Scientific Computing WS 2017/2018

Lecture 5

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

With material from “Introduction to High-Performance Scientific Computing” by Victor Eijkhout (http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html)

Recap from last time

Floating point limits

- symmetry wrt. 0 because of sign bit
- smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \dots t-1$
  $x_{min} = \beta^L$
- smallest positive denormalized number: $d_i = 0, i = 0 \dots t-2, d_{t-1} = 1$
  $x_{min} = \beta^{1-t}\beta^L$
- largest positive normalized number: $d_i = \beta - 1, 0 \dots t-1$
  $x_{max} = \beta(1 - \beta^{1-t})\beta^U$

Machine precision

- Exact value x
- Approximation $\tilde{x}$
- Then: $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = \beta^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}\beta^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in partiular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

Matrix + Vector norms

- Vector norms: let $x = (x_i) \in \mathbb{R}^n$
  - $||x||_1 = \sum_i =^n |x_i|$: sum norm, l_1-norm
  - $||x||_2 = \sqrt{\sum_{i=1}^n x_i^2}$: Euclidean norm, l_2-norm
  - $||x||_\infty = \max_{i=1\dots n} |x_i|$: maximum norm, l_∞-norm
- Matrix $A = (a_{ij}) \in \mathbb{R}^n \times \mathbb{R}^n$
  - Representation of linear operator $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^n$ defined by $\mathcal{A} : x \mapsto y = Ax$ with

$$y_i = \sum_{j=1}^n a_{ij} x_j$$

  - Induced matrix norm:

$$\begin{aligned} ||A||_\nu &= \max_{x \in \mathbb{R}^n, x \neq 0} \frac{||Ax||_\nu}{||x||_\nu} \\ &= \max_{x \in \mathbb{R}^n, ||x||_\nu = 1} \frac{||Ax||_\nu}{||x||_\nu} \end{aligned}$$

Matrix norms

- $||A||_1 = \max_{j=1\dots n} \sum_{i=1}^{n} |a_{ij}|$ maximum of column sums
- $||A||_\infty = \max_{i=1\dots n} \sum_{j=1}^{n} |a_{ij}|$ maximum of row sums
- $||A||_2 = \sqrt{\lambda_{max}}$ with λ_{max}: largest eigenvalue of $A^T A$.

Matrix condition number and error propagation

Problem: solve $Ax = b$, where b is inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\left\{ \begin{array}{cl} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{rl} ||A|| \cdot ||x|| & \geq ||b|| \\ ||\Delta x|| & \leq ||A^{-1}|| \cdot ||\Delta b|| \end{array} \right.$$

$$\Rightarrow \frac{||\Delta x||}{||x||} \leq \kappa(A) \frac{||\Delta b||}{||b||}$$

where $\kappa(A) = ||A|| \cdot ||A^{-1}||$ is the *condition number* of A.

Approaches to linear system solution

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

numcxx

numcxx is a small C++ library developed for and during this course which implements the concepts introduced

- Shared smart pointers vs. references
- 1D/2D Array class
- Matrix class with LAPACK interface
- Expression templates
- Interface to triangulations
- Sparse matrices + UMFPACK interface
- Iterative solvers
- Python interface

numcxx classes

- `TArray1`: templated 1D array class
  `DArray1`: 1D double array class
- `TArray2`: templated 2D array class
  `DArray2`: 2D double array class
- `TMatrix`: templated dense matrix class
  `DMatrix`: double dense matrix class
- `TSolverLapackLU`: LU factorization based on LAPACK
  `DSolverLapackLU`

Obtaining and compiling the examples

- Copy files, creating subdirectory part2
  - the . denotes the current directory

```
$ ls /net/wir/numxx/examples/10-numcxx-basicx/*.cxx
$ cp -r /net/wir/examples/10-numcxx-basicx/numcxx-expressions.cxx .
```

- Compile sources (for each of the .cxx files) (integrates with codeblocks)

```
$ numcxx-build -o example numcxx-expressions.cxx
$ ./example
```

CMake

What is behind `numcxx-build`?

