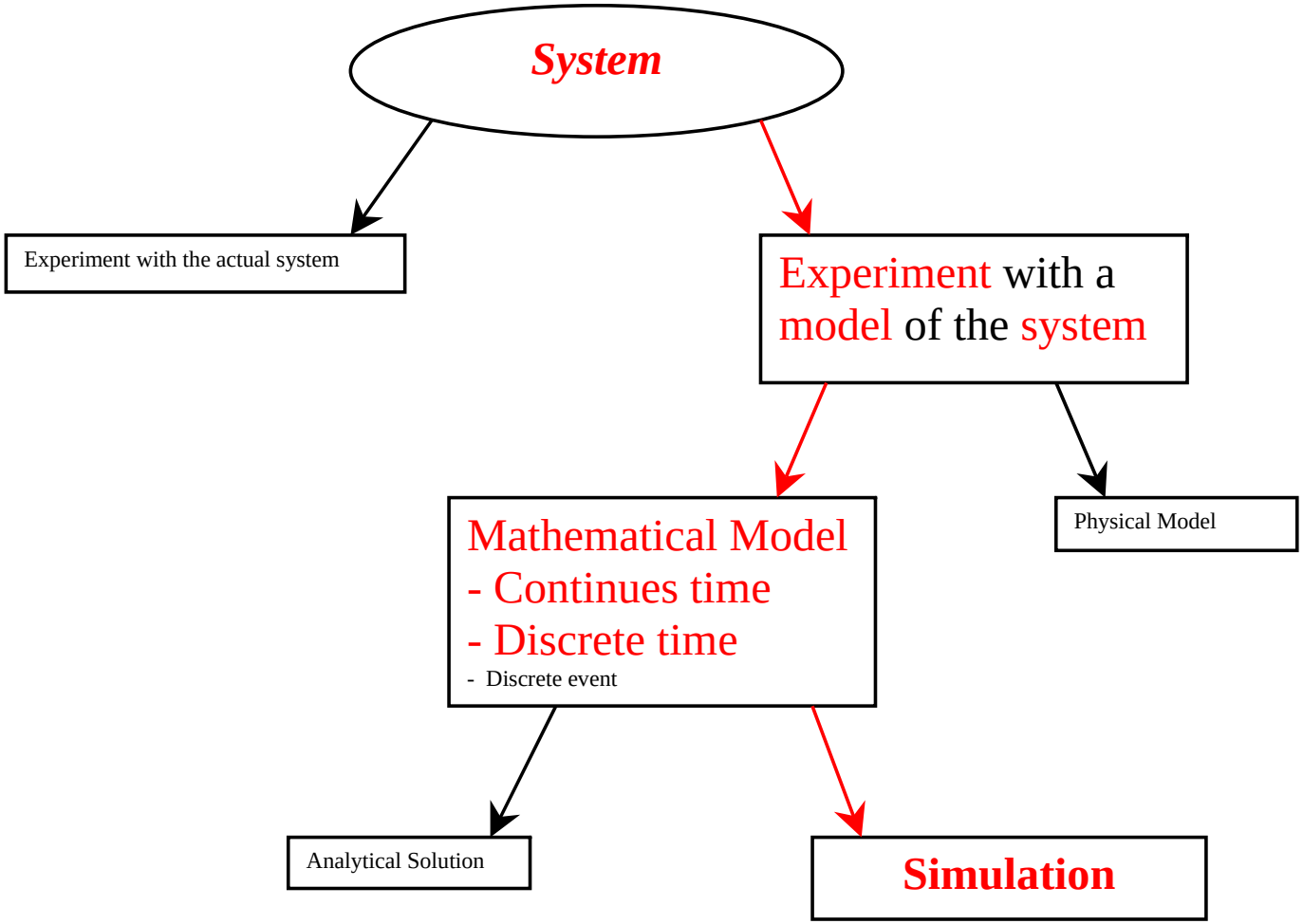
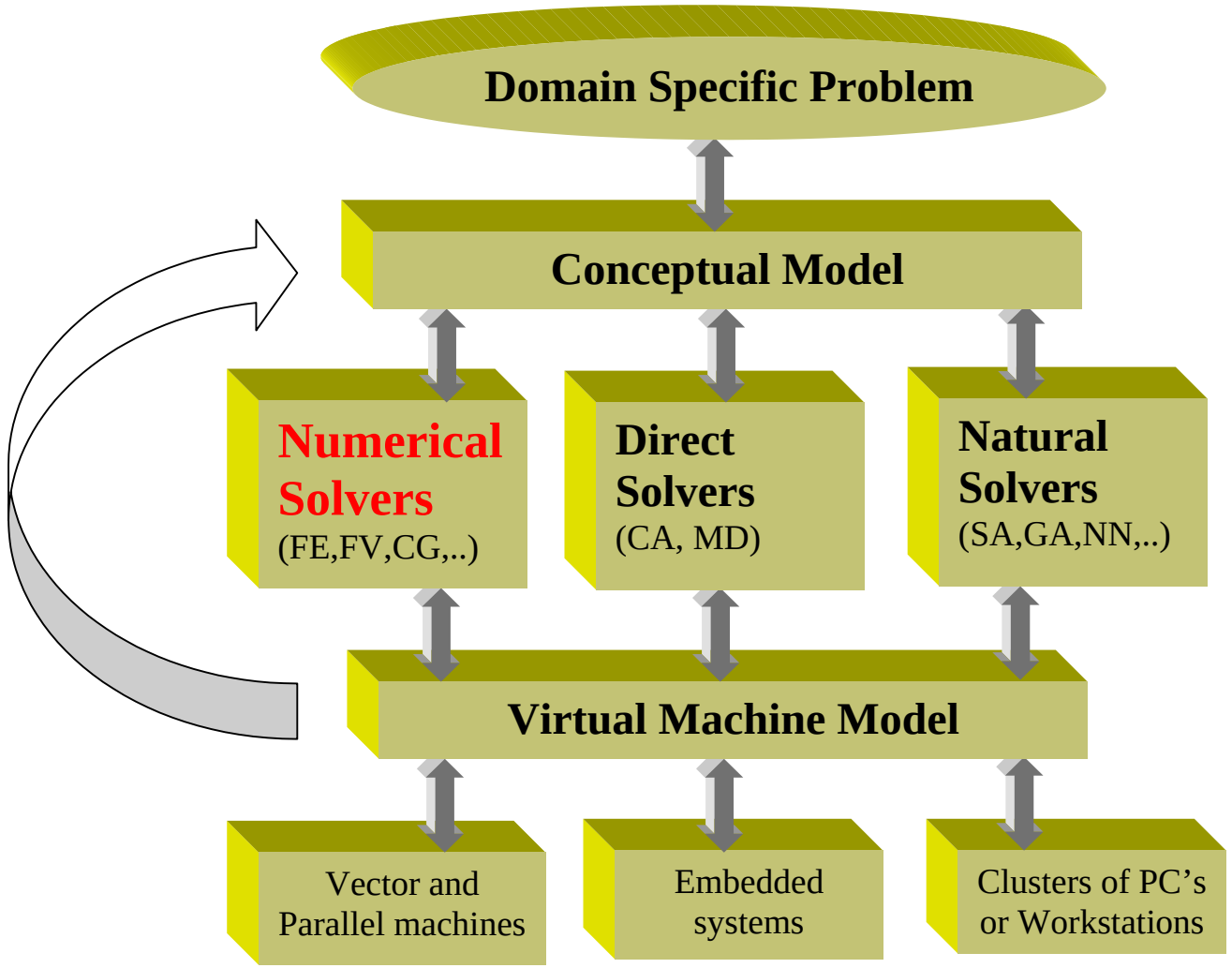
Solving the Diffusion equation in parallel

- The Diffusion Equation
- Parallel simulation of time dependent diffusion
- Parallel simulation of time independent diffusion
  - –direct methods
  - –iterative methods
- Diffusion Limited Aggregation

Ways to study a system



Modeling and Simulation Cycle



Diffusion Equation

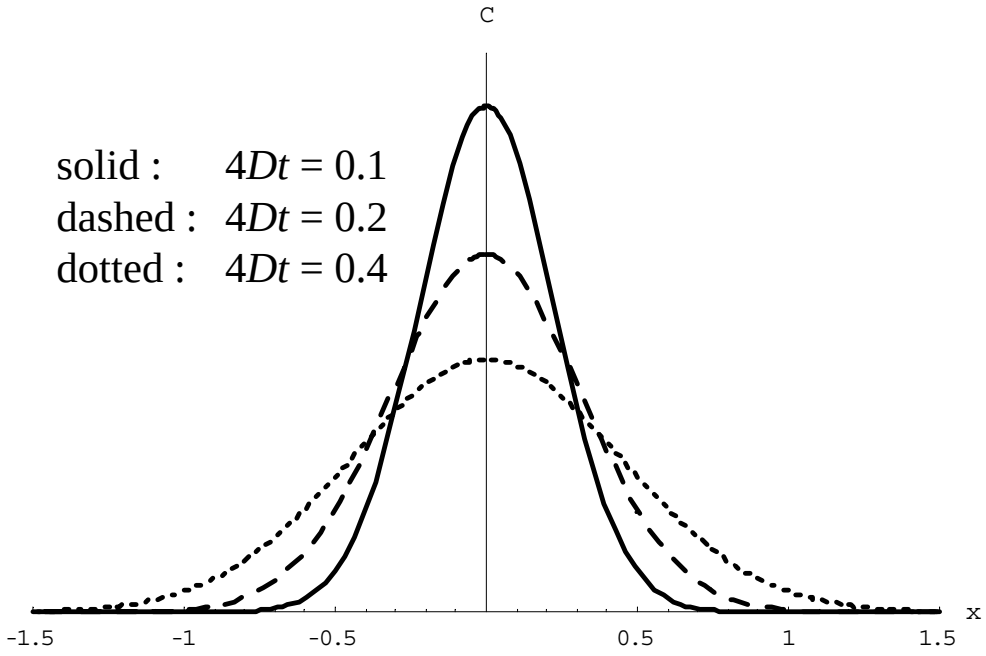
- Our case will be diffusion
  - derivation on black board

