Multigrid Methods on Parallel Computers

U. Rüde (LSS Erlangen, ruede@cs.fau.de)

Lehrstuhl für Informatik 10 (Systemsimulation)
Universität Erlangen-Nürnberg

www10.informatik.uni-erlangen.de

CEMRACS'12
Numerical Methods and Algorithms for
High Performance Computing
CIRM, Marseille
July 17,2012

Overview

- Motivation
- A tutorial intro to Multigrid (based on Irad Yavneh's tutorial)
  - „Sergeant Jacobi's" soldier alignment problem
  - The multigrid algorithm
  - How fast should our solvers be
- How fast are parallel computers today:
  - The race to Exa-Scale
- Scalable Parallel Multigrid
- Matrix-Free Multigrid FE solver: Hierarchical Hybrid Grids (HHG)
- Other things we do
  - Flow Simulation with Lattice Boltzmann Methods
- Conclusions

A (too) brief introduction to Multigrid

following the wonderful tutorial and using slides by I. Yavneh

Further Acknowledgements

- Multigrid Lecture Notes by S. McCormick
- Slides from Multigrid Tutorial by V. Henson
- „Why Multigrid Methods are so Efficient“ by I. Yavneh
- Multigrid Tutorial by B. Briggs, S. McCormick, V. Henson
- „The Multigrid Guide“ by A. Brandt

What is „Multigrid“?

A framework of efficient iterative methods for solving problems with many variables and many scales.

- Framework: common concept, different methods.
- Efficient: usually $O(N)$ or $O(N \log N)$ operations

  The importance of efficient methods becomes greater as computers grow stronger!
- Iterative: most nontrivial problems in our field cannot be solved directly efficiently.
- Solving: approximately, subject to appropriate convergence criteria, constraints, etc.
- Many variables: the larger the number of variables, the greater the gain of efficient methods
- Many scales: typical spatial and/or temporal sizes.

What can multigrid achieve?

- Solve elliptic PDE in *asymptotically optimal complexity*
- Poisson‘s eqn in 2D: <30 FLOPs per unknown
  - Cheaper than computing

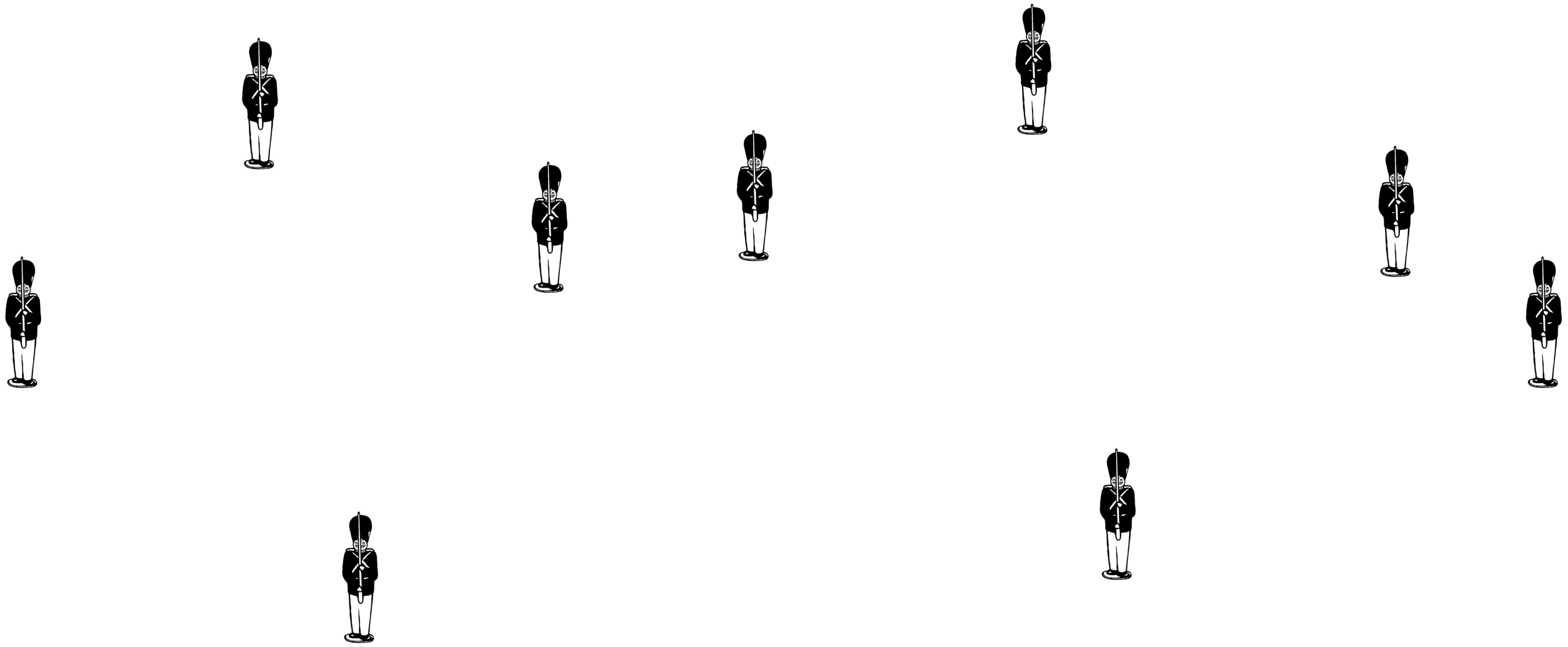
$$u_{i,j} = \sin(x_i)\sin(y_j)$$

- Solve FE problems (linear, scalar, elliptic PDE) with more than 10^{12} tetrahedral elements
  - on a massively parallel supercomputer
  - reminder: $10^{12} \neq 2 \times 10^6$

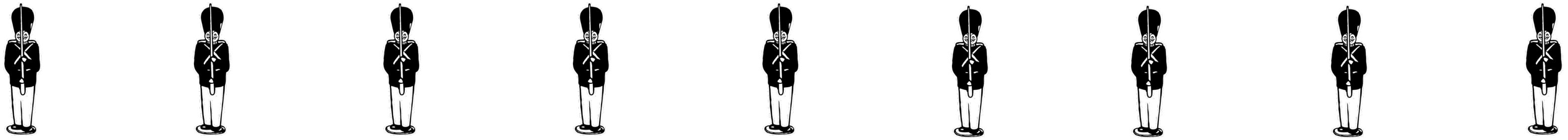
Basic Concepts: Local vs. Global processing.

Imagine a large number of soldiers who need to be arranged in a straight line and at equal distances from each other.

The two soldiers at the ends of the line are fixed. Suppose we number the soldiers 0 to N , and that the length of the entire line is L.

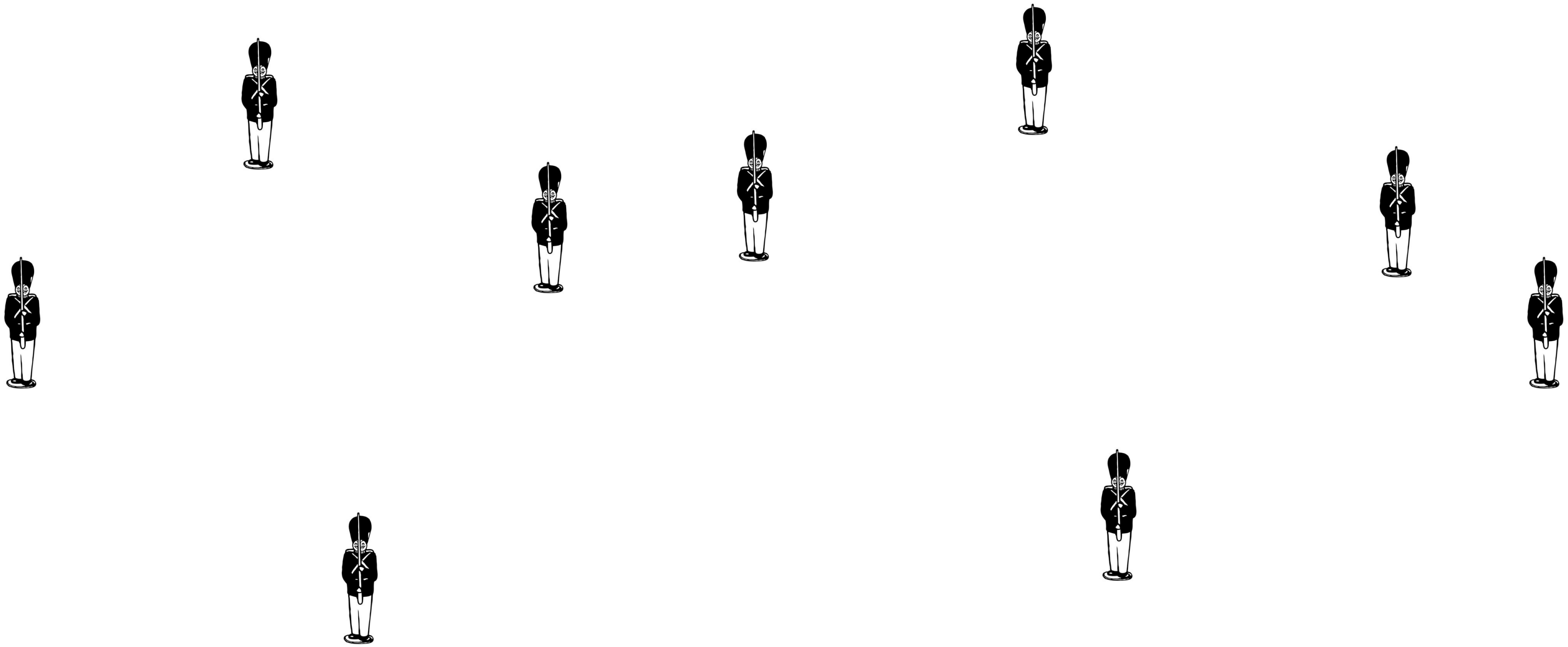


Initial Position



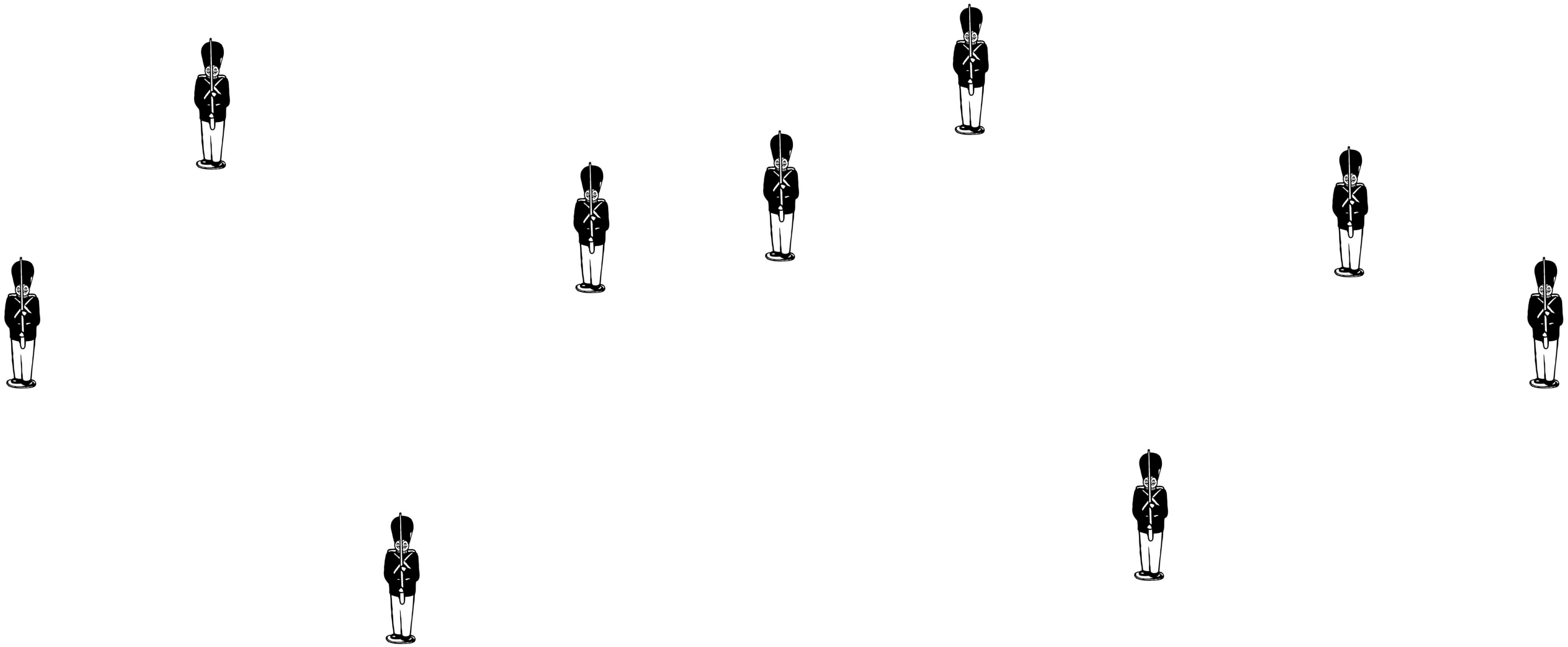
Final Position

Global processing. Let soldier number j stand on the line connecting soldier 0 to soldier N at a distance jL/N from soldier number 0.

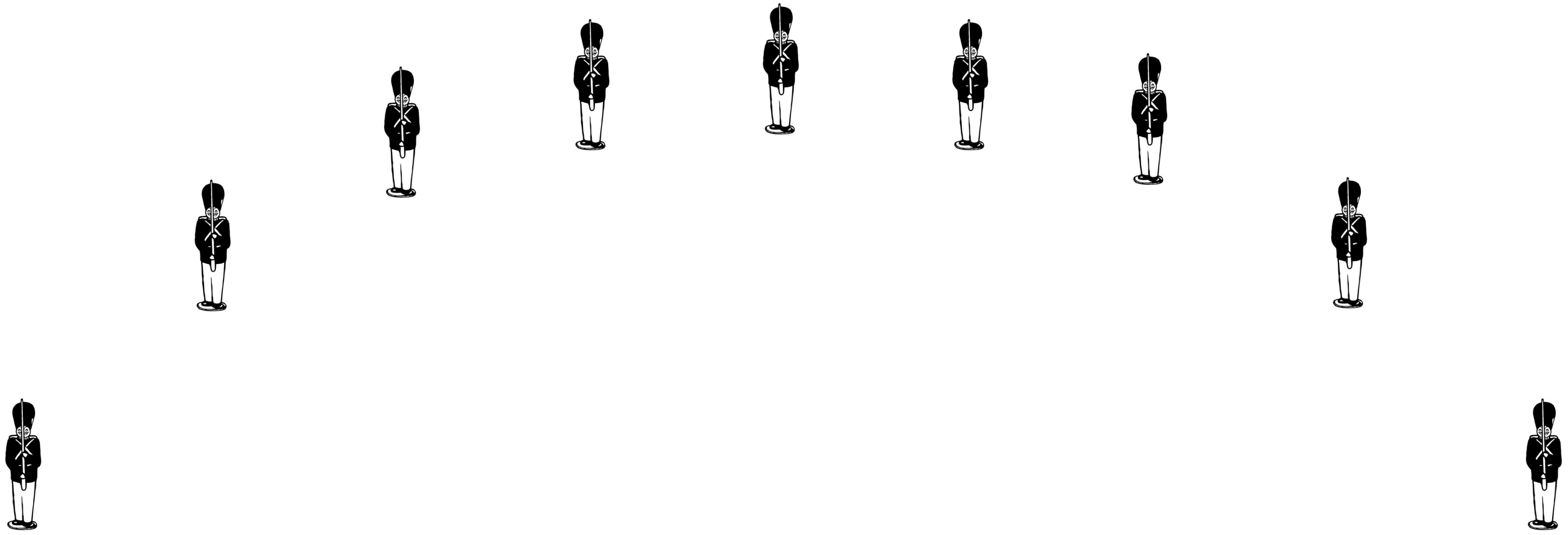


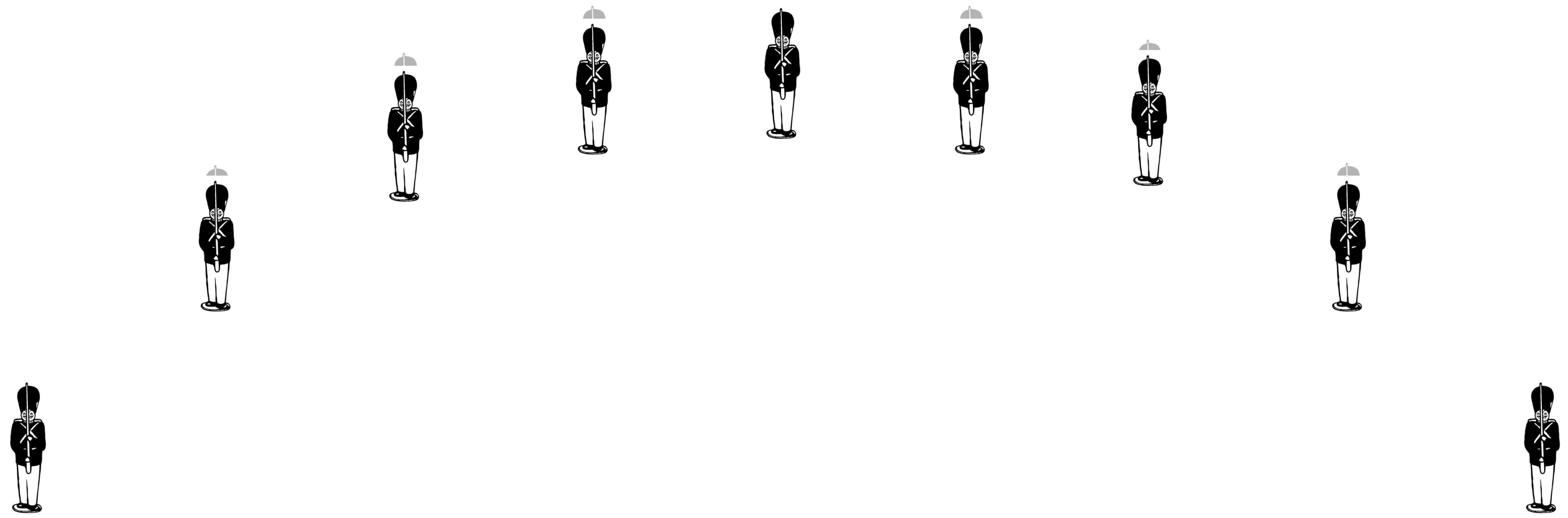
Global processing. Let soldier number *j* stand on the line connecting soldier 0 to soldier *N* at a distance *jL*/*N* from soldier number 0.

This method solves the problem directly, but it requires a high degree of sophistication: recognition of the extreme soldiers and some pretty fancy arithmetic.

