsilon u''(x),\ x \in (0,1), \quad u(0) = 0,\ u(1) = 1$$

and linear (P1) elements leads to a centered scheme (show it!)

$$\frac{u_{i+1} - u_{i-1}}{2h} = \epsilon \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2},\ i = 2, \ldots, n-1$$

$$u_1 = 0, \quad u_n = 1$$

or

$$[D_{2x}u = \epsilon D_x D_x u]_i$$

- Stability problems when $h > 2\epsilon$

Finite elements and upwind differences

- How to construct upwind differences in a finite element context?
- One possibility: add artificial diffusion ($h/2$)

$$u'(x) = (\epsilon + \frac{h}{2})u''(x),\ x \in (0,1), \quad u(0) = 0,\ u(1) = 1$$

- Can be solved by a Galerkin method
- Another, equivalent strategy: use of perturbed weighting functions

Perturbed weighting functions in 1D

- Take

$$w_i(x) = \varphi_i(x) + \tau\varphi_i'(x)$$

or alternatively written

$$w(x) = v(x) + \tau v'(x)$$

where v is the standard test function in a Galerkin method

- Use this w_i or w as test function *for the convective term* u':

$$\int_0^1 u'w dx = \int_0^1 u'v dx + \int_0^1 \tau u'v' dx$$

- The new term $\tau u'v'$ is the weak formulation of an artificial diffusion term $\tau u''v$
- With $\tau = h/2$ we then get the upwind scheme

Optimal artificial diffusion

- Try a weighted sum of a centered and an upwind discretization:

$$[u']_i \approx [\theta D_x^- u + (1-\theta) D_{2x} u]_i, \quad 0 \leq \theta \leq 1$$

$$[\theta D_x^- u + (1-\theta) D_{2x} u = \epsilon D_x D_x u]_i$$

- Is there an optimal θ?
- Yes, for

$$\theta(h/\epsilon) = \coth \frac{h}{2\epsilon} - \frac{2\epsilon}{h}$$

  we get exact u_i (i.e. u exact at nodal points)
- Equivalent artificial diffusion $\tau_o = 0.5h\theta(h/\epsilon)$
- Exact finite element method: $w(x) = v(x) + \tau_o v'(x)$ for the convective term u'

Multi-dimensional problems

- Model problem:

$$\boldsymbol{v} \cdot \nabla u = \alpha \nabla^2 u$$

  or written out:

$$v_x \frac{\partial u}{\partial x} + v_y \frac{\partial u}{\partial y} = \alpha \nabla^2 u$$

- Non-physical oscillations occur with centered differences or Galerkin methods when the left-hand side terms are large
- Remedy: upwind differences
- Downside: too much diffusion
- Important result: extra stabilizing diffusion is needed only in the streamline direction, i.e., in the direction of $\boldsymbol{v} = (v_x, v_y)$

Streamline diffusion

- Idea: add diffusion in the streamline direction
- Isotropic physical diffusion, expressed through a diffusion *tensor*:

$$\sum_{i=1}^{d}\sum_{j=1}^{d}\alpha\delta_{ij}\frac{\partial^2 u}{\partial x_i \partial x_j} = \alpha\nabla^2 u$$

  $\alpha\delta_{ij}$ is the diffusion *tensor* (here: same in all directions)
- Streamline diffusion makes use of an anisotropic diffusion tensor α_{ij}:

$$\sum_{i=1}^{d}\sum_{j=1}^{d}\frac{\partial}{\partial x_i}\left(\alpha_{ij}\frac{\partial u}{\partial x_j}\right), \quad \alpha_{ij} = \tau\frac{v_i v_j}{||\boldsymbol{v}||^2}$$

- Implementation: artificial diffusion term or perturbed weighting function

Perturbed weighting functions (1)

- Consider the weighting function

$$w = v + \tau^* \boldsymbol{v} \cdot \nabla v$$

  for the convective (left-hand side) term: $\int w\, \boldsymbol{v} \cdot \nabla u \, d\Omega$

- This expands to

$$\int v \boldsymbol{v} \cdot \nabla u d\Omega + \int \tau^* \boldsymbol{v} \cdot \nabla u \, \boldsymbol{v} \cdot \nabla v d\Omega$$

- The latter term can be viewed as the Galerkin formulation of (write $\boldsymbol{v} \cdot \nabla u = \sum_i \partial u / \partial x_i$ etc.)

$$\sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial}{\partial x_i} \left(\tau^* v_i v_j \frac{\partial u}{\partial x_j} \right)$$

Perturbed weighting functions (2)

$\Rightarrow$ Streamline diffusion can be obtained by perturbing the weighting function

- Common name: SUPG (streamline-upwind/Petrov-Galerkin)

Consistent SUPG

- Why not just add artificial diffusion?
- Why bother with perturbed weighting functions?
- In standard FEM (method of weighted residuals),

$$\int_{\Omega} \mathcal{L}(u) w \Omega = 0$$

  the exact solution is a solution of the FEM equations (it fulfills $\mathcal{L}(u)$)
- This no longer holds if we
  - add an artificial diffusion term ($\sim h/2$)
  - use different weighting functions on different terms
- Idea: use consistent SUPG
  - no artificial diffusion term
  - same (perturbed) weighting function applies to all terms

A step back to 1D

- Let us try to use

$$w(x) = v(x) + \tau v'(x)$$

  on both terms in $u' = \epsilon u''$:

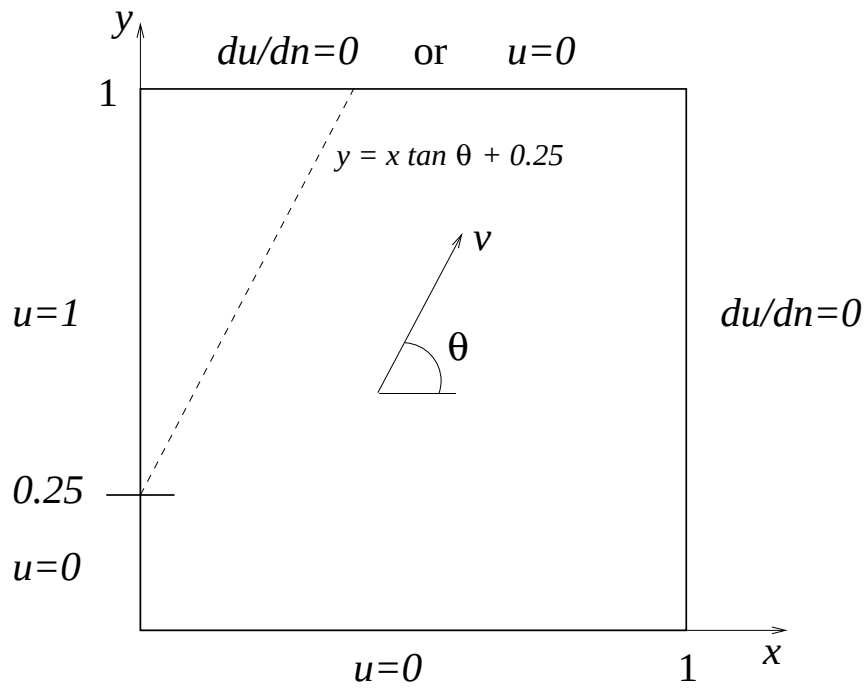
$$\int_0^1 (u'v + (\epsilon + \tau)u'v')dx + \tau \int_0^1 v''u'dx = 0$$

- Problem: last term contains v''
- Remedy: drop it (!)
- Justification: $v'' = 0$ on each linear (P1) element
- Drop 2nd-order derivatives of v in 2D/3D too
- Consistent SUPG is not so consistent...