$$\frac{\partial c}{\partial t} = D\nabla^2 c$$

- $c(x, y, z, t)$ - concentration
- D - diffusion coefficient

Exact solutions 1

- One dimensional domain, $-\infty < x < \infty$
- delta pulse

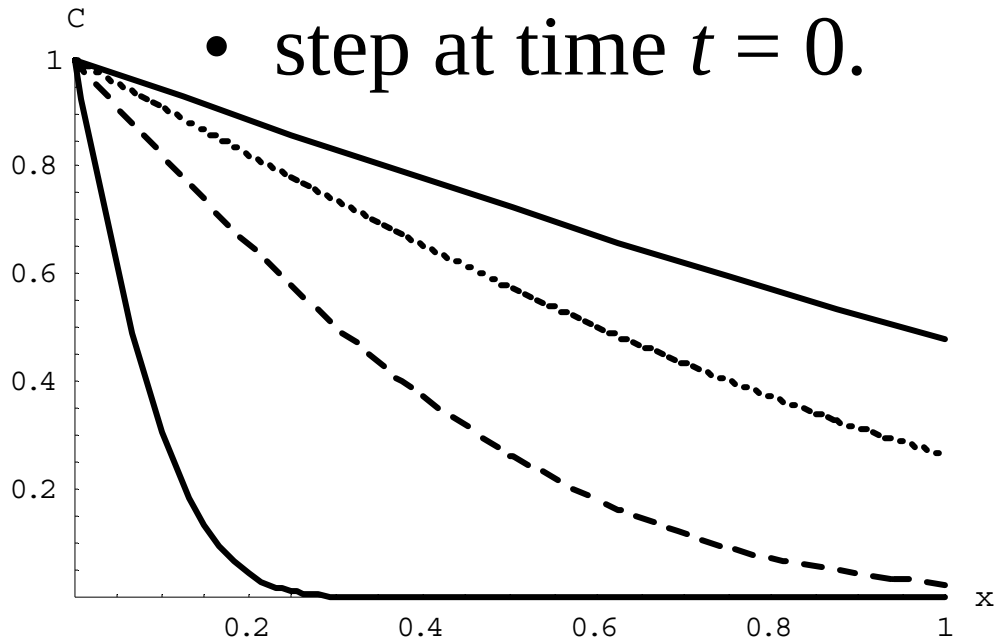


$$c(x,t=0)=\delta(x)$$

$$c(x;t)=\frac{1}{\sqrt{4\pi Dt}}\exp\left(\frac{-x^2}{4Dt}\right)$$

Exact solutions 2

- One dimensional domain, $0 \leq x < \infty$
- step at time $t = 0$.



solid : $Dt = 0.005$
dashed : $Dt = 0.1$
dotted : $Dt = 0.4$
solid : $Dt = 1.0$

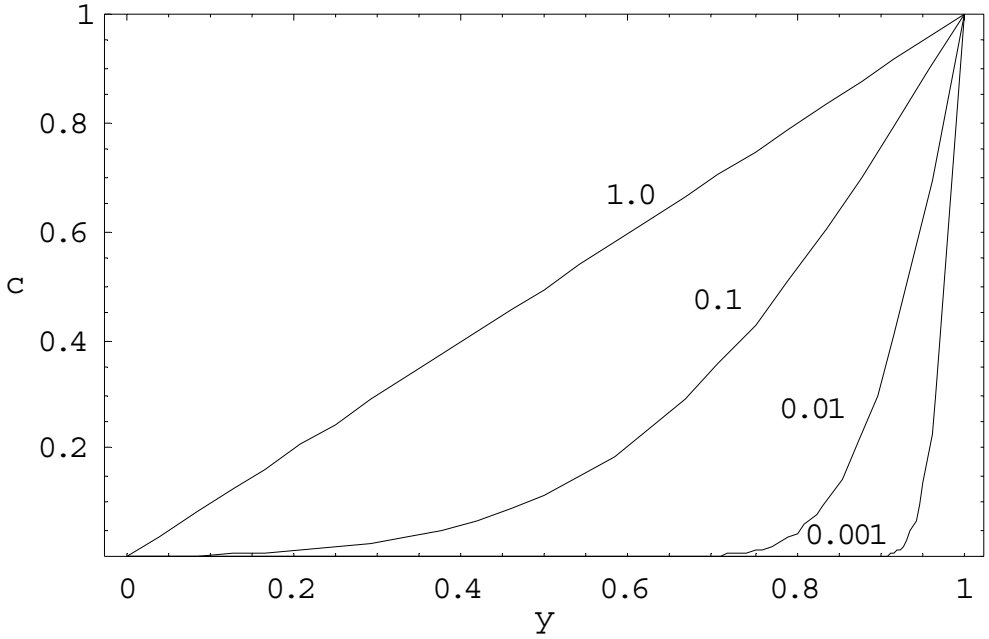
$$c(x=0,t) = H(t)$$

$$c(x;t) = \operatorname{erfc}\left(\frac{x}{2\sqrt{Dt}}\right)$$

where $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$

Exact solutions 3

- One dimensional bounded domain, $0 \le x \le 1$



step at time $t = 0$.

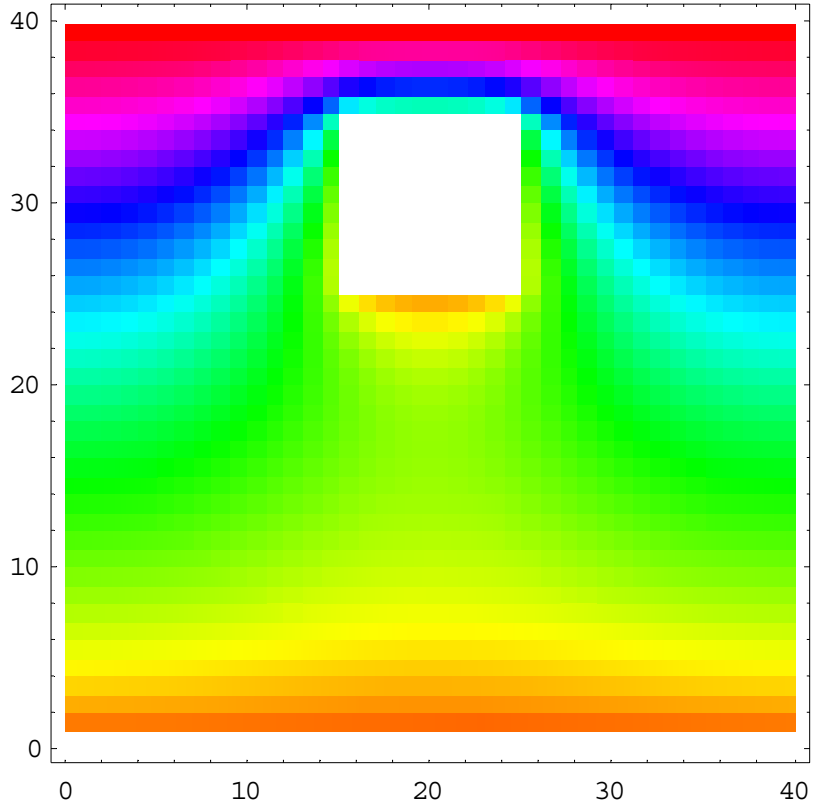
$$c(x, t = 0) = 0$$

$$c(x = 1, t) = H(t)$$

$$c(x;t) = \sum_{i=0}^{\infty} \left[\operatorname{erfc}\left(\frac{1 - x + 2i}{2\sqrt{Dt}} \right) + \operatorname{erfc}\left(\frac{1 + x + 2i}{2\sqrt{Dt}} \right) \right]$$

No exact solution...

- two dimensional bounded domain, $0 \leq x, y \leq 1$, with object immersed in it.



$c(x, y = 0, t) = 0$
$c(x, y = 1, t) = H(t)$
$c(x, y, t = 0) = 0$

Final solution for $t \rightarrow \infty$

Numerical Solution

- With a discretization $x = l\delta x$, $y = m\delta y$, $z = n\delta z$ with (l, m, n) integers, and writing $c(x, y, t) \equiv c^n_{l,m}$

- finite differencing gives
  - with time step δt and spatial steps δx
  - derived on blackboard

$$c^{n+1}_{l,m} = c^n_{l,m} + \frac{D\delta t}{\delta x^2}\left[c^n_{l+1,m} + c^n_{l-1,m} + c^n_{l,m+1} + c^n_{l,m-1} - 4c^n_{l,m}\right]$$

  - explicit finite difference schema with a local 5 point stencil

Consequences

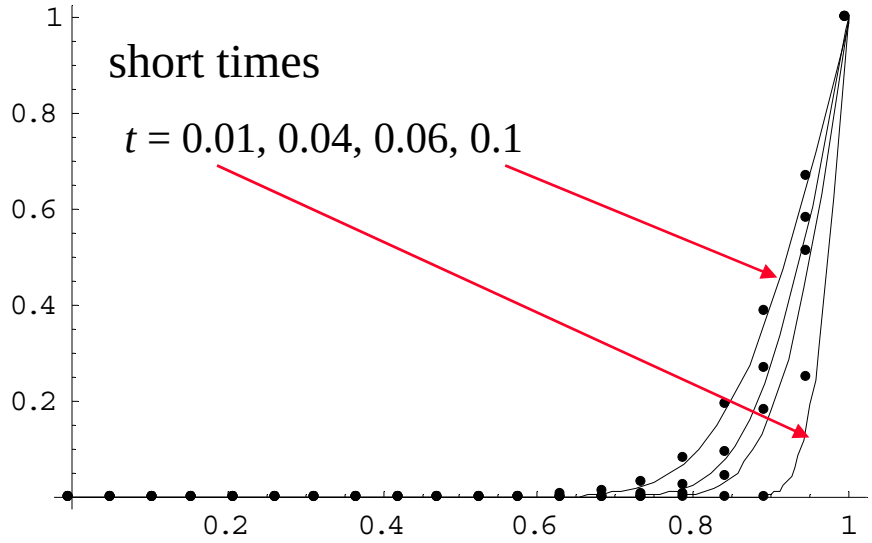
- The maximum allowed time step δt is limited by the "diffusion time" over the lattice $t_D \sim \delta x^2/D$.
- Assume we simulate in a domain with spatial scales $\lambda = L\delta x$. The diffusion time through the domain is $\tau \sim \lambda^2/D$.
- The number of time steps therefore is in the order of $\tau/t_D = \lambda^2/\delta x^2 = L^2$.
  - Bad scaling, huge amount of time steps…
  - Solution lies in implicit methods - not discussed here.