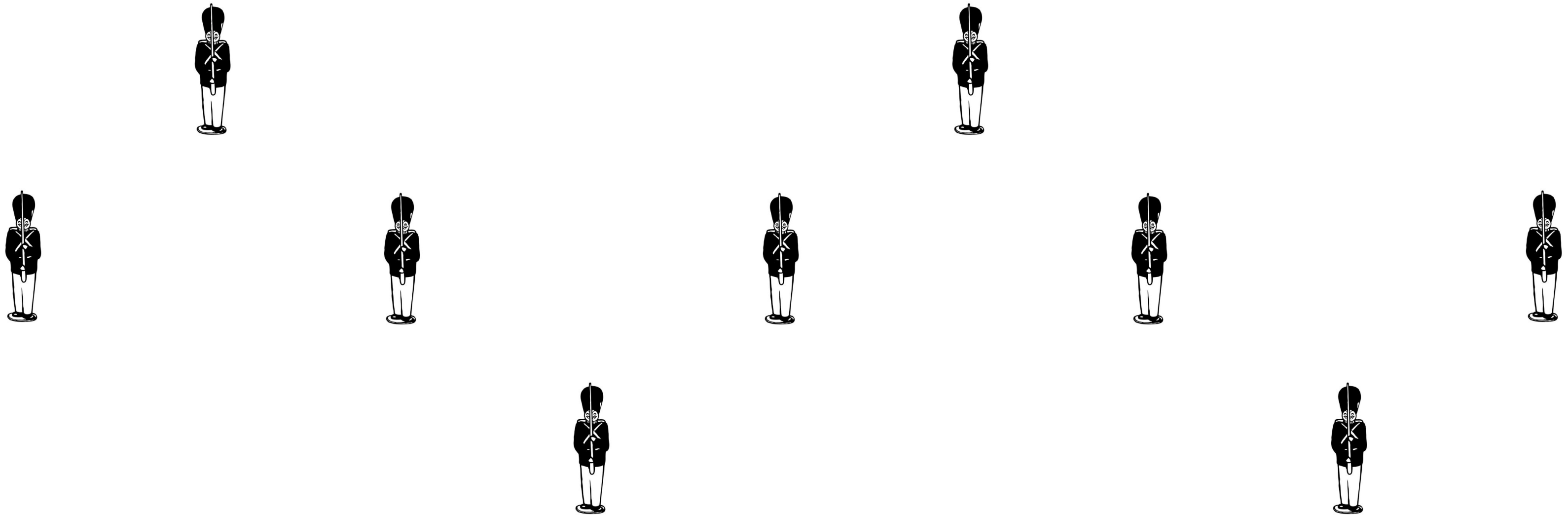


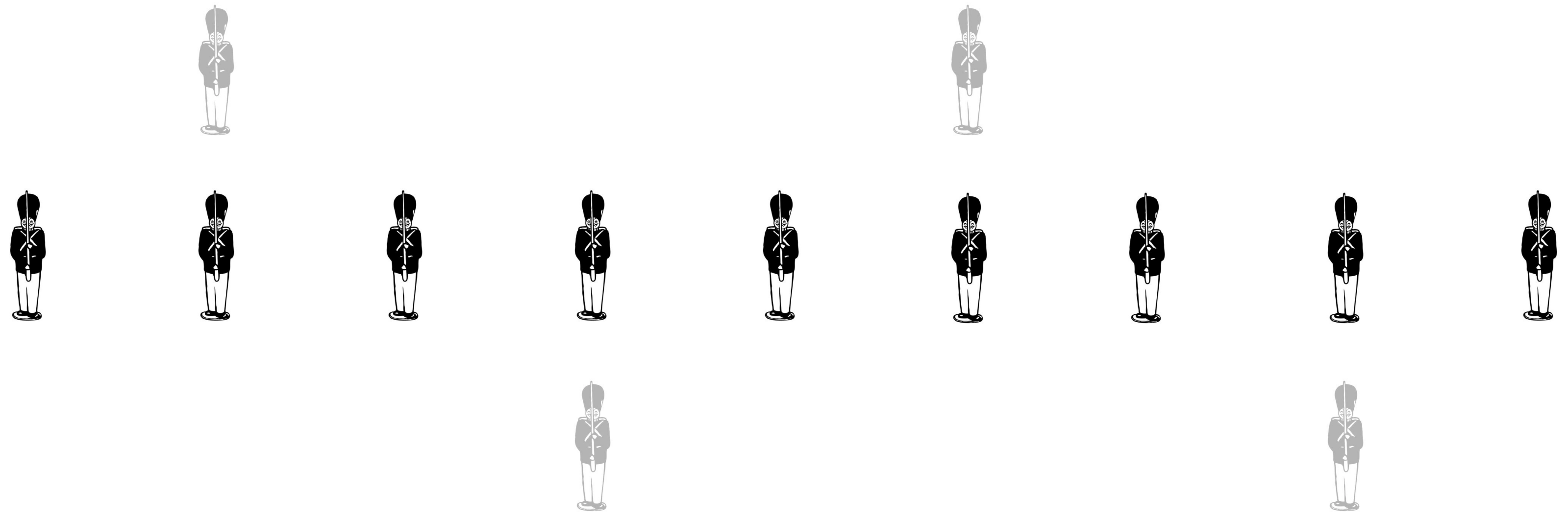
A step in the right direction



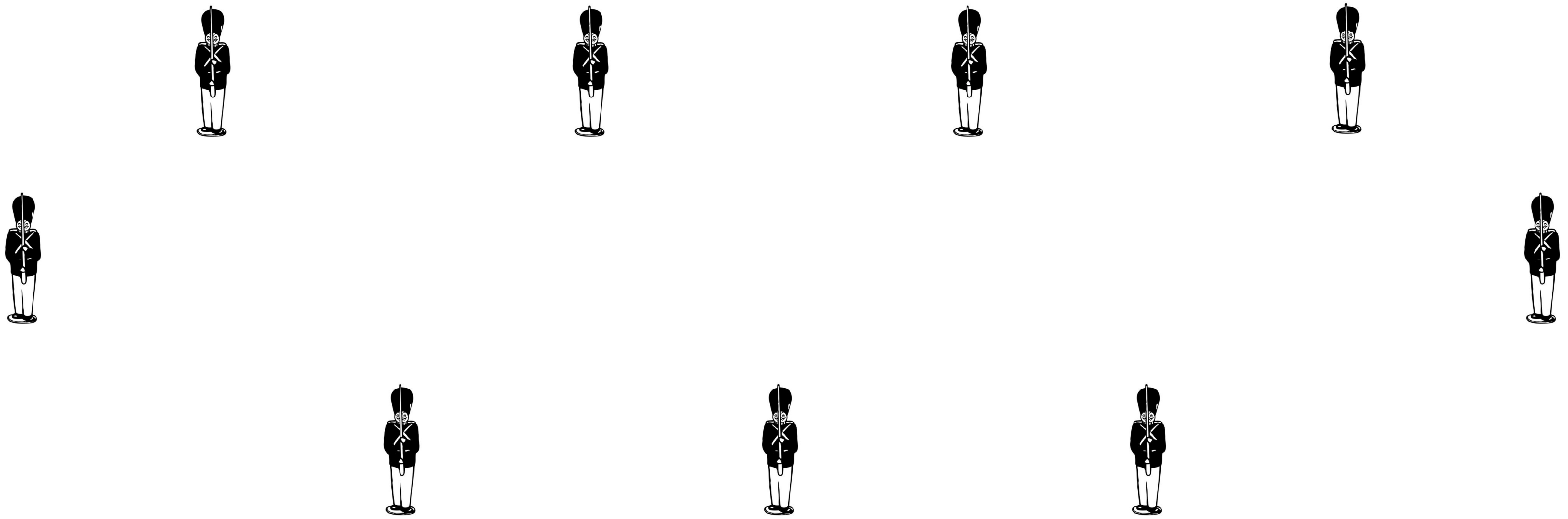


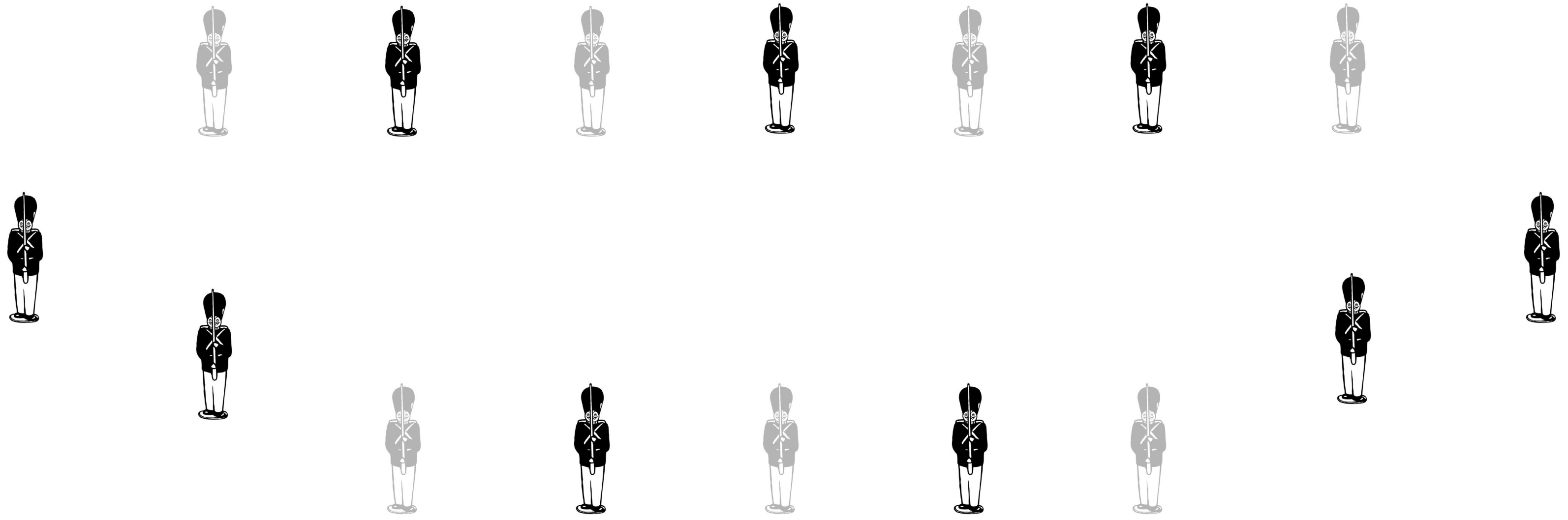
Slow convergence



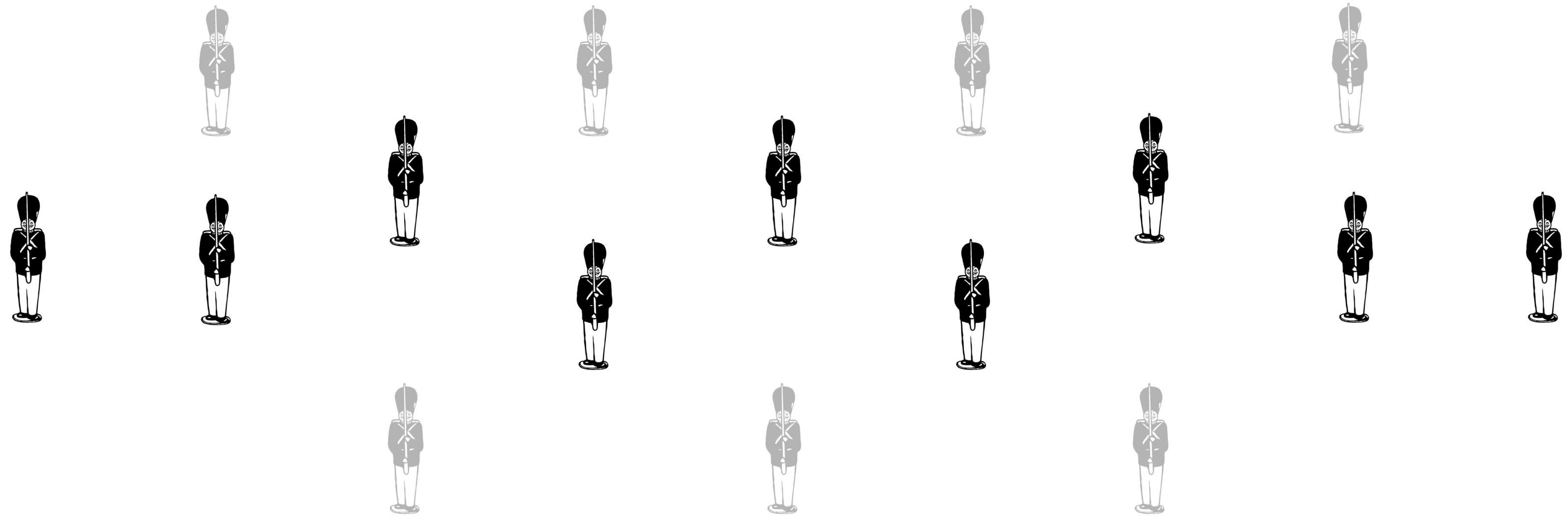


Fast convergence

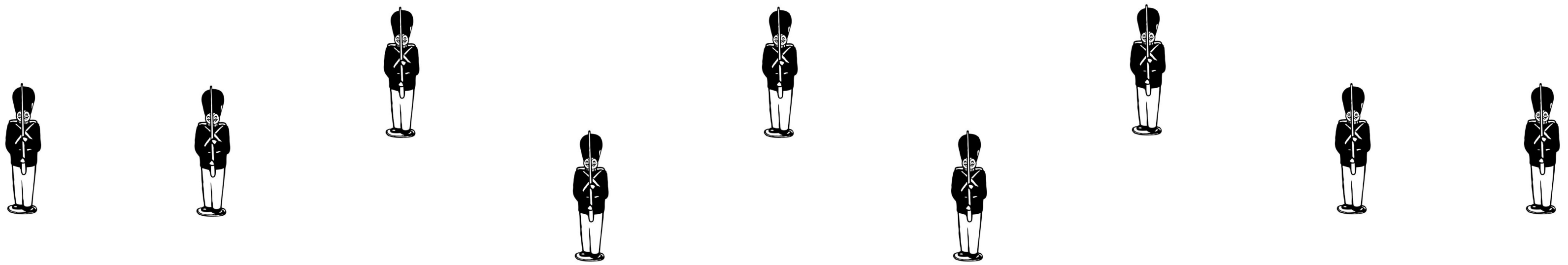




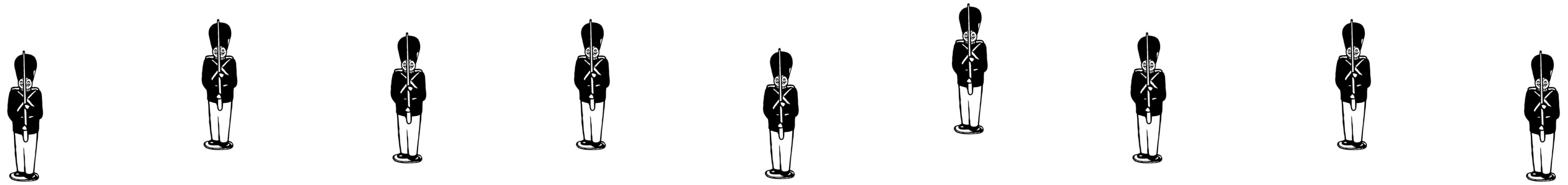
Slow convergence



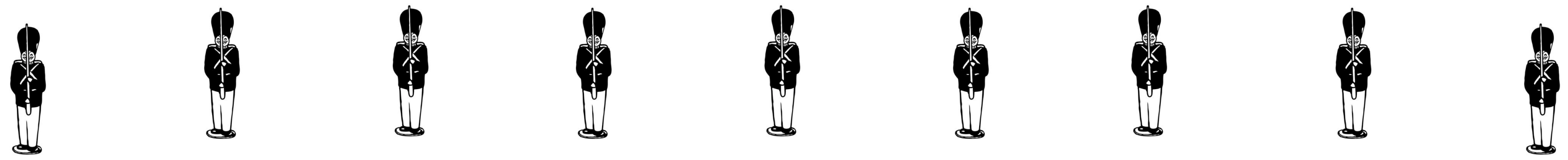
Local solution: damping



Local solution: damping

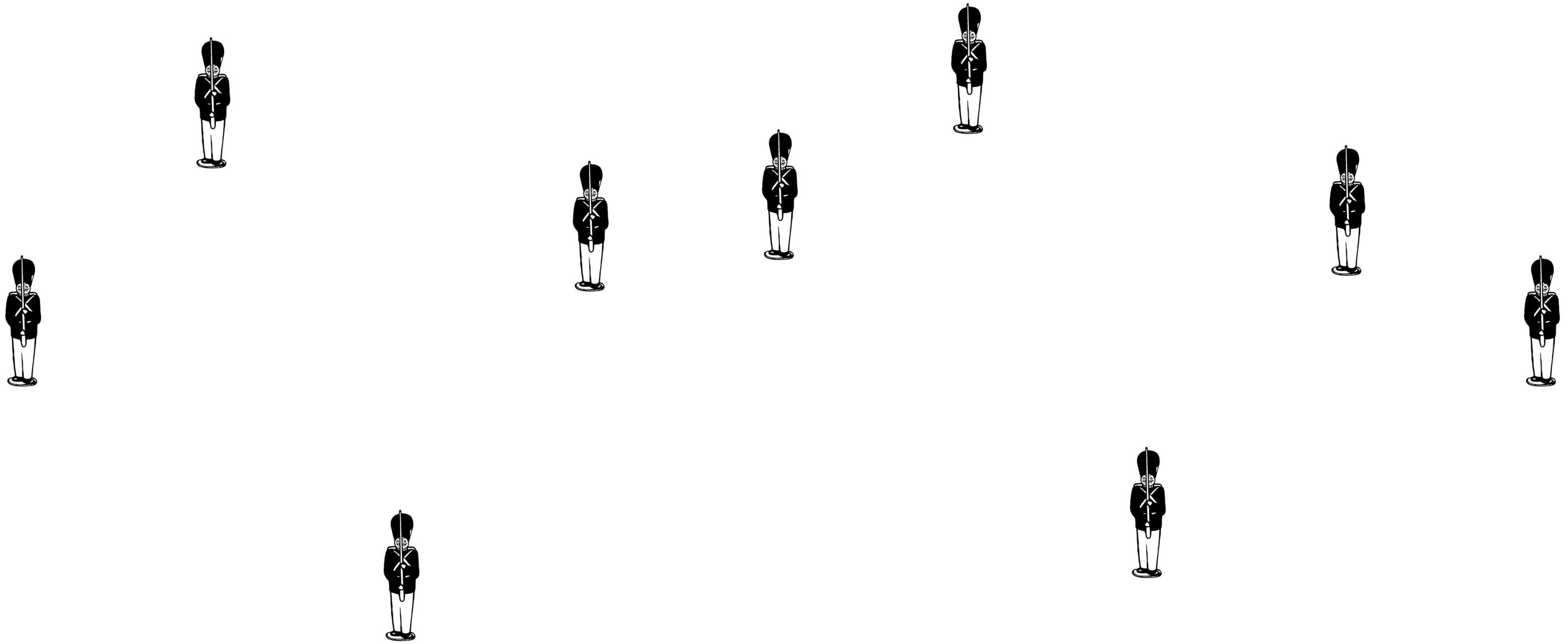


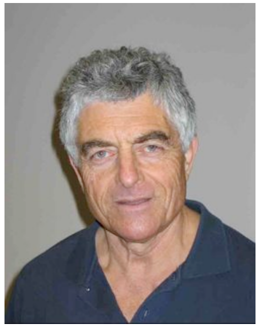
Local solution: damping

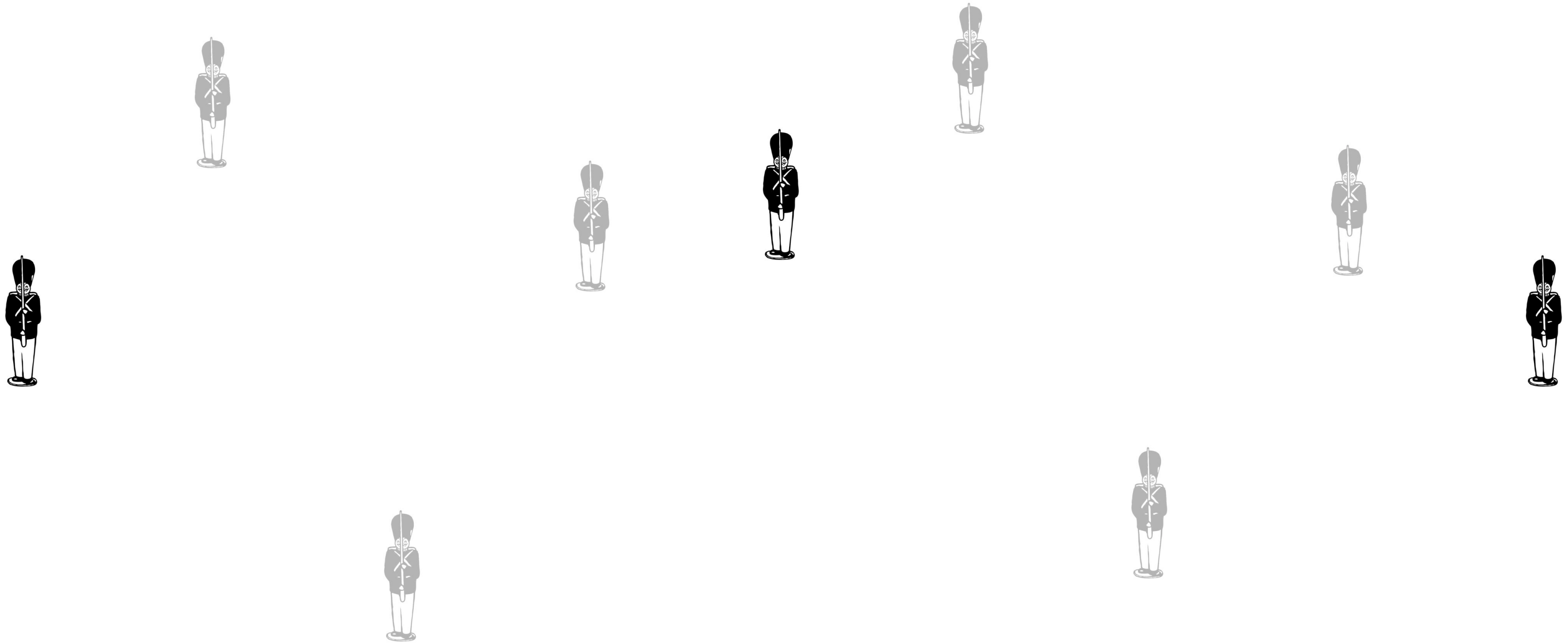


Local solution: damping

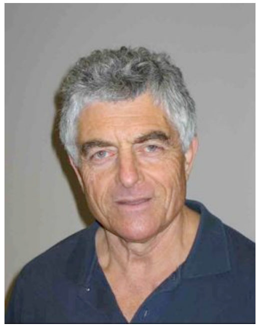
The multiscale idea: Employ the local processing with simple arithmetic. But do this on all the different scales.