- CMake - the current best way to build code
- Describe project in a file called `CMakeLists.txt`

```
cmake_minimum_required(VERSION 2.8.12)
PROJECT(example C CXX)
find_package(NUMCXX REQUIRED)
include_directories("${NUMCXX_INCLUDE_DIRS}")
link_libraries("${NUMCXX_LIBRARIES}")
add_executable(example example.cxx)
```

- Set up project (only once)

```
$ mkdir builddir
$ cd buildir
$ cmake ..
$ cd ..
```

- build code

```
$ cmake --build builddir
```

- run code

```
$ ./builddir/example
```

Let's have some naming conventions

- lowercase letters: scalar values
  - `i,j,k,l,m,n` standalone or as prefixes: integers, indices
  - others: floating point
- Upper_case_letters: class objects/references

```
std::vector<double> X(n);
numcxx::DArray1<double> Y(n);
```

- pUpper_case_letters: smart pointers to objects

```
auto pX=std::make_shared<std::vector<double>>(n);
auto pY=numcxx::TArray1<double>::create(n);
auto pZ=numcxx::TArray1<double>::create({1,2,3,4});

// getting references from smart pointers
auto &X=*pX;
auto &Y=*pY;
auto &Z=*pZ;

auto W=std::make_shared<std::vector<double>>({1,2,3,4}); // doesn't work...
```

C++ code using vectors, C++-style with smart pointers

File
/net/wir/numcxx/examples/00-cxx-basics/05-cxx-style-sharedptr.cxx

```
#include <cstdio>
#include <vector>
#include <memory>
void initialize(std::vector<double> &x)
{  for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);}
double sum_elements(std::vector<double> & x)
{    double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;
}
int main()
{   const int n=12345678;
    // call constructor and wrap pointer into smart pointer
    auto x=std::make_shared<std::vector<double>>(n);
    initialize(*x);
    double s=sum_elements(*x);
    printf("sum=%e\n",s);
    // smartpointer calls destructor if reference count reaches zero
}
```

- Heap memory management controlled by smart pointer lifetime
- If method or function does not store the object, pass by reference ⇒ API stays the same as for previous case.

C++ code using numcxx with references

File /net/wir/examples/10-numcxx-basicx/numcxx-ref.cxx

```
#include <cstdio>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  numcxx::TArray1<double> X(n);
  initialize(X);
  double s=sum_elements(X);
  printf("sum=%e\n",s);
}
```

C++ code using numcxx with smart pointers

File /net/wir/examples/10-numcxx-basics/numcxx-sharedptr.cxx

```
#include <cstdio>
#include <memory>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  // call constructor and wrap pointer into smart pointer
  auto pX=numcxx::TArray1<double>::create(n);
  initialize(*pX);
  double s=sum_elements(*pX);
  printf("sum=%e\n",s);
}
```

Solution of linear systems of equations

Approaches to linear system solution

Let A: $n \times n$ matrix, $b \in \mathbb{R}^n$.

Solve $Ax = b$

- Direct methods:
  - Exact
    - up to machine precision
    - condition number
  - Expensive (in time and space)
    - where does this matter ?
- Iterative methods:
  - Only approximate
    - with good convergence and proper accuracy control, results are not worse than for direct methods
  - May be cheaper in space and (possibly) time
  - Convergence guarantee is problem dependent and can be tricky

Complexity: "big O notation"

- Let $f, g : \mathbb{V} \to \mathbb{R}^+$ be some functions, where $\mathbb{V} = \mathbb{N}$ or $\mathbb{V} = \mathbb{R}$.

  We write
  $$f(x) = O(g(x)) \quad (x \to \infty)$$
  if there exist a constant $C > 0$ and $x_0 \in \mathbb{V}$ such that
  $$\forall x > x_0, \quad |f(x)| \leq C|g(x)|$$
- Often, one skips the part "$(x \to \infty)$"
- Examples:
  - Addition of two vectors: $O(n)$
  - Matrix-vector multiplication (for matrix where all entries are assumed to be nonzero): $O(n^2)$

Really bad example of direct method

Solve $Ax = b$ by Cramer's rule

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A| \quad (i = 1 \dots n)$$

This takes $O(n!)$ operations...

Gaussian elimination

- Essentially the only feasible direct solution method
- Solve $Ax = b$ with square matrix A.
- While formally, the algorithm is always the same, its implementation depends on
  - data structure to store matrix
  - possibility to ignore zero entries for matrices with many zeroes
  - sorting of elements

Gaussian elemination: pass 1

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} x = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

Step 1: $\mathrm{equation}_2 \leftarrow \mathrm{equation}_2 - 2\,\mathrm{equation}_1$
$\mathrm{equation}_3 \leftarrow \mathrm{equation}_3 - \frac{1}{2}\,\mathrm{equation}_1$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix}$$

Step 2: $\mathrm{equation}_3 \leftarrow \mathrm{equation}_3 - 3\,\mathrm{equation}_2$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

