High Performance and Parallel Computing (MA 6252)

Lecture 6 & 7

Algorithms for the Fast Parallel Solution of PDE

Ulrich Rüde

Lehrstuhl für Simulation
Universität Erlangen-Nürnberg

visiting Professor at
Department of Mathematics
National University of Singapore

Contents of High Performance and Parallel Computing

- Introduction
  - Computational Science and Engineering
  - High Performance Computing
- Computer Architecture
  - memory hierarchy
  - pipeline
  - instruction level parallelism
  - multi-core systems
  - parallel clusters
- Efficient Programming
  - computational complexity
  - efficient coding
  - architecture aware programming
  - shared memory parallel programming (OpenMP)
  - distributed memory parallel programming (MPI)
  - parallel programming with Graphics Cards

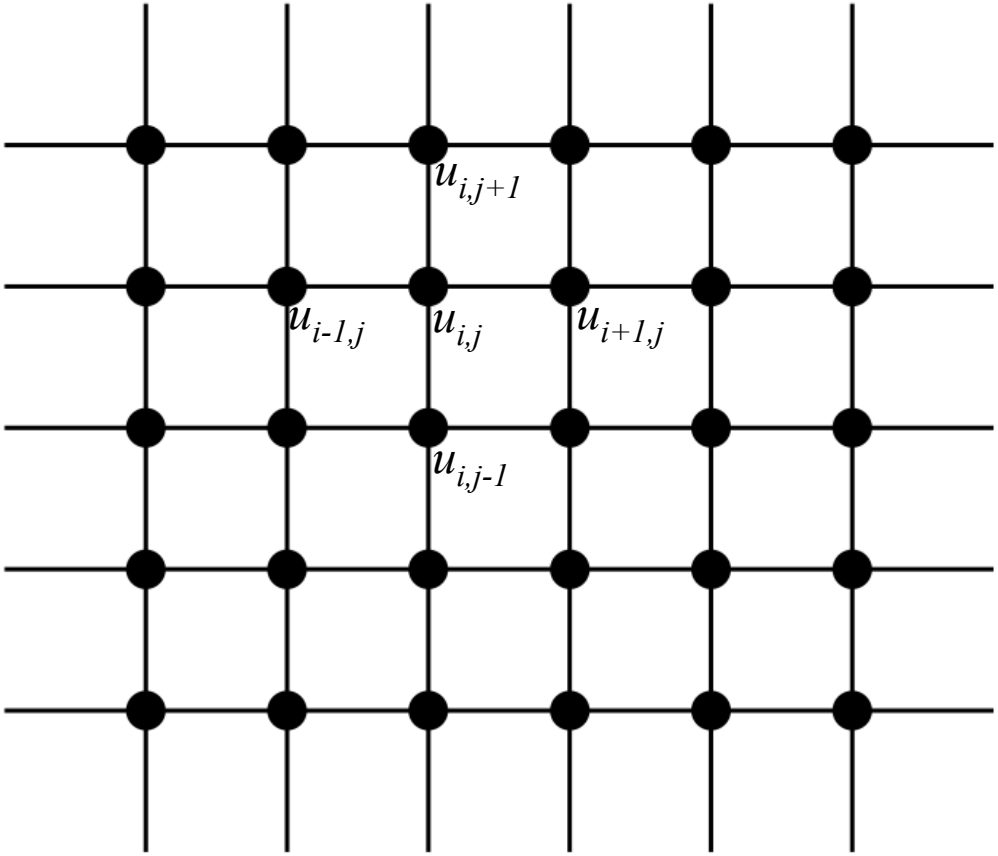
Algorithms

for the fast parallel

Solution of PDE

Example for use of an iterative method

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) \quad \text{for } i,j = 1...n$$



More about matrix representation later.

Grid of wires (as replacement of a 2D plate)

Temperature at each node = Average of neighboring temperatures

Boundary values given.

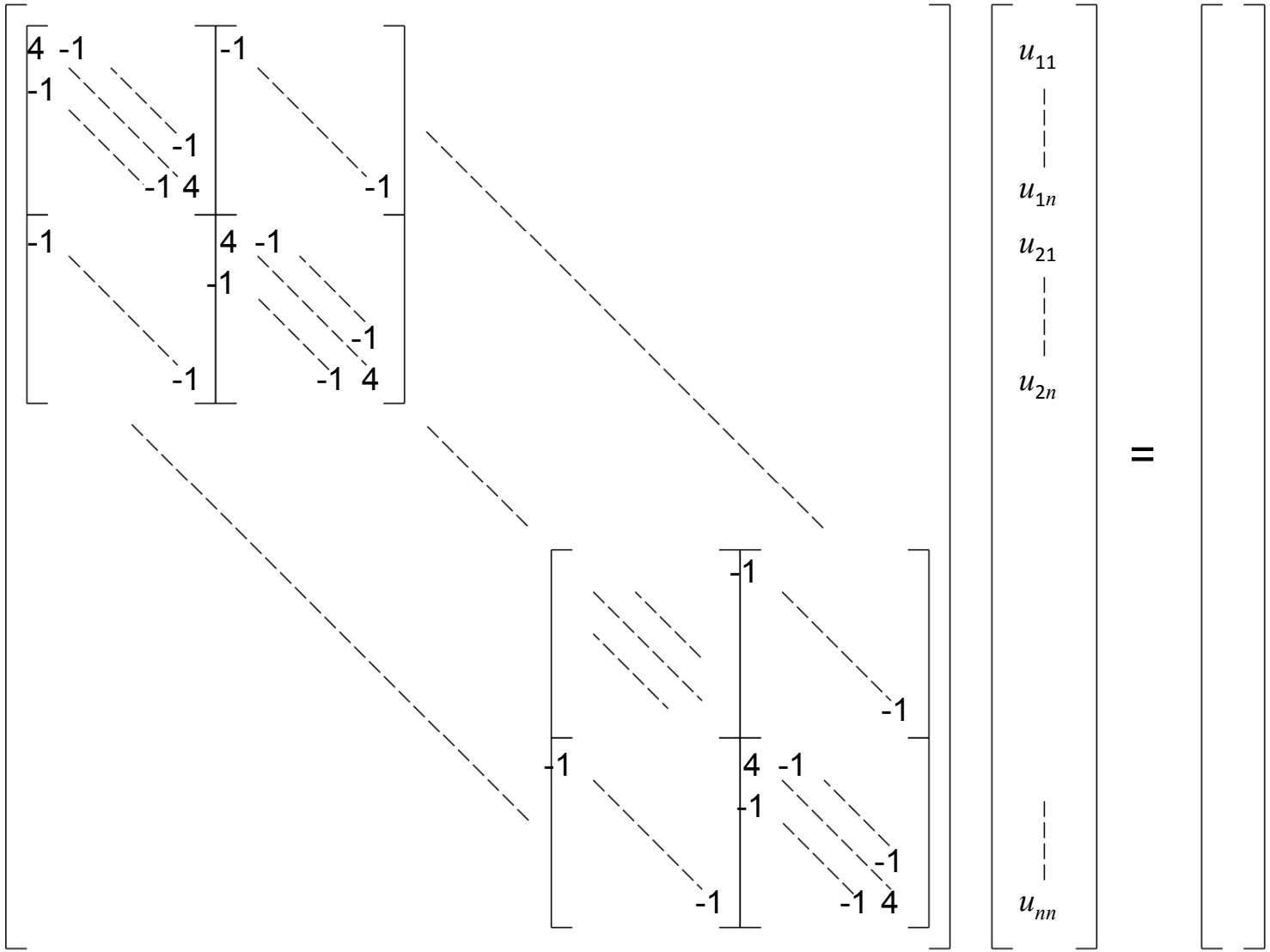
This is (indirectly) a discretization of the Laplace- or Poisson-equation.