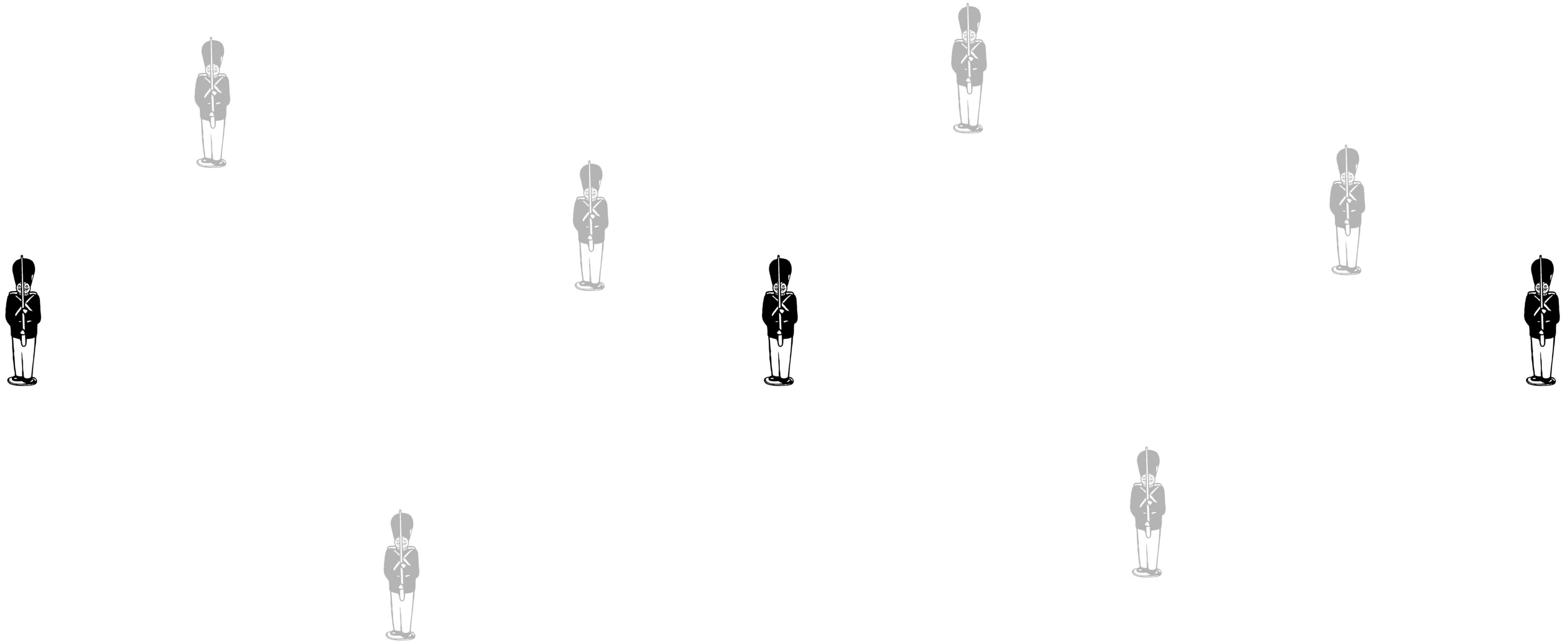




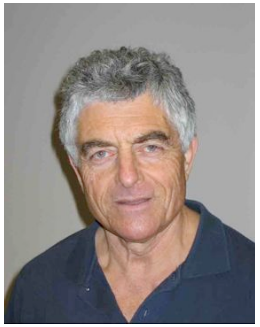


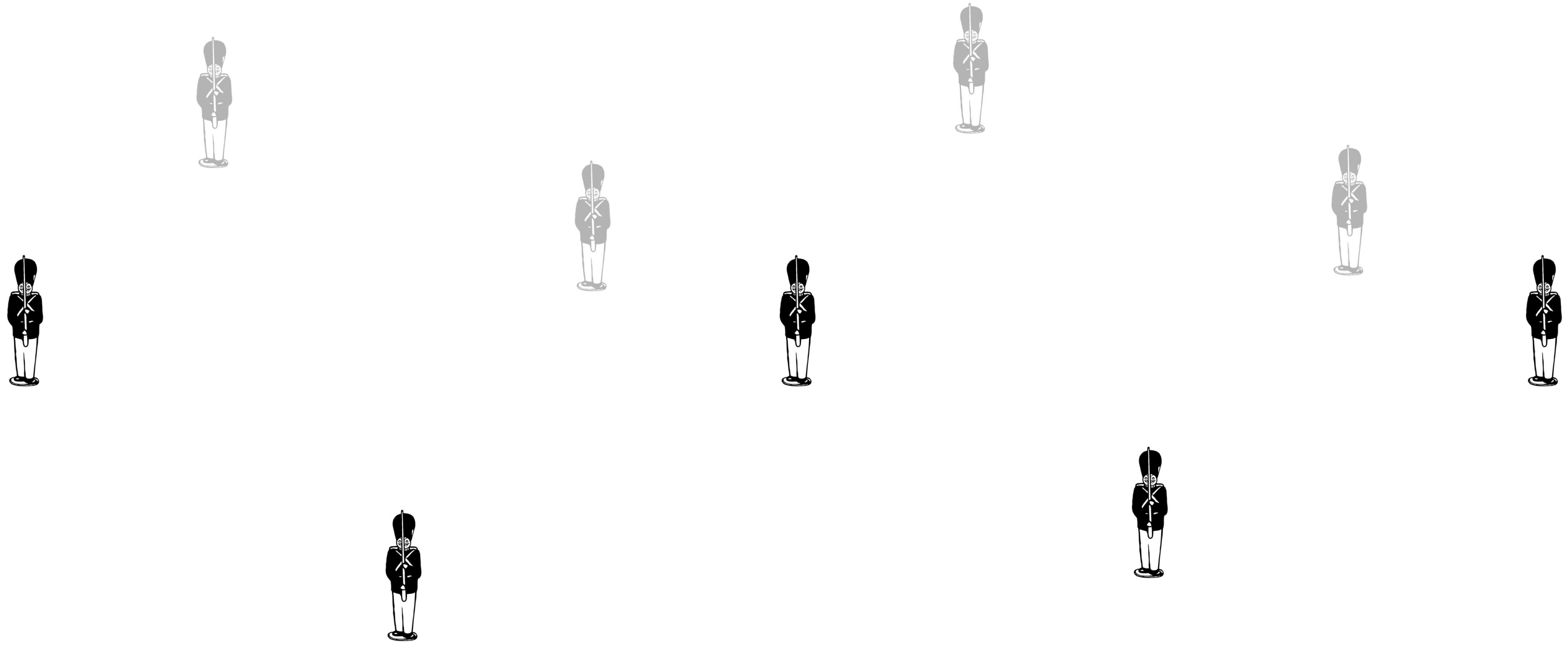
Large scale



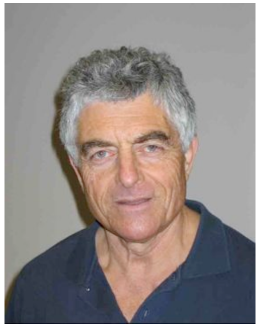


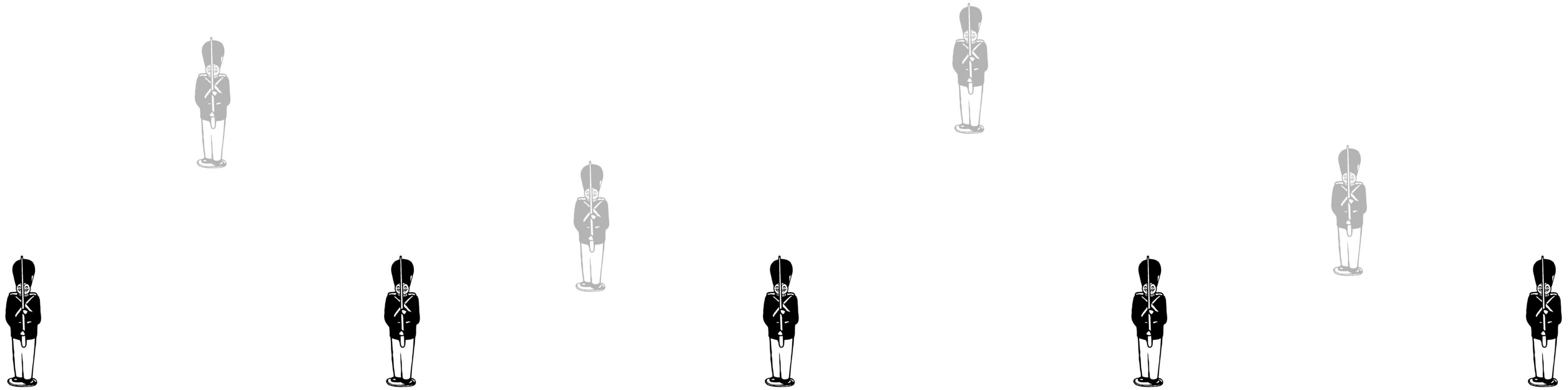
Large scale



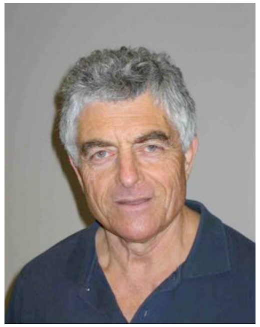


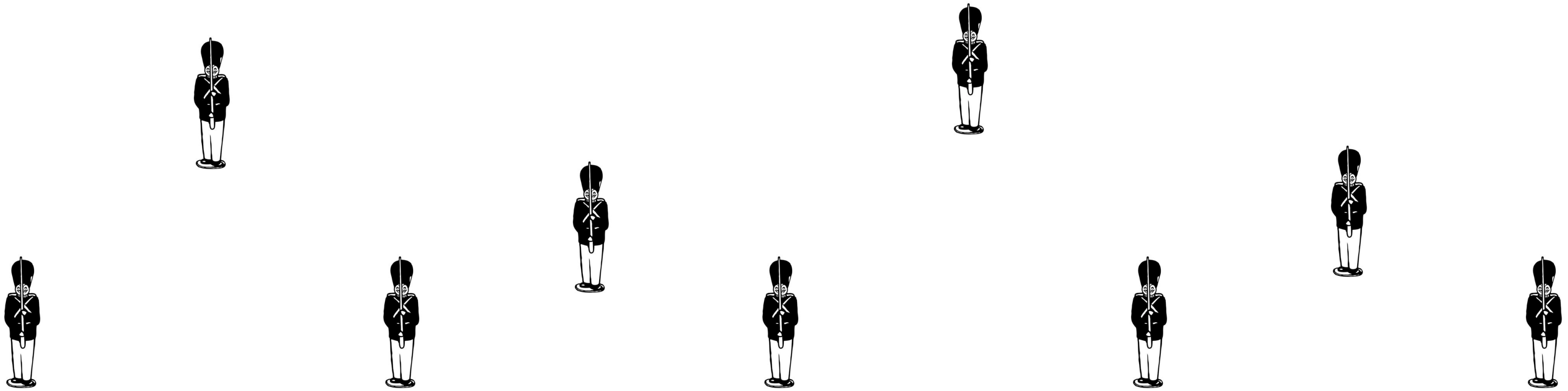
Intermediate scale



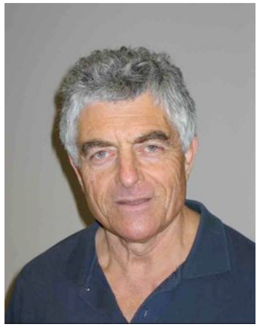


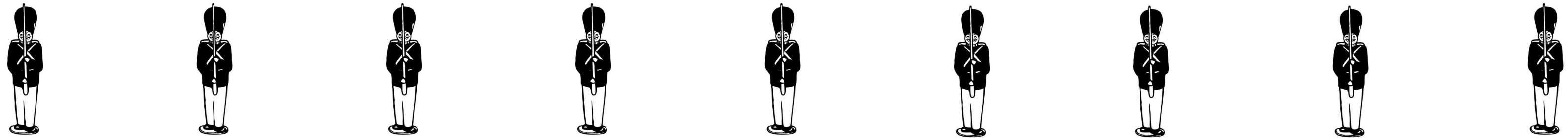
Intermediate scale

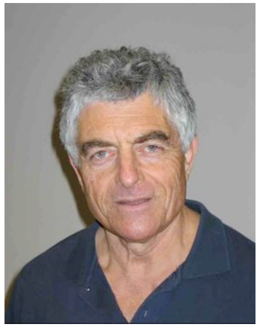




Small scale







How much work do we save?

Jacobi’s method requires about N^2 iterations and $N^2 *N = N^3$ operations to improve the accuracy by an order of magnitude.

The multiscale approach solves the problem in about $\text{Log}_2(N)$ iterations (whistle blows) and only about N operations.

Example: for $N = 1000$ we require about:

10 iterations and 1000 operations

instead of about

1,000,000 iterations and 1,000,000,000 operations

How important is computational efficiency?

Suppose that we have three different algorithms for a given problem, with different computational complexities for input size *N* :

Algorithm 1: 10^6 *N* operations

Algorithm 2: 10^3 N^2 operations

Algorithm 3: N^3 operations

Suppose that the problem size, *N*, is such that Algorithm 1 requires one second.

How long do the others require?

Algorithm 3 $O(N^3)$	Algorithm 2 $O(N^2)$	Algorithm 1 $O(N)$	N	Computer Speed (ops/sec)
0.000001 sec	0.001 sec	1 sec	1	1M (~10^6) (1980's)
1 sec	1 sec	1 sec	1K	1G (~10^9) (1990's)
12 days	17 min	1 sec	1M	1T (~10^{12}) (2000's)
31,710 years	12 days	1 sec	1G	1P (~10^{15}) (2010's)

Stronger Computers ⇒

Greater Advantage of Efficient Algorithms!

The catch: in less trivial problems, we cannot construct appropriate equations on the large scales without first propagating information from the small scales.

Skill in developing efficient multilevel algorithms is required for:

1. Choosing a good local iteration.
2. Choosing appropriate coarse-scale variables.
3. Choosing inter-scale transfer operators.
4. Constructing coarse-scale approximations to the fine-scale problem.

Multigrid is not the answer to everything!

+ Sparse, low dimension, large, stiff, elliptic PDE, geometric, smooth long-range effects, structured, isotropic, smoothly varying coefficients, symmetric positive definite.

~ Nonlinear, disordered, anisotropic, discontinuous coefficients, singular-perturbation and non-elliptic PDE, PDE systems, non-symmetric, indefinite, non-deterministic.

- Dense, high-dimensional, small, single-scale.

2D Model Problem

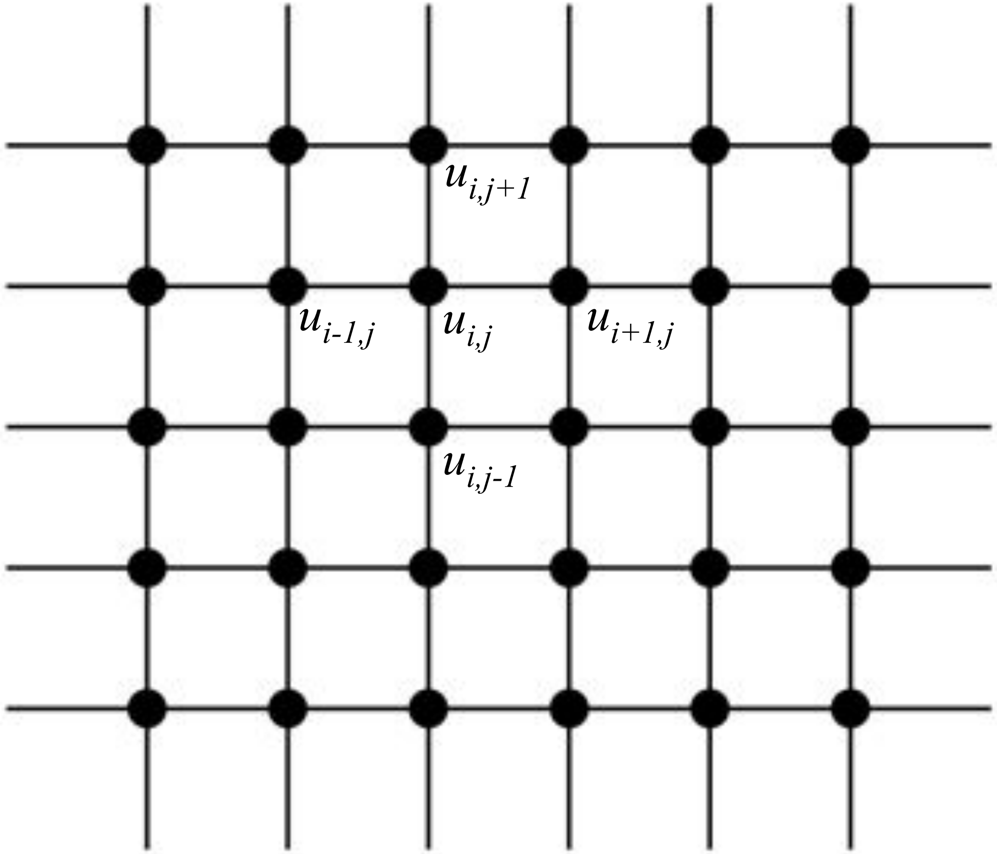
Find *u* which satisfies:

$$\begin{aligned} Lu = u_{xx} + u_{yy} &= f(x,y), \quad (x,y) \in \Omega, \\ u &= g(x,y), \quad (x,y) \in \partial\Omega. \end{aligned} \tag{4}$$

This is the 2D Poisson equation, with Dirichlet boundary conditions. It is an elliptic partial differential equation which appears is many models.

Example for Use of an Iterative Method

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



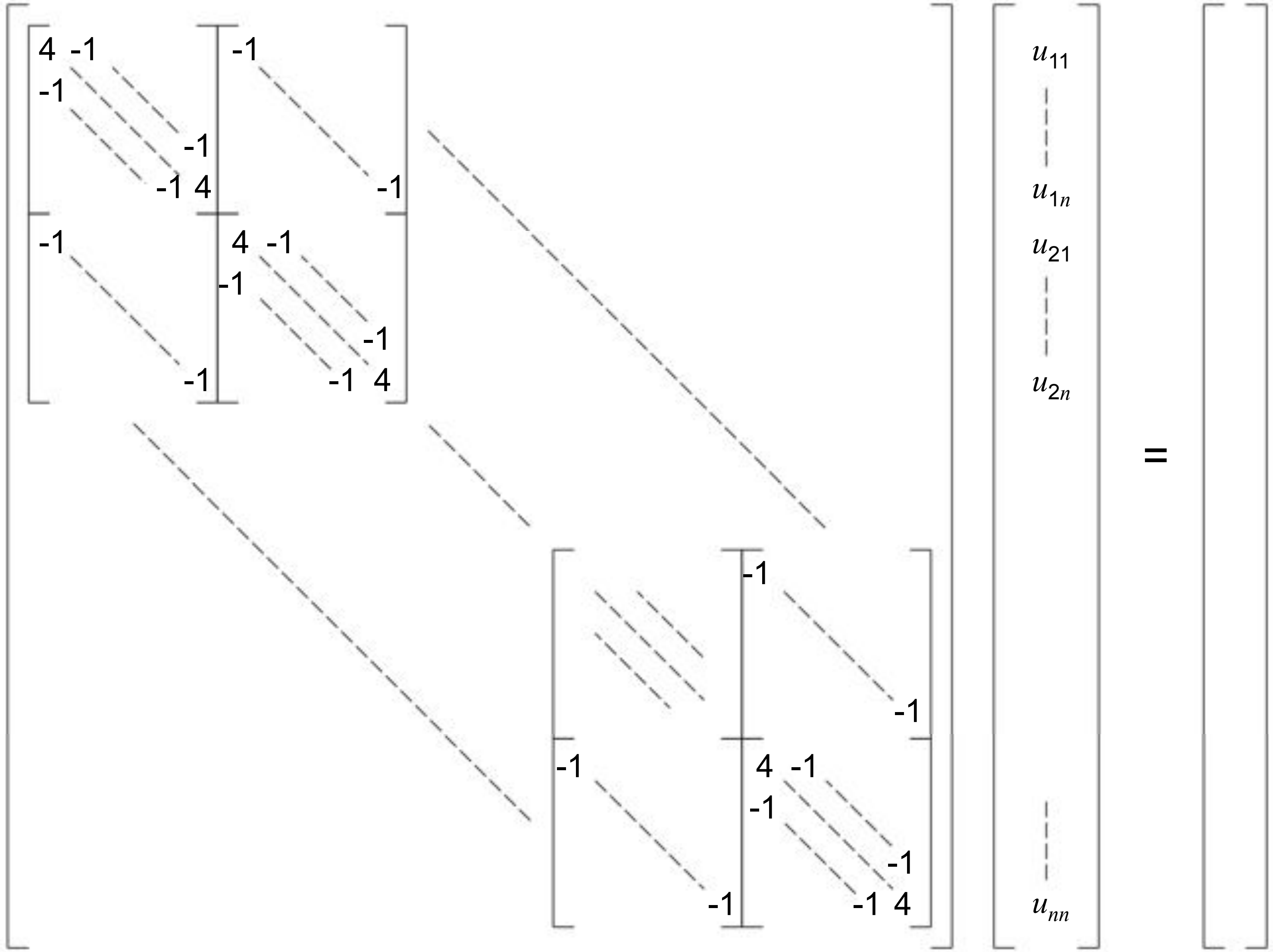
Grid of wires

Solution at each node =
Average of neighboring values

Boundary values given.

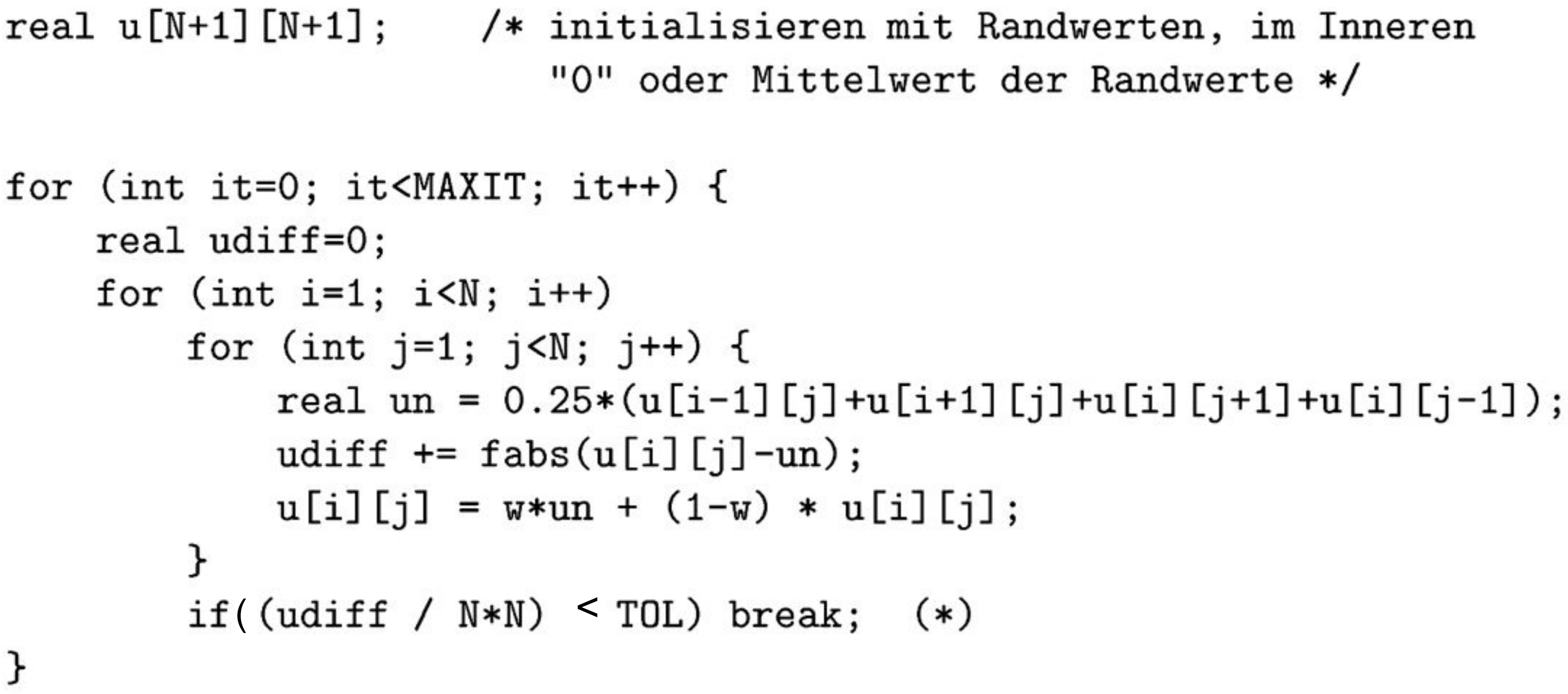
This is the standard 5-point discretization of the Laplace- or Poisson-equation in 2D.