Choosing τ^*

- Choosing τ^* is a research topic
- Many suggestions
- Two classes:
  - $\tau^* \sim h$
  - $\tau^* \sim \Delta t$ (time-dep. problems)
- Little theory

A test problem (1)



A test problem (2)

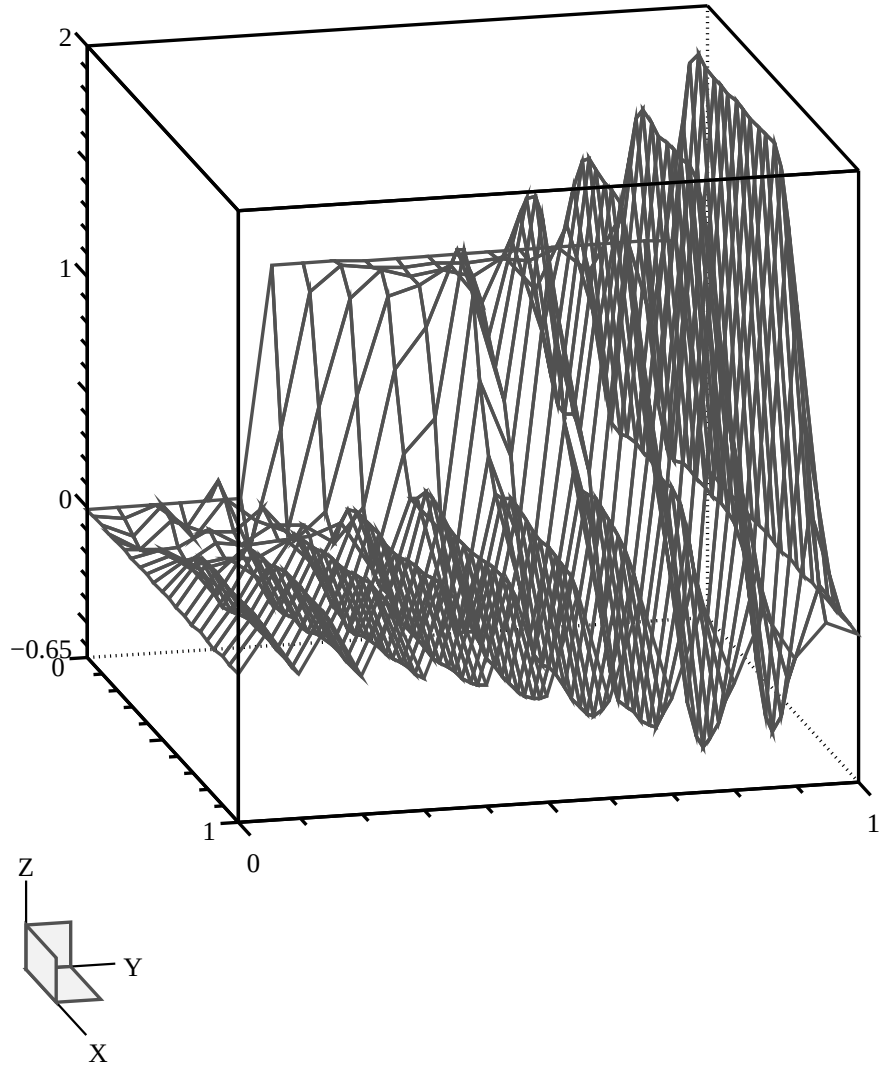
Methods:

1. Classical SUPG:
   Brooks and Hughes: "A streamline upwind/Petrov-Galerkin finite element formulation for advection domainated flows with particular emphasis on the incompressible Navier-Stokes equations", Comp. Methods Appl. Mech. Engrg., 199-259, 1982.
2. An additional discontinuity-capturing term

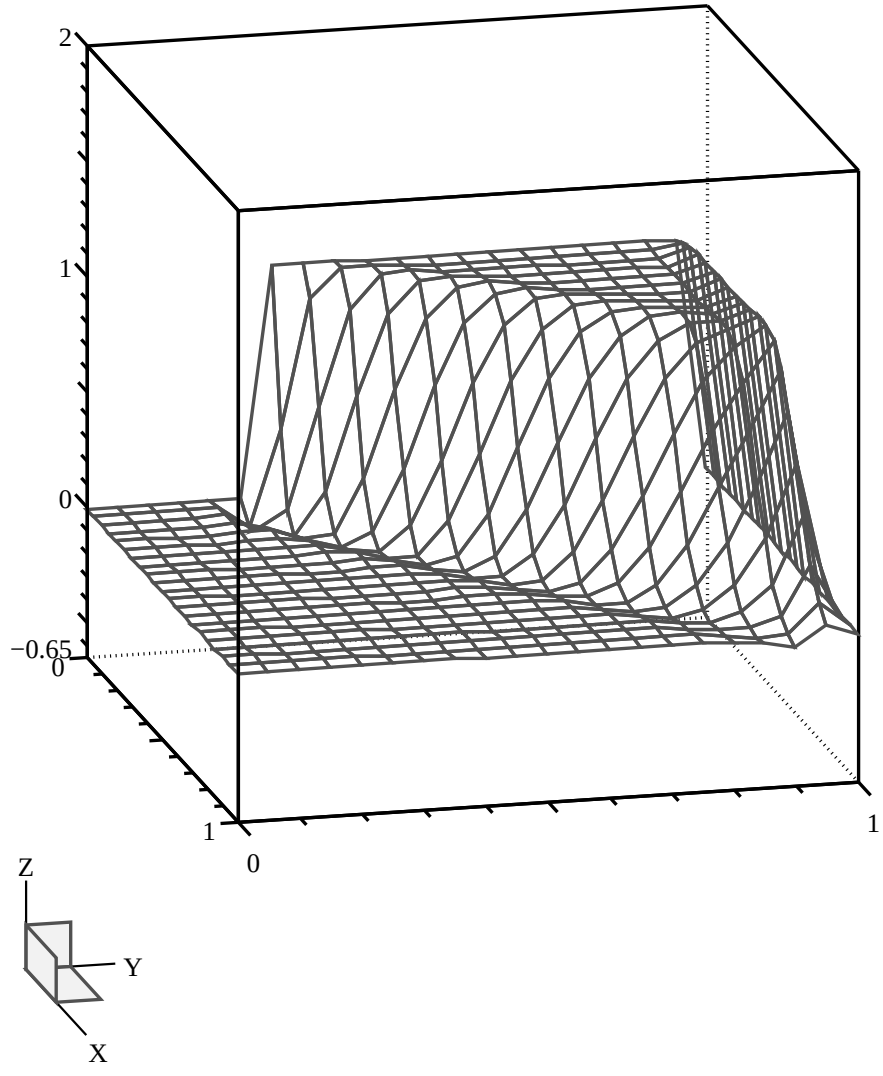
$$w = v + \tau^* \boldsymbol{v} \cdot \nabla v + \hat{\tau} \frac{\boldsymbol{v} \cdot \nabla u}{||\nabla u||^2} \nabla u$$

   was proposed in
   Hughes, Mallet and Mizukami: "A new finite element formulation for computational fluid dynamics: II. Beyond SUPG", Comp. Methods Appl. Mech. Engrg., 341-355, 1986.