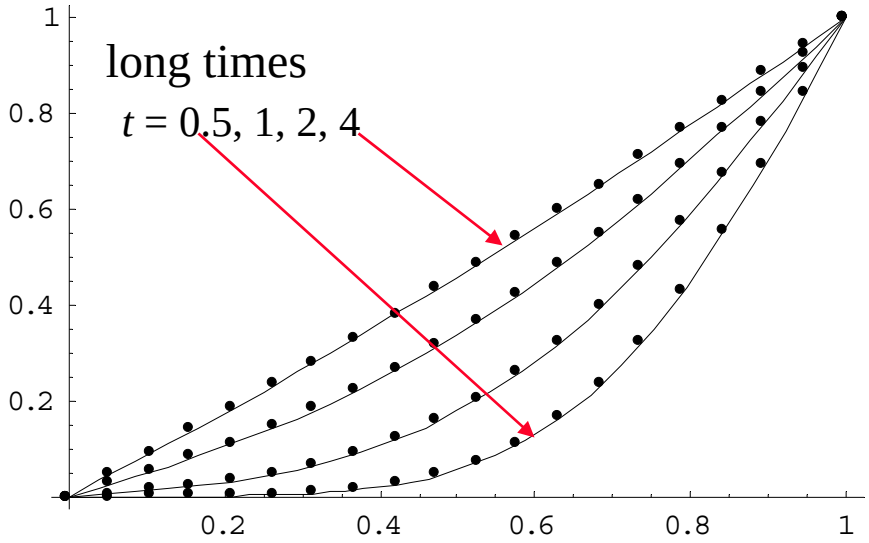
An example 1

- Two dimensional domain, $0 \le x, y \le 1$
  - periodic boundary conditions in x-direction
    - $c(x,y,t) = c(x+1,y,t)$
  - other boundary conditions
    - $c(x,y=0,t) = 0$
    - $c(x,y,t \le 0) = 0$ for $0 \le x \le 1$ and $0 \le y < 1$
    - $c(x,y=1,t) = \mathrm{H(t)}$ (i.e. a step from 0 to 1 at time $t = 0$)
  - Note that because of symmetry the solution will not depend on the x-coordinate, and that the exact solution number 3 applies in y-direction.

An example 2

- Compare the finite difference simulation with the exact solution.
  - δx 1/20; $D\delta x^2/\delta t = 1/4$; $D = 1$; $\delta t = 0.01$.

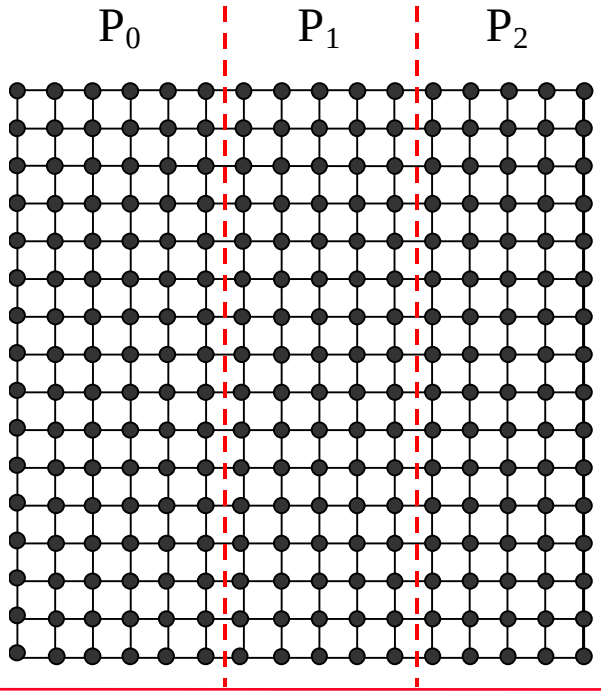


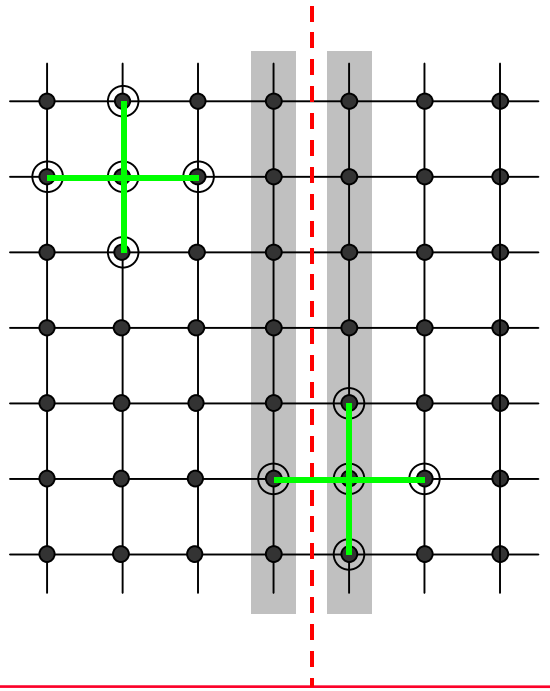


lines : exact solution
dots : numerical results

Parallelization

- Domain decomposition of spatial domain
  - here, as an example, strip wise decomposition
- one strip of grid points on the edge of the sub-domains (gray zone) must be communicated to neighbors, due to 5-point stencil (green lines).





Pseudo code

```
main () /* pseudo code for parallel FD simulation */
{
    decompose lattice;
    initialize lattice sites;
    set boundary conditions;
    for N time steps {
        exchange boundary strips with neighboring processors;
        for all grid points in this processor {
            update according to FD scheme
        }
        print results to file; /* e.g. after every n time steps */
    }
}
```

Time complexity for single iteration

N^2 grid points (assume square $N \diamondsuit N$ domain)

assume strip wise decomposition

τ_{calc} : time to update 1 grid point

τ_{setup} : latency of exchange communication

$\tau_{exchange}$: time to exchange 1 date point with neighbor

$$T_1 = N^2 \tau_{calc}$$

$$T_p = \frac{N^2}{p} \tau_{calc} + T_{comm}$$

$$= \frac{N^2}{p} \tau_{calc} + 2(\tau_{setup} + N \tau_{exchange})$$

$$\varepsilon_p = \frac{S_p}{p} = \frac{T_1}{pT_p} = \frac{1}{1 + \frac{2p}{N^2} \frac{\tau_{setup}}{\tau_{calc}} + \frac{2p}{N} \frac{\tau_{exchange}}{\tau_{calc}}}$$

Time independent diffusion

- No transient behavior, all time derivatives are zero: $\nabla^2 c = 0 \quad \rightarrow$ the Laplace equation.
- Finite differencing:

$$c_{l,m} = \frac{1}{4}\left[c_{l+1,m} + c_{l-1,m} + c_{l,m+1} + c_{l,m-1}\right], \forall (l,m)$$

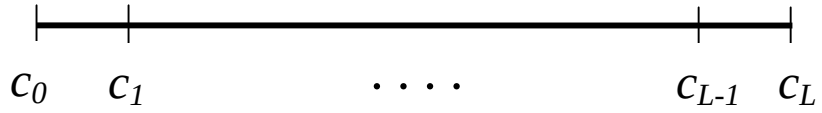
- The equations can be reformulated as

$$\mathbf{Ax} = \mathbf{b}$$

A: $N \times N$ matrix, **x**, **b** are vectors of length N.

- i.e. a set of coupled linear equations.

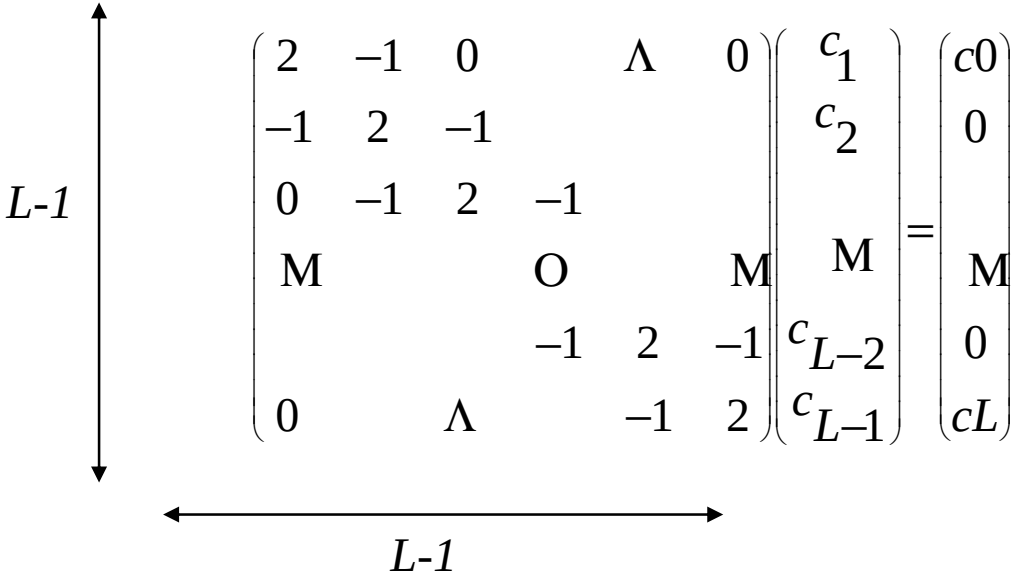
Example in 1 dimension



$$\frac{c_{l-1} - 2c_l + c_{l+1}}{\delta x^2} = 0, \quad 0 \le l \le L$$

$$\text{boundary}: \; c_0 = C0$$

$$c_L = CL$$

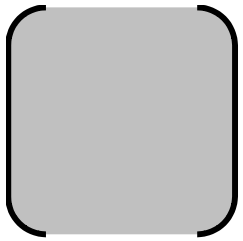
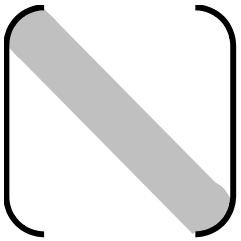


$$\begin{pmatrix} 2 & -1 & 0 & & \Lambda & 0 \\ -1 & 2 & -1 & & & \\ 0 & -1 & 2 & -1 & & \\ M & & & O & & M \\ & & & -1 & 2 & -1 \\ 0 & & \Lambda & & -1 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \\ M \\ c_{L-2} \\ c_{L-1} \end{pmatrix} = \begin{pmatrix} c0 \\ 0 \\ \\ M \\ 0 \\ cL \end{pmatrix}$$

- Banded matrix, $N = L - 1$
- more complicated structure in higher dimensions, but still banded (try yourself !)