Iterative Methods (example)

- Many Generalizations
  - Modifiied equations with weighted averages can simulate a grid (wire frame) with thicker or thinner wires
  - Asymmetry in equations could be used to simulate convective transport
  - Have more connections, e.g. wires that run diagonally.
  - Etc. etc.
- The mathematical problem is the solution of elliptic PDE of type

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \quad \text{in } \Omega = (0,1)^2$$

Matrix representation



Towards the Linear System

- Please note: the $n \times n$ grid is not a matrix. The u-values in the grid are the unknowns.
- Use well-defined terms: Grid or Mesh versus *Matrix*
- The matrix has size $n^2 \times n^2$.
- The right hand side contains
  - given „boundary values“ and
  - right hand side values (if the problem is a bit generalized).
- We could now assemble the matrix in a suitable sparse matrix data structure, but here it is more efficient (and simpler) to use a „matrix-free“ approach
- Implementation of an iterative method

Some properties of the matrix

- symmetric (reflecting that the PDE is self-adjoint)
- positive definite (reflecting that it discretizes a coercive elliptic)
  - thus no pivoting is needed when performing elimination
- sparse (reflecting that we discretize a PDE)
- banded, reflecting that the grid is cartesian
- the theorem of Gershgorin tells us that all eigenvalues ≥ 0
- in the special case, in fact all eigenvalues can be computed analytically (using a discrete Fourier transform that diagonalizes the matrix).
- Or a strengthened form of Gershgorin theorem due to O. Taussky can be applied that states:
  - If the matrix is irreducible and an eigenvalue lies on the boundary of the union of all Gershgorin discs then it lies on the bounday of each Gershgorin circle.

Further generalizations of the example

- Scalar, linear, elliptic partial differential equations in 2D (oder also 3D).
- Applications:
  - Electrostatics, Mechanics, Energy and Heat Transfer, Fundamantal Physics and Chemistry (e.g. Gravitaty potential in astrophysics or Coulomb potential in molecular dynamics),
  - Fluid mechanics (pressure equation)
  - Environmental engineering (ground water flow, pollution dynamics),
  - Medicine (bioelectrical fields),
  - ... and many more
- Here in an elementary context:
  - Approximations for the partial derivatives by finite differences
  - Similar structure for Finite Elements or Finite Volumes

Solvers for sparse linear systems

Direct Methods

- Elimination-based
  - full matrix
  - banded matrix
  - sparse matrix
  - nested dissection
  - multi-frontal
- Transform based
  - FFT
  - Fast direct solvers (Bunemann)

Iterative Methods

- Linear (splitting-type)
  - Richardson
  - Gauss-Seidel
  - Jacobi
  - SOR
- Krylov-Space
  - Conjugate Gradient
  - GMRES
  - ... many variants
- Multilevel
  - multigrid
  - multipole

Recapitulation: Direct Linear Systems Solvers

This is *THE* Standard Algorithm: Gaussian Elimination

- direct method: i.e. it computes the exact result (except for roundoff errors) after a finite number of operations
- complexity (for dense matrices): $O(n^3)$
- how does it work:
  - for all $O(n^2)$ elements of A below the diagonal: a_{ij}
    - perform a „*row elimination step*“ by adding a suitable multiple of the „pivot“ row to the row i with the goal of annihilating the element a_{ij}
  - Usually pivoting necessary (to keep roundoff errors low)
  - Must be done in the right order as not to destroy already created zeros
  - Elimination also for rhs b
  - or LU-decomposition (and forward subsitution)
  - backward substitution

General Goal

- Solve the linear system

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

- If the matrix is non-singular, the system has a unique solution x
- But ... not all linear systems are created equal.
  - Developing specialised solvers for large finite elment problems has been (and will stay) an active research topic for decades
- The key is to exploit the special features of FE systems

Gaussian Elimination for FE-Systems?

- In its classical form, Gaussian Elimination is much too expensive for FE problems: Complexity $O(n^3)$
- Example: $n=10^6$:
  - $n^3 = 10^{18}$ - this corresponds to 10^8 Seconds (on a 10 Gigaflops PC) = 3.1 years of compute time
  - The best known multigrid algorithms (for the 2D problem from this class) needs about $30n$ operations, so it should compute the solution in less about 3 milliseconds (which is NOT at all easy to do!)
  - But: even with the fastest supercomputers Gaussian Elimination would be much slower than multigrid on a laptop.
- So Gaussian Elimination cannot be used for FE systems!?
  - Gaussian Elimination *CAN be used* - in fact it is state of the art in most commerical FE packages (such as NASTRAN)
  - It must be modified to *exploit existing zeros* in the FE-matrix

Gaussian Elimination for FE-Systems (1)?

- In its classical form, Gaussian Elimination is much too expensive for FE problems: Complexity $O(n^3)$
- Example: $n=10^6$:
  - $n^3 = 10^{18}$ - this corresponds to 10^8 Seconds (on a 10 Gigaflops PC) = 3.1 years of compute time
  - The best known multigrid algorithms (for the 2D problem from this class) needs about $30n$ operations, so it should compute the solution in less about 3 milliseconds (which is NOT at all easy to do!)
  - But: even with the fastest supercomputers Gaussian Elimination would be much slower than multigrid on a laptop.
- So Gaussian Elimination cannot be used for FE systems!?
  - Gaussian Elimination *CAN be used* - in fact it is state of the art in most commerical FE packages (such as NASTRAN)
  - It must be modified to *exploit existing zeros* in the FE-matrix

Gaussian Elimination for FE-Systems?