Galerkin's method



SUPG



Time-dependent problems

- Model problem:

$$\frac{\partial u}{\partial t} + \boldsymbol{v} \cdot \nabla u = \epsilon \nabla^2 u$$

- Can add artificial streamline diffusion term
- Can use perturbed weighting function

$$w = v + \tau^* \boldsymbol{v} \cdot \nabla v$$

  on all terms
- How to choose τ^*?

Taylor-Galerkin methods (1)

- Idea: Lax-Wendroff + Galerkin
- Model equation:

$$\frac{\partial u}{\partial t} + U\frac{\partial u}{\partial x} = 0$$

- Lax-Wendroff: 2nd-order Taylor series in time,

$$u^{n+1} = u^n + \Delta t \left[\frac{\partial u}{\partial t}\right]^n + \frac{1}{2}\Delta t^2 \left[\frac{\partial^2 u}{\partial t^2}\right]^n$$

- Replace temporal by spatial derivatives,

$$\frac{\partial}{\partial t} = -U\frac{\partial}{\partial x}$$

- Result:

$$u^{n+1} = u^n - U\Delta t \left[\frac{\partial u}{\partial x}\right]^n + \frac{1}{2}U^2\Delta t^2 \left[\frac{\partial^2 u}{\partial x^2}\right]^n$$

Taylor-Galerkin methods (2)

- We can write the scheme on the form

$$\left[D_t^+ u + U \frac{\partial u}{\partial x} = \frac{1}{2} U^2 \Delta t \frac{\partial^2 u}{\partial x^2} \right]^n$$

$\Rightarrow$ Forward scheme with artificial diffusion

- Lax-Wendroff: centered spatial differences,

$$[\delta_t^+ u + U D_{2x} u = \frac{1}{2} U^2 \Delta t D_x D_x u]_i^n$$

- Alternative: Galerkin's method in space,

$$[\delta_t^+ u + U D_{2x} u = \frac{1}{2} U^2 \Delta t D_x D_x u]_i^n$$

provided that we lump the mass matrix

- This is Taylor-Galerkin's method

Taylor-Galerkin methods (3)

- In multi-dimensional problems,

$$\frac{\partial u}{\partial t} + \boldsymbol{v} \cdot \nabla u = 0$$

we have

$$\frac{\partial}{\partial t} = -\boldsymbol{v} \cdot \nabla$$

and ($\nabla \cdot \boldsymbol{v} = 0$)

$$\frac{\partial^2}{\partial t^2} = \nabla \cdot (\boldsymbol{v}\boldsymbol{v} \cdot \nabla) = \sum_{r=1}^{d} \sum_{s=1}^{d} \frac{\partial}{\partial x_r} \left(v_r v_s \frac{\partial}{\partial x_s} \right)$$

- This is streamline diffusion with $\tau^* = \Delta t/2$:

$$[D_t^+ u + \boldsymbol{v} \cdot \nabla u = \frac{1}{2} \Delta t \nabla \cdot (\boldsymbol{v}\boldsymbol{v} \cdot \nabla u)]^n$$

Taylor-Galerkin methods (4)

- Can use the Galerkin method in space
  (gives centered differences)
- The result is close to that of SUPG, but τ^* is diferent

$\Rightarrow$ The Taylor-Galerkin method points to $\tau^* = \Delta t/2$ for SUPG in time-dependent problems

Solving linear systems

The importance of linear system solvers

- PDE problems often (usually) result in linear systems of algebraic equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

- Special methods utilizing that $\boldsymbol{A}$ is sparse is much faster than Gaussian elimination!
- Most of the CPU time in a PDE solver is often spent on solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

$\Rightarrow$ Important to use fast methods

Example: Poisson eq. on the unit cube (1)

- $-\nabla^2 u = f$ on an $n = q \times q \times q$ grid
- FDM/FEM result in $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ system
- FDM: 7 entries pr. row in $\boldsymbol{A}$ are nonzero
- FEM: 7 (tetrahedras), 27 (trilinear elms.), or 125 (triquadratic elms.) entries pr. row in $\boldsymbol{A}$ are nonzero
- $\boldsymbol{A}$ is sparse (mostly zeroes)
- Fraction of nonzeroes: Rq^{-3}
  (R is nonzero entries pr. row)
- Important to work with nonzeroes only!

Example: Poisson eq. on the unit cube (2)

- Compare Banded Gaussian elimination (BGE) versus Conjugate Gradients (CG)
- Work in BGE: $\mathcal{O}(q^7) = \mathcal{O}(n^{2.33})$
- Work in CG: $\mathcal{O}(q^3) = \mathcal{O}(n)$ (multigrid; optimal),
  for the numbers below we use incomplete factorization preconditioning: $\mathcal{O}(n^{1.17})$
- $n = 27000$:
  - CG 72 times faster than BGE
  - BGE needs 20 times more memory than CG
- $n = 8$ million:
  - CG 10^7 times faster than BGE
  - BGE needs 4871 times more memory than CG

Classical iterative methods

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \quad \boldsymbol{A} \in \mathbb{R}^{n,n},\ \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n\ .$$

- Split $\boldsymbol{A}$: $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$
- Write $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ as

  $$\boldsymbol{M}\boldsymbol{x} = \boldsymbol{N}\boldsymbol{x} + \boldsymbol{b},$$

  and introduce an iteration

  $$\boldsymbol{M}\boldsymbol{x}^k = \boldsymbol{N}\boldsymbol{x}^{k-1} + \boldsymbol{b}, \quad k = 1, 2, \ldots$$

- Systems $\boldsymbol{M}\boldsymbol{y} = \boldsymbol{z}$ should be easy/cheap to solve
- Different choices of $\boldsymbol{M}$ correspond to different classical iteration methods:
  - Jacobi iteration
  - Gauss-Seidel iteration
  - Successive Over Relaxation (SOR)
  - Symmetric Successive Over Relaxation (SSOR)

Convergence

$$\boldsymbol{M}\boldsymbol{x}^k = \boldsymbol{N}\boldsymbol{x}^{k-1} + \boldsymbol{b}, \quad k = 1, 2, \ldots$$