Direct vs iterative methods

- **Ax = b** : solution via direct or iterative methods
  - direct: solve the equation directly within numerical error
  - iterative : approximate the solution using some iterative procedure

	dense matrix	banded matrix
		
direct	$O(N^3)$	$O(N^2)$
iterative	$O(N^2)$	$O(N)$

Direct Methods

- Gaussian elimination
- LU factorisation
- BLAS, (parallel) software libraries

Gaussian Elimination

Approach : reduce the system to an equivalent upper triangular system

Easily solved through backward substitution

$$\begin{pmatrix} a_{11} & \Lambda & a_{1N} \\ \text{M} & \text{O} & \text{M} \\ a_{N1} & \Lambda & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \text{M} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \text{M} \\ b_N \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} a'_{11} & a'_{12} & \Lambda & a'_{1N} \\ 0 & a'_{22} & & \text{M} \\ \text{M} & & \text{O} & a'_{N-1,N} \\ 0 & \Lambda & 0 & a'_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \text{M} \\ x_N \end{pmatrix} = \begin{pmatrix} b'_1 \\ \text{M} \\ b'_N \end{pmatrix}$$

$$x_N = \frac{b'_N}{a'_{NN}}$$

$$x_{N-1} = \frac{b'_{N-1} - a'_{N-1,N} x_N}{a'_{N-1,N-1}}, \quad \text{etc.}$$

or

$$x_i = \frac{b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j}{a'_{ii}}$$

Sweeping the matrix 1

Consider the composed matrix $(\mathbf{A}, \mathbf{b}) = \begin{pmatrix} a_{11} & \Lambda & a_{1N} & b_1 \\ \text{M} & \text{O} & \text{M} & \text{M} \\ a_{N1} & \Lambda & a_{NN} & b_N \end{pmatrix}$

a) Determine an element $a_{r1} \neq 0$, proceed with (b). If such element does not exist, **A** is singular, stop.

b) Interchange row r and row 1 of (**A**, **b**), result is $(\overline{\mathbf{A}}, \overline{\mathbf{b}})$

c) For $i = 2, 3, \ldots, N$, subtract multiple $l_{i1} = \overline{a}_{i1} / \overline{a}_{11}$ of row 1 from row i.

Result is $(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \Lambda & a_{1N}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(1)} & \text{M} & \text{M} & \text{M} \\ \text{M} & & \text{O} & & \\ 0 & a_{N1}^{(1)} & & a_{NN}^{(1)} & b_N^{(1)} \end{pmatrix} = \left[\begin{array}{c|c|c} a_{11}^{(1)} & (\mathbf{a}^{(1)})^T & b_1^{(1)} \\ \hline 0 & \tilde{\mathbf{A}} & \tilde{\mathbf{b}} \end{array}\right]$

Sweeping the matrix 2

$$(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \left[\begin{array}{c|c|c} a_{11}^{(1)} & (\mathbf{a}^{(1)})^T & b_1^{(1)} \\ \hline 0 & \tilde{\mathbf{A}} & \tilde{\mathbf{b}} \end{array}\right]$$

Apply the same procedure to $(\tilde{\mathbf{A}}, \tilde{\mathbf{b}})$, etc.

$$(\mathbf{A}, \mathbf{b}) := (\mathbf{A}^{(0)}, \mathbf{b}^{(0)}) \rightarrow (\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) \rightarrow \Lambda \rightarrow (\mathbf{A}^{(N-1)}, \mathbf{b}^{(N-1)}) =: (\mathbf{U}, \mathbf{c})$$

The matrix (**U**, **c**) has the desired upper triangular form, and the solution **x** can now be found using backward substitution.

Pivoting

- The element a_{r1} in step (a) is called the pivot element.
- Not a good idea to choose any element, or e.g. always $a_{r1} = a_{11}$.
  - Leads to loss of accuracy, or even instability of the Gaussian elimination.
- Always apply partial pivoting.

$$|a_{r1}| = \max_i |a_{i1}|$$

  - i.e. pick the maximum element (in absolute measure) as the pivot.

More on sweeping the matrix 1

Note that step (b), i.e. the interchanging of rows, can be formulated as

$$(\overline{\mathbf{A}}, \overline{\mathbf{b}}) = \mathbf{P}_1(\mathbf{A}, \mathbf{b})$$

where P_1 is a permutation matrix,

e.g. for $N = 4$, exchange row 1 and 2: $\mathbf{P}_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Note that $\mathbf{P}_1\mathbf{P}_1 = \mathbf{I} \Rightarrow \mathbf{P}_1^{-1} = \mathbf{P}_1$

More on sweeping the matrix 2

Note that step (c), i.e. the actual sweeping, can be formulated as $(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \mathbf{G}_1(\overline{\mathbf{A}}, \overline{\mathbf{b}})$

where $\mathbf{G}_1 = \begin{pmatrix} 1 & 0 & \Lambda & 0 \\ -l_{21} & 1 & & \\ \mathrm{M} & & \mathrm{O} & \\ -l_{N1} & & 0 & 1 \end{pmatrix}$

Note that

$$\mathbf{G}_1^{-1} = \begin{pmatrix} 1 & 0 & \Lambda & 0 \\ l_{21} & 1 & & \\ \mathrm{M} & & \mathrm{O} & \\ l_{N1} & & 0 & 1 \end{pmatrix}$$

More on sweeping the matrix 3

So, we find $(\mathbf{A}^{(1)},\mathbf{b}^{(1)}) = \mathbf{G}_1\mathbf{P}_1(\mathbf{A},\mathbf{b})$

and in general $(\mathbf{A}^{(j)},\mathbf{b}^{(j)}) = \mathbf{G}_j\mathbf{P}_j(\mathbf{A}^{(j-1)},\mathbf{b}^{(j-1)})$

with $\mathbf{G}_j = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{j+1,j} & \ddots & \\ & & \vdots & & \\ 0 & & -l_{N,j} & & 1 \end{pmatrix}$ and $\mathbf{P}_j$ a permutation matrix

final solution $(\mathbf{U},\mathbf{c}) = \mathbf{G}_{N-1}\mathbf{P}_{N-1}\mathbf{G}_{N-2}\mathbf{P}_{N-2}\Lambda\ \mathbf{G}_1\mathbf{P}_1(\mathbf{A},\mathbf{b})$

LU factorization 1

Now, suppose no row exchanges are needed, i.e. $\mathbf{P}_j = \mathbf{I}$.

In that case we find $\mathbf{U} = \mathbf{G}_{N-1}\mathbf{G}_{N-2}\Lambda\ \mathbf{G}_1\mathbf{A}$

$$\Rightarrow \mathbf{A} = \mathbf{L}\mathbf{U}, \text{ where } \mathbf{L} = \mathbf{G}_1^{-1}\mathbf{G}_2^{-1}\Lambda\ \mathbf{G}_{N-1}^{-1}$$

It is easily verified that $\mathbf{L} = \begin{pmatrix} 1 & 0 & \Lambda & 0 \\ l_{21} & \mathrm{O} & & \mathrm{M} \\ \mathrm{M} & & 1 & 0 \\ l_{N1} & \Lambda & l_{N,N-1} & 1 \end{pmatrix}$

So, $\mathbf{L}$ is a lower triangular matrix, and the result of the sweeping process $\mathbf{U}$ is an upper triangular matrix.

LU factorization 2

This LU factorization is important, because with it we can now easily solve for many arguments **b**.

original system : $\mathbf{Ax} = \mathbf{b}$

after factorization : $\mathbf{LUx} = \mathbf{b}$

first solve : $\mathbf{Ly} = \mathbf{b}$ $\rightarrow$ forward substitution

then solve : $\mathbf{Ux} = \mathbf{y}$ $\rightarrow$ backward substitution

LU factorization 3

- In general $\mathbf{LU} = \mathbf{PA}$
- with $\mathbf{P} = \mathbf{P}_{N-1}\,\mathbf{P}_{N-2}\,\ldots\,\mathbf{P}_{1}$
- and $\mathbf{L}$ some permutation of elements from the $\mathbf{G}_j$ matrices.
- Many algorithms exist to find LU factorizations
  - See e.g. Stoer and Bulirsch, *Introduction to Numerical Analyis*.
- Finally note that also many optimized algorithms exist for special matrices, e.g for banded matrices.

Example

$$\begin{pmatrix} 3 & 1 & 6 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 23 \\ 12 \\ 6 \end{pmatrix}$$

Solution : $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 1/3 & 1 & 1 \end{pmatrix}$$

Work this out yourself !
and check all results….

$$\mathbf{U} = \begin{pmatrix} 3 & 1 & 6 \\ 0 & 2/3 & 1 \\ 0 & 0 & -2 \end{pmatrix}$$

BLAS

- Basic Linear Algebra Subprograms
  - BLAS 1 - O(N) operations on O(N) data items
    - typically a vector update, e.g. : $\mathbf{y} = \alpha\,\mathbf{x} + \mathbf{y}$
  - BLAS 2 - O(N^2) operations on O(N^2) data items
    - typically matrix vector products, e.g. : $\mathbf{y} = \alpha\mathbf{A}\,\mathbf{x} + \beta\,\mathbf{y}$
  - BLAS 3 - O(N^3) operations on O(N^2) data items
    - typically matrix matrix products, e.g. : $\mathbf{C} = \alpha\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}$
- Typically highly optimized BLAS libraries available.
  - More information via http://www.netlib.org

Solving equations with BLAS

- Gaussian elimination, or LU factorisation can be implemented using either BLAS-1, BLAS-2 or BLAS-3.
  - Typically performance increases in going from BLAS-1 to BLAS-3 due to better ratio between amount of calculations and needed memory references.
  - Modern linear algebra libraries use BLAS-3, where the algorithms are formulated as block algorithms.
    - Example of such a block LU factorization, see reader.