- For 1D (ODE) problems, the stiffness matrix is „tridiagonal“:

$$A = \begin{bmatrix} * & * & & & \\ * & * & \ddots & & \\ & \ddots & \ddots & * & \\ & & * & * & * \\ & & & * & * \end{bmatrix}$$

  - The 0-s below the subdiagonal can be exploited: they need not be eliminated - they are never touched - they need not be stored - a row elimination reduces to a few elementary operations - the whole Gaussian Elimination then has reduced complexity of $O(n)$ - the algorithm is perfectly suited for this - no need to use an iterative method here.

Gaussian Elimination for banded FE-Systems

- Fill in destroys zeros within „band“ when eliminating.
- In our case in 2D: $N = n^2$: band width n
  - Cost: $O(n^2 \times n)$ elements to eliminate, each cost $O(n)$ results in total cost $O(n^4) = O(N^2)$ Flops
  - Cost $O(n^2 \times n) = O(N^{3/2})$ memory
- In 3D: $N = n^3$: band width n^2
  - Cost: $O(n^3 \times n^2)$ elements to eliminate, each cost $O(n^2)$ results in total cost $O(n^7) = O(N^{7/3})$ Flops
  - Cost $O(n^3 \times n^2) = O(N^{5/3})$ memory

- Are there more efficient alternatives
  - note that ordering of unknows is arbitrary and other orderings may lead to better elimination schedules, reducing fill in.

Gaussian Elimination for FE-Systems

- For 2D (PDE) problems, the stiffness matrix has more complicated structure
- The matrix is still *sparse*: e.g. only 5 or 9 entries in each row may be nonzero.
- A matrix is called *sparse*, if it contains „many more“ zero entries than non-zero entries
  - typical: the number of nonzeros in each row is less than a constant (such as 10)
  - sparse matrices are common in many practical applications!
  - even if a matrix is sparse, its inverse is usually not: do not attempt to compute the inverse
    (actually computing the inverse is a bad idea in almost all cases, even for dense matrices)
  - we must develop and study algorithms that exploit sparsity (to save storage and compute time)

Gaussian Elimination for FE-Systems

- For problems on a structured grid, the FE stiffness matrix will have „banded“ structure
- in general, however, the FE stiffnes matrix is unstructured
- Elimination suffers from „fill-in“, that is, originally existing zeros are destroyed in the elimination process and must be re-eliminated
- For 2D-FE-problem with n unknowns (and in lexicographic ordering), $O(n^{1.5})$ unknowns must be eliminated
- better orderings exist: e.g. Minimum degree ordering, nested dissection ordering
  - finding orderings that reduce fill-in has become research topic in ist own right and is still active (especially e.g. for parallel computing!)
  - For 2D-problems, good elimination based algorithms are practically useful and are routinely used in practice, though they fail to be „optimal“
- In (large) 3D problems - elimination based algorithms are not sufficiently efficient

Nested Dissection

- Split unknowns in two halfs and a separator
- order unknows
  - first part A_{11}, second part A_{22}, separator S
- Exploit that unknowns of first and second part do not depend on each other (directly)
  - „domain decomposition“
- Form Schur complement for separator unknowns
  - leads to dense system for Schur complement
  - solve for Schur complement, i.e. separator unknowns
  - solve for A_{11} unknowns and A_{22} unknowns (in parallel)
- Classical Divide and Conquer-Strategy:
  - Can be applied recursively

Computational Complexity of Nested Dissection

In 2D:

- #unknowns in S is $O(n)$
  solving the Schur complement thus costs
  $O(n^3) = O(N^{3/2})$
- The A_{11} unknowns and A_{22} unknowns are each $O(n^2/2)$ but are treated by recursion (until they are so small that solving becomes trival (only one unknown).
- The recursion (over two levels in 2D) leads to a complexity estimate of the form

$$\text{Cost}(n) = R(n) + 4R\left(\frac{n}{2}\right) + 16R\left(\frac{n}{2}\right) + \dots$$

where

$$R(n) = c\left(n^3 + 2(\frac{n}{2})^3\right)$$

The cost of solving the first separator Schur complement dominates the cost (up to a constant)

Nested Dissection for 3D

- Here the separator S has size $O(n^2)$
- Solving the (dense) Schur complement system costs $O(n^6) = O(N^2)$
- This is again the dominating complexity

- Summary: Nested dissection has better complexity than banded solvers, but in 3D the advantage is not overwhelming
- Nested dissection requires more complex data structures
- For general FE meshes, finding a good separator is a difficult graph problem, but in many engineering designs good separators exist (unit square is actually the worst case)

Nested Dissection and Alternatives

- Minimum Degree Ordering ... trying to find elimination orders that minimize fill-in
  - can be in conflict with Pivoting orders (for numerical stability when systems are non-SPD)
- Partitioning software exists
- Multifrontal Method
  (state of the art in current direct solvers for FE problems)
- Parallelization possible
  but increases complexity further
- Idea: do not form the Schur complements, but treat them approximately:
  - Leads to domain decomposition methods, where an iteration must be performed

Complexity of iterative nested dissection (aka domain decomposition)

- Assume a partitioning in two subsets, each with $N/2$ unknowns
- and an iterative scheme that must visit each partition k times
- Assume that an optimal algorithm exists that solves the problem with cost

$$\text{Cost}(N) = cN$$

- Then a domain decomposition algorithm with 2 subdomains and k visits to each subdomain will cost

$$\text{DD-Cost}(N) = 2k\text{Cost}(N/2) = kcN$$

- Consequence: for $k \geq 2$ (i.e. if each subdomain is visited more than once), a domain decomposition algorithm cannot be optimal.

Iterative Methods for sparse systems

- C.F. Gauss (Göttingen)
- L. Seidel (München)
- R.V. Southwell (Cambridge)
- D. M. Young (UT Texas)
- M. Hestness, E. Stiefel (and many others):
  Conjugate gradients, Krylov space methods
- Brandt, Hackbusch, Trottenberg (and many others):
  Multigrid

Iterative Linear System Solver Algorithms

- For large 2D and especially for 3D problems iterative linear systems solvers are preferable - because they can have (much) better complexity.
- Idea:
  - generate a sequence of approximate solution vectors
  - that aproach the true solution.

$$x^0, x^1, x^2, \ldots \text{ such that } x^* := \lim_{i \to \infty} x^{(i)} \text{ solves } Ax^* = b.$$

- Each iterate $x^{(i)}$ must be easy (that is: cheap) to compute
  - low compute requirements
  - low memory requirements

Historical Background

[Über Stationsausgleichungen.]

GAUSS an GERLING. Göttingen, 26. December 1823.

Mein Brief ist zu spät zur Post gekommen und mir zurückgebracht. Ich erbreche ihn daher wieder, um noch die praktische Anweisung zur Elimination beizufügen. Freilich gibt es dabei vielfache kleine Localvortheile, die sich nur ex usu lernen lassen.