- The iteration converges if $\boldsymbol{G} = \boldsymbol{M}^{-1}\boldsymbol{N}$ has its largest eigenvalue, $\varrho(\boldsymbol{G})$, less than 1
- Rate of convergence: $R_\infty(\boldsymbol{G}) = -\ln \varrho(\boldsymbol{G})$
- To reduce the initial error by a factor ϵ,

$$||\boldsymbol{x} - \boldsymbol{x}^k|| \leq \epsilon ||\boldsymbol{x} - \boldsymbol{x}^0||$$

one needs

$$-\ln \epsilon / R_\infty(\boldsymbol{G})$$

iterations

Some classical iterative methods

- Split: $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$
- $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower and upper triangular parts, $\boldsymbol{D}$ is $\boldsymbol{A}$'s diagonal
- Jacobi iteration: $\boldsymbol{M} = \boldsymbol{D}$ ($\boldsymbol{N} = -\boldsymbol{L} - U$)
- Gauss-Seidel iteration: $\boldsymbol{M} = \boldsymbol{L} + \boldsymbol{D}$ ($\boldsymbol{N} = -U$)
- SOR iteration: Gauss-Seidel + relaxation
- SSOR: two (forward and backward) SOR steps
- Rate of convergence $R_\infty(\boldsymbol{G})$ for $-\nabla^2 u = f$ in 2D with $u = 0$ as BC:
  - Jacobi: $\pi h^2/2$
  - Gauss-Seidel: πh^2
  - SOR: $\pi 2h$
  - SSOR: $> \pi h$

  SOR/SSOR is superior (h vs. h^2, $h \to 0$ is small)

Jacobi iteration

- $\boldsymbol{M = D}$
- Put everything, except the diagonal, on the rhs
- 2D Poisson equation $-\nabla^2 u = f$:

$$u_{i,j-1} + u_{i-1,j} + u_{i+1,j} + u_{i,j+1} - 4u_{i,j} = -h^2 f_{i,j}$$

- Solve for diagonal element and use old values on the rhs:

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

for $k = 1, 2, \ldots$

Relaxed Jacobi iteration

- Idea: Computed new $\boldsymbol{x}$ approximation $\boldsymbol{x}^*$ from

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

- Set

$$\boldsymbol{x}^k = \omega\boldsymbol{x}^* + (1-\omega)\boldsymbol{x}^{k-1}$$

- weighted mean of $\boldsymbol{x}^{k-1}$ and $\boldsymbol{x}^k$ if $\omega \in (0,1)$

Relation to explicit time stepping

- Relaxed Jacobi iteration for $-\nabla^2 u = f$ is equivalent with solving

$$\alpha \frac{\partial u}{\partial t} = \nabla^2 u + f$$

  by an explicit forward scheme until $\partial u/\partial t \approx 0$, provided $\omega = 4\Delta t/(\alpha h)^2$
- Stability for forward scheme implies $\omega \leq 1$
- In this example: $\omega = 1$ best ($\Leftrightarrow$ largest Δt)
- Forward scheme for $t \to \infty$ is a slow scheme, hence Jacobi iteration is slow

Gauss-Seidel/SOR iteration

- $\boldsymbol{M} = \boldsymbol{L} + \boldsymbol{D}$
- For our 2D Poisson eq. scheme:

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

  i.e. solve for diagonal term and use the most recently computed values on the right-hand side

- SOR is relaxed Gauss-Seidel iteration:
  - compute $\boldsymbol{x}^*$ from Gauss-Seidel it.
  - set $\boldsymbol{x}^k = \omega \boldsymbol{x}^* + (1-\omega)\boldsymbol{x}^{k-1}$

  $\omega \in (0,2)$, with $\omega = 2 - \mathcal{O}(h^2)$ as optimal choice
- Very easy to implement!

Symmetric/double SOR: SSOR

- SSOR = Symmetric SOR
- One (forward) SOR sweep for unknowns $1, 2, 3, \ldots, n$
- One (backward) SOR sweep for unknowns $n, n-1, n-2, \ldots, 1$
- $\boldsymbol{M}$ can be shown to be

$$\boldsymbol{M} = \frac{1}{2-\omega}\left(\frac{1}{\omega}\boldsymbol{D} + \boldsymbol{L}\right)\left(\frac{1}{\omega}\boldsymbol{D}\right)^{-1}\left(\frac{1}{\omega}\boldsymbol{D} + \boldsymbol{U}\right)$$

- Notice that each factor in $\boldsymbol{M}$ is diagonal or lower/upper triangular ($\Rightarrow$ very easy to solve systems $\boldsymbol{M}\boldsymbol{y} = \boldsymbol{z}$)

Status: classical iterative methods

- Jacobi, Gauss-Seidel/SOR, SSOR are too slow for paractical PDE computations
- The simplest possible solution method for $-\nabla^2 u = f$ and other stationary PDEs in 2D/3D is to use SOR
- Classical iterative methods converge quickly in the beginning but slow down after a few iterations
- Classical iterative methods are important ingredients in multigrid methods

Conjugate Gradient-like methods

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \quad \boldsymbol{A} \in \mathbb{R}^{n,n},\ \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n\ .$$

- Use a Galerkin or least-squares method to solve a linear system (!)
- Idea: write

$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} + \sum_{j=1}^{k} \alpha_j \boldsymbol{q}_j$$

  α_j: unknown coefficients, $\boldsymbol{q}_j$: known vectors

- Compute the residual:

$$\boldsymbol{r}^k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k = \boldsymbol{r}^{k-1} - \sum_{j=1}^{k} \alpha_j \boldsymbol{A}\boldsymbol{q}_j$$

  and apply the *ideas* of the Galerkin or least-squares methods

Galerkin

- Residual:

$$\boldsymbol{r}^k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k = \boldsymbol{r}^{k-1} - \sum_{j=1}^{k} \alpha_j \boldsymbol{A}\boldsymbol{q}_j$$

$$(\boldsymbol{r}^k, \boldsymbol{q}_i) = 0$$

- Galerkin's method ($\boldsymbol{r} \sim R$, $\boldsymbol{q}_j \sim N_j$, $\alpha_j \sim u_j$):

$$(\boldsymbol{r}^k, \boldsymbol{q}_i) = 0, \quad i = 1, \ldots, k$$

$(\cdot, \cdot)$: Eucledian inner product

- Result: linear system for α_j,

$$\sum_{j=1}^{k} (\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{q}_j)\alpha_j = (\boldsymbol{r}^{k-1}, \boldsymbol{q}_i), \quad i = 1, \ldots, k$$