Linear Algebra Libraries 1

- EISPACK
  - Fortran routines for calculation of eigenvectors and eigenvalues for dense and banded matrices.
  - Based on BLAS-1.

- LINPACK
  - Fortran routines to solve linear equations for dense and banded matrices.
  - Column oriented BLAS-1 approach.

Linear Algebra Libraries 2

- LAPACK
  - Successor of EISPACK and LINPACK
  - Fortran routines for eigenvalues and solving linear equation, for dense and banded matrices.
  - Exploit BLAS-3 (improve speed).
  - Runs efficiently on vector machines and distributed memory parallel computers.

Linear Algebra Libraries 3

- ScaLAPACK
  - Extend LAPACK to run efficiently on distributed memory parallel computers
  - Fundamental building blocks are distributed memory versions of BLAS-2 and BLAS-3 routines and a set of Basic Linear Algebra Communication Subprograms (BLACS).

Parallel Linear Algebra routines

- Detailed discussion of parallel algorithms for e.g. LU factorisation is beyond the scope of this lecture.
  - Main features are
    - block oriented BLAS-3 approach
    - in combination with special (scattered) decompositions of the matrices.
  - For more information we refer to e.g.
    - G. Fox et al., *Parallel Computing Works*, chapter 9.5
      - http://www.npac.syr.edu/copywrite/pcw/
    - J.J. Dongarra and D.W. Walker, “Constructing Numerical Software Libraries for High Performance Computer Environments”, in A. Zomaya (editor), *Parallel & Distributed Computing Handbook*, chapter 32.

Iterative methods 1

- General idea:

  consider an iteration of the form

  $$\mathbf{x}^{(n+1)} = \Phi(\mathbf{x}^{(n)}), \quad n = 0, 1, 2, \ldots$$

  Construct as follows

  $$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{Bx} + (\mathbf{A} - \mathbf{B})\mathbf{x} = \mathbf{b}$$

  then put

  $$\mathbf{Bx}^{(n+1)} + (\mathbf{A} - \mathbf{B})\mathbf{x}^{(n)} = \mathbf{b}$$

  or

  $$\mathbf{x}^{(n+1)} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^{(n)} - \mathbf{B}^{-1}\mathbf{b}$$

  Many choices are possible, see e.g. Stoer and Bulirsch, *Introduction to numerical analysis*, chapter 8.

  We will return to this point later, first go back to the Finite Difference scheme

Jacobi iteration

- Finite difference scheme for Laplace equation..

$$\nabla^2 c = 0 \rightarrow \quad c_{l,m} = \frac{1}{4}\left[c_{l+1,m} + c_{l-1,m} + c_{l,m+1} + c_{l,m-1}\right], \forall (l,m)$$

- .. Immediately suggests an iterative procedure.

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n)}\right]$$

- This is the Jacobi iteration.
  - (n) is now the iteration number.
- Note that the Jabobi iteration is immediately found from the time dependent FD scheme with a maximal time step, i.e.

$D\delta x^2/\delta t = 1/4$, so Jacobi still suffers from large amount of iterations.

Sequential Jacobi iteration

```
/* Jacobi update, square domain, periodic in x, fixed upper and lower boundaries */
do {
  for i=0 to max {
    for j=0 to max {
      if(c_ij is a source) c_ij^(n+1) = 1.0
      else if(c_ij is a sink) c_ij^(n+1) = 0.0   /* e.g. an object in the lattice */
      else {
        west = (i==0) ? c_max-1,j^(n) : c_i-1,j^(n)      /* periodic boundaries */
        east = (i==max) ? c_1,j^(n) : c_i+1,j^(n)
        south = (j==0) ? c0 : c_i,j-1^(n)          /* fixed boundaries */
        north = (j==max) ? cL : c_i,j+1^(n)
        c_ij^(n+1) = 0.25 * (west + east + south + north)
      }
      if(|c_ij^(n+1) - c_ij^(n)| > tolerance) δ = |c_ij^(n+1) - c_ij^(n)| /* stopping criterion */
    }
  }
while (δ > tolerance)
```

Example 1

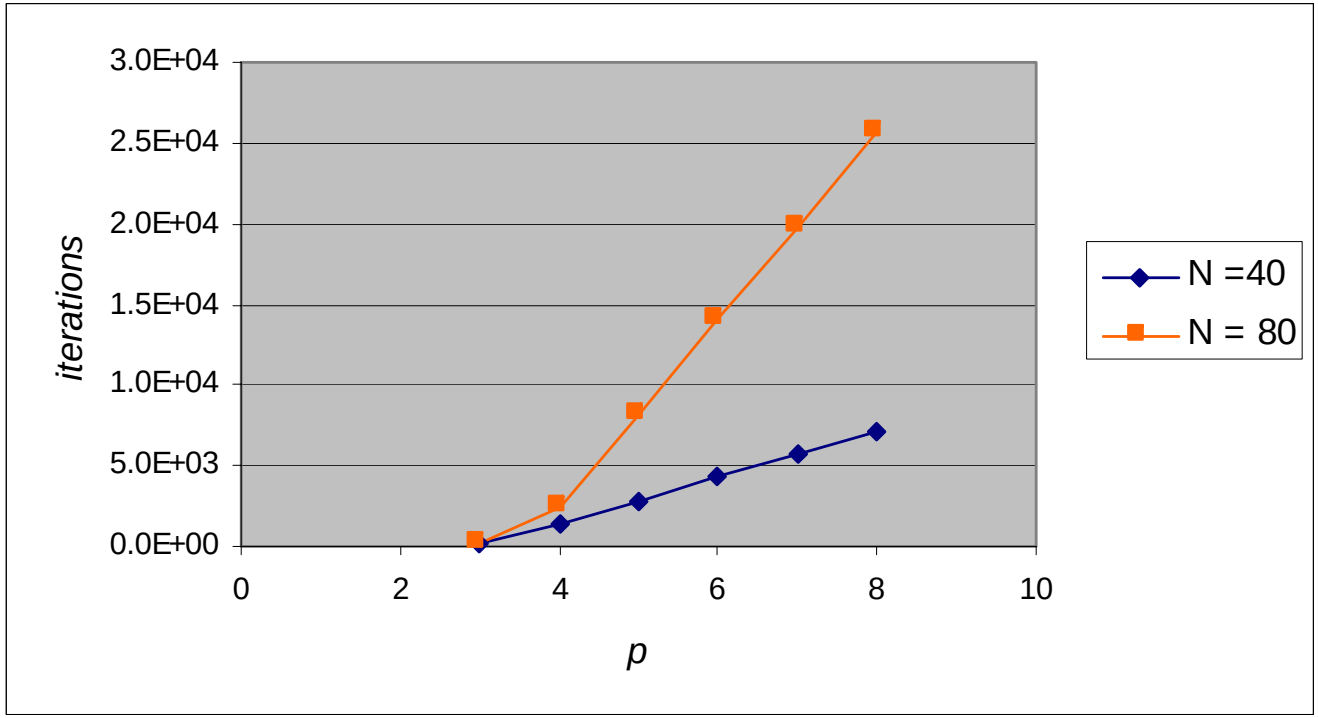
- As before, the two dimensional domain, $0 \leq x, y \leq 1$
  - periodic boundary conditions in x-direction
    - $c(x,y) = c(x+1,y)$
  - other boundary conditions
    - $c(x,y=0) = 0$
    - $c(x,y=1) = 1$
    - Note that because of symmetry the solution will not depend on the x-coordinate, and that the exact solution is a simple linear concentration profile:

      $c(x,y) = y$

      derive this result yourself !!

Example 2

- Set the stop condition $\delta = 10^{-p}$
- measure number of iterations as function of number of gridpoints (N^2) and p.



Number of iterations approximately as O(N^2)

Compare to number of expected time steps in the time dependent scheme.

Parallel Jacobi

- Update procedure is straightforward, just like the time dependent FD scheme.
  - Thanks to the local five-point stencil operation.
- However, the stopping condition requires a global communication operation.
  - This will reduce the efficiency of the parallel iteration.
  - May need some clever procedures to fine-tune
    - e.g. only check for convergence after every q iterations.
    - this is part of the lab course.

Pseudo code

```
main () /* pseudo code for parallel Jacobi iteration */
{
    decompose lattice;
    initialize lattice sites;
    set boundary conditions;
    do {
        exchange boundary strips with neighboring processors;
        for all grid points in this processor {
            update according to Jacobi iteration;
            calculate local δ_l parameter; /* stopping criterion */
        }
        obtain the global maximum δ of all local δ_l values
    }
    while (δ > tolerance)
    print results to file;
}
```