Matrix representation



Iterative Methods

Matrix free implementation of the Gauss-Seidel Method

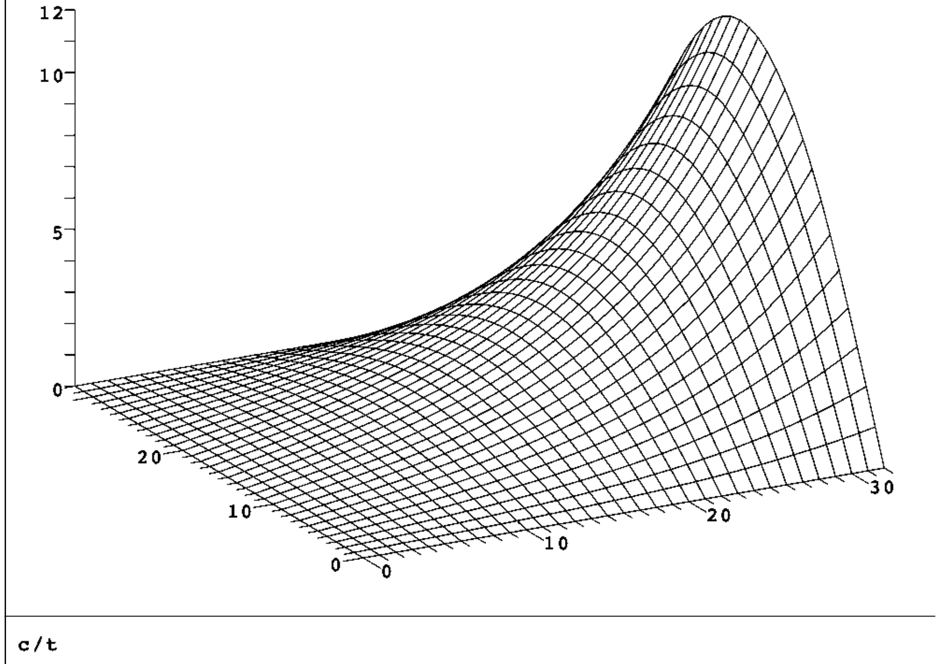


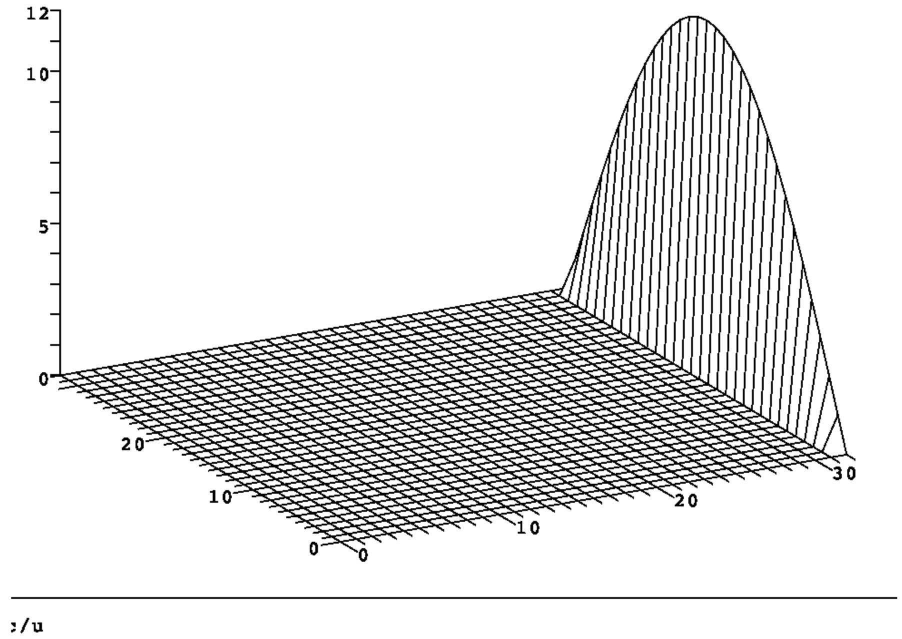
```
real u[N+1][N+1];    /* initialisieren mit Randwerten, im Inneren
                        "0" oder Mittelwert der Randwerte */

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)

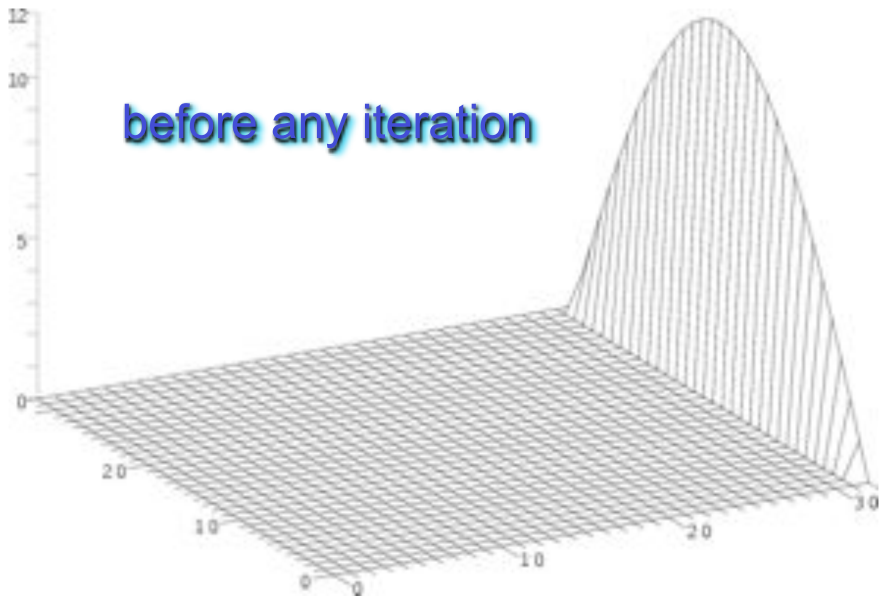
$$u = \sinh(x)\sin(y)$$

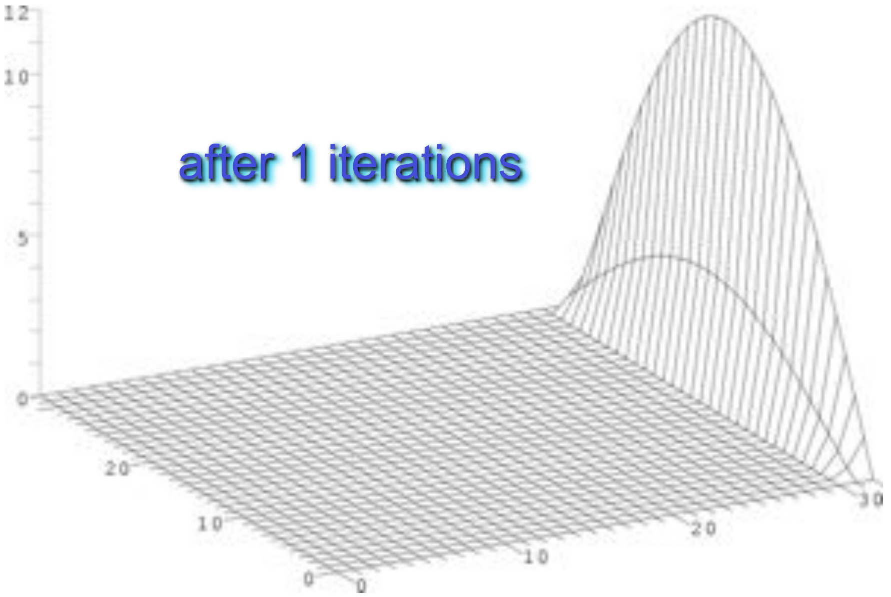


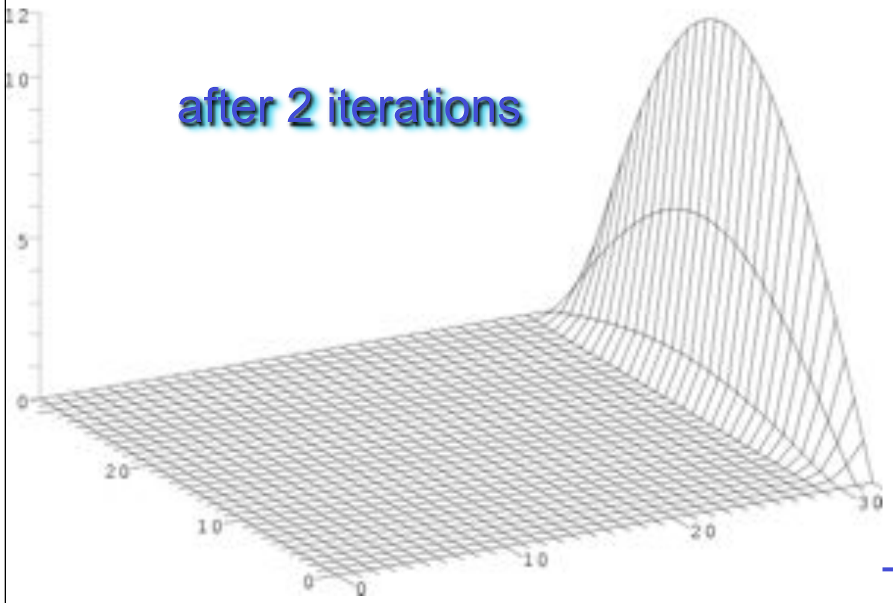


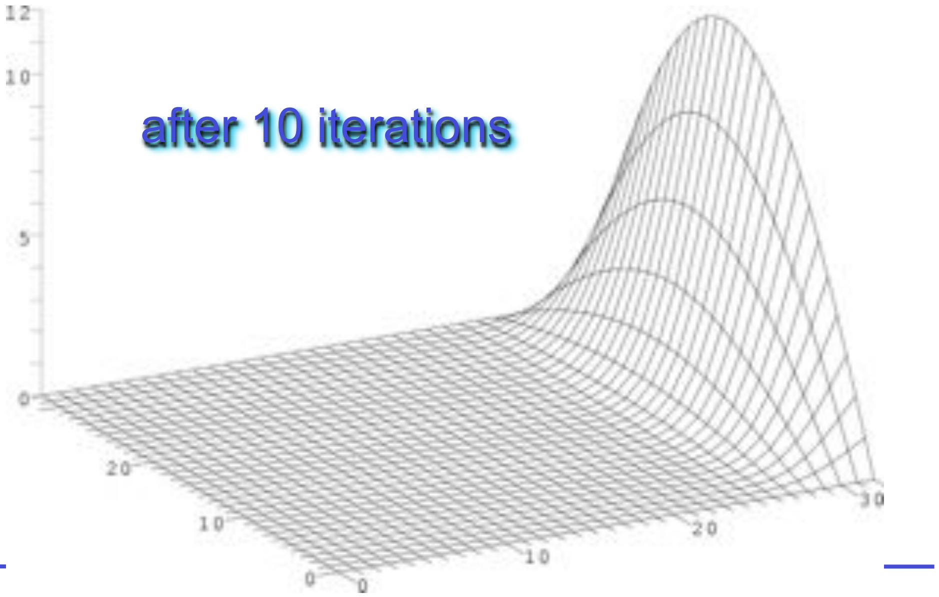
- Exact Solution (of PDE)
- Boundary values to start the iteration

Visualization of Convergence

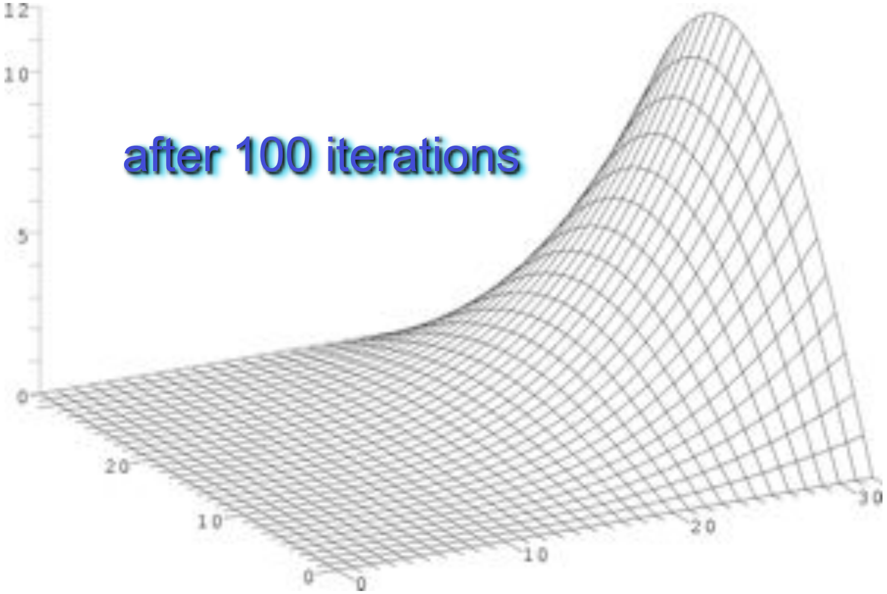

before any iteration


after 1 iterations

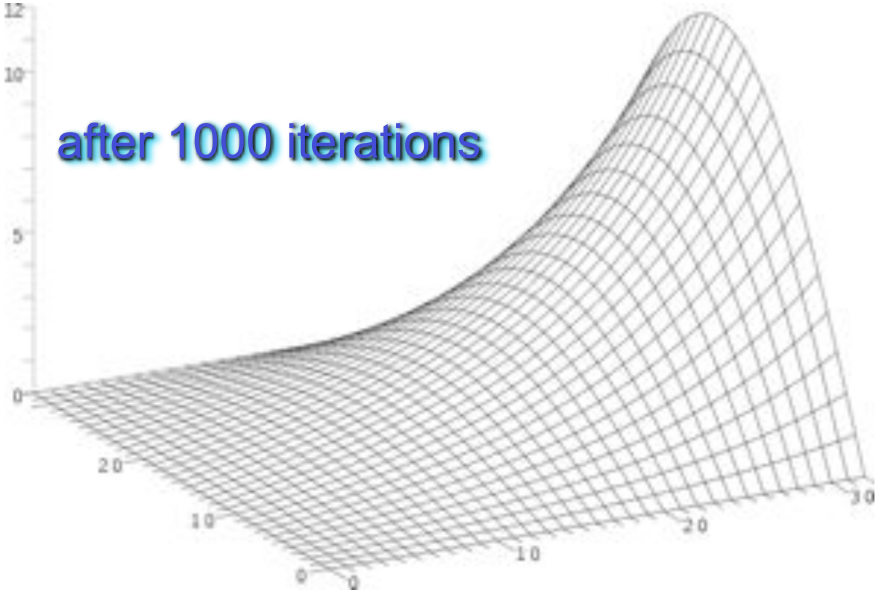

after 2 iterations


after 10 iterations

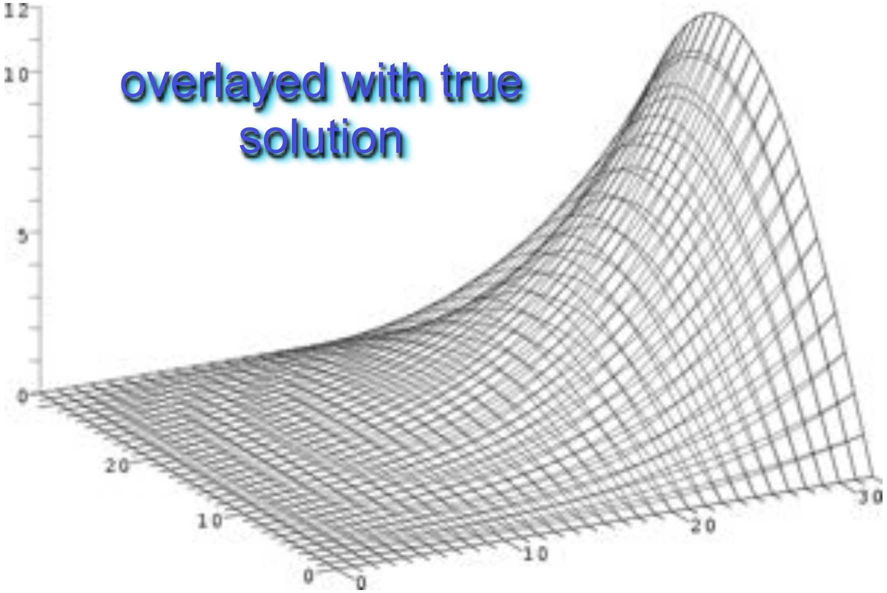
Visualization of Convergence



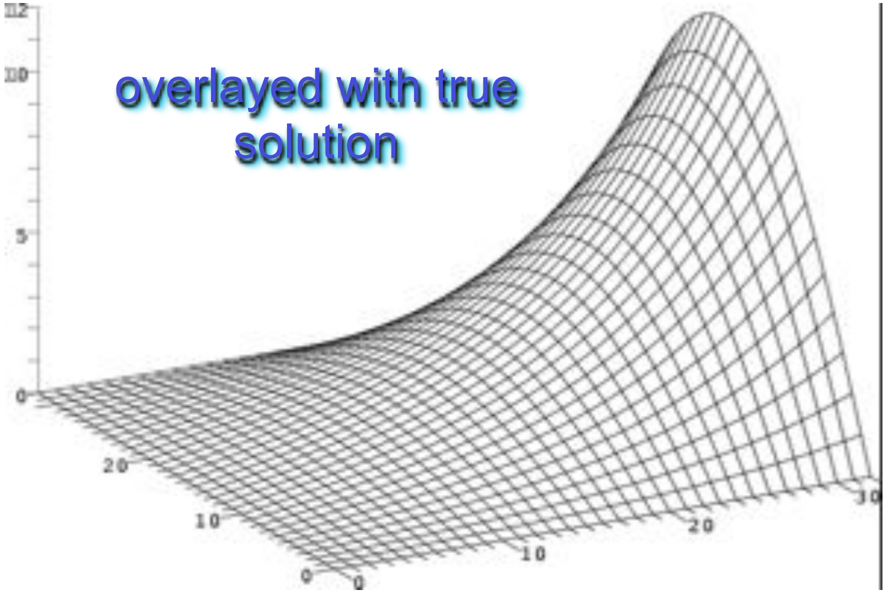
after 100 iterations



after 1000 iterations

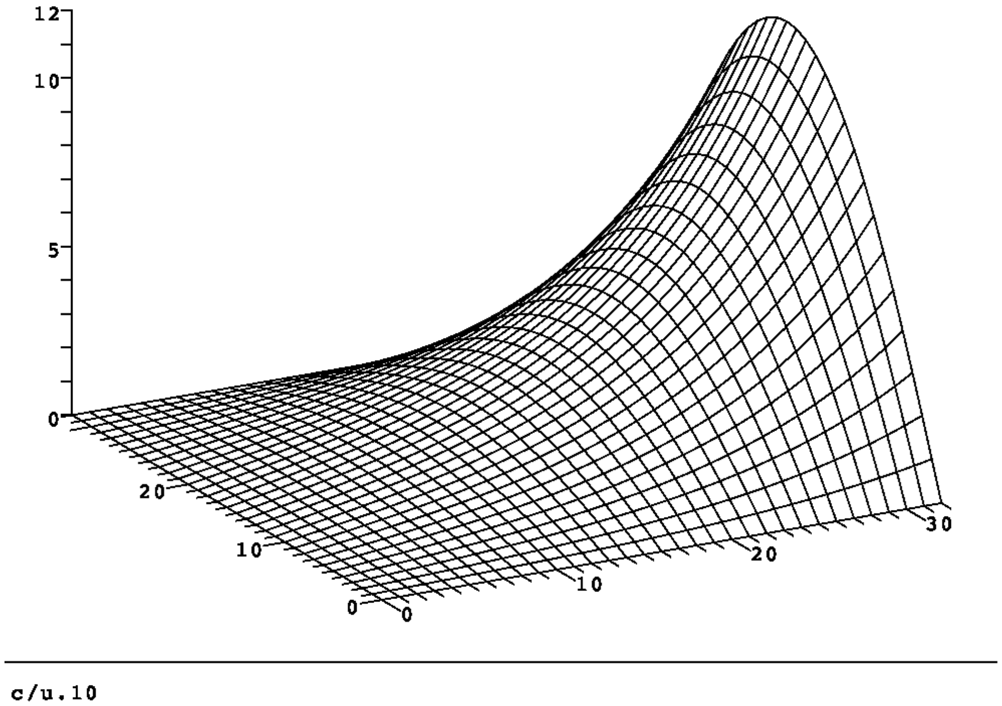


overlayed with true solution



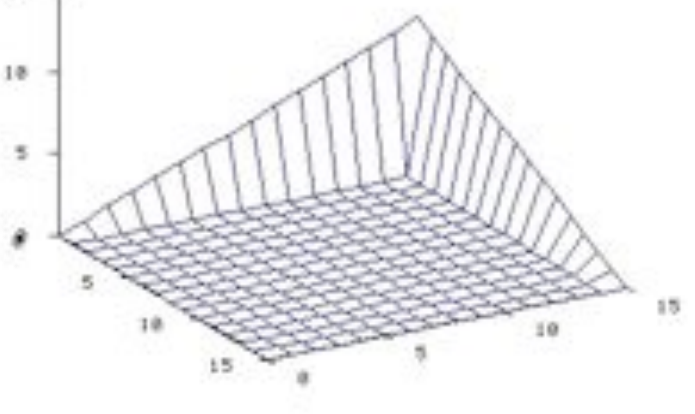
overlayed with true solution

Beispiel: Iterations -- Visualisation

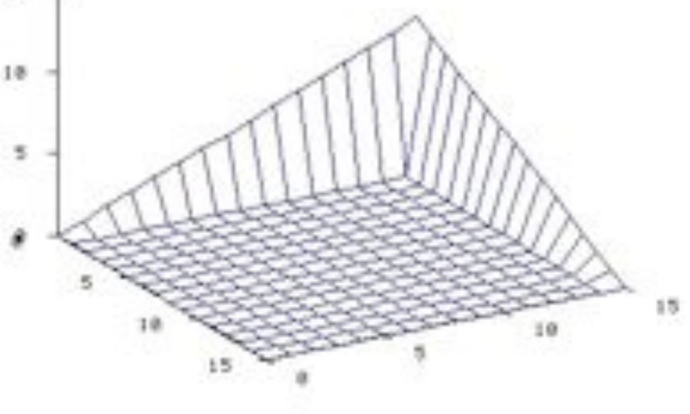


- View of iterative solution after 1000. Gauss-Seidel-Iterations
- The Gauss-Seidel-method needs for this problem $\mathrm{O}(N)$ iterations, where *N* (=n^2) is the number of unknowns (grid points).

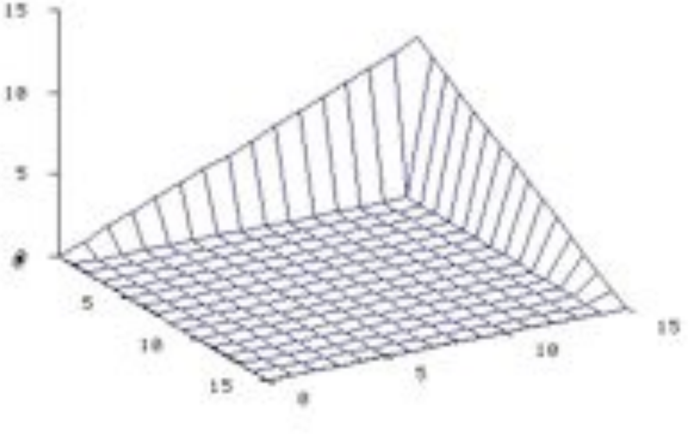
ω = 1.0



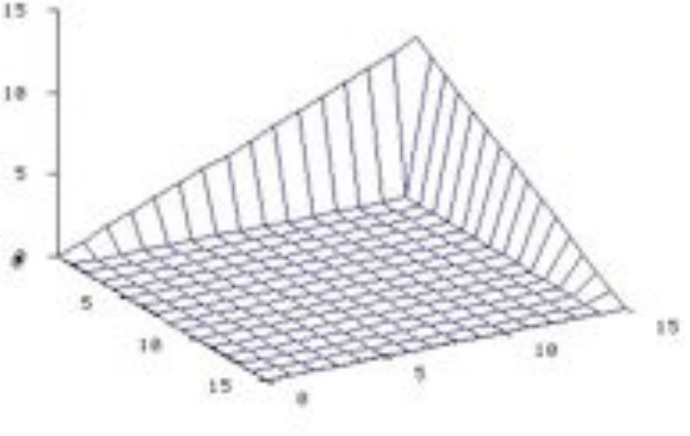
ω = 1.5



ω = 1.8



ω = 1.95



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 parameter $\omega>1$ is chosen optimally.
  - For 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)

Key Observation re-worded: Relaxation cannot be generally efficient for reducing the error (i.e., the difference field $\tilde{u}^h - u^h$). But relaxation may be extremely efficient for smoothing the error relative to the grid.

Practical conclusion:

1. A smooth error can be approximated well on a coarser grid.
2. A coarser grid implies less variables, hence less computation.
3. On the coarser grid the error is no longer as smooth relative to the grid, so relaxation may once again be efficient.

The solution, u^h, depends only on the equation and the data, so it is not, of course, smoothed by relaxation. Only the error is smoothed. Hence, we reformulate our problem:

Denote $v^h = u^h - \tilde{u}^h$.

Recall $L^h u^h = f^h$.

Subtract $L^h \tilde{u}^h$ from both sides, and use the linearity of L^h to obtain:

$$L^h v^h = f^h - L^h \tilde{u}^h \equiv r^h$$

It is this equation that we shall approximate on the coarse grids.

As we have seen, we need to smooth the error on the fine grid first, and only then solve the coarse-grid problem. Hence, we need two types of intergrid transfer operations:

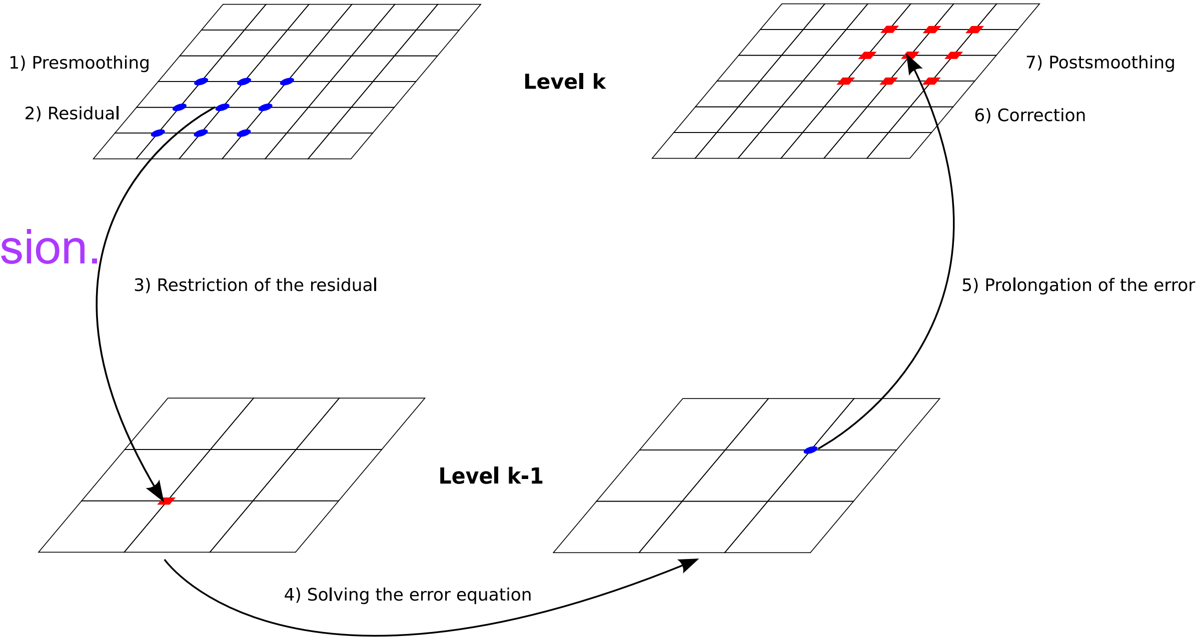
1. A Restriction (fine-to-coarse) operator: I_h^H.
2. A Prolongation (coarse-to-fine) operator: I_H^h.

For restriction we can often use simple injection, but full-weighted (local averaging) transfers are preferable.

For prolongation, linear interpolation (bi-linear in 2D) is simple and usually effective.

Two-grid Algorithm

- ✧ Relax several times on grid *h*, obtaining with a smooth corresponding error. $\tilde{u}^h$
- ✧ Calculate the residual: $r^h = f^h - L^h\tilde{u}^h.$
- ✧ Solve approximate error-equation on the coarse grid: $L^H v^H = f^H \equiv I_h^H r^h.$
- ✧ Interpolate and add correction: $\tilde{u}^h \leftarrow \tilde{u}^h + I_H^h v^H.$
- ✧ Relax again on grid h.


1) Presmoothing
2) Residual
Level k
7) Postsmoothing
6) Correction
3) Restriction of the residual
5) Prolongation of the error
Level k-1
4) Solving the error equation

Multi-grid is obtained by recursion.

Multi-grid Cycle $V(\nu_1, \nu_2)$

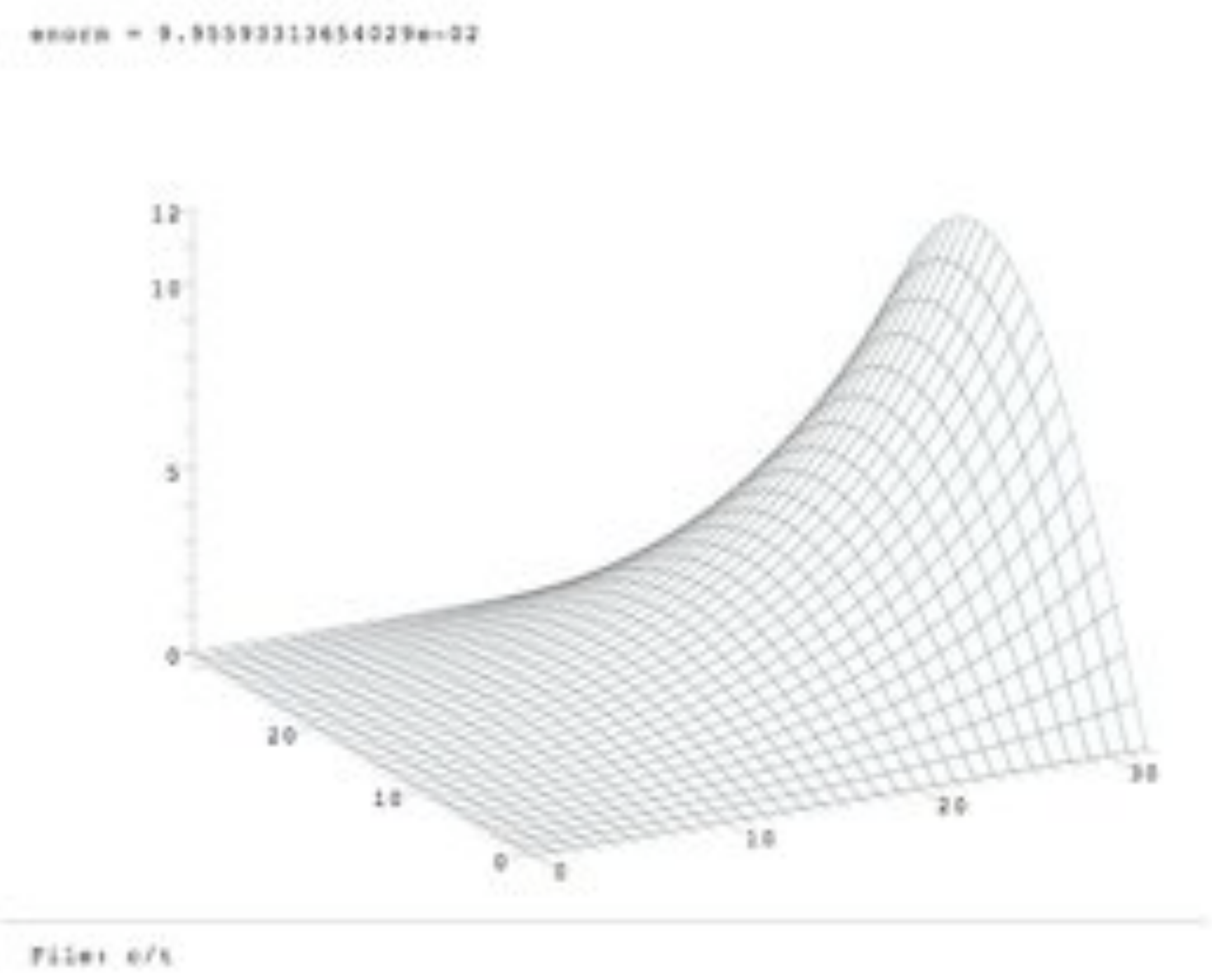
Let u^{2h} approximate v^{2h}, u^{4h} approximate the error on grid 2h, etc.