Least squares

- Residual:

$$\boldsymbol{r}^k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k = \boldsymbol{r}^{k-1} - \sum_{j=1}^{k} \alpha_j \boldsymbol{A}\boldsymbol{q}_j$$

$$\frac{\partial}{\partial \alpha_i}(\boldsymbol{r}^k, \boldsymbol{r}^k) = 0$$

- Least squares: minimize $(\boldsymbol{r}^k, \boldsymbol{r}^k)$
- Result: linear system for α_j:

$$\sum_{j=1}^{k} (\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{A}\boldsymbol{q}_j)\alpha_j = (\boldsymbol{r}^{k-1}, \boldsymbol{A}\boldsymbol{q}_i), \quad i = 1, \ldots, k$$

The nature of the methods

- Start with a guess $\boldsymbol{x}^0$
- In iteration k: seek $\boldsymbol{x}^k$ in a k-dimensional vector space V_k
- Basis for the space: $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k$
- Use Galerkin or least squares to compute the (optimal) approximation $\boldsymbol{x}^k$ in V_k
- Extend the basis from V_k to V_{k+1} (i.e. find $\boldsymbol{q}_{k+1}$)

Extending the basis

- V_k is normally selected as a so-called Krylov subspace:

$$V_k = \text{span}\{\boldsymbol{r}^0, \boldsymbol{A}\boldsymbol{r}^0, \ldots, \boldsymbol{A}^{k-1}\boldsymbol{r}^0\}$$

- Alternatives for computing $\boldsymbol{q}_{k+1} \in V_{k+1}$:

$$\boldsymbol{q}_{k+1} = \boldsymbol{r}^k + \sum_{j=1}^{k} \beta_j \boldsymbol{q}_j$$

$$\boldsymbol{q}_{k+1} = \boldsymbol{A}q_k + \sum_{j=1}^{k} \beta_j \boldsymbol{q}_j$$

  The first dominates in frequently used algorithms – only that choice is used hereafter

- How to choose β_j?

Orthogonality properties

- Bad news: must solve a $k \times k$ linear system for α_j in each iteration (as $k \to n$ the work in each iteration approach the work of solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$!)
- The coefficient matrix in the α_j system:

$$(\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{q}_j), \quad (\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{A}\boldsymbol{q}_j)$$

- Idea: make the coefficient matrices diagonal
- That is,
  - Galerkin: $(\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{q}_j) = 0$ for $i \neq j$
  - Least squares: $(\boldsymbol{A}\boldsymbol{q}_i, \boldsymbol{A}\boldsymbol{q}_j) = 0$ for $i \neq j$
- Use β_j to enforce orthogonality of $\boldsymbol{q}_i$

Formula for updating the basis vectors

- Define

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle \equiv (\boldsymbol{A}\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^T \boldsymbol{A}\boldsymbol{v}$$

  and

$$[\boldsymbol{u}, \boldsymbol{v}] \equiv (\boldsymbol{A}\boldsymbol{u}, \boldsymbol{A}\boldsymbol{v}) = \boldsymbol{u}^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{v}$$

- Galerkin: require $\boldsymbol{A}$-orthogonal q_j vectors, which then results in

$$\beta_i = -\frac{\langle \boldsymbol{r}^k, \boldsymbol{q}_i \rangle}{\langle \boldsymbol{q}_i, \boldsymbol{q}_i \rangle}$$

- Least squares: require $\boldsymbol{A}^T \boldsymbol{A}$–orthogonal q_j vectors, which then results in

$$\beta_i = -\frac{[\boldsymbol{r}^k, \boldsymbol{q}_i]}{[\boldsymbol{q}_i, \boldsymbol{q}_i]}$$

Simplifications

- Galerkin: $\langle \boldsymbol{q}_i, \boldsymbol{q}_j \rangle = 0$ for $i \neq j$ gives

$$\alpha_k = \frac{(\boldsymbol{r}^{k-1}, \boldsymbol{q}_k)}{\langle \boldsymbol{q}_k, \boldsymbol{q}_k \rangle}$$

  and $\alpha_i = 0$ for $i < k$ (!):

$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} + \alpha_k \boldsymbol{q}_k$$

- That is, hand-derived formulas for α_j
- Least squares:

$$\alpha_k = \frac{(\boldsymbol{r}^{k-1}, \boldsymbol{A}\boldsymbol{q}_k)}{[\boldsymbol{q}_k, \boldsymbol{q}_k]}$$

  and $\alpha_i = 0$ for $i < k$

Symmetric A

If $\boldsymbol{A}$ is symmetric ($\boldsymbol{A}^T = \boldsymbol{A}$) and positive definite (positive eigenvalues $\Leftrightarrow \boldsymbol{y}^T \boldsymbol{A}\boldsymbol{y} > 0$ for any $\boldsymbol{y} \neq 0$), also $\beta_i = 0$ for $i < k$
$\Rightarrow$ need to store $\boldsymbol{q}_k$ only
($\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{k-1}$ are not used in iteration k)

Summary: least squares algorithm

- given a start vector $\boldsymbol{x}^0$,
  compute $\boldsymbol{r}^0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^0$ and set $\boldsymbol{q}_1 = \boldsymbol{r}^0$.
  for $k = 1, 2, \ldots$ until termination criteria are fulfilled:
  $\alpha_k = (\boldsymbol{r}^{k-1}, \boldsymbol{A}\boldsymbol{q}_k)/[\boldsymbol{q}_k, \boldsymbol{q}_k]$
  $\boldsymbol{x}^k = \boldsymbol{x}^{k-1} + \alpha_k \boldsymbol{q}_k$
  $\boldsymbol{r}^k = \boldsymbol{r}^{k-1} - \alpha_k \boldsymbol{A}\boldsymbol{q}_k$
  if $\boldsymbol{A}$ is symmetric then
  $\beta_k = [\boldsymbol{r}^k, \boldsymbol{q}_k]/[\boldsymbol{q}_k, \boldsymbol{q}_k]$
  $\boldsymbol{q}_{k+1} = \boldsymbol{r}^k - \beta_k \boldsymbol{q}_k$
  else
  $\beta_j = [\boldsymbol{r}^k, \boldsymbol{q}_j]/[\boldsymbol{q}_j, \boldsymbol{q}_j], \quad j = 1, \ldots, k$
  $\boldsymbol{q}_{k+1} = \boldsymbol{r}^k - \sum_{j=1}^{k} \beta_j \boldsymbol{q}_j$
- The Galerkin-version requires $\boldsymbol{A}$ to be symmetric and positive definite and results in the famous Conjugate Gradient method

Truncation and restart

- Problem: need to store $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k$
- Much storage and computations when k becomes large
- Truncation: work with a truncated sum for $\boldsymbol{x}^k$,

$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} + \sum_{j=k-K+1}^{k} \alpha_j \boldsymbol{q}_j$$

  where a possible choice is $K = 5$
- Small K might give convergence problems
- Restart: restart the algorithm after K iterations (alternative to truncation)

Family of methods

- Generalized Conjugate Residual method
  = least squares + restart
- Orthomin method
  = least squares + truncation
- Conjugate Gradient method
  = Galerkin + symmetric and positive definite A
- Conjugate Residuals method
  = Least squares + symmetric and positive definite A
- Many other related methods: BiCGStab, Conjugate Gradients Squared (CGS), Generalized Minimum Residuals (GMRES), Minimum Residuals (MinRes), SYMMLQ
- Common name: Conjugate Gradient-like methods

Convergence

- Conjugate Gradient-like methods converge slowly (but usually faster than SOR/SSOR)
- To reduce the initial error by a factor ϵ,

$$\frac{1}{2}\ln\frac{2}{\epsilon}\sqrt{\kappa}$$

iterations are needed, where κ is the condition number:

$$\kappa = \frac{\text{largest eigenvalue of } \boldsymbol{A}}{\text{smalles eigenvalue of } \boldsymbol{A}}$$

- $\kappa = \mathcal{O}(h^{-2})$ when solving 2nd-order PDEs (incl. elasticity and Poisson eq.)