Gauss-Seidel iteration

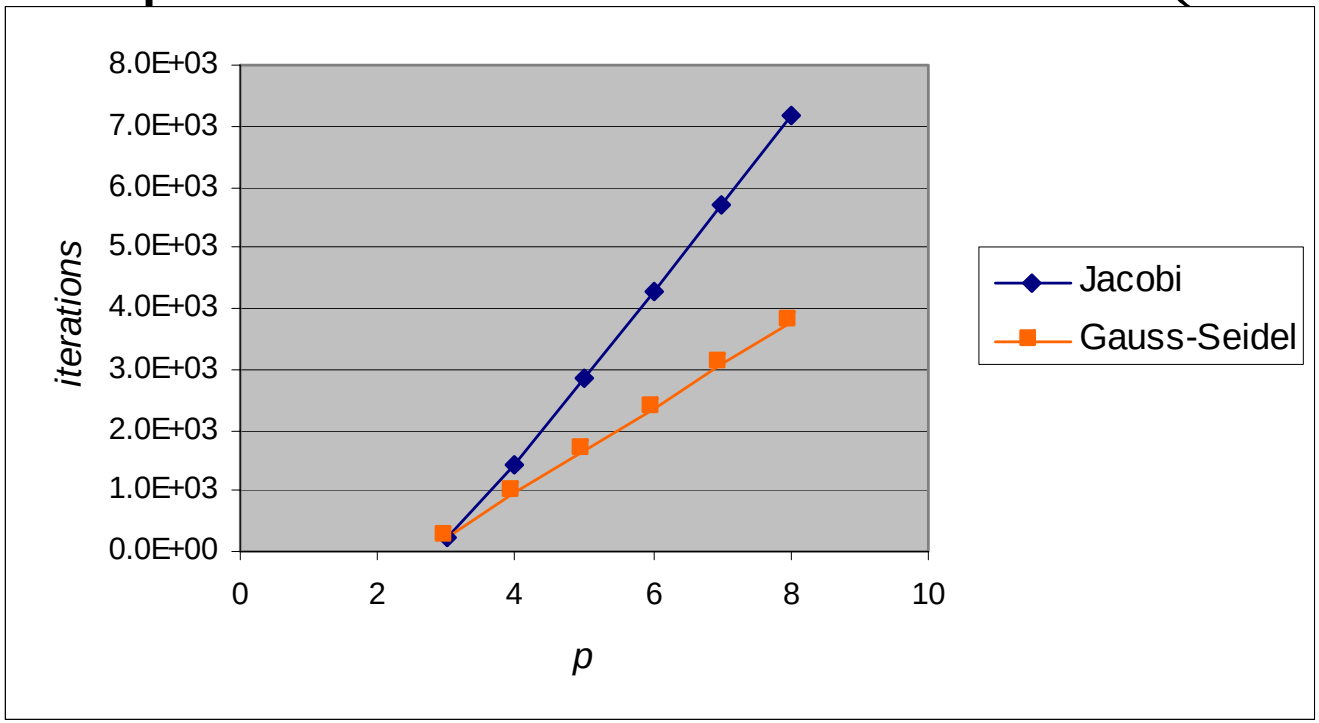
- Idea : use new values as soon as they are calculated:
  - proceed along the rows, incrementing l for fixed m

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right]$$

  - Note that Gauss-Seidel iteration is done in place
    - no need to have a buffer holding a copy of all lattice points, as in the Jacobi iterations, saves a lot of memory !

Example Continued

- Compare Jacobi and Gauss-Seidel ($N = 40$)



Gauss-Seidel needs a factor 2 less iteration than Jacobi.

Also expected from theory, see e.g. Press et al., *Numerical Recipes*, chapter 17.5

http://www.ulib.org/webRoot/Books/Numerical_Recipes/bookcpdf.html

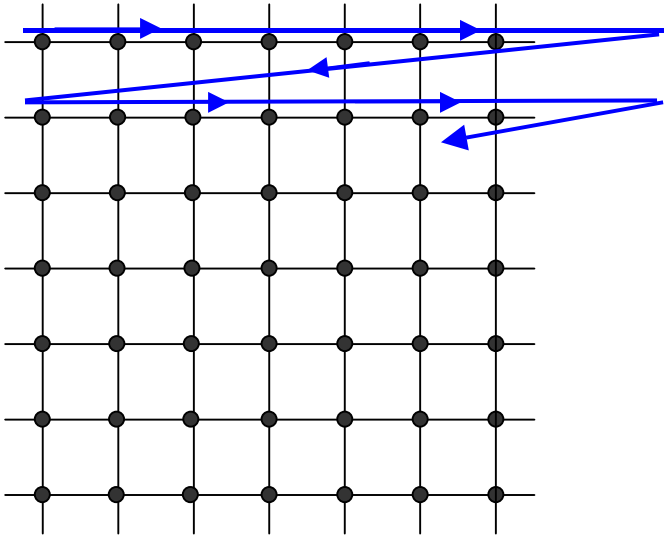
Parallel Gauss-Seidel

- The inherent parallelism in the Jacobi iteration is destroyed, and at first sight the Gauss-Seidel iteration is sequential…
- …however, this is only true because of the specific order (proceeding along the rows) in which the points are updated.
- Changing this update order will again restore the inherent parallelism.

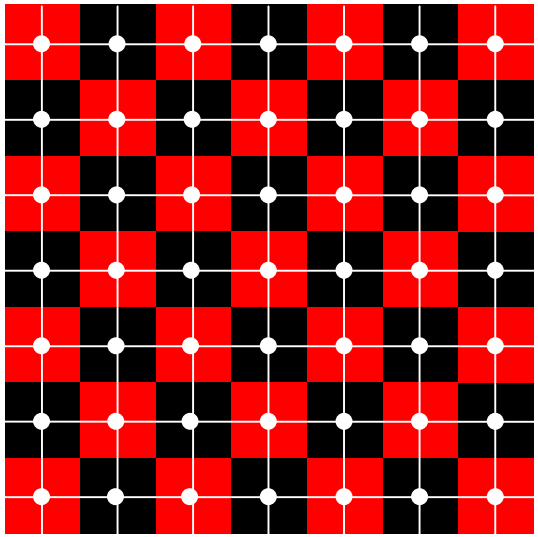
Red Black ordering 1

- Color the grid as a checkerboard, with red and black lattice points.
- All the red points are independent from each other (only depend on black points), and also all the black points are independent from each other (only depend on red points).
- So, a parallel procedure would be to first update, in parallel, all red points, and next all black points, etc..

Red Black ordering 2



Row wise ordening



Red Black ordening

Parallel Gauss-Seidel with R/B ordering

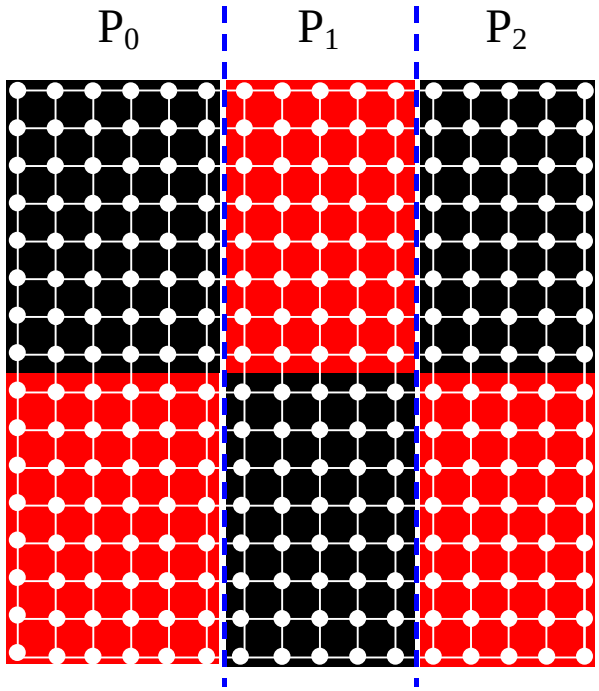
```
/* only the inner loop of the parallel Gauss-Seidel method with */
/* Red Black ordering */
    do {
        exchange boundary strips with neighboring processors;
        for all red grid points in this processor {
            update according to Gauss-Seidel iteration;
        }
        exchange boundary strips with neighboring processors;
        for all black grid points in this processor {
            update according to Gauss-Seidel iteration;
        }
        obtain the global maximum δ of all local δ_l values
    }
    while (δ > tolerance)
```

Efficiency

- Note that the Red Black ordening requires one extra exchange operation !
  - However, the exchange operation only needs to sent half the amount of grid points (why?).
- So, we recover parallelism, at the expense of twice as much communication setup time
  - need larger grain size for good parallel performance
  - make a detailed analysis yourself !

Coarse grained Red Black ordering

- A coarse grained R/B ordering is off course also possible, e.g. for a strip-decomposition.
- within each red or black block the natural row wise ordering can be used again (why ?).



Successive Over Relaxation

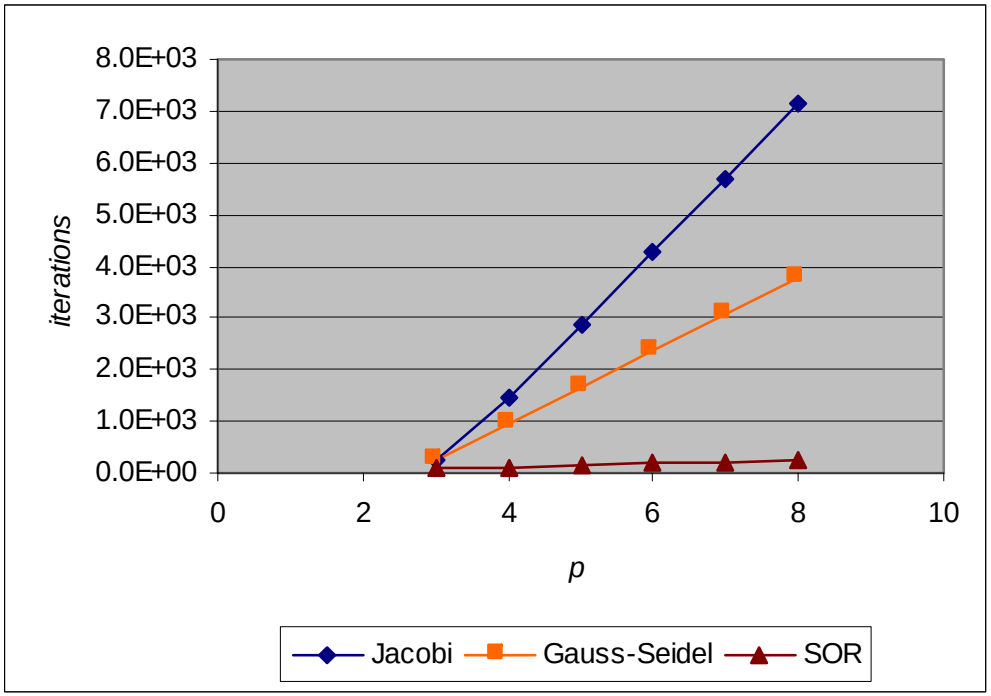
- An final improvement is to make an *over correction* of the Gauss-Seidel iteration, thus creating the Successive Over Relaxation method (SOR)