Ich nehme Ihre Messungen auf Orber-Reisig zum Beispiel [*)].

Ich mache zuerst

$$[\text{Richtung nach}]\ 1 = 0,$$

nachher aus 1 . 3

First reference to iterative solvers

Die Bedingungsgleichungen sind also:

$$
\begin{aligned}
0 &= +\ 6 + 67a - 13b - 28c - 26d \\
0 &= -\ 7558 - 13a + 69b - 50c - 6d \\
0 &= -\ 14604 - 28a - 50b + 156c - 78d \\
0 &= +\ 22156 - 26a - 6b - 78c + 110d;
\end{aligned}
$$

Summe $= 0$.

- Linear system scaled to use integer values (mixed precision!)
- singular, rows and columns linearly dependent
- Matrix-notation (vector spaces) was not yet invented
- clever adaptive strategy

An adaptive algorithm

Um nun indirect zu eliminiren, bemerke ich, dass, wenn 3 der Grössen a, b, c, d gleich 0 gesetzt werden, die vierte den grössten Werth bekommt, wenn d dafür gewählt wird. Natürlich muss jede Grösse aus ihrer eigenen Gleichung, also d aus der vierten, bestimmt werden. Ich setze also $d = -201$ und substituire diesen Werth. Die absoluten Theile werden dann: $+5232$, -6352, $+1074$, $+46$; das Übrige bleibt dasselbe.

Jetzt lasse ich b an die Reihe kommen, finde $b = +92$, substituire und finde die absoluten Theile: $+4036$, -4, -3526, -506. So fahre ich fort, bis nichts mehr zu corrigiren ist. Von dieser ganzen Rechnung schreibe ich aber in der Wirklichkeit bloss folgendes Schema:

Data structures, efficient execution, mixed precision

	$d = -201$	$b = +92$	$a = -60$	$c = +12$	$a = +5$	$b = -2$	$a = -1$
+ 6	+ 5232	+ 4036	+ 16	− 320	+ 15	+ 41	− 26
− 7558	− 6352	− 4	+ 776	+ 176	+ 111	− 27	− 14
− 14604	+ 1074	− 3526	− 1846	+ 26	− 114	− 14	+ 14
+ 22156	+ 46	− 506	+ 1054	+ 118	− 12	0	+ 26.

Insofern ich die Rechnung nur auf das nächste 2000^{tel} [der] Secunde führe, sehe ich, dass jetzt nichts mehr zu corrigiren ist. Ich sammle daher

$a = -60$	$b = +92$	$c = +12$	$d = -201$
$+5$	-2		
-1			
-56	$+90$	$+12$	-201

und füge die Correctio communis $+56$ bei, wodurch wird:

$$a = 0 \qquad b = +146 \qquad c = +68 \qquad d = -145,$$

Efficiency, Performance, Multitasking

Fast jeden Abend mache ich eine neue Auflage des Tableaus, wo immer leicht nachzuhelfen ist. Bei der Einförmigkeit des Messungsgeschäfts gibt dies immer eine angenehme Unterhaltung; man sieht dann auch immer gleich, ob etwas zweifelhaftes eingeschlichen ist, was noch wünschenswerth bleibt, etc. Ich empfehle Ihnen diesen Modus zur Nachahmung. Schwerlich werden Sie je wieder direct eliminiren, wenigstens nicht, wenn Sie mehr als 2 Unbekannte haben. Das indirecte Verfahren lässt sich halb im Schlafe ausführen, oder man kann während desselben an andere Dinge denken.

Even though almost 200 years old this is a good inspiration!

The algorithm is guaranteed to converge for spd systems:

Iterative Methods

Matrix free implementation of the Gauss-Seidel Method

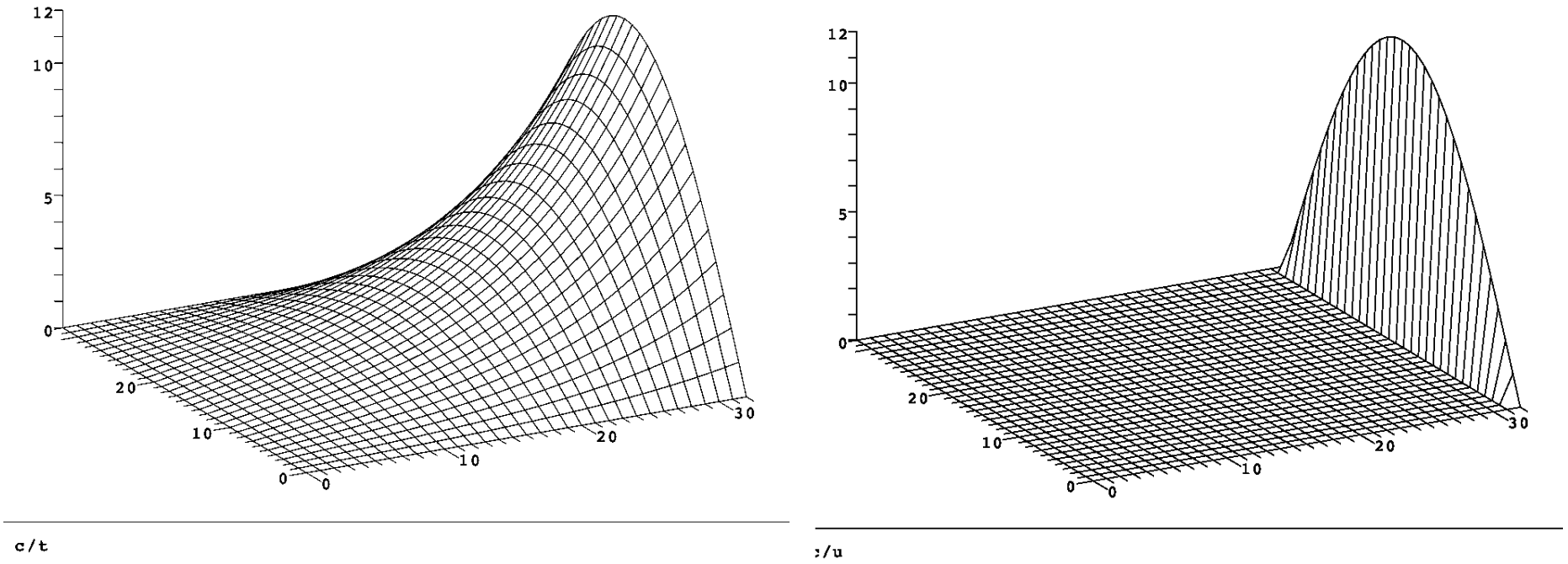
```
real u[N+1][N+1];     /* initialisieren mit Randwerten, im Inneren
                         "0" oder Mittelwert der Randwerte */

for (int it=0; it<MAXIT; it++) {
    real udiff=0;
    for (int i=1; i<N; i++)
        for (int j=1; j<N; j++) {
            real un = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j+1]+u[i][j-1]);
            udiff += fabs(u[i][j]-un);
            u[i][j] = w*un + (1-w) * u[i][j];
        }
        if( (udiff / N*N) < TOL) break;   (*)
}
```