```
Relax on $L^h u^h = f^h$  $\nu_1$ times
Set $f^{2h} = I_h^{2h}(f^h - L^h u^h)$, $u^{2h} = 0$
   Relax on $L^{2h} u^{2h} = f^{2h}$  $\nu_1$ times
   Set $f^{4h} = I_{2h}^{4h}(f^{2h} - L^{2h} u^{2h})$, $u^{4h} = 0$
      Relax on $L^{4h} u^{4h} = f^{4h}$  $\nu_1$ times
      Set $f^{8h} = I_{4h}^{8h}(f^{4h} - L^{4h} u^{4h})$, $u^{8h} = 0$
         ...
         Solve $L^{Mh} - u^{Mh} = f^{Mh}$
         ...
      Correct $u^{4h} \leftarrow u^{4h} + I_{8h}^{4h} u^{8h}$
      Relax on $L^{4h} u^{4h} = f^{4h}$  $\nu_2$ times
   Correct $u^{2h} \leftarrow u^{2h} + I_{4h}^{2h} u^{4h}$
   Relax on $L^{2h} u^{2h} = f^{2h}$  $\nu_2$ times
Correct $u^h \leftarrow u^h + I_{2h}^h u^{2h}$
Relax on $L^h u^h = f^h$  $\nu_2$ times
```

Remarks:

1. Simple recursion yields a *V* cycle. More generally, we can choose a cycle index ã, and define a ã–cycle recursively as follows: Relax; transfer to next coarser grid; perform ã cycles; interpolate and correct; Relax. (On the coarser grid define the cycle as an exact solution).
2. The best number of pre-relaxation + post-relaxation sweeps is normally 2 or 3.
3. The boundary conditions for all coarse-grid problems is zero (because the coarse-grid variable is the error). The initial guess for the coarse-grid solution must be zero.

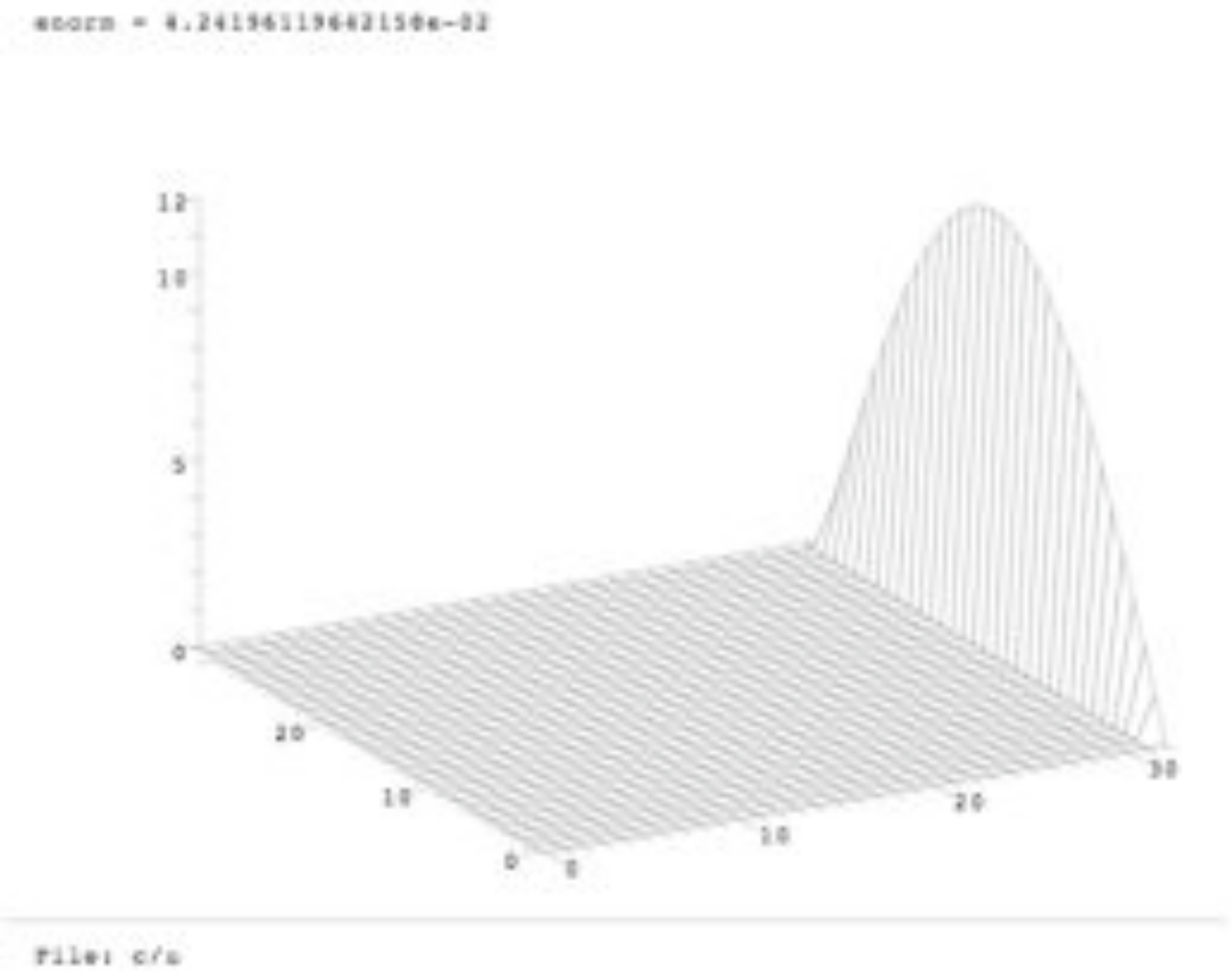
Solution of Poisson‘s eqn



```
enorm = 9.95593313654029e-02
```

```
File: e/t
```

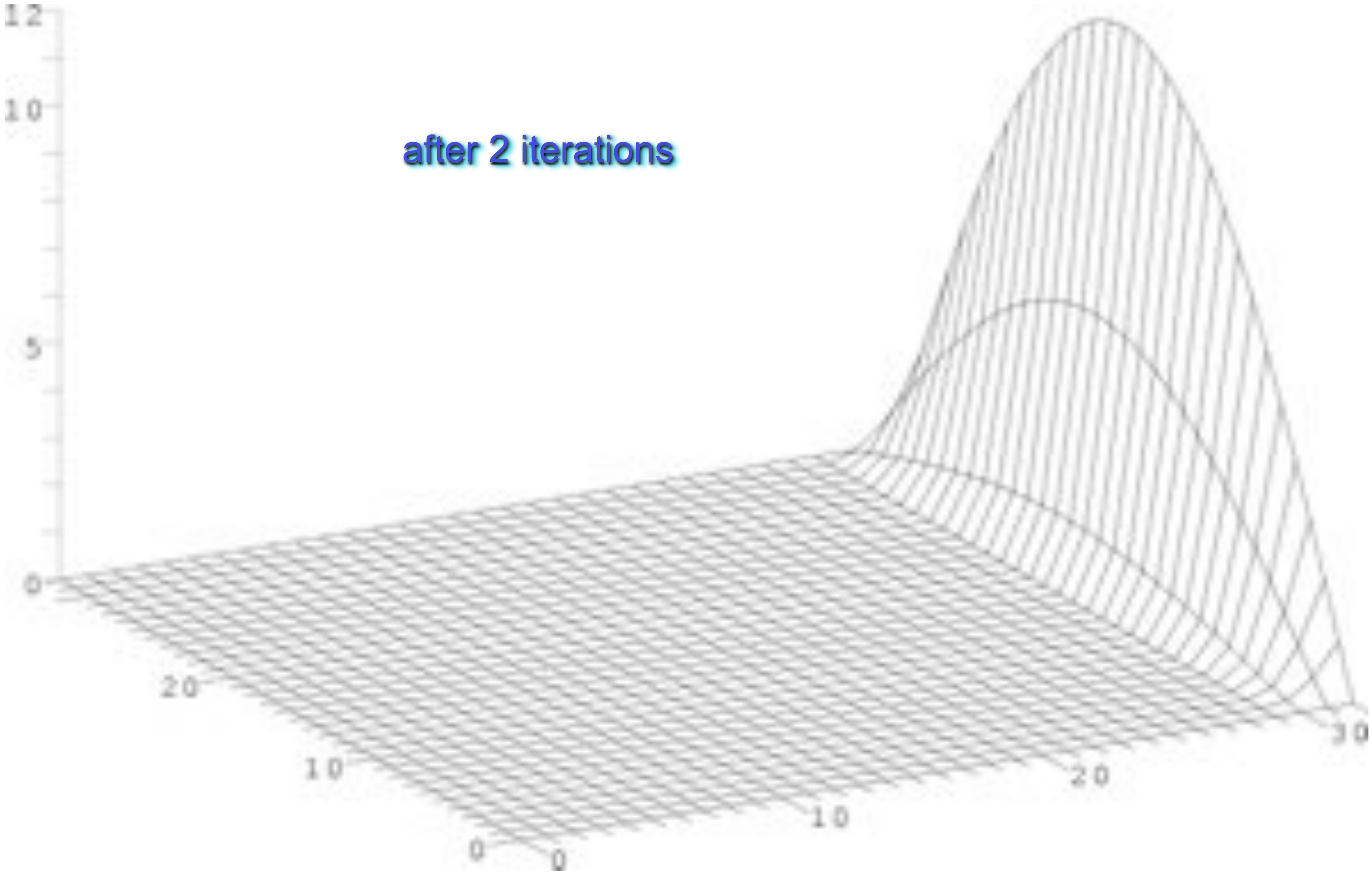
Dirichlet boundary conditions



```
enorm = 4.24196119642150e-02
```

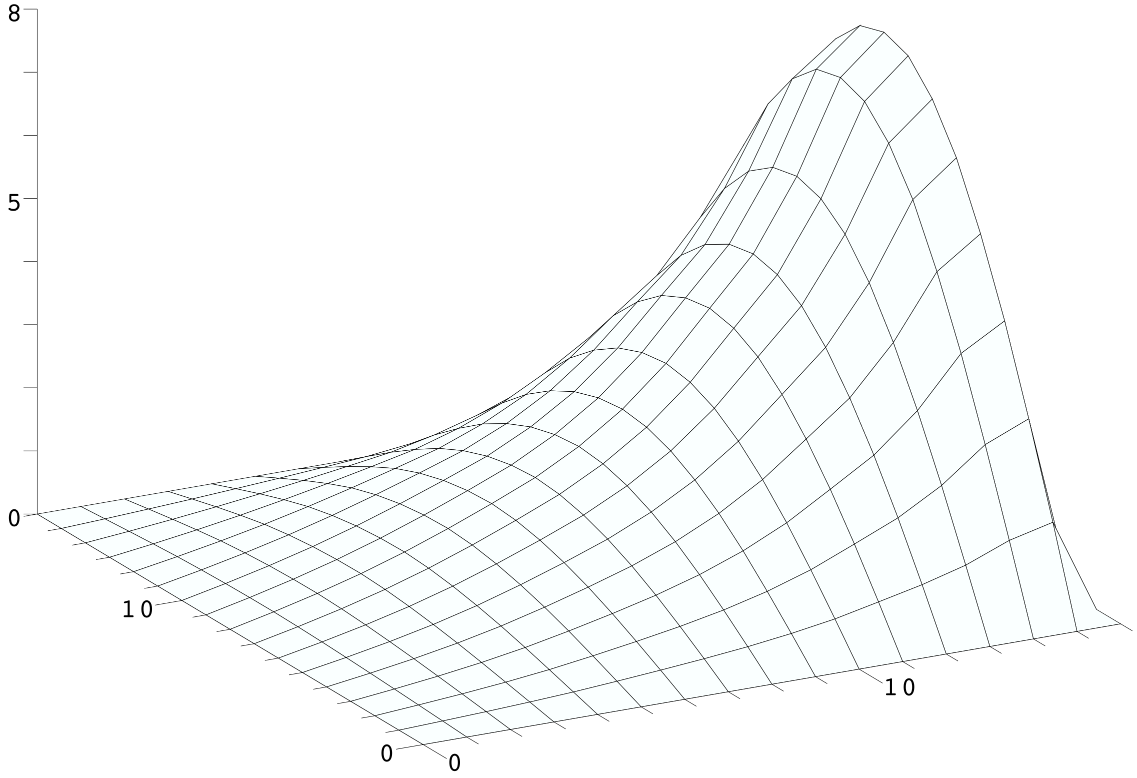
```
File: c/u
```

After 2 steps of Gauss-Seidel smoothing



On the next coarser grid, approximate solution (after „V-cycle“ recursion)

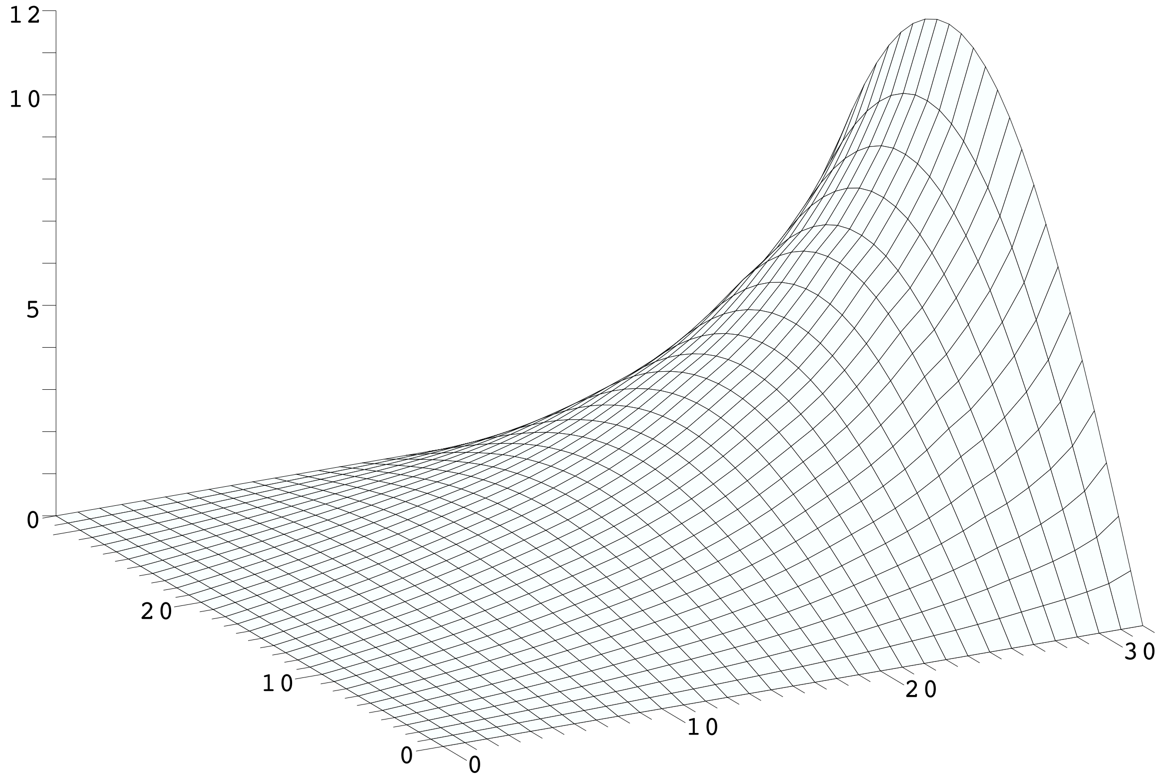
```
enorm = 1.38371516388512e-01
```