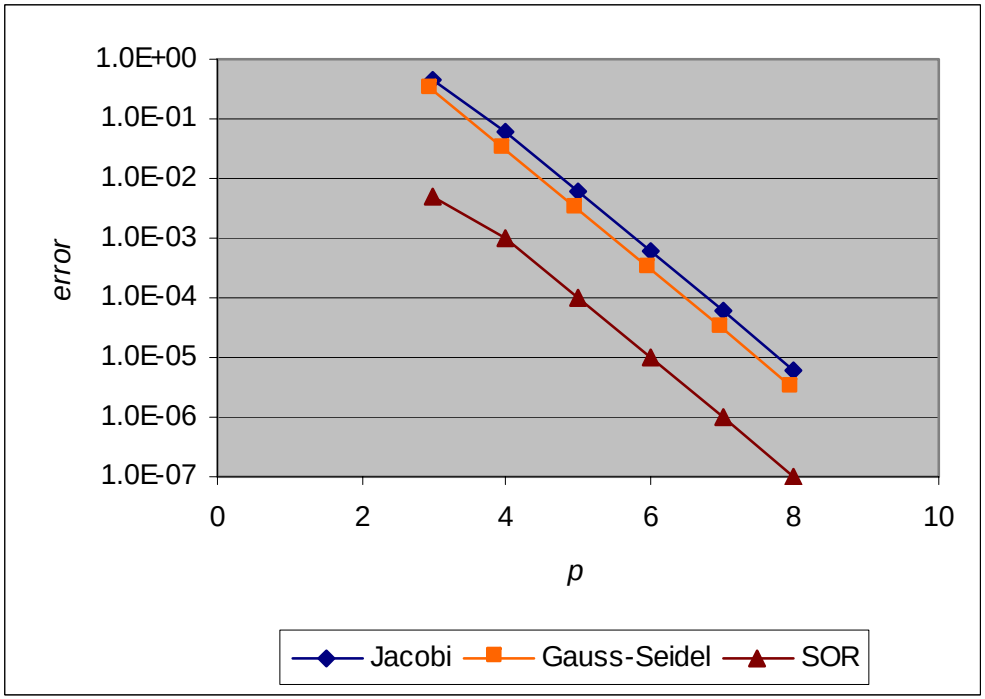
$$c_{l,m}^{(n+1)} = \frac{\omega}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right] + (1-\omega)\,c_{l,m}^{(n)}$$

- One can prove that:
  - method is convergent only for $0 < \omega < 2$
    - $0 < \omega < 1$: under relaxation
    - $\omega = 1$: Gauss-Seidel
    - $1 < \omega < 2$: over relaxation
  - usually, for finite differencing, over relaxation gives a faster convergence then Gauss-Seidel.

Compare all three methods

- Again for the example of the bounded domain ($N = 40$)
- Number of iterations
- error (defined as maximum between numerical solution and exact solution)





Iterative methods in matrix notation

- Remember that iterative methods can be constructed through
  - $\mathbf{B}\mathbf{x}^{(n+1)} + (\mathbf{A} - \mathbf{B})\mathbf{x}^{(n)} = \mathbf{b}$
- Now, write

$$\mathbf{A} = \mathbf{D} + \mathbf{E} + \mathbf{F}, \qquad \mathbf{D} = \begin{pmatrix} a_{11} & & 0 \\ & \mathrm{O} & \\ 0 & & a_{NN} \end{pmatrix}$$

$$\mathbf{E} = \begin{pmatrix} 0 & & & 0 \\ a_{21} & \mathrm{O} & & \\ \mathrm{M} & & \mathrm{O} & \\ a_{N1} & \Lambda & a_{N,N-1} & 0 \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} 0 & a_{12} & \Lambda & a_{1N} \\ & \mathrm{O} & & \mathrm{M} \\ & & \mathrm{O} & a_{N-1,N} \\ 0 & & & 0 \end{pmatrix}$$

Matrix notation 2

- Jacobi

$$\mathbf{B} = \mathbf{D} \Rightarrow a_{ii} x_i^{(n+1)} = -\sum_{i \neq j} a_{ij} x_j^{(n)} + b_i$$

- Gauss-Seidel

$$\mathbf{B} = \mathbf{D} + \mathbf{E} \Rightarrow a_{ii} x_i^{(n+1)} = -\sum_{i<j} a_{ij} x_j^{(n+1)} - \sum_{i>j} a_{ij} x_j^{(n)} + b_i$$

- SOR

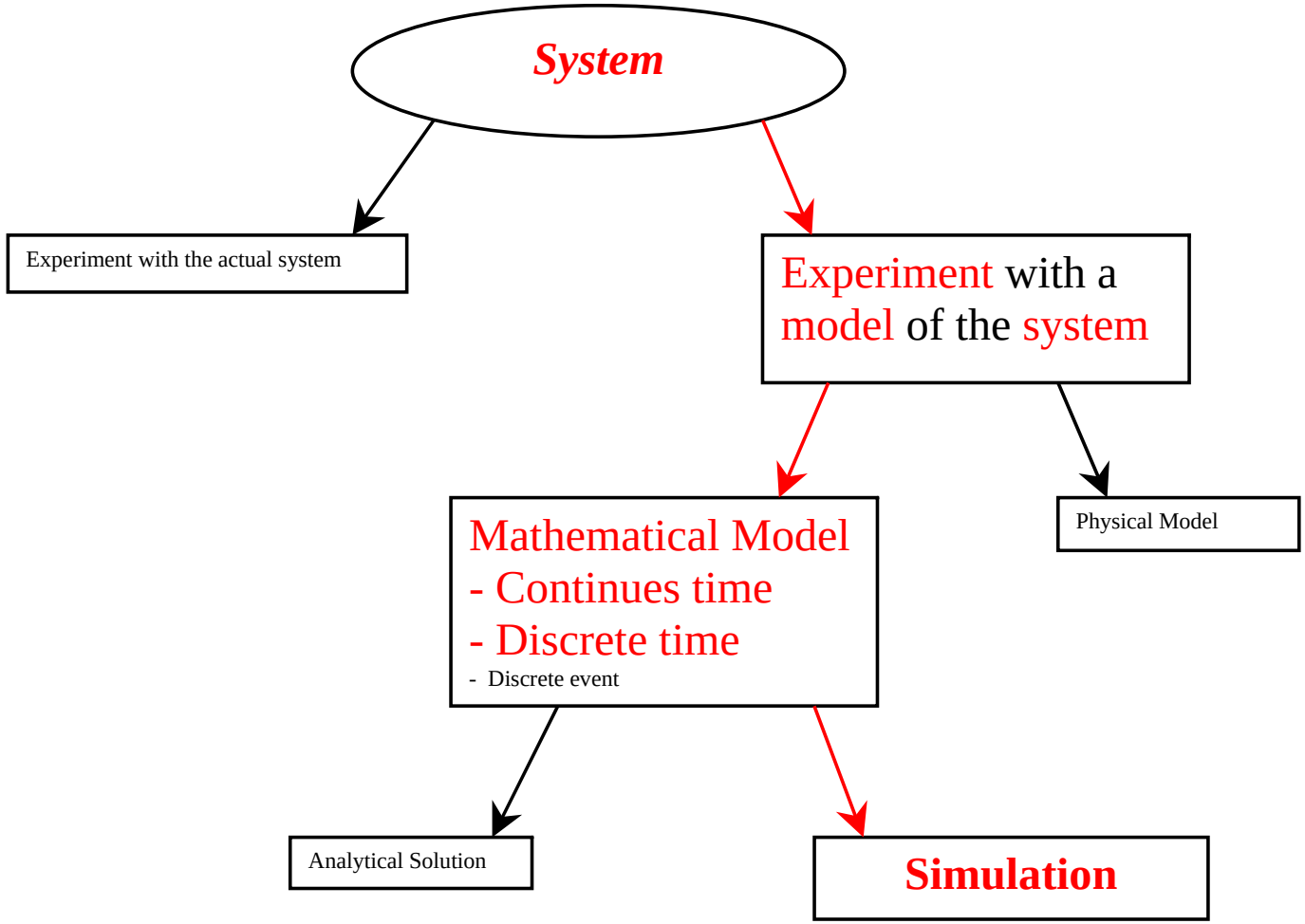
$$\mathbf{B} = \frac{1}{\omega}\mathbf{D} + \mathbf{E} \Rightarrow$$

$$a_{ii} x_i^{(n+1)} = \omega(-\sum_{i<j} a_{ij} x_j^{(n+1)} - \sum_{i>j} a_{ij} x_j^{(n)} + b_i) + (1-\omega) a_{ii} x_i^{(n)}$$

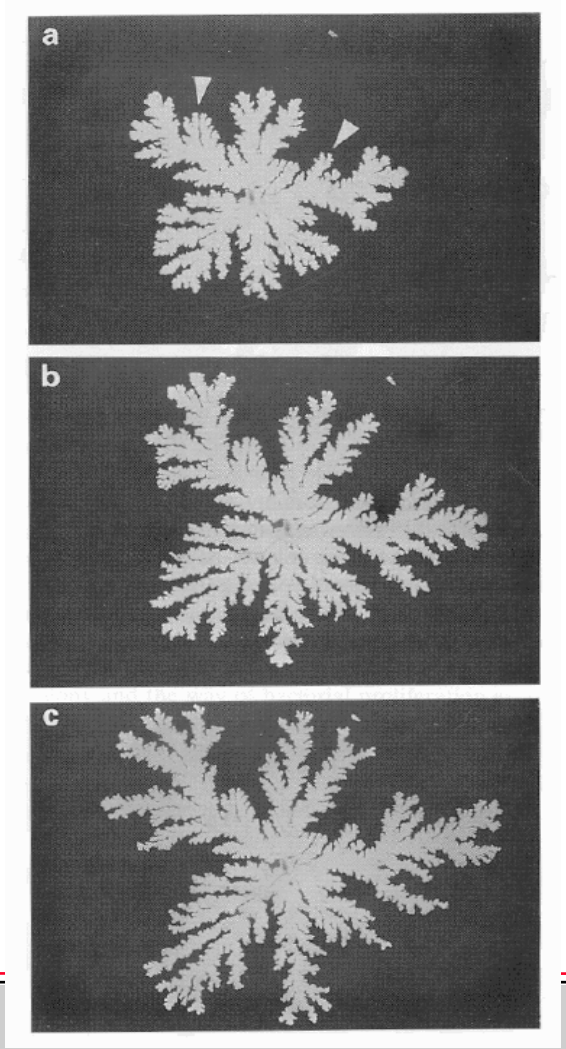
Diffusion Limited Aggregation

- Diffusion Limited Aggregation (DLA)
  - model for non-equilibrium growth
    - e.g. a *Bacillus subtilis* bacteria colony in a petri dish
    - colony feeds on nutrients in the immediate environment
    - probability of growth is determined by the concentration of nutrients.
    - concentration of nutrients in its turn is determined by diffusion
  - DLA algorithm
    1. Solve Laplace equation to get distribution of nutrients, assume that the object is a sink (i.e. $c = 0$ on the object)
    2. Let the object grow
    3. Go back to (1)