w=1: Gauss-Seidel; w>1: SOR

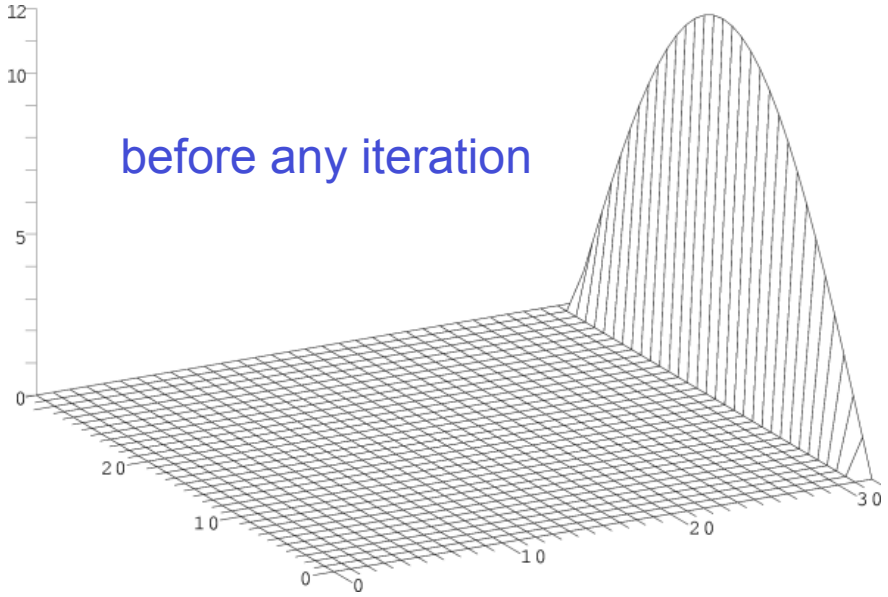
Graphical Illustration (Visualization)

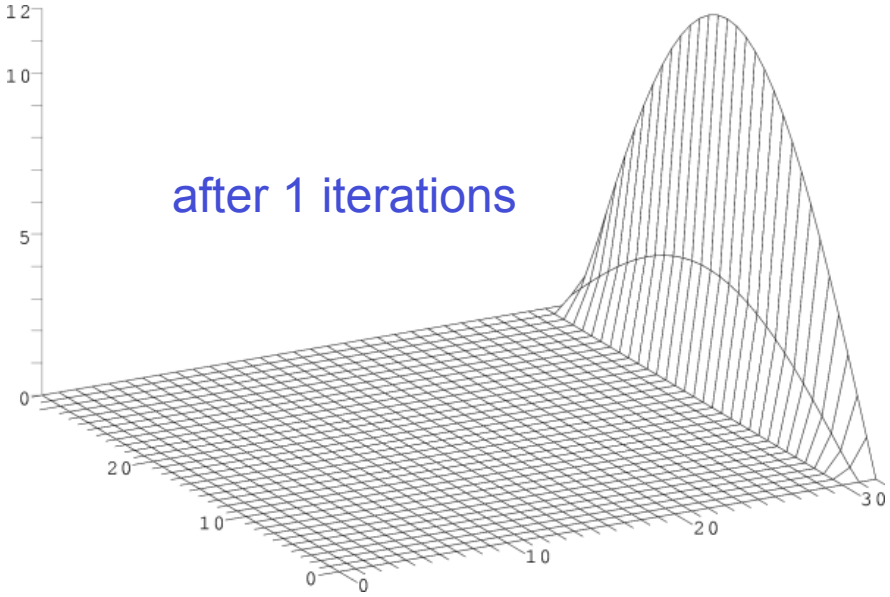
$$u = \sinh(x)\sin(y)$$

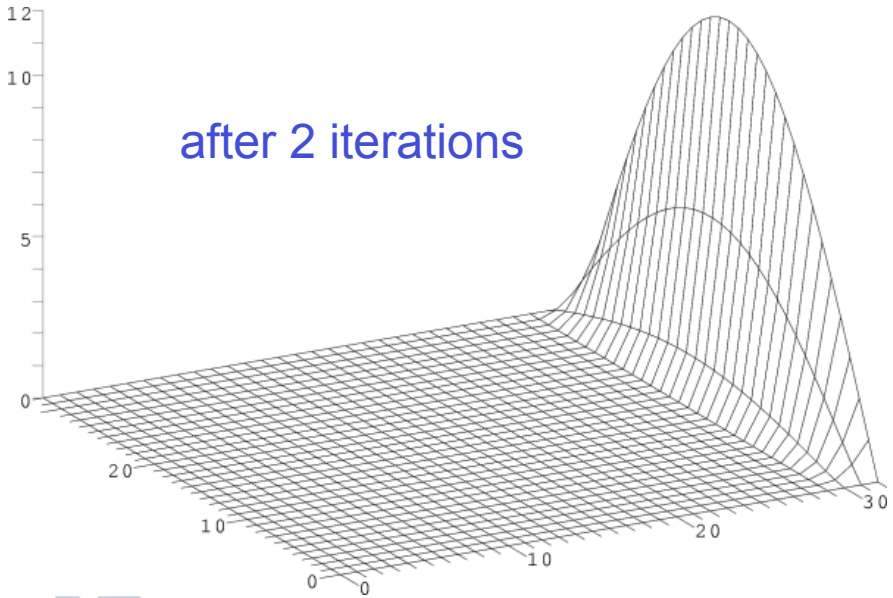


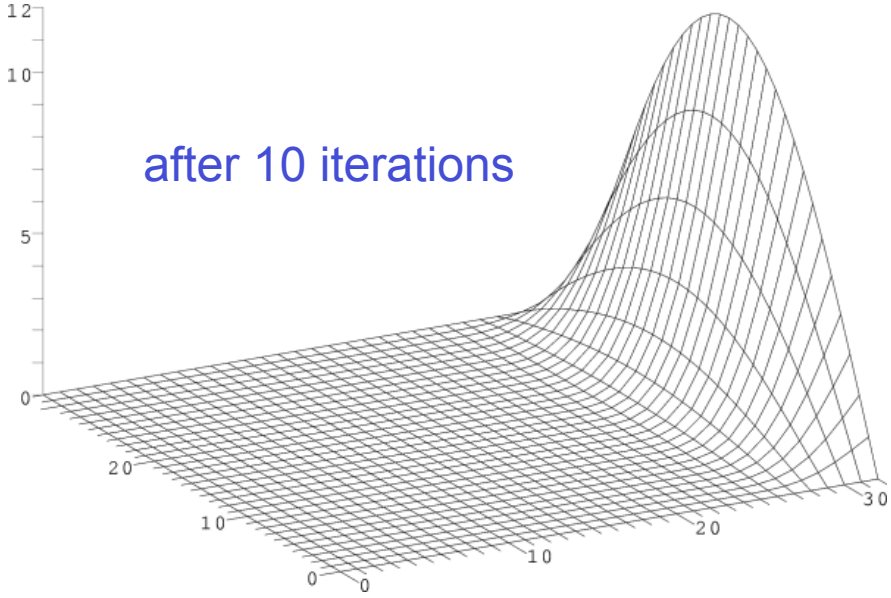
- Exact Solution (of PDE)
- Boundary values to start the iteration