Preconditioning

- Idea: Introduce an equivalent system

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}$$

  solve it with a Conjugate Gradient-like method and construct $\boldsymbol{M}$ such that

  1. $\kappa = \mathcal{O}(1) \;\Rightarrow\; \boldsymbol{M} \approx \boldsymbol{A}$ (i.e. fast convergence)
  2. $\boldsymbol{M}$ is cheap to compute
  3. $\boldsymbol{M}$ is sparse (little storage)
  4. systems $\boldsymbol{M}\boldsymbol{y} = \boldsymbol{z}$ (occuring in the algorithm due to $\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{v}$-like products) are efficiently solved ($\mathcal{O}(n)$ op.)

  Contradictory requirements!

- The preconditioning business: find a good balance between 1-4

Classical methods as preconditioners

- Idea: "solve" $\boldsymbol{My = z}$ by one iteration with a classical iterative method (Jacobi, SOR, SSOR)
- Jacobi preconditioning: $\boldsymbol{M = D}$ (diagonal of $\boldsymbol{A}$)
- No extra storage as $\boldsymbol{M}$ is stored in $\boldsymbol{A}$
- No extra computations as $\boldsymbol{M}$ is a part of $\boldsymbol{A}$
- Efficient solution of $\boldsymbol{My = z}$
- But: $\boldsymbol{M}$ is probably not a good approx to $\boldsymbol{A}$
  $\Rightarrow$ poor quality of this type of preconditioners?
- Conjugate Gradient method + SSOR preconditioner is widely used

M as a factorization of A

- Idea: Let M be an LU-factorization of A, i.e.,

$$M = LU$$

  where L and U are lower and upper triangular matrices resp.

- Implications:
  1. $M = A$ ($\kappa = 1$): very efficient preconditioner!
  2. M is not cheap to compute
     (requires Gaussian elim. on A!)
  3. M is not sparse (L and U are dense!)
  4. systems $My = z$ are not efficiently solved
     ($\mathcal{O}(n^2)$ process when L and U are dense)

M as an incomplete factorization of A

- New idea: compute sparse $\boldsymbol{L}$ and $\boldsymbol{U}$
- How? compute only with nonzeroes in $\boldsymbol{A}$

$\Rightarrow$ Incomplete factorization, $\boldsymbol{M} = \widehat{\boldsymbol{L}}\widehat{\boldsymbol{U}} \neq \boldsymbol{LU}$

- $\boldsymbol{M}$ is not a perfect approx to $\boldsymbol{A}$
- $\boldsymbol{M}$ is cheap to compute and store ($\mathcal{O}(n)$ complexity)
- $\boldsymbol{My} = \boldsymbol{z}$ is efficiently solved ($\mathcal{O}(n)$ complexity)
- This method works well - much better than SOR/SSOR preconditioning

How to compute M

- Run through a standard Gaussian elimination, which factors $\boldsymbol{A}$ as $\boldsymbol{A} = \boldsymbol{LU}$
- Normally, $\boldsymbol{L}$ and $\boldsymbol{U}$ have nonzeroes where $\boldsymbol{A}$ has zeroes
- Idea: let $\boldsymbol{L}$ and $\boldsymbol{U}$ be as sparse as $\boldsymbol{A}$
- Compute only with the nonzeroes of $\boldsymbol{A}$
- Such a preconditioner is called Incomplete LU Factorization, ILU
- Option: add contributions outside $\boldsymbol{A}$'s sparsity pattern to the diagonal, multiplied by ω
- Relaxed Incomplete Factorization (RILU): $\omega > 1$
- Modified Incomplete Factorization (MILU): $\omega = 1$
- See algorithm C.3 in the book

Numerical experiments

- Two test cases:
  - $-\nabla^2 u = f$ on the unit cube and FDM
  - $-\nabla^2 u = f$ on the unit cube and FEM

Test case 1: 3D FDM Poisson eq.

- Equation: $-\nabla^2 u = 1$
- Boundary condition: $u = 0$
- 7-pt star standard finite difference scheme
- Grid size: $20 \times 20 \times 20 = 8000$ points and $20 \times 30 \times 30 = 27000$ points

Jacobi vs. SOR vs. SSOR

- $n = 20^3 = 8000$ and $n = 30^3 = 27000$
- Jacobi: not converged in 1000 iterations
- SOR($\omega = 1.8$): 2.0s and 9.2s
- SSOR($\omega = 1.8$): 1.8s and 9.8s
- Gauss-Seidel: 13.2s and 97s
- SOR's sensitivity to relax. parameter ω:
  1.0: 96s, 1.6: 23s, 1.7: 16s, 1.8: 9s, 1.9: 11s
- SSOR's sensitivity to relax. parameter ω:
  1.0: 66s, 1.6: 17s, 1.7: 13s, 1.8: 9s, 1.9: 11s

$\Rightarrow$ relaxation is important,
great sensitivity to ω

Conjugate Residuals or Gradients?

- Compare Conjugate Residuals with Conjugate Gradients
- Or: least squares vs. Galerkin
- MinRes: not converged in 1000 iterations
- ConjGrad: 0.7s and 3.9s

$\Rightarrow$ ConjGrad is clearly faster than the best SOR/SSOR

- Add ILU preconditioner
- MinRes: 0.7s and 4s
- ConjGrad: 0.6s and 2.7s
- The importance of preconditioning grows as n grows

Different preconditioners

- ILU, Jacobi, SSOR preconditioners ($\omega = 1.2$)
- MinRes:
  Jacobi: not conv., SSOR: 11.4s, ILU: 4s
- ConjGrad:
  Jacobi: 4.8s, SSOR: 2.8s, ILU: 2.7s
- Sensitivity to relax. parameter in SSOR, with ConjGrad as solver:
  1.0: 3.3s, 1.6: 2.1s, 1.8: 2.1s, 1.9: 2.6s
- Sensitivity to relax. parameter in RILU, with ConjGrad as solver:
  0.0: 2.7s, 0.6: 2.4s, 0.8: 2.2s, 0.9: 1.9s,
  0.95: 1.9s, 1.0: 2.7s

$\Rightarrow$ ω slightly less than 1 is optimal,
RILU and SSOR are equally fast (here)

Jacobi vs. SOR vs. SSOR

- $n = 9261$ and $n = 30^3 = 29791$, trilinear and triquadratic elms.
- Jacobi: not converged in 1000 iterations
- SOR($\omega = 1.8$): 9.1s and 81s, 42s and 338s
- SSOR($\omega = 1.8$): 47s and 248s, 138s and 755s
- Gauss-Seidel: not converged in 1000 iterations
- SOR's sensitivity to relax. parameter ω:
  1.0: not conv., 1.6: 200s, 1.8: 83s, 1.9: 57s
  ($n = 29791$ and trilinear elements)
- SSOR's sensitivity to relax. parameter ω:
  1.0: not conv., 1.6: 212s, 1.7: 207s, 1.8: 245s, 1.9: 435s
  ($n = 29791$ and trilinear elements)

$\Rightarrow$ relaxation is important,
great sensitivity to ω

Conjugate Residuals or Gradients?