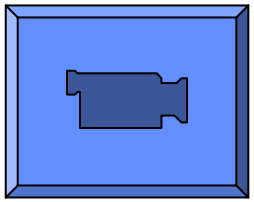
Ways to study a system



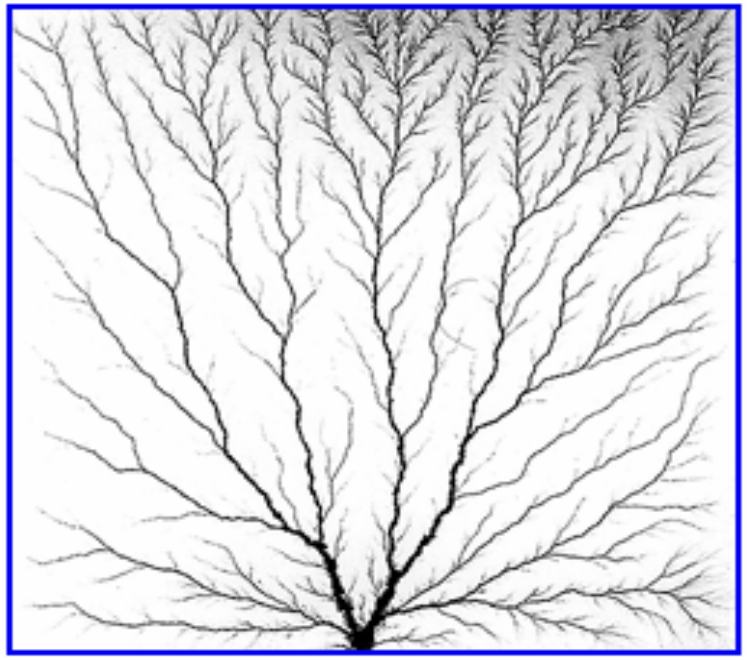
Growth of a colony of the bacterial species Bacillus subtilis *(after Matsushita & Fujikawa,1990*



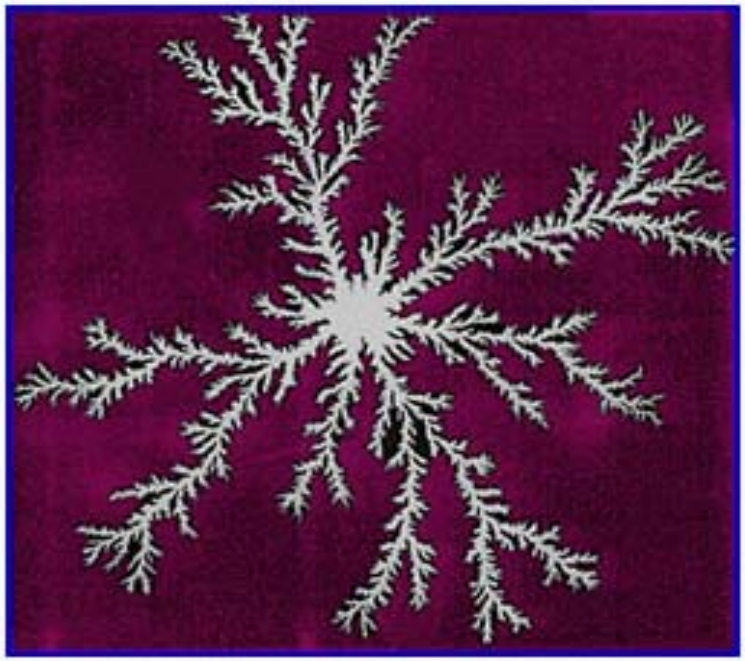
Abiotic growth and form:Viscous fingering



Abiotic growth and form:Lichtenberg patterns



Abiotic growth and form:metal deposit

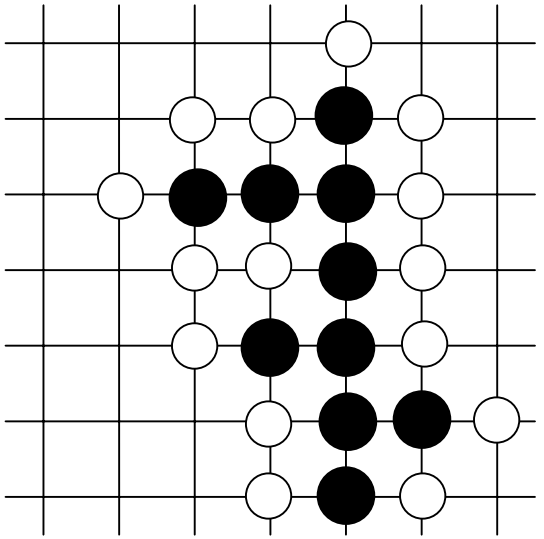


Construction of the object

1. Determine growth candidates

   black circles are part of the object

   open circles are growth candidates



2. Determine growth probabilities

$$p_g\left((i,j)\in \mathrm{o}\rightarrow(i,j)\in\bullet\right)=\frac{\left(c_{i,j}\right)^{\eta}}{\sum_{(i,j)\in \mathrm{o}}\left(c_{i,j}\right)^{\eta}}$$

   η is a free parameter, usually $0 \leq \eta \leq 2$, for classical DLA, $\eta = 1$.

3. Grow the object

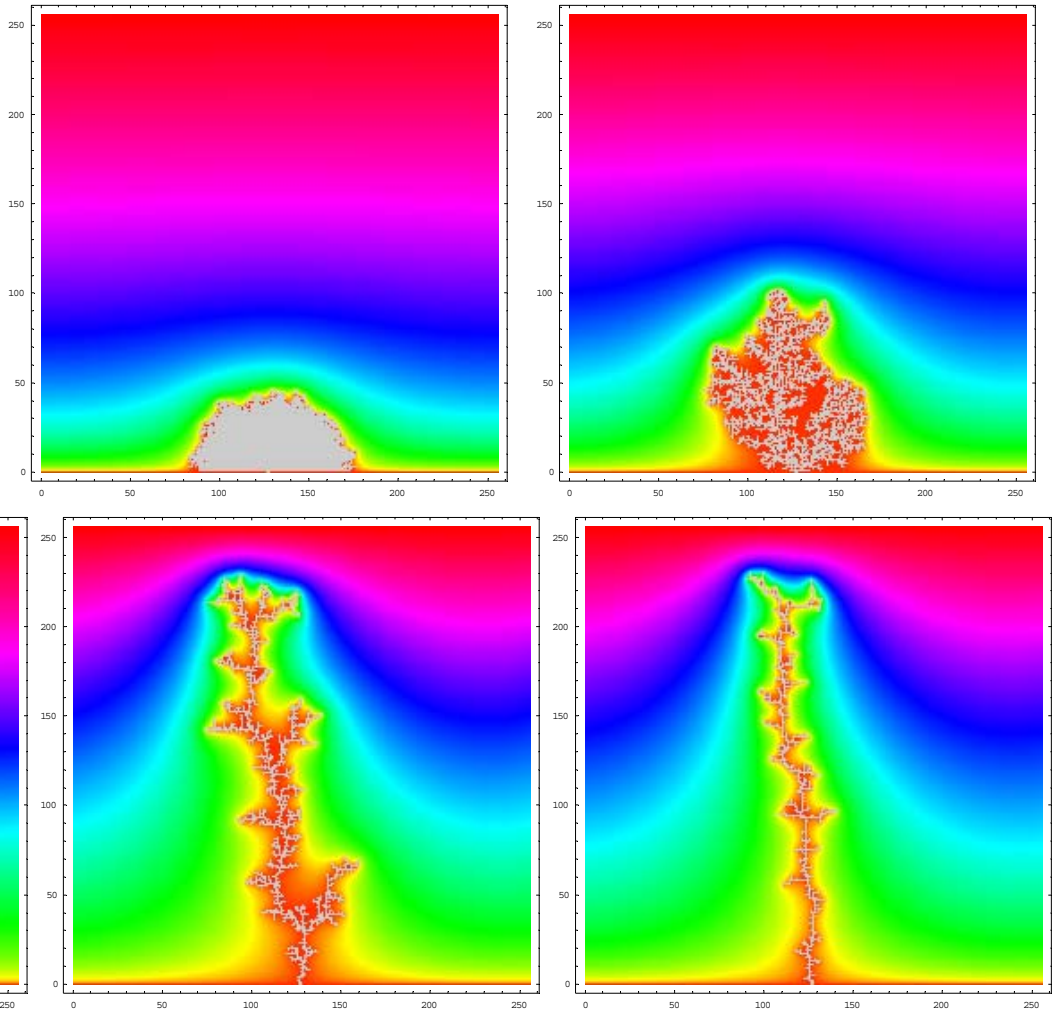
   for each grow candidate, draw a random number between 0 and 1, and if this number is smaller then p_g, add the candidate to the object.

Some results

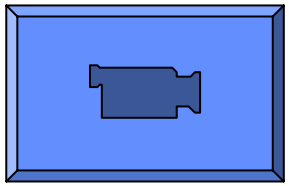
256^2 grid, Laplace equation solved with SOR.

Top lane: $\eta = 0.0$ (Eden cluster), $\eta = 0.5$

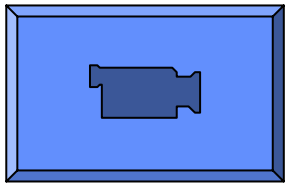
bottom lane : $\eta = 1.0$, $\eta = 1.5$, $\eta = 2.0$



Diffusion limited aggregation in 3D



Flow limited aggregation in 3D



Parallel DLA

- Easiest is to keep using the same decomposition as for the iterative solvers
  - growth step is local, completely in parallel, but
  - calculation of the growth probabilities requires a global communication (for the normalization, i.e. the denominator of the equation for the probability)
- However, due to the growing object the load balancing gets worse and worse during the simulation.
  - We will return to this issue in the part on domain decomposition.