Visualization of Convergence

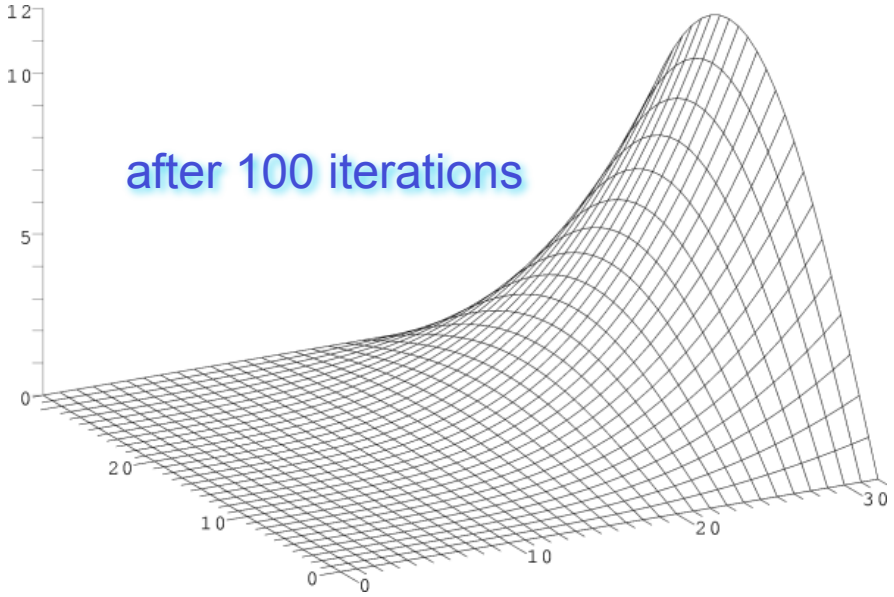


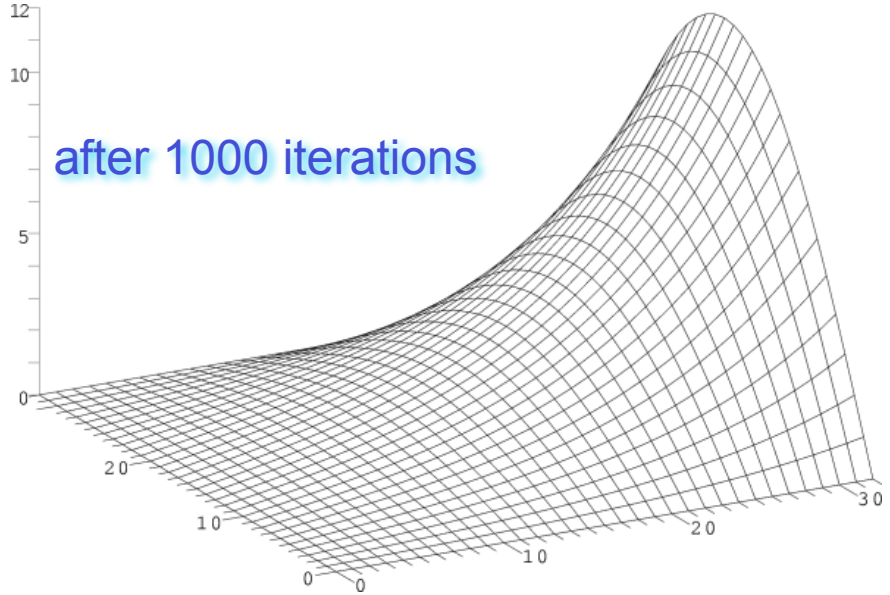


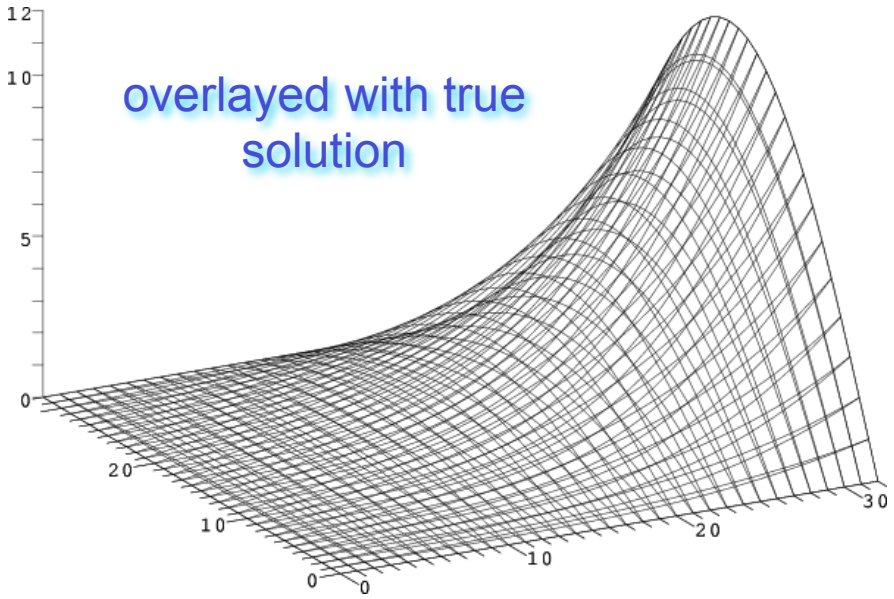


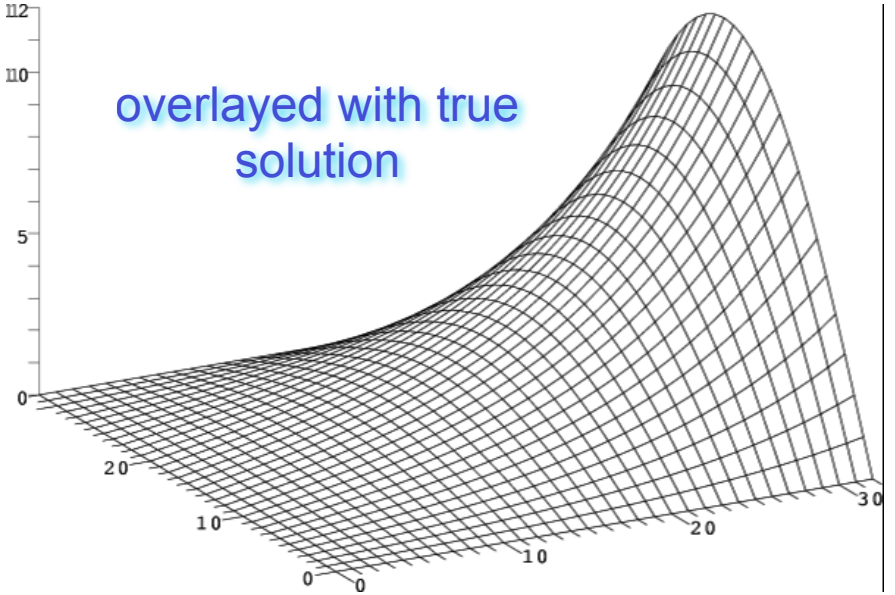


Visualization of Convergence









Iterative Methods: Discussion of Example

- Gauss-Seidel-Method:
  - When N iterations are required, each of which needs $O(N)$ operations then the total cost is $O(N^2)$ operations
  - $N=n^2$, n number of grid points in one mesh dimensions
- SOR-method: only $n=N^{1/2}$ iterations necessary, if the relaxation parametre $\omega>1$ is chosen optimally.
  - FFor the model problem thsi can be shown mathmatically, see e.g. Stoer/ Bulirsch.
- Be clear that there are different types of error:
  - Rounding errors (hier of secondary importance)
  - Iteration error: stopping the iteration after finitely many steps
  - Discretisation: even after ∞ many iterationenan error relative to the partial differential equation remains (discretization error: griod vs. plate)

Ideas for improvement (1)

- Choose ordering of grid traversals more inteligently. However this unfortunately only helps substantially, when the problem at hand has a „preferred direction. (Physically this corresponds to convection rather than diffusion)
- Search systematically for equations/unknows which are far from equlibrium (large residual) and iterate preferably on those (search algorithm often too expensive)
- Search for better initial guess.
  - interpolate boundary values
  - Start on coarser grid, compute a approximate solution there, and interpolate to finer grid. (Cascade algorithm, Nested iteration)

Ideas for accelerating the algorithms

- Acceleration possible?:
  - Store several iterates x^i, x^{i+1}, x^{i+2},... and search for better solution by taking (linear) combinations -> leads to the method of conjugate gradients with preconditioning or more generally to Krylov-space methods such as (GMRES, etc.)
  - Use coarser grids to accelerate fine grid iteration process: Multigrid methods.
  - ....
- Many books, e.g.: Wolfgang Hackbusch: *Iterative Lösung großer schwachbesetzter Gleichungssysteme,* Teubner, Stuttgart, 2. Auflage (1993)

Linear Iterative Methods in Matrix Notation

$$Ax = f$$

$$Bx^{k+1} = f - (A - B)x^k$$

$$x^{k+1} = x^k + B^{-1}(f - Ax^k)$$

- The matrix B is characteristic for the method:
  - $B = \mathrm{diag}(A)$: Gauss-Jacobi method
  - $B = \mathrm{lowertriang}(A)$: Gauss-Seidel Method
- Why iterative methods?