```
File: c/c.4/u.26
```

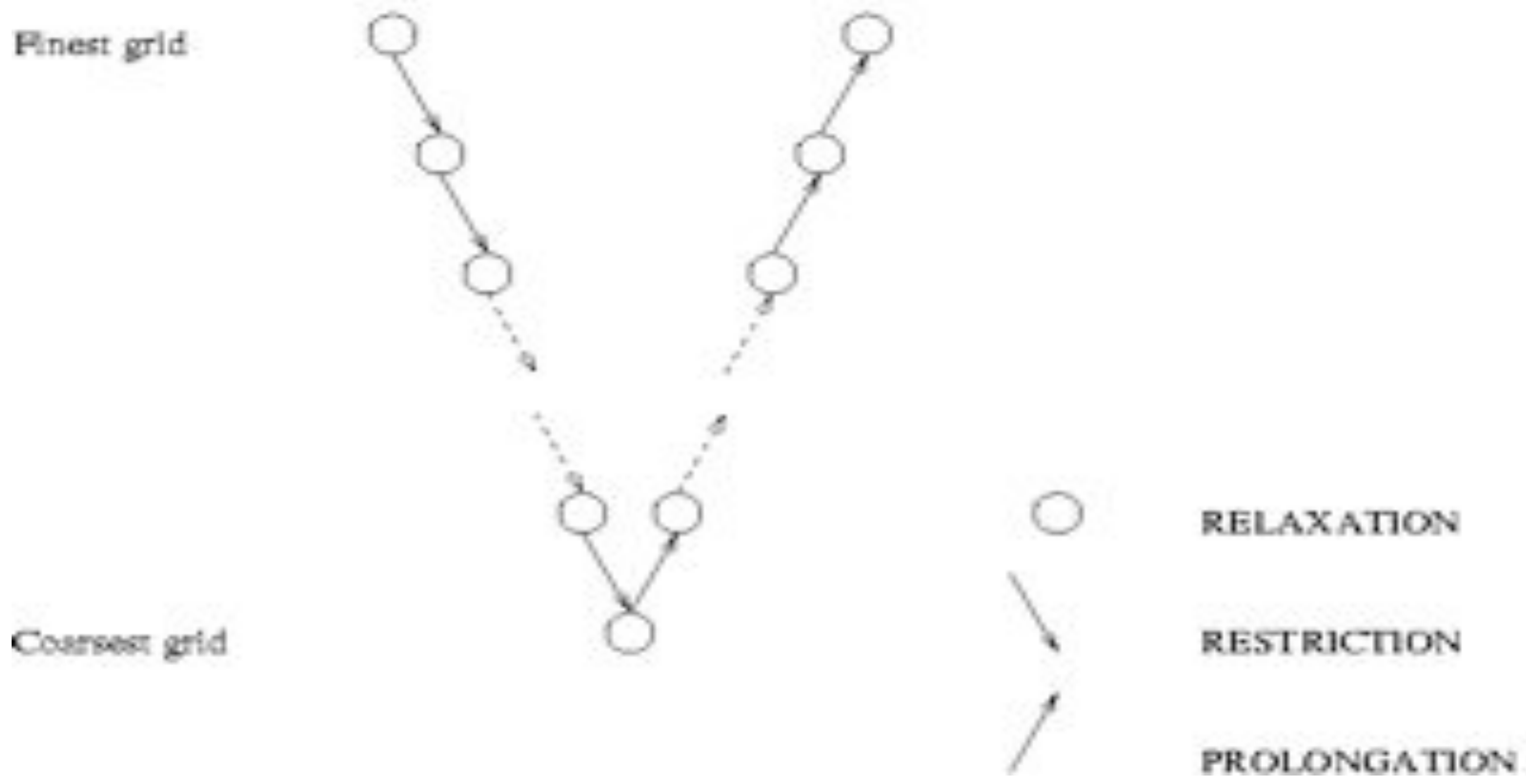
On finest grid, after coarse grid correction

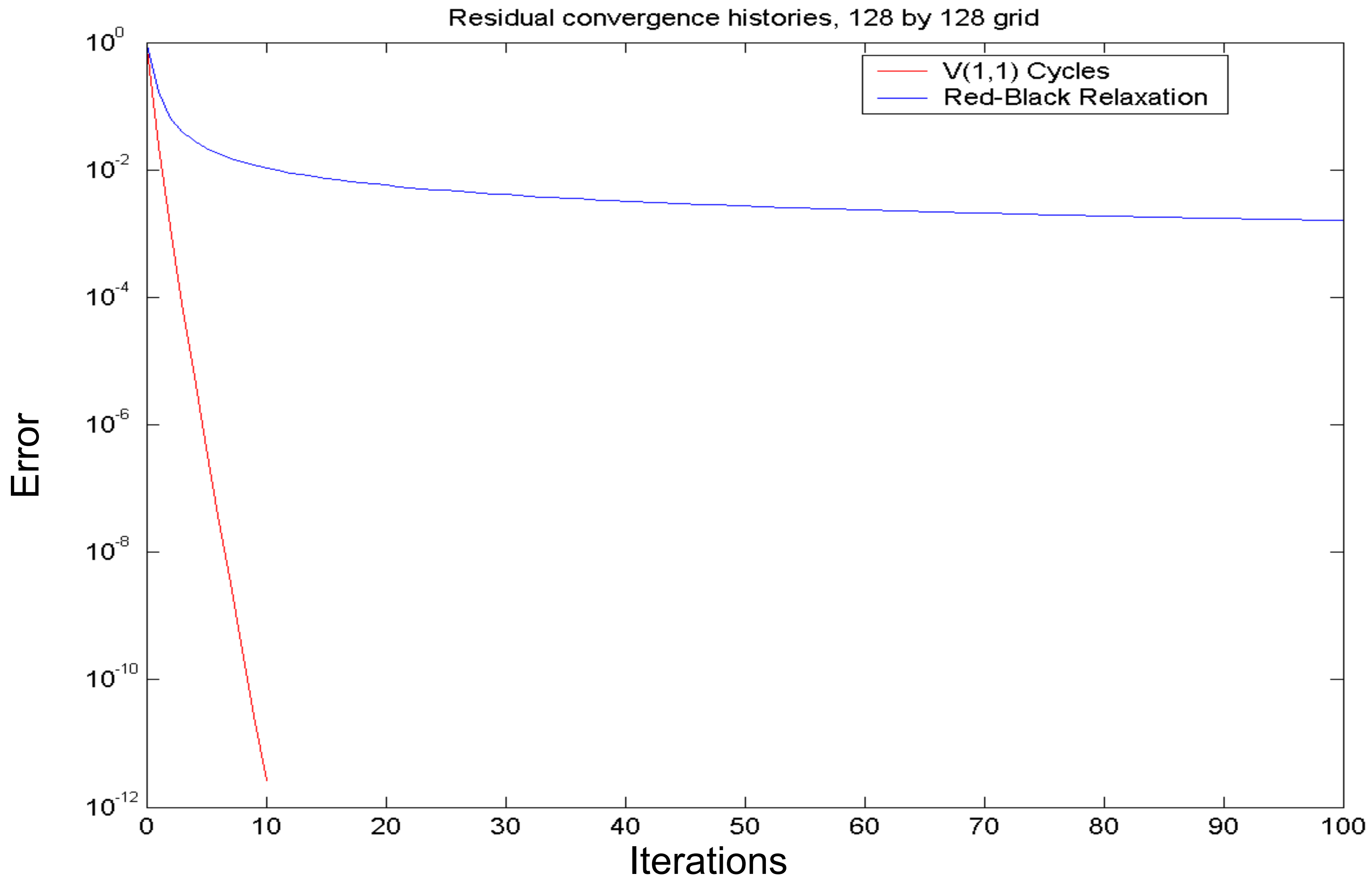
```
enorm = 9.11869751280490e-02
```



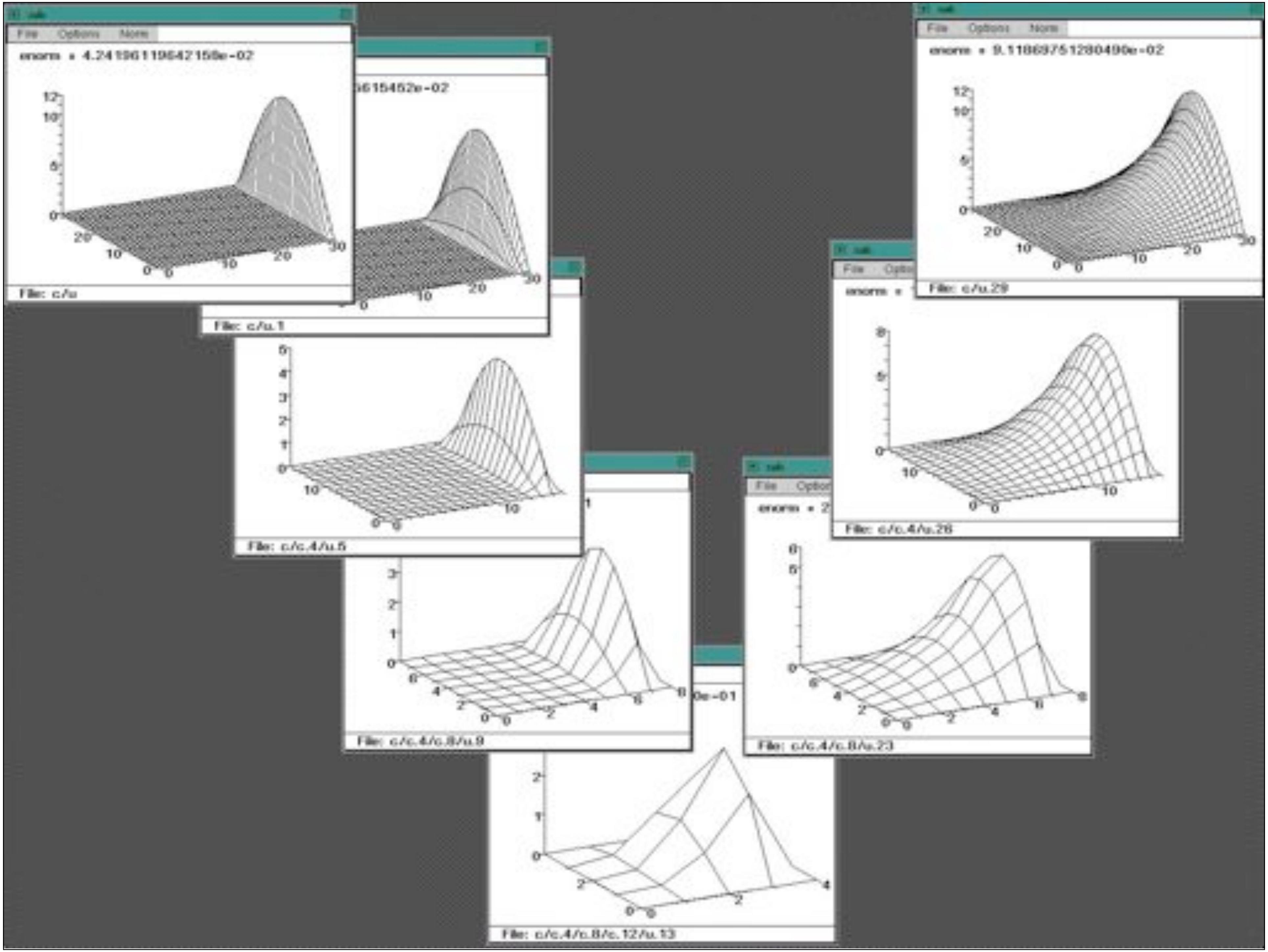
```
File: c/u.29
```

V cycle


Finest grid
Coarsest grid
○ RELAXATION
RESTRICTION
PROLONGATION



Multigrid vs. Relaxation



enorm = 4.24196119642158e-02

File: c/u

5615452e-02

File: c/u.1

File: c/c.4/u.5

File: c/c.4/c.8/u.9

0e-01

File: c/c.4/c.8/c.12/u.13

enorm = 2

File: c/c.4/c.8/u.23

enorm = 1

File: c/c.4/u.26

enorm = 9.11869751280490e-02

File: c/u.29

The **cost** of the *V* cycle in terms of computation and storage is given by

$$cN \sum_{k=0}^{levels-1} 2^{-dk} < cN \frac{2^d}{2^d - 1}$$

Where *d* is the dimension and *N* is the number of variables on the finest grid. Here, *c* is some constant that depends on the discrete operators and the number of relaxation sweeps per level.

Thus, for a 2D problem, the *V*-cycle with one pre-relaxation and one post-relaxation requires approximately the same number of operations as 3-5 relaxation sweeps.

The **convergence rate** of a V-cylce is <1 and bounded away from one for a wide class of elliptic PDE independent of the mesh size. In practice, we try (and often succeed) to achieve $\mu \approx 0.1$.

The **cost** of the *V* cycle in terms of computation and storage is given by

$$cN \sum_{k=0}^{levels-1} 2^{-dk} < cN \frac{2^d}{2^d - 1}$$

Where *d* is the dimension and *N* is the number of variables on the finest grid. Here, *c* is some constant that depends on the discrete operators and the number of relaxation sweeps per level.

Thus, for a 2D problem, the *V*-cycle with one pre-relaxation and one post-relaxation requires approximately the same number of operations as 3-5 relaxation sweeps.

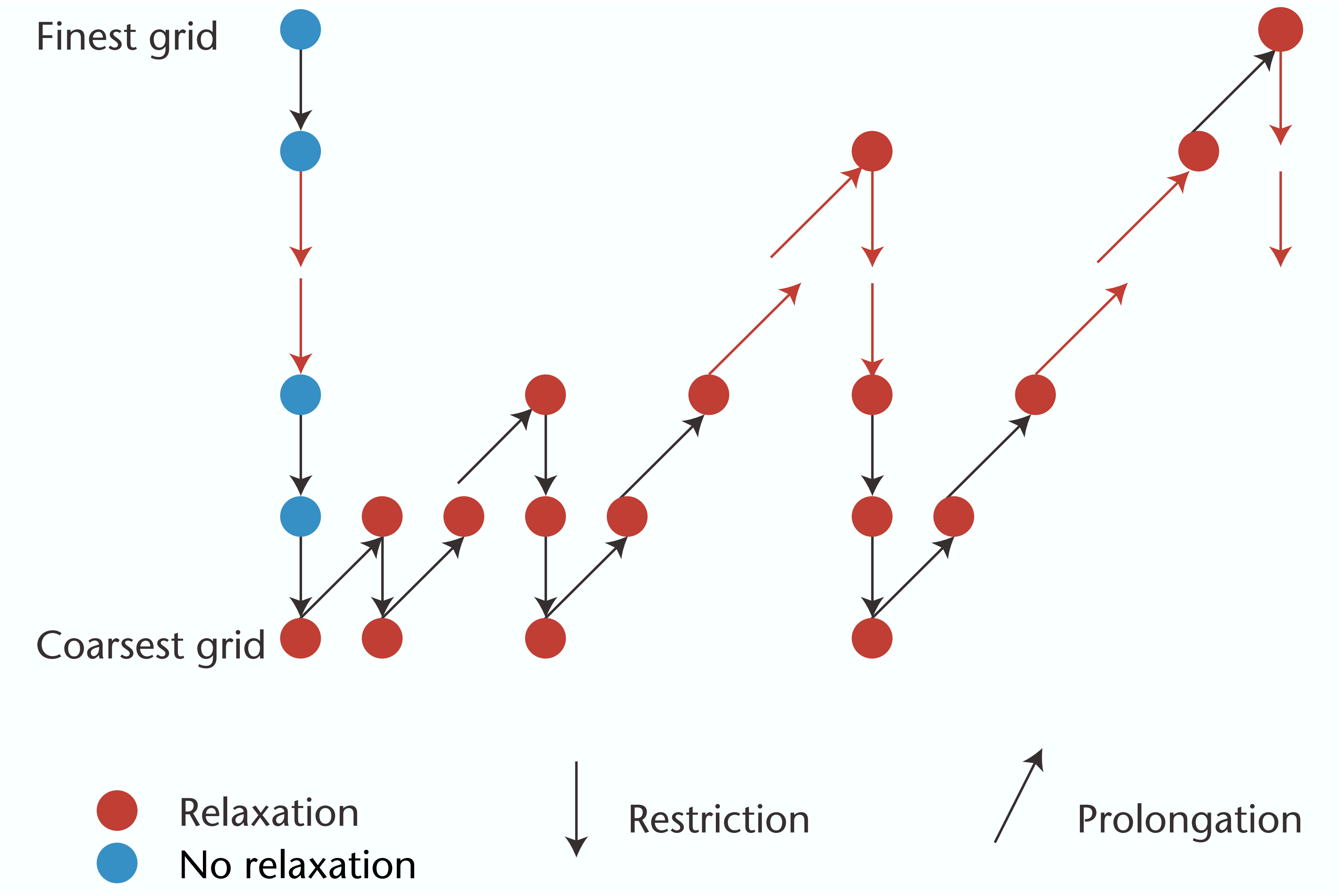
The **convergence rate** of a V-cylce is <1 and bounded away from one for a wide class of elliptic PDE independent of the mesh size. In practice, we try (and often succeed) to achieve $\mu \approx 0.1$.

The Full Multi-Grid (FMG) Algorithm

The multigrid V-cycle is an iterative method, and hence it requires an initial guess for the solution. This initial approximation is obtained from a coarser grid, and so on recursively.

The FMG algorithm combines the grid-refinement approach with the V-cycle.

For many problems, FMG with just a single V-cycle per level suffices to reduce the error below truncation level. In this case, only $O(N)$ operations are required overall.


Finest grid
Coarsest grid
Relaxation
No relaxation
Restriction
Prolongation

High Performance Systems (on the way to Exa-Flops)

How much is ExaFlops?

- 10^6 = 1 MegaFlops: Intel 486

  33MHz PC (~1989)
- 10^9 = 1 GigaFlops: Intel Pentium III

  1GHz (~2000)
  - If every person on earth computes one operation every 7 seconds, all humans together have ~1 GigaFlops performance (less than a current laptop)
- 10^{12}= 1 TeraFlops: HLRB-I

  1344 Proc., ~ 2000
- 10^{15}= 1 PetaFlops
  - 122 400 Cores (Roadrunner, 2008)
  - 294 912 Cores (Jugene, Jülich, 1.44 10^{14} Bytes Memory)
  - 155 000 Cores (SuperMuc, 3 PFlops, 3.33 10^{14} Bytes Memory)
- If every person on earth runs a 486 PC, we all together have an aggregate Performance of 7 PetaFlops.
- ExaScale (~10^{18} Flops) around 2018?

Example Peta-Scale System: Jugene @ Jülich

Extreme Scaling Workshop 2010
at Jülich Supercomputing Center

- PetaFlops = 10^{15} operations/second
- IBM Blue Gene
- Theoretical peak performance: 1.0027 Petaflop/s
- 294 912 cores
- 144 TBytes = $1.44 \, 10^{14}$
- #9 on TOP 500 List in Nov. 2010

- For comparison: Current fast desktop PC is ~ 20.000 times slower
- > 1 000 000 cores expected 2011
- Exa-Scale System expected by 2018/19 ... likely with ~10^9 cores

What will Computers Look Like in 2020?

- Super Computer (Heroic Computing)
  - Cost: 200 Million €
  - Parallel Threads: 10^8 - 10^9
  - 10^{18} FLOPS, Mem: 10^{15}-10^{17} Byte (1-100 PByte)
  - Power Consumption: 20 MW
- Departmental Server (Mainstream Computing for R&D)
  - Cost: 200 000 €
  - Parallel Threads: 10^5 - 10^6
  - 10^{15} FLOPS, Mem: 10^{12}-10^{14} Byte (1-100 TByte)