- Compare Conjugate Residuals with Conjugate Gradients
- Or: least squares vs. Galerkin
- MinRes: not converged in 1000 iterations
- 9261 vs 29791 unknowns, trilinear elements
- ConjGrad: 5s and 22s

$\Rightarrow$ ConjGrad is clearly faster than the best SOR/SSOR!

- Add ILU preconditioner
- MinRes: 5s and 28s
- ConjGrad: 4s and 16s
- ILU prec. has a greater impact when using triquadratic elements (and when n grows)

Different preconditioners

- ILU, Jacobi, SSOR preconditioners ($\omega = 1.2$)
- MinRes:
  Jacobi: 68s., SSOR: 57s, ILU: 28s
- ConjGrad:
  Jacobi: 19s, SSOR: 14s, ILU: 16s
- Sensitivity to relax. parameter in SSOR, with ConjGrad as solver:
  1.0: 17s, 1.6: 12s, 1.8: 13s, 1.9: 18s
- Sensitivity to relax. parameter in RILU, with ConjGrad as solver:
  0.0: 16s, 0.6: 15s, 0.8: 13s, 0.9: 12s, 0.95: 11s, 1.0: 16s

$\Rightarrow$ ω slightly less than 1 is optimal,
RILU and SSOR are equally fast (here)

More experiments

- Convection-diffusion equations:
  $NOR/doc/Book/src/app/Cd/Verify
- Files: linsol_a.i etc as for LinSys4 and Poisson2
- Elasticity equations:
  $NOR/doc/Book/src/app/Elasticity1/Verify
- Files: linsol_a.i etc as for the others
- Run experiments and learn!

Multigrid methods

- Multigrid methods are the most efficient methods for solving linear systems
- Multigrid methods have optimal complexity $\mathcal{O}(n)$
- Multigrid can be used as stand-alone solver or preconditioner
- Multigrid applies a hierarchy of grids
- Multigrid is not as robust as Conjugate Gradient-like methods and incomplete factorization as preconditioner, but faster when it works
- Multigrid is complicated to implement

The rough ideas of multigrid

- Observation: e.g. Gauss-Seidel methods are very efficient during the first iterations
- High-frequency errors are efficiently damped by Gauss-Seidel
- Low-frequence errors are slowly reduced by Gauss-Seidel
- Idea: jump to a coarser grid such that low-frequency errors get higher frequency
- Repeat the procedure
- On the coarsest grid: solve the system exactly
- Transfer the solution to the finest grid
- Iterate over this procedure

Damping in Gauss-Seidel's method (1)

- Model problem: $-u'' = f$ by finite differences:

$$-u_{j-1} + 2u_j - u_{j+1} = h^2 f_j$$

solved by Gauss-Seidel iteration:

$$2u_j^\ell = u_{j-1}^\ell + u_{j+1}^{\ell-1} + h^2 f_j$$

- Study the error $e_i^\ell = u_i^\ell - u_i^\infty$:

$$2e_j^\ell = e_{j-1}^\ell + e_{j+1}^{\ell-1}$$

- This is like a time-dependent problem, where the iteration index ℓ is a pseudo time

Damping in Gauss-Seidel's method (2)

- Can find e_j^ℓ with techniques from Appendix A.4:

$$e_j^\ell = \sum_k A_k \exp\left(i(kjh - \tilde{\omega}\ell\Delta t)\right)$$

or (easier to work with here):

$$e_j^\ell = \sum_k A_k \xi^\ell \exp\left(ikjh\right), \quad \xi = \exp\left(-i\tilde{\omega}\Delta t\right)$$

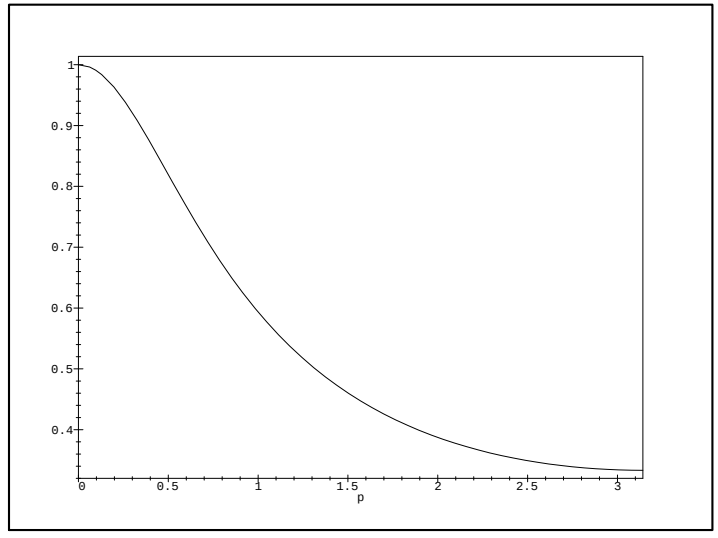
- Inserting a wave component in the scheme:

$$\xi = \exp\left(-i\tilde{\omega}\Delta t\right) = \frac{\exp\left(ikh\right)}{2 - \exp\left(-ikh\right)}, \quad |\xi| = \frac{1}{\sqrt{5 - 4\cos kh}}$$