  - One step of iteration requires only a multiplication of a vector with A (which is cheap, since A is sparse) and with B^{-1}. Therefore B^{-1} must be easily accessible and cheap to apply: design goal for iterative methods.
- Even if one iteration is cheap, we must also not need too many iterations.
- Wanted: **Cheap** iterations with **fast** convergence.
- Keep in mind: Good iterative methods will usually neither explicitly form the matrices nor perform matrix-vector multiplications explicitly!

Linear Iterative Methods in Matrix Notation

- Clearly, the exact solution $x^* = A^{-1} f$ is a fixed point of the iteration
- To check whether the iteration converges $x^k = x^* + e^k$ and $x^{k+1} = x^* + e^{k+1}$ may be inserted:

$$e^{k+1} = (I - B^{-1}A)e^k$$

- Convergence is obtained, when the error norm is reduced.

$$||e^{k+1}|| \leq ||I - B^{-1}A||\ ||e^k||$$

- Convergence is obtained, when the error norm is reduced.

$$|||I - B^{-1}A|| \leq 1$$

  - Extreme (but useless) case: $A=B$
  - Design goal: good compromise between cost for each iteration and speed of convergence

Linear Iterative Methods in Matrix Notation

- The classical choices for linear iterative methods are based on a matrix splitting

$$A = L + D + R$$

- Jacobi Iteration:

$$x^{k+1} = x^k + D^{-1}(f - Ax^k)$$

- Gauss-Seidel Iteration

$$x^{k+1} = x^k + (L + D)^{-1}(f - Ax^k)$$

- SOR method (for $0 < \omega < 2$)

$$x^{k+1} = x^k + (L + \frac{1}{\omega}D))^{-1}(f - Ax^k)$$

SOR-method in elementary form

For $i=1,\ldots,n$:

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right)$$

- For typical systems (such as the Laplace operator in 2D), an optimally chosen ω can reduce the number of required iterations from $O(n)$ to $O(n^{0.5})$.
  - The overall complexity is then $O(n^{1.5})$ (instead of $O(n^2)$) - provided the sparsity of the matrix is exploited
- The optimal parameter can in practice be determined experimentally
- For model problems an analysis exists that allows to calculate the optimal ω analytically

Explicitly writing the linear system ...

$$
\begin{array}{ccccccccccccc}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & & & \cdots & & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & b_1 \\
\vdots & & \vdots & & & & & & & \vdots & & \vdots & & \vdots \\
a_{k,1}x_1 & + & a_{k,2}x_2 & + & \cdots & + & a_{k,k-1}x_{k-1} & + & a_{k,k}x_k & + \cdots + & a_{k,n-1}x_{n-1} & + & a_{k,n}x_n & = & b_k \\
\vdots & & \vdots & & & & & & & \vdots & & \vdots & & \vdots \\
a_{n,1}x_1 & + & a_{n,2}x_2 & + & & & \cdots & & + & a_{n,n-1}x_{n-1} & + & a_{n,n}x_n & = & b_n
\end{array}
$$