  - Power Consumption: 20 KW
- (mobile) Workstation (Computing for the Masses)
  - ... scale down by another factor 100

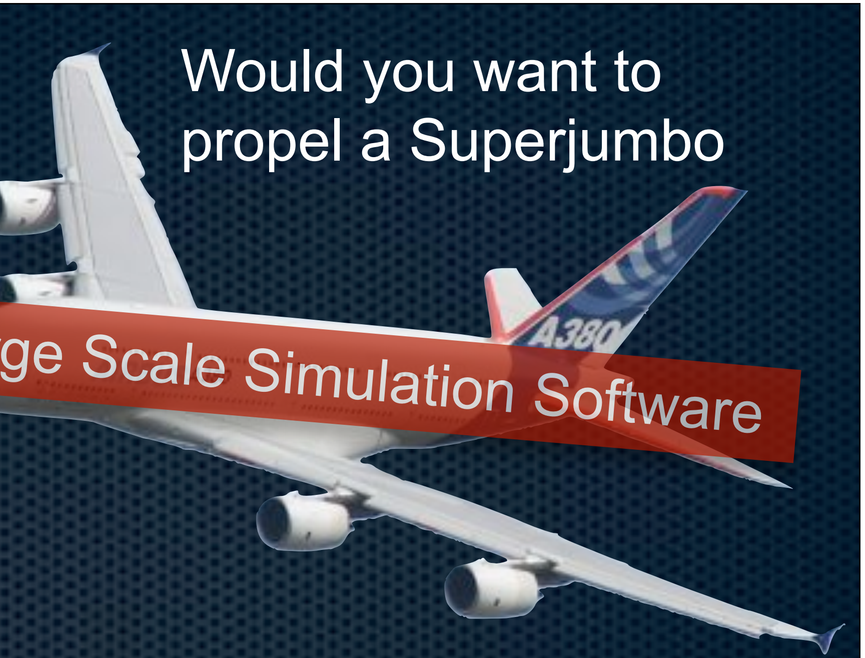
But remember: Predictions are difficult ...
especially those about the future

What‘s the problem?

with four strong jet engines

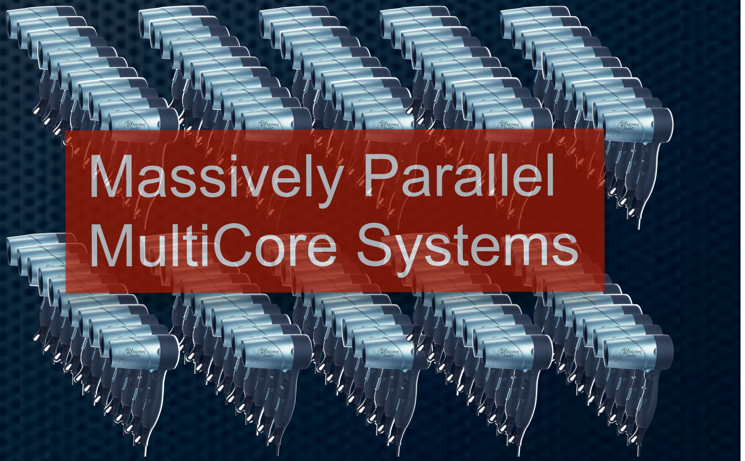
Moderately Parallel Computing

Would you want to
propel a Superjumbo



Large Scale Simulation Software

or with 300,000
blow dryer fans?



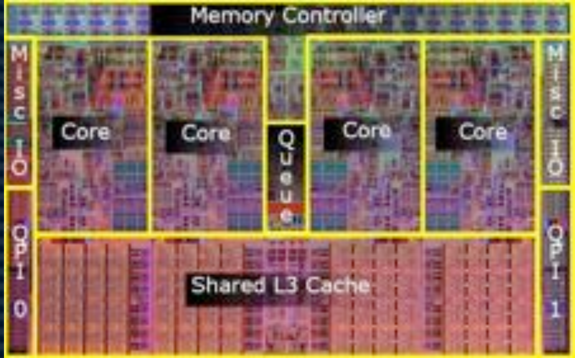
Massively Parallel
MultiCore Systems

What are the problems?

- Unprecedented levels of parallelism
  - maybe billions of cores/threads needed
- Hybrid architectures
  - standard CPU
  - vector units (SSE)
  - accelerators (GPU)
- Memory wall
  - memory response slow: latency
  - memory transfer limited: bandwith
- Power considerations dictate
  - limits to clock speed => multi core
  - limits to memory size (byte/flop)
  - limits to address references per operation
  - limits to resilience

Why Parallel Programming?

- All standard processors are multicore processors
  - “The free lunch is over”
  - To exploit multicore performance, parallel algorithms are essential
  - CPUs will have 2, 4, 8, 16, ..., 128, ..., ??? cores
  - Exa-Scale Systems will have many millions of cores



Current Exa-Scale development dictated by

- power consumption: 10 PicoJoule/Flop:
  - we cannot afford communication
  - we cannot afford memory access
- fault tolerance
  - we must learn to live with system failures and errors

What are the consequences?

- For the application developers “the free lunch is over”
  - Without explicitly parallel algorithms, the performance potential cannot be used any more
- For HPC
  - CPUs will have 2, 4, 8, 16, ..., 128, ..., ??? cores - maybe sooner than we are ready for this
  - We will have to deal with systems with millions of cores
- The memory wall grows higher

Towards Scalable FE Software

Scalable Algorithms and Data Structures

How Fast

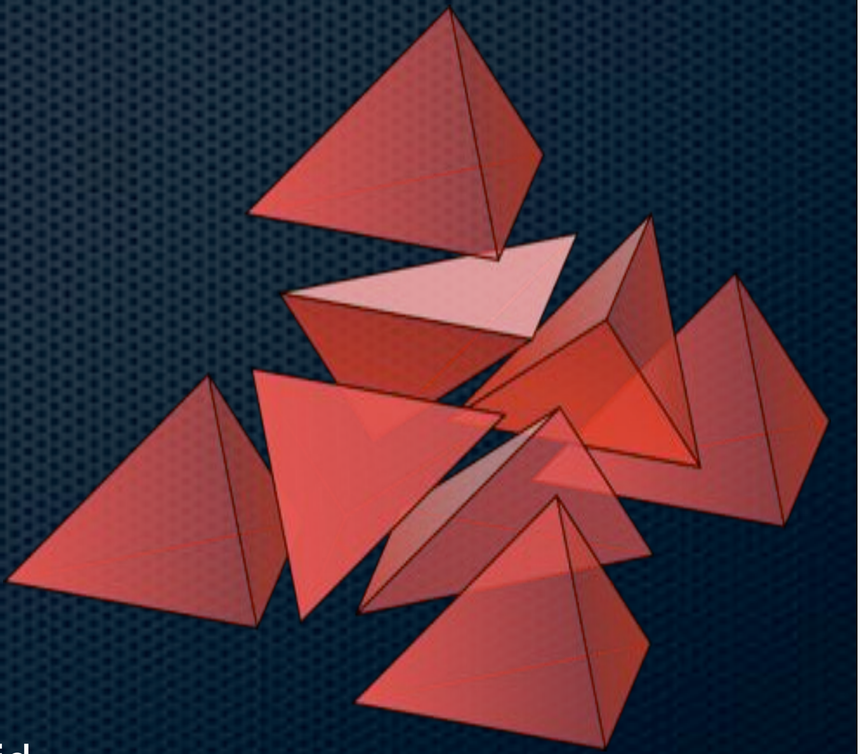
should our simulations be

... and why they aren‘t

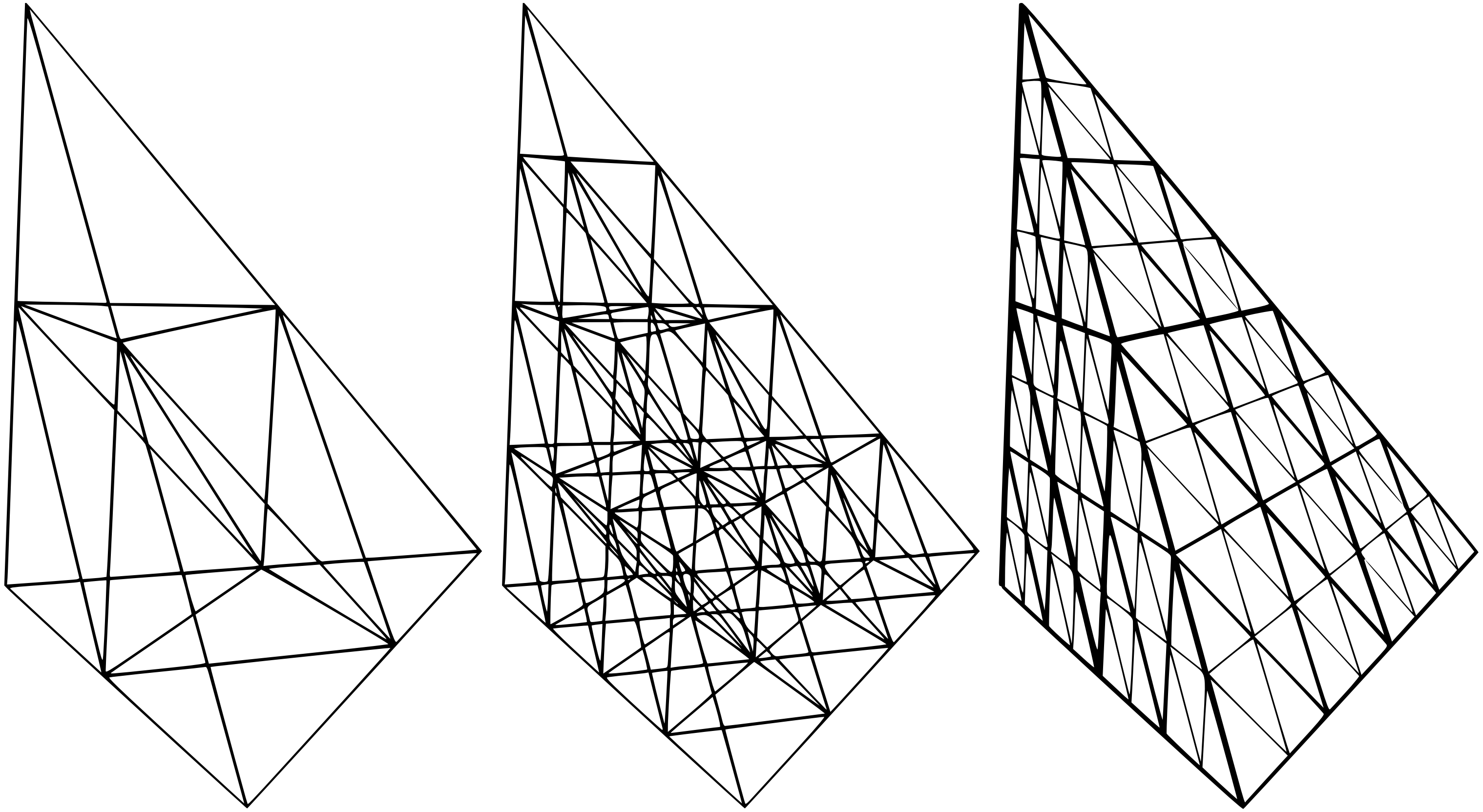
How fast can we make FE multigrid

- Parallelize „plain vanilla“ multigrid for tetrahedral finite elements
  - partition domain
  - parallelize all operations on all grids
  - use clever data structures
  - matrix free implementation
- Do not worry (so much) about Coarse Grids
  - idle processors?
  - short messages?
  - sequential dependency in grid hierarchy?
- Elliptic problems always require global communication. This cannot be accomplished by
  - local relaxation or
  - Krylov space acceleration or
  - domain decomposition without coarse grid

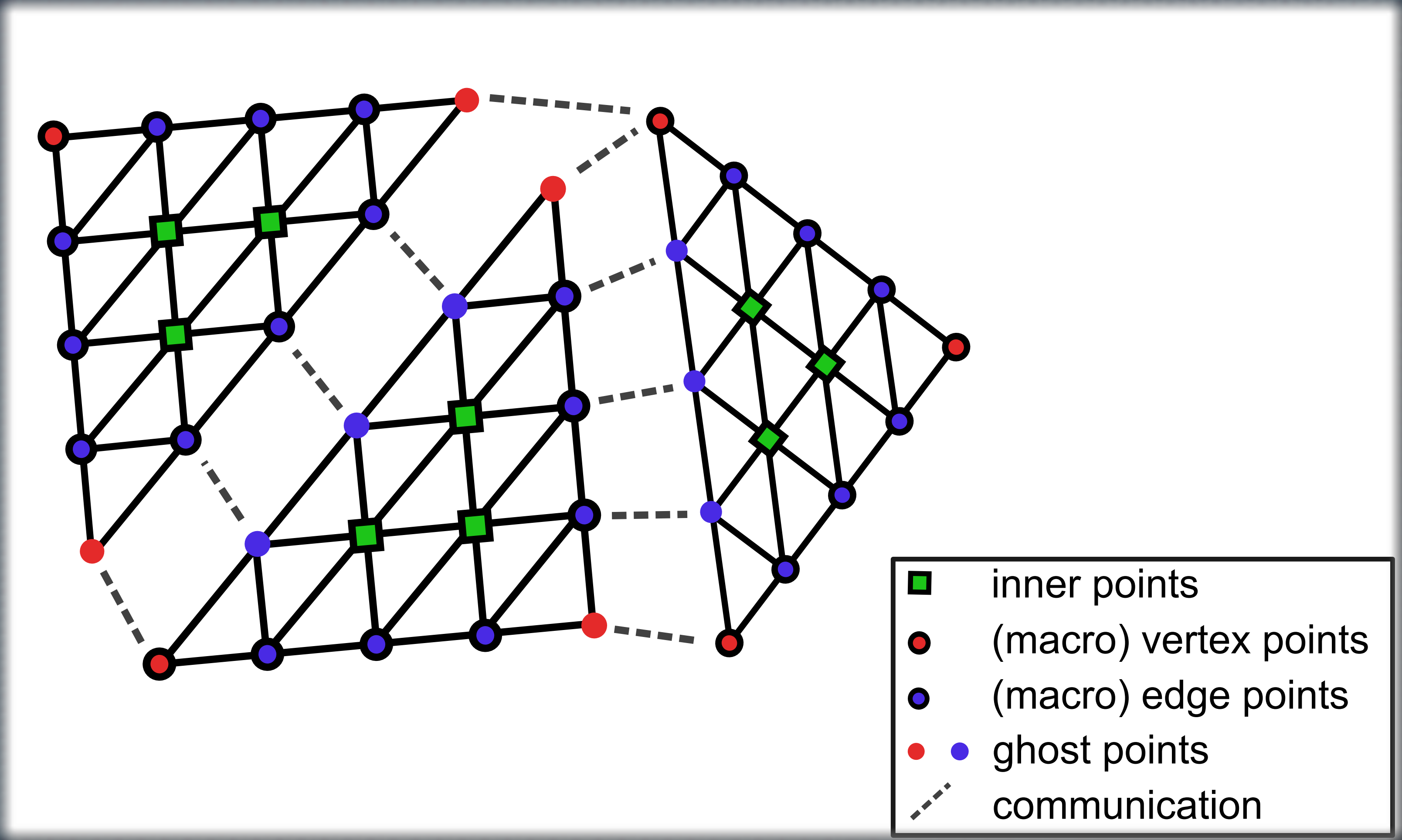
Bey‘s Tetrahedral Refinement



Regular tetrahedral refinement



Grid Partitioning - Communication Pattern



- 🟩 inner points
- 🔴 (macro) vertex points
- 🔵 (macro) edge points
- ● ● ghost points
- ⋰ communication

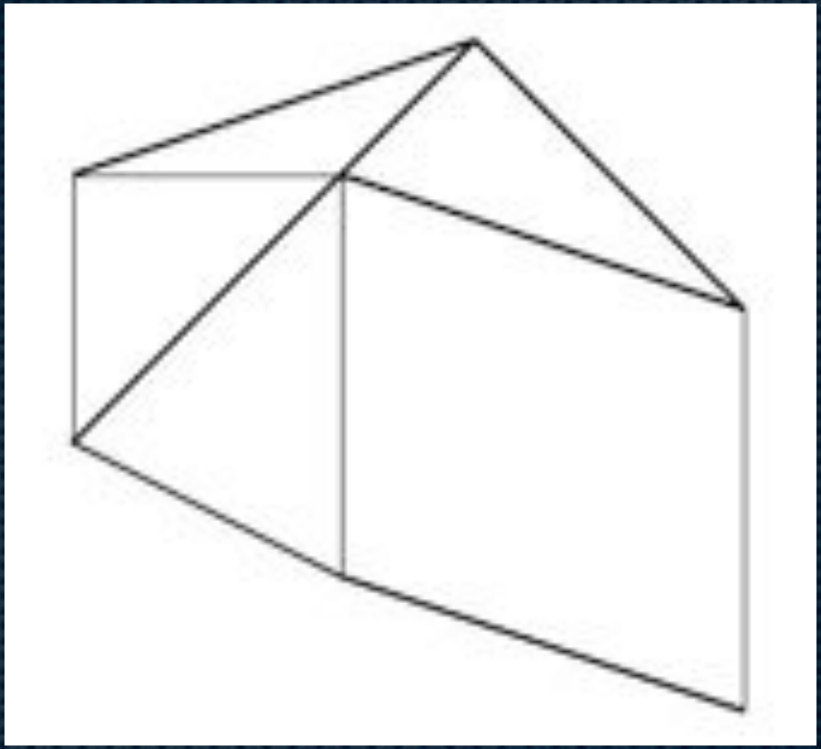
Hierarchical Hybrid Grids (HHG)

- Joint work with
- Frank Hülsemann (now EDF, Paris), Ben Bergen (now Los Alamos), T. Gradl (Erlangen), B. Gmeiner (Erlangen)

HHG Goal: Ultimate Parallel FE Performance!

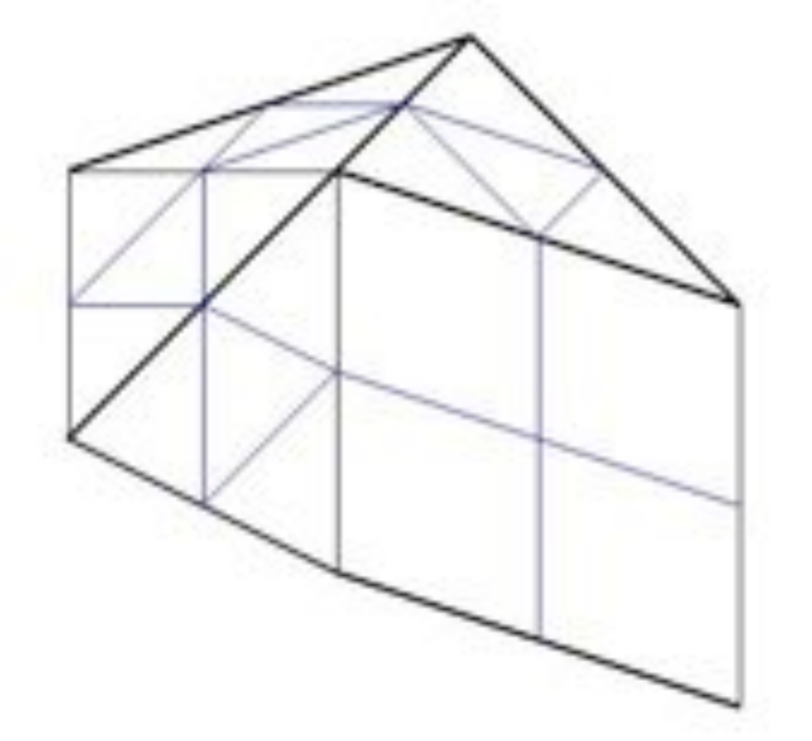
- unstructured adaptive refinement grids with
  - regular substructures for
  - efficiency
  - superconvergence effects
  - matrix-free implementation

HHG refinement example



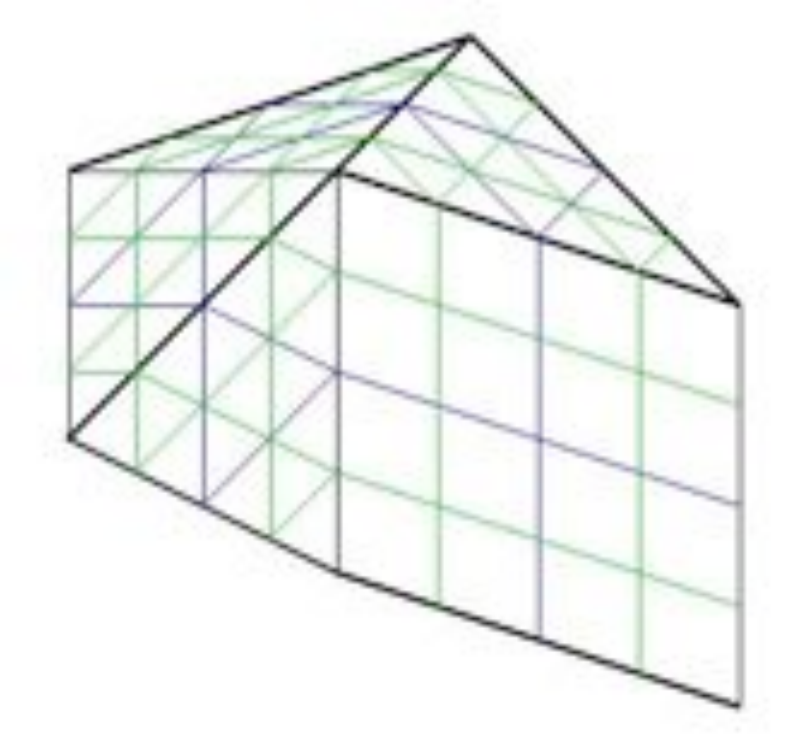
Input Grid

HHG Refinement example



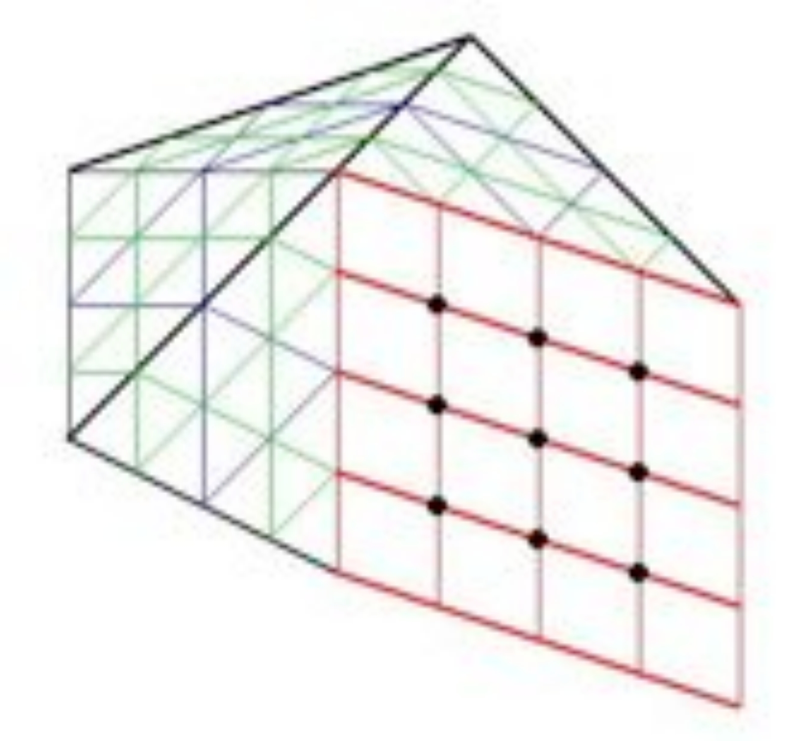
Refinement Level one

HHG Refinement example



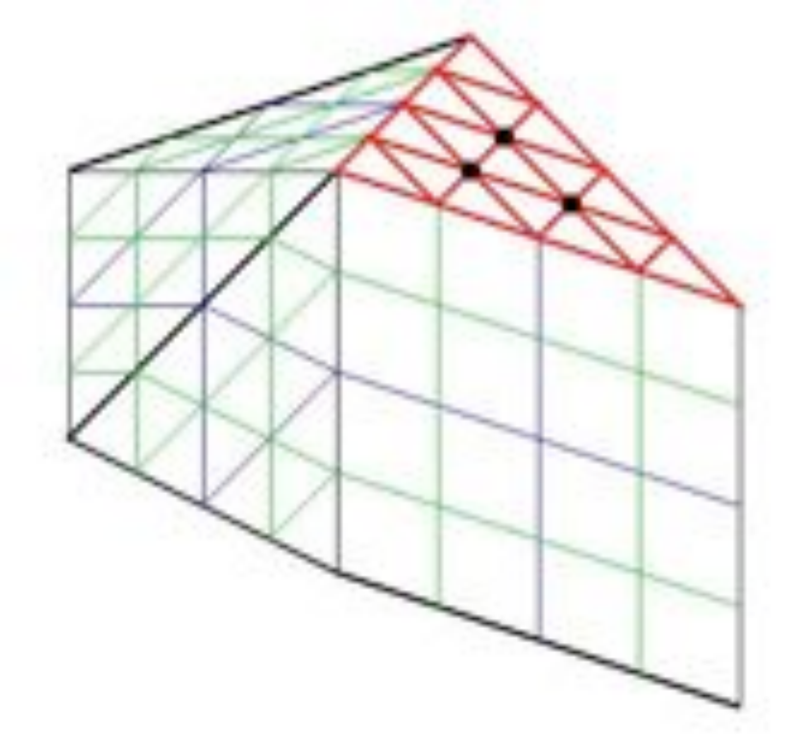
Refinement Level Two

HHG Refinement example



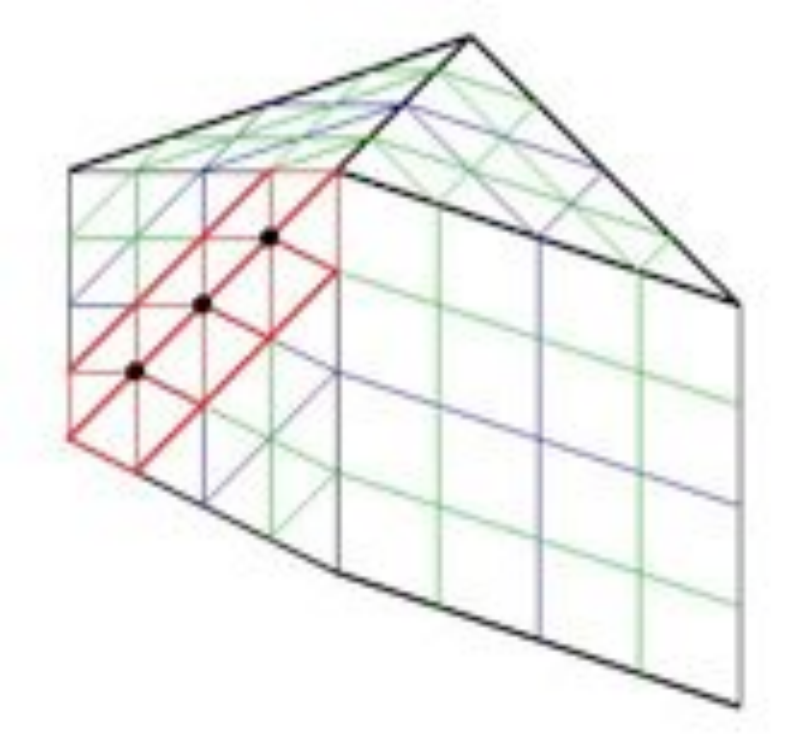
Structured Interior

HHG Refinement example



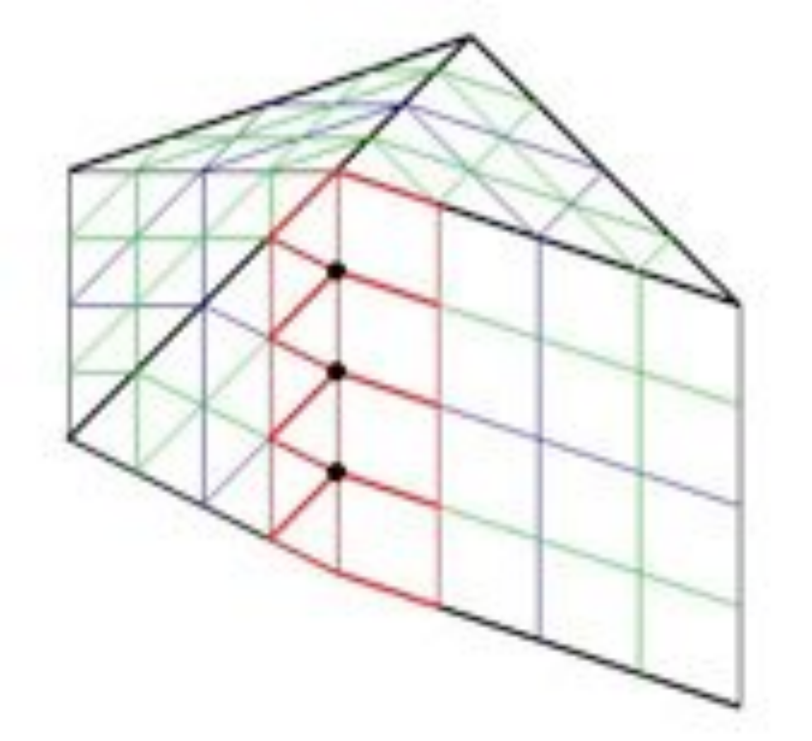
Structured Interior

HHG Refinement example



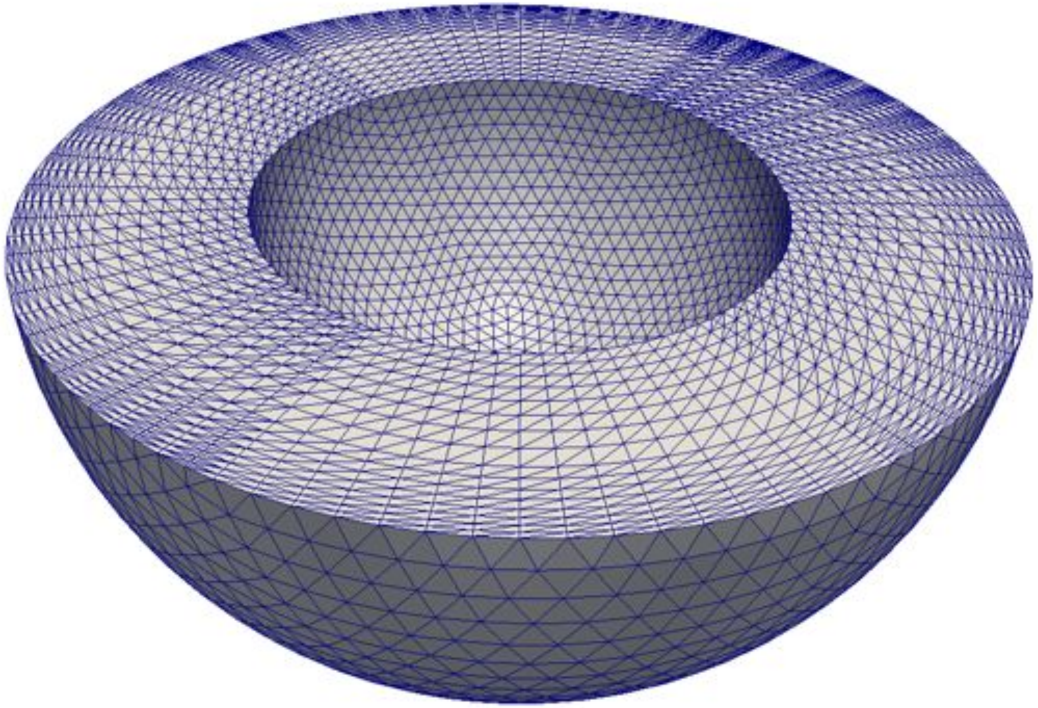
Edge Interior

HHG Refinement example



Edge Interior

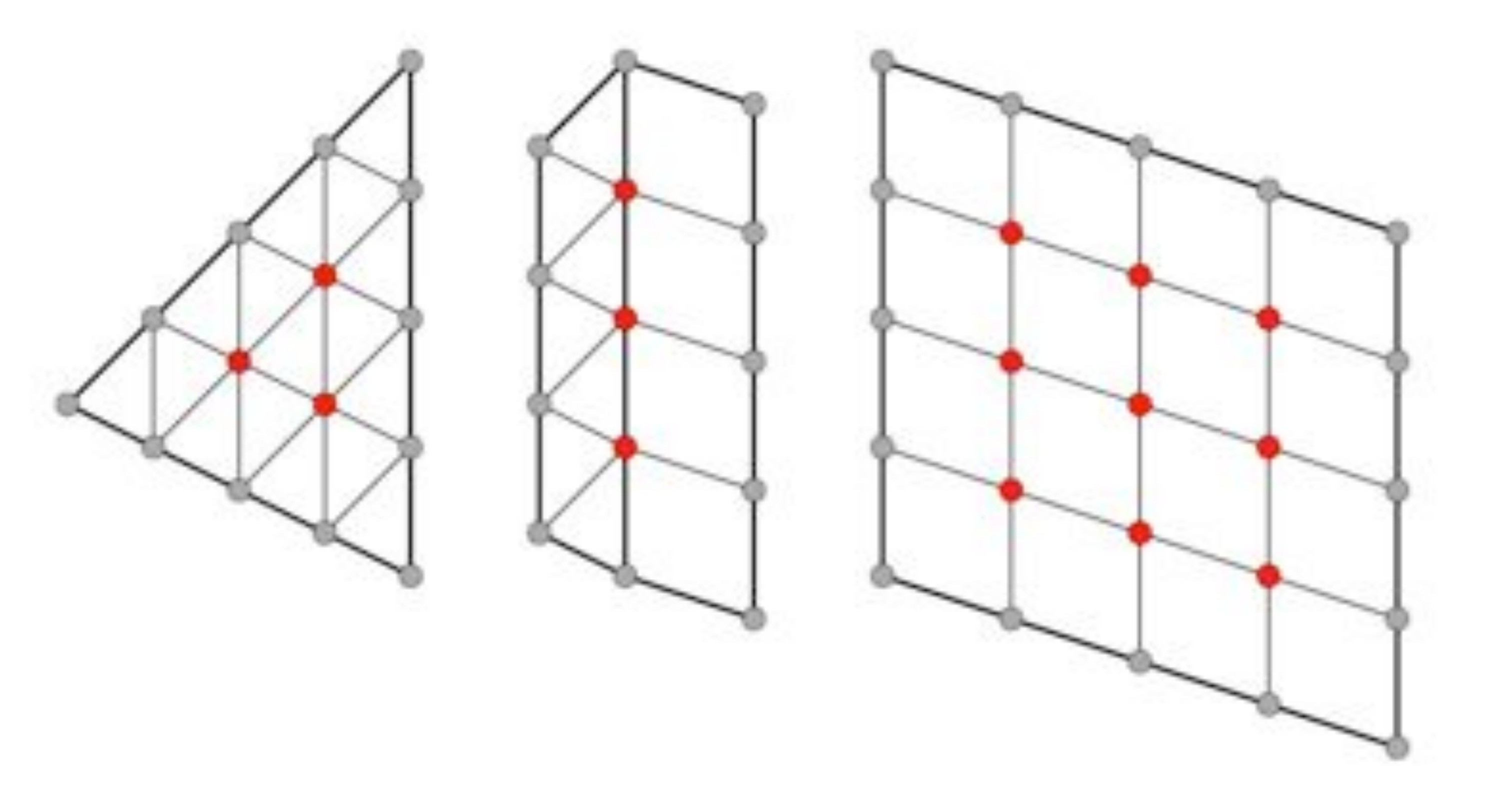
Typical HHG Input mesh



- Each tetrahedral element ($\approx 132k$) was assigned to one Jugene compute core.

HHG for Parallelization

- Use regular HHG patches for partitioning the domain



HHG Parallel Update Algorithm