- Interpretation of $|\xi|$: reduction in the error per iteration

Gauss-Seidel's damping factor

$$|\xi| = \frac{1}{\sqrt{5 - 4\cos p}}, \quad p = kh \in [0, \pi]$$



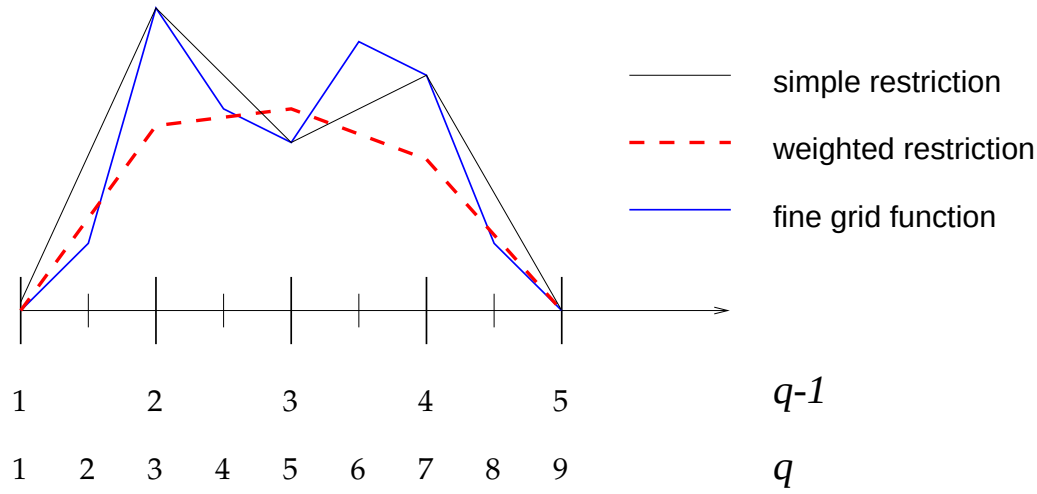
- Small $p = kh \sim h/\lambda$: low frequency (relative to the grid) and small damping
- Large ($\to \pi$) $p = kh \sim h/\lambda$: high frequency (relative to the grid) and efficient damping

More than one grid

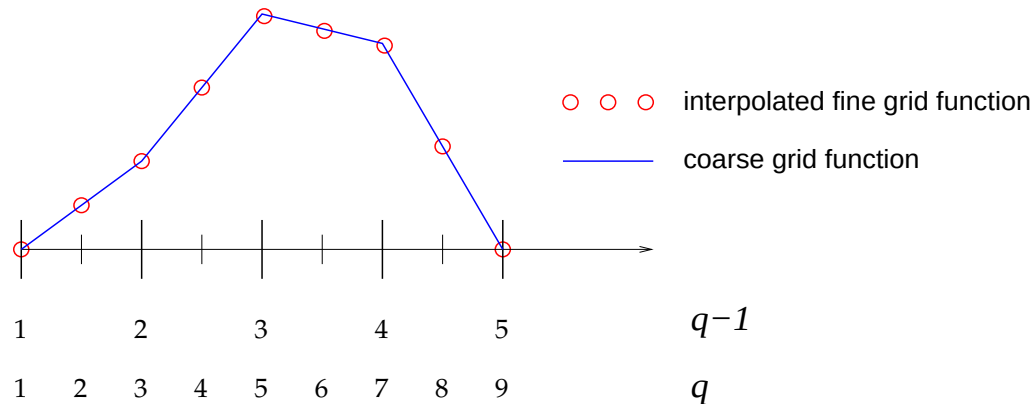
- From the previous analysis: error components with high frequency are quickly damped
- Jump to a coarser grid, e.g. $h' = 2h$
- p is increased by a factor of 2, i.e., not so high-frequency waves on the h grid is efficiently damped by Gauss-Seidel on the h' grid
- Repeat the procedure
- On the coarsest grid: solve by Gaussian elimination
- Interpolate solution to a finer grid, perform Gauss-Seidel iterations, and repeat until the finest grid is reached

Transferring the solution between grids

- From fine to coarser: restriction



- From coarse to finer: prolongation



Smoothers

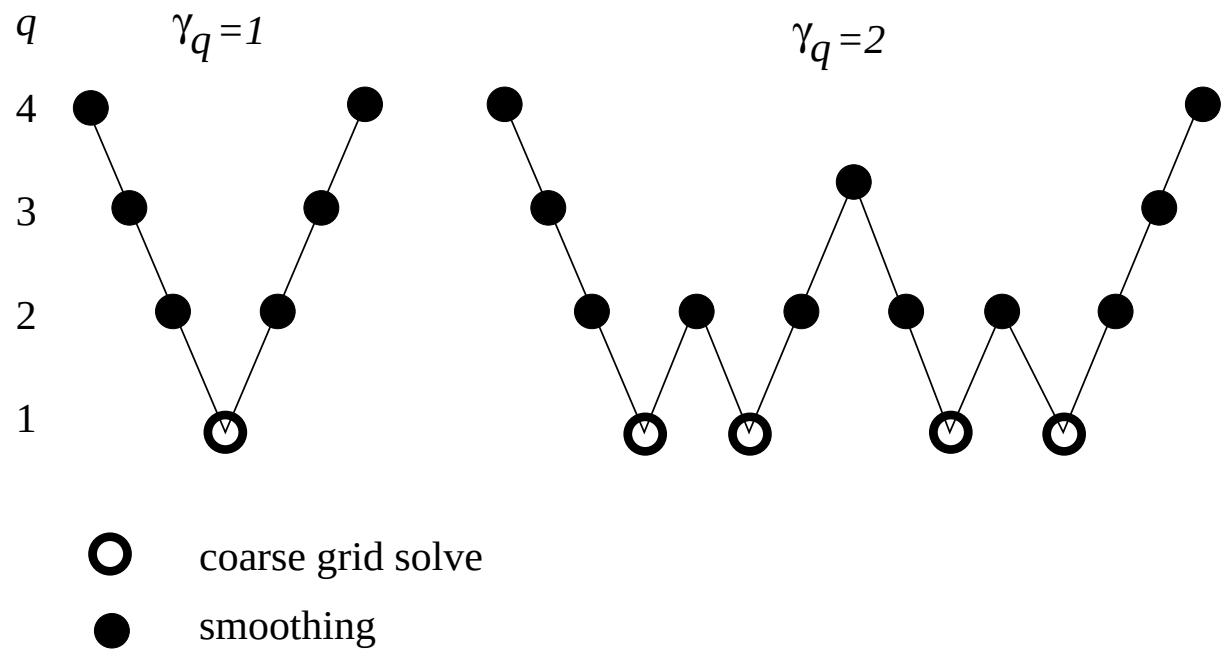
- The Gauss-Seidel method is called a smoother when used to damp high-frequency error components in multigrid
- Other smoothers: Jacobi, SOR, SSOR, incomplete factorization
- No of iterations is called no of smoothing sweeps
- Common choice: one sweep

A multigrid algorithm

- Start with the finest grid
- Perform smoothing (pre-smoothing)
- Restrict to coarser grid
- Repeat the procedure (recursive algorithm!)
- On the coarsest grid: solve accurately
- Prolongate to finer grid
- Perform smoothing (post-smoothing)
- One cycle is finished when reaching the finest grid again
- Can repeat the cycle
- Multigrid solves the system in $\mathcal{O}(n)$ operations

V- and W-cycles

Different strategies for constructing cycles:



Multigrid requires flexible software

- Many ingredients in multigrid:
  - pre- and post-smoother
  - no of smoothing sweeps
  - solver on the coarsest level
  - cycle strategy
  - restriction and prolongation methods
  - how to construct the various grids?
- There are also other variants of multigrid (e.g. for nonlinear problems)
- The optimal combination of ingredients is only known for simple model problems (e.g. the Poisson eq.)
- In general: numerical experimentation is required!