Jacobi-Iteration

$$
\begin{array}{ccccccccccccc}
\boxed{a_{1,1}x_1^{i+1}} & + & a_{1,2}x_2^i & + & & & \cdots & & + & a_{1,n-1}x_{n-1}^i & + & a_{1,n}x_n^i & = & b_1 \\
\vdots & & \vdots & & & & & & & \vdots & & \vdots & & \vdots \\
a_{k,1}x_1^i & + & a_{k,2}x_2^i & + & \cdots & + & a_{k,k-1}x_{k-1}^i & + & \boxed{a_{k,k}x_k^{i+1}} & + \cdots + & a_{k,n-1}x_{n-1}^i & + & a_{k,n}x_n^i & = & b_k \\
\vdots & & \vdots & & & & & & & & \vdots & & \vdots & & \vdots \\
a_{n,1}x_1^i & + & a_{n,2}x_2^i & + & & & \cdots & & + & a_{n,n-1}x_{n-1}^i & + & \boxed{a_{n,n}x_n^{i+1}} & = & b_n
\end{array}
$$

Here all x^i are brought to the „right hand side“, leading to a simple formula to compute the x^{i+1}

Gauss-Seidel-Iteration (same for SOR)

$$
\begin{aligned}
a_{1,1}x_1^{i+1} &+ a_{1,2}x_2^{i} &+ \cdots &+ a_{1,n-1}x_{n-1}^{i} &+ a_{1,n}x_n^{i} &= b_1 \\
\vdots & \vdots & & \vdots & \vdots & \vdots \\
a_{k,1}x_1^{i+1} &+ a_{k,2}x_2^{i+1} &+ \cdots + a_{k,k-1}x_{k-1}^{i+1} + a_{k,k}x_k^{i+1} &+ \cdots + a_{k,n-1}x_{n-1}^{i} &+ a_{k,n}x_n^{i} &= b_k \\
\vdots & \vdots & & \vdots & \vdots & \vdots \\
a_{n,1}x_1^{i+1} &+ a_{n,2}x_2^{i+1} &+ \cdots &+ a_{n,n-1}x_{n-1}^{i+1} &+ a_{n,n}x_n^{i+1} &= b_n
\end{aligned}
$$

Here also all x^i are brough to the „right hand side“, but additionally all x^{i+1} that have already been computed (that is the lower triangle - as marked); only the x^{i+1} on the diagonal remain as unknowns on the left side.

Once more formally in matrix notation

- we split the Matrix $A = L+D+R$
  - L strictly lower triangle
  - D diagonal
  - R strictly upper triangle
- and write:
  - $(L + D)\, x^{i+1} + R\, x^i = b$ Gauss-Seidel-Method (suucessive relaxation)
  - $D\, x^{i+1} + (L + R)\, x^i = b$ Jacobi-Method (simultaneous iteration)
- Generally:
  - Split $A = \hat{A} + E$
  - $\hat{A}$ easily „invertible“
    (but only abstractly there is never any matrix inversion the code!)
  - E Remainder
  - Iteration: $\hat{A}\, x^{i+1} = b - E\, x^i$

Classical Iteration Methods

- Richardson: $\hat{A} = I$
- Jacobi: $\hat{A} = D$
- Gauss-Seidel: $\hat{A} = (D+L)$
- SOR: $\hat{A} = (1/\omega\, D + L)$; ω: relaxation parameter
  - The latter is not trivial to see, but will be shown on the next slide
- Observe that formally a triangular matrix is inverted in both the GS and SOR algorithms, but that this step is NOT executed explicitly!
  - The solution of system with triangular matirix is performed by forward (L) or backward substitution, respectively (R).

Matrix-decomposition for the SOR-Method

$$
\begin{aligned}
x_k^{i+1} &= x_k^i + \omega \left(b_k - \sum_{j=1}^{k-1} a_{k,j} x_j^{i+1} + \sum_{j=k}^{n} a_{k,j} x_j^i \right) / a_{kk} \\
\Leftrightarrow \quad & x^{i+1} = x^i + \omega D^{-1} \left(b - L x^{i+1} - (D+R) x^i \right) \\
\Leftrightarrow \quad & \frac{1}{\omega} D x^{i+1} = \frac{1}{\omega} D x^i + b - L x^{i+1} - (D+R) x^i \\
\Leftrightarrow \quad & \left(\frac{1}{\omega} D + L \right) x^{i+1} + \left((1 - \frac{1}{\omega}) D + R \right) x^i = b \\
\Leftrightarrow \quad & \hat{A} x^{i+1} + E x^i = b
\end{aligned}
$$

Efficient implementation of iterative methods

- We must not use the matrix data structure as a two-dimensional $n \times n$ array.
- We must avoid to store and operate on all the zeros
- Banded matrices:
  - storing the diagonal bands in a collection of 1D-vectors is the standard technique.
  - The grid and coresponding matrix structure is „hard-coded“ with this data structure
- Stencil based storage
  - In many applications - when element shapes and material parameters are identical, the matrix rows will be identical
  - No matrix needs to be stored, just a „stencil“ that represents the entries in a matrix row for all (or at least many) rows of the matrix

Compressed row matrix data structure

- Make no assumption on sparsity structure
- they don't store any unnecessary elements
- are effcient (but not very efficient)
  - needing an indirect addressing step for every single scalar operation in a matrix-vector product
- CRS: nnz = number of nonzeros in A.
  - puts the subsequent nonzeros of the matrix rows in contiguous memory locations
  - 3 vectors
    - one for floating-point numbers (`val`)
    - other two for integers (`col_ind`, `row_ptr`)
  - convention: row_ptr(n+1) = nnz+1

Example of matrix in CRS format

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -1 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val	10	-1	3	9	3	7	8	7	3	...	9	13	4	2	-1
col-ind	1	5	1	2	6	2	3	4	1	...	5	6	2	5	6

row-ptr	1	3	6	9	13	17	20

Properties of the CRS format

- Memory cost proportional to number nonszeros (1 int + 1 float/double)
- Matrix-vector multiply (or Gauss-Seidel iteration) costs proportional to number nonzeros
- adding new matrix coefficients is costly
- deleting matrix coefficients may be costly
- access to element (i,j) requires search in row
- indirection in memory access may be slow on some machines

Compressed row matrix data structure

- Using CRS for iterative solver
- Example: matrix-vector multiply $y = A\,x$
  - `initialize y= 0`
  - `for i=1 to number_of_rows`
    - `for j=row_ptr(i) to row-ptr(i+1)-1`
      - `y(i)= y(i) + val(j) * x(col_ind(j))`
- Complexity: $O(nnz) \ll O(n^2)$
- Variants of CRS
  - store diagonal element first in each row.
  - store the inverse of the diagonal element instead of element itself
  - helps to replace repeated divisions by cheaper multiplications in Gauss-Seidel and similar iterative schemes