```
for each vertex do
    apply operation to vertex
end for
update vertex primary dependencies
for each edge do
    copy from vertex interior
    apply operation to edge
    copy to vertex halo
end for
update edge primary dependencies
for each element do
    copy from edge/vertex interiors
    apply operation to element
    copy to edge/vertex halos
end for
update secondary dependencies
```

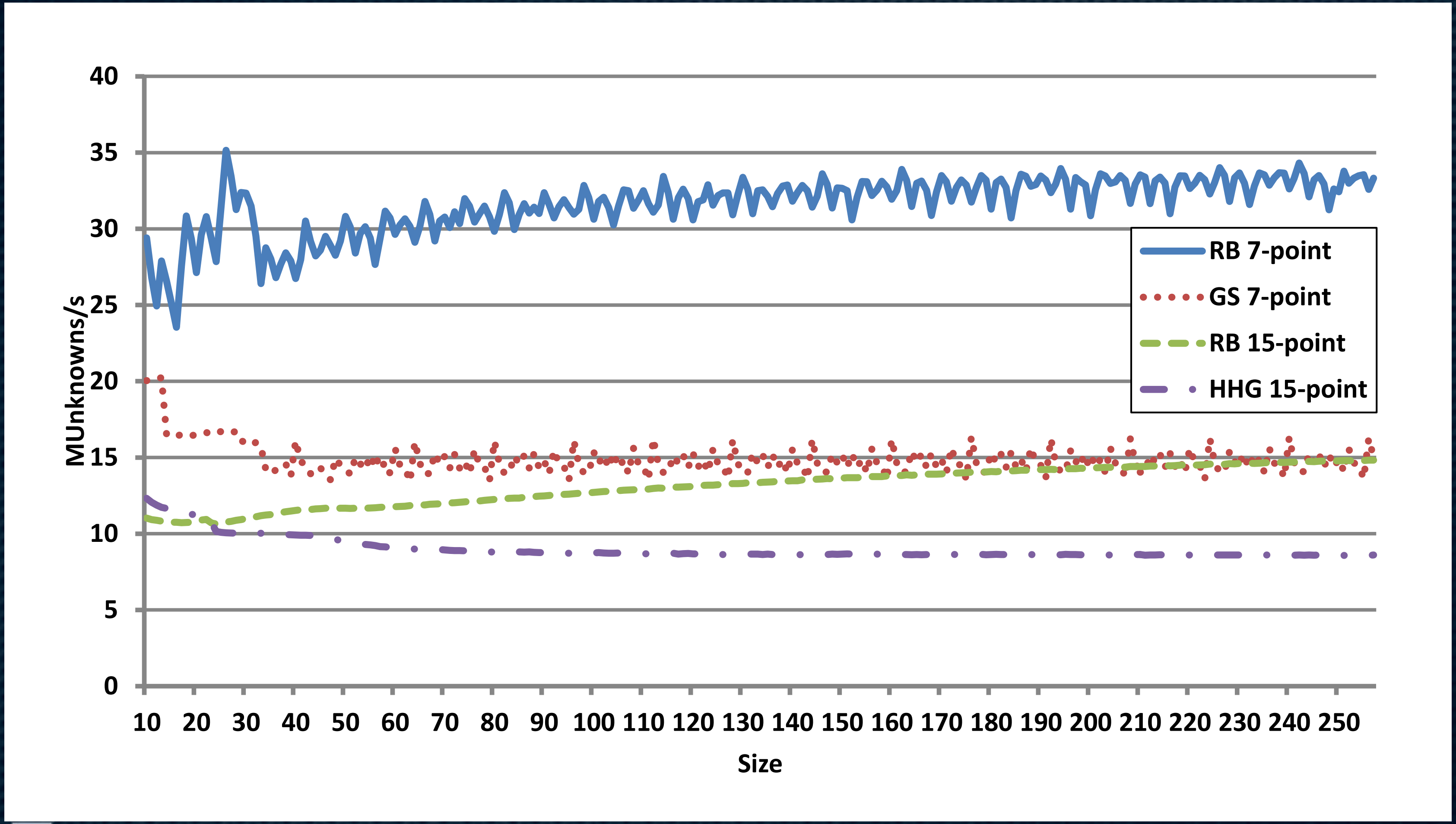
HHG Pros and Cons

- Pro:
  - performance
    - within node: SIMD, superscalar execution, etc.
  - better accuracy through local superconvergence effects
  - well suited for parallelization
  - tau-extrapolation for higher order
  - local line/plane smoothers for better efficiency
- Con:
  - only restricted adaptivity possible
  - only limited ability to handle complex shapes
  - how to solve the coarse grid problem
  - high implementation effort
  - less flexible

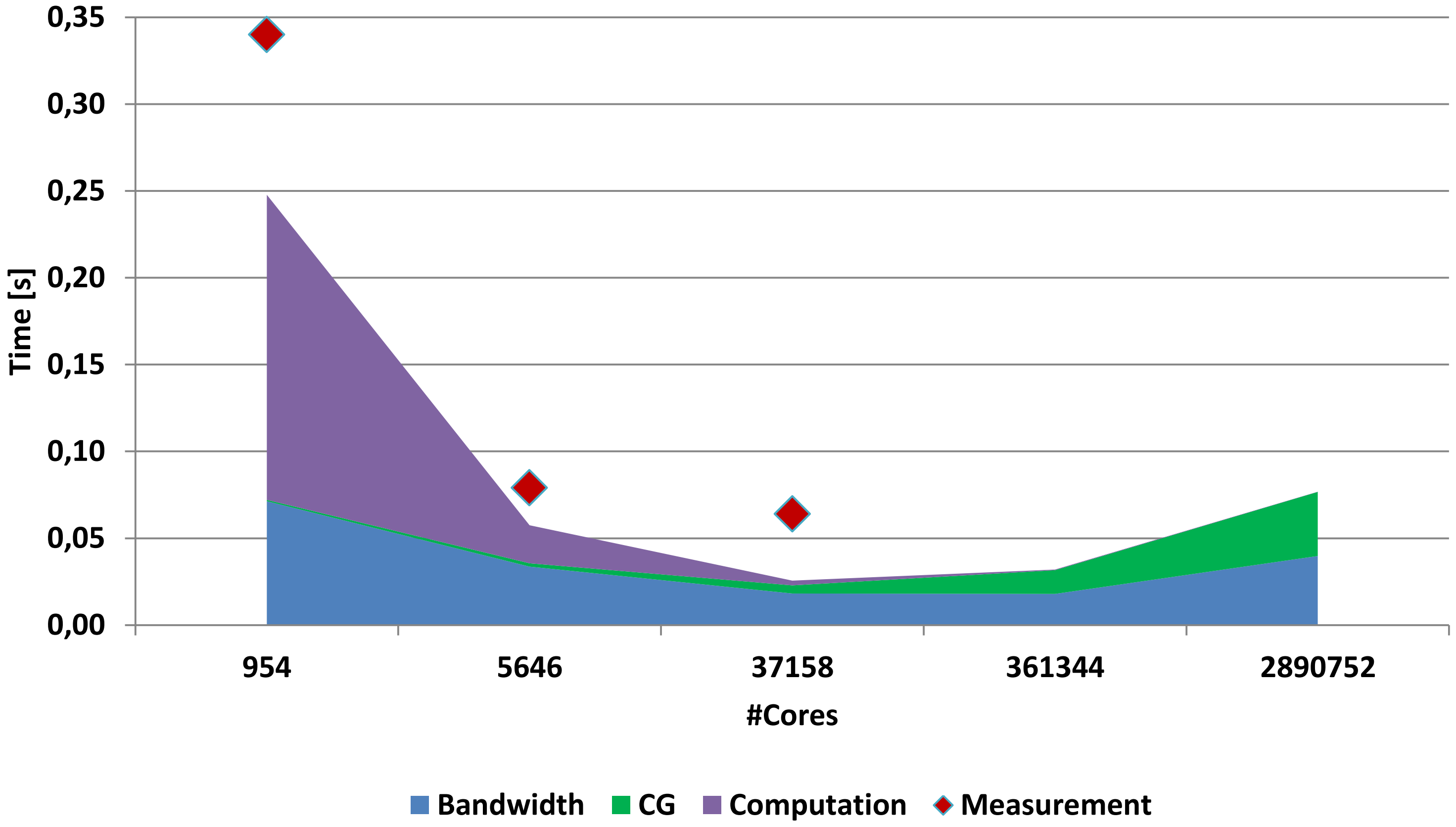
Performance Engineering

design - model - measure - revise - tune

node performance first!



System Performance Model and Measurement



#Cores	Coarse Grid	Unkn (x 10^6)	Crse Grd Its	Tme to soln
? ? ?	1536	535	15	5,64
256	3072	1070	20	5,66
512	6144	2142	25	5,69
1024	12288	4286	30	5,71
2028	24576	8577	45	5,75
4096	49152	17158	60	5,92
8192	98304	34326	70	5,86
16384	196608	68669	90	5,91
32768	393216	137355	105	6,17
65536	786432	274743	115	6,41
131072	1572864	549554	145	6,42
262144	3145728	1099276	280	6,82
294912	294912	824365	110	3,80

Parallel scalability of scalar elliptic problem in 3D discretized by tetrahedral finite elements.

Times to solution on Jugene.

Largest problem solved:
1.099 x 10^{12} DOFS (6 trillion tetrahedra) on 262000 processors in roughly 100 secs.

B. Bergen, F. Hülsemann, U. Rüde, G. Wellein: ISC Award 2006, also: „*Is 1.7× 10^{10} unknowns the largest finite element system that can be solved today?*“, SuperComputing, Nov‘ 2005. Runs done on SGI Altix at LRZ.

Conclusions

Theory versus Practice

- Assumptions:
  - Multigrid requires 27.5 Ops/unknown to solve an elliptic PDE (Griebel ´89 for Poisson)
  - A modern laptop CPU delivers >10 GFlops peak
- Consequence:
  - We should solve one million unknowns in 0.00275 seconds
  - ~ 3 ns per unknown
- Revised Assumptions:
  - Multigrid takes 500 Ops/unknown to solve your favorite PDE
  - you can get 5% of 10 Gflops performance
- Consequence: On your laptop you should
  - solve one million unknowns in 1.0 second
  - ~ 1 microsecond per unknown
- Consider Banded Gaussian Elimination on the Play Station (Cell Processor), single Prec. 250 GFlops, for 1000 x 1000 grid unknowns
  - ~2 Tera-Operations for factorization - will need about 10 seconds to factor the system
  - requires 8 GB Mem.
  - Forward-backward substitution should run in about 0.01 second, except for bandwidth limitations

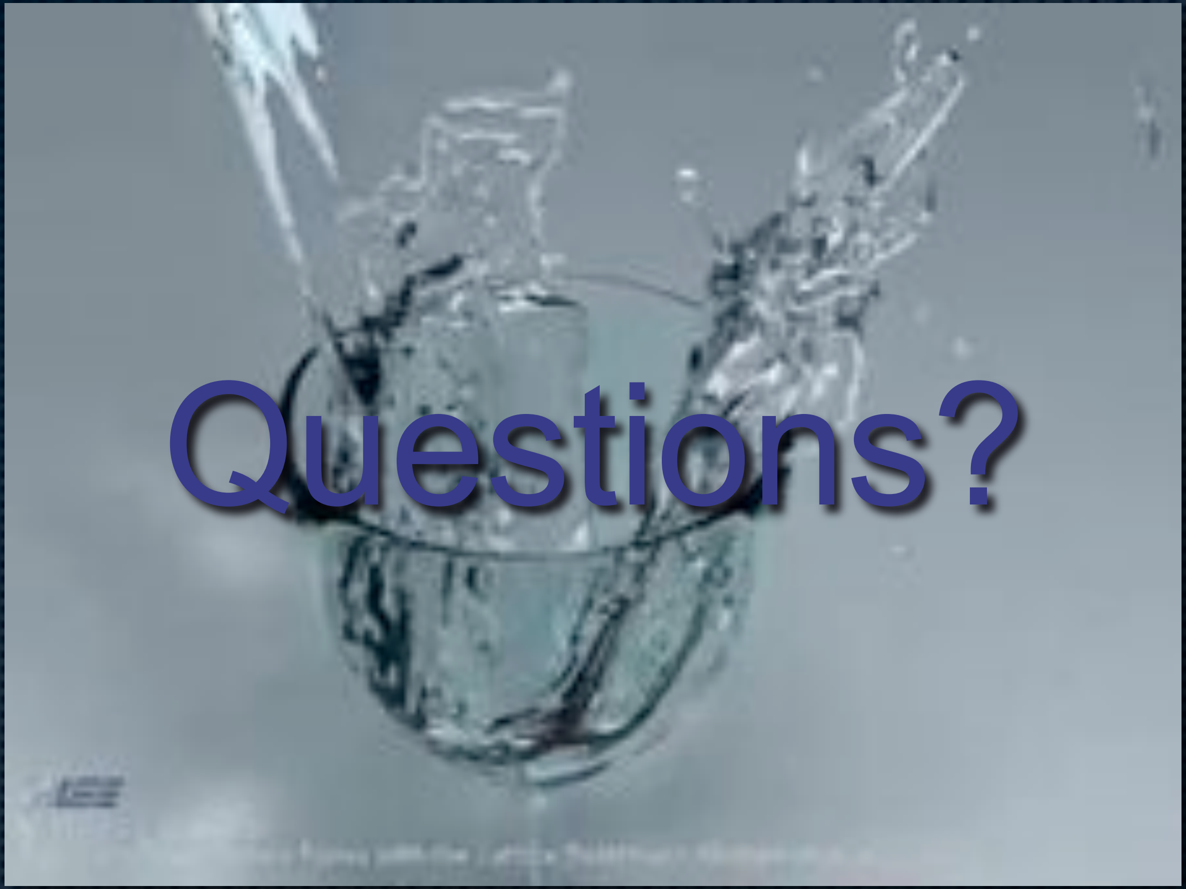
***Pessimizing** the Performance*

Pessimize →

- Bring loops in wrong order, ignore caches, randomize memory access, use many small MPI messages
  - $10^{12} \rightarrow 10^{11}$ unknowns
- Do not use a matrix-free implementation (keep in mind that a single multiplication with the mass and stiffness can easily cost 50 mem accesses per unknown):
  - $10^{11} \rightarrow 10^{10}$ unknowns
- Gain additional flexibility by using unoptimized unstructured grids (indirect mem access costs!)
  - $10^{10} \rightarrow 10^{9}$ unknowns
- Increase algorithmic overhead, e.g. permanently checking convergence, use the most expensive error estimator, etc. etc.
  - $10^{9} \rightarrow 10^{8}$ unknowns (... *still a large system* ...)

Optimize →

Thank you for your attention!



Questions?

Slides, reports, thesis, animations available for download at:

www10.informatik.uni-erlangen.de