Gaussian elimination: pass 2

Solve upper triangular system

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

$$\begin{aligned} -4x_3 &= -9 & & & \Rightarrow x_3 &= \frac{9}{4} \\ -4x_2 + 2x_3 &= -6 & \Rightarrow -4x_2 &= -\frac{21}{2} & \Rightarrow x_2 &= \frac{21}{8} \\ 6x_1 - 2x_2 + 2x_3 &= 2 & \Rightarrow 6x_1 &= 2 + \frac{21}{4} - \frac{18}{4} = \frac{11}{4} & \Rightarrow x_1 &= \frac{11}{4} \end{aligned}$$

LU factorization

Pass 1 expressed in matrix operation

$$L_1Ax = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix} = L_1 b, \qquad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix}$$

$$L_2L_1Ax = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix} = L_2L_1 b, \qquad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

- Let $L = L_1^{-1}L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 3 & 1 \end{pmatrix}$, $U = L_2L_1A$. Then $A = LU$
- Inplace operation. Diagonal elements of L are always 1, so no need to store them $\Rightarrow$ work on storage space for A and overwrite it.

LU factorization

Solve $Ax = b$

- Pass 1: factorize $A = LU$ such that L, U are lower/upper triangular
- Pass 2: obtain $x = U^{-1}L^{-1}b$ by solution of lower/upper triangular systems
  - 1. solve $L\tilde{x} = b$
  - 2. solve $Ux = \tilde{x}$
- We never calculate A^{-1} as this would be more expensive

Problem example

- Consider $\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 1 \end{pmatrix}$
- Solution: $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Machine arithmetic: Let $\epsilon << 1$ such that $1+\epsilon = 1$.
- Equation system in machine arithmetic:
  $1 \cdot \epsilon + 1 \cdot 1 = 1 + \epsilon$

  $1 \cdot 1 + 1 \cdot 1 = 2$
- Still fulfilled!

Problem example II: Gaussian elimination

- Ordinary elimination: $\mathrm{equation}_2 \leftarrow \mathrm{equation}_2 - \frac{1}{\epsilon}\,\mathrm{equation}_1$
  $\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 2-\frac{1+\epsilon}{\epsilon} \end{pmatrix}$
- In exact arithmetic:

$$\Rightarrow x_2 = \frac{1-\frac{1}{\epsilon}}{1-\frac{1}{\epsilon}} = 1 \Rightarrow x_1 = \frac{1+\epsilon-x_2}{\epsilon} = 1$$

- In floating point arithmetic: $1+\epsilon=1$, $1-\frac{1}{\epsilon}=-\frac{1}{\epsilon}$, $2-\frac{1}{\epsilon}=-\frac{1}{\epsilon}$:
  $\begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{\epsilon} \end{pmatrix}$
  $\Rightarrow x_2 = 1 \Rightarrow \epsilon x_1 + 1 = 1 \Rightarrow x_1 = 0$

Problem example III: Partial Pivoting

- Before elimination step, look at the element with largest absolute value in current column and put the corresponding row "on top" as the "pivot"
- This prevents near zero divisions and increases stability

$$\begin{pmatrix}1 & 1\\ \epsilon & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}2\\ 1+\epsilon\end{pmatrix}\Rightarrow\begin{pmatrix}1 & 1\\ 0 & 1-\epsilon\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}2\\ 1-\epsilon\end{pmatrix}$$

- Independent of ϵ:

$$x_2=\frac{1-1\epsilon}{1-\epsilon}=1, \qquad x_1=2-x_2=1$$

- Instead of A, factorize PA: $PA=LU$, where P is a permutation matrix which can be encoded using an integer vector

Gaussian elimination and LU factorization

- Full pivoting: in addition to row exchanges, perform column exchanges to ensure even larger pivots. Seldomly used in practice.
- Gaussian elimination with partial pivoting is the "working horse" for direct solution methods
- Complexity of LU-Factorization: $O(N^3)$, some theoretically better algorithms are known with e.g. $O(N^{2.736})$
- Complexity of triangular solve: $O(N^2)$
  $\Rightarrow$ overall complexity of linear system solution is $O(N^3)$

Cholesky factorization

- $A = LL^T$ for symmetric, positive definite matrices

BLAS, LAPACK

- BLAS: Basic Linear Algebra Subprograms `http://www.netlib.org/blas/`
  - Level 1 - vector-vector: $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$
  - Level 2 - matrix-vector: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
  - Level 3 - matrix-matrix: $C \leftarrow \alpha AB + \beta C$
- LAPACK: Linear Algebra PACKage `http://www.netlib.org/lapack/`
  - Linear system solution, eigenvalue calculation etc.
  - `dgetrf`: LU factorization
  - `dgetrs`: LU solve
- Used in overwhelming number of codes (e.g. matlab, scipy etc.). Also, C++ matrix libraries use these routines. Unless there is special need, they should be used.
- Reference implementations in Fortran, but many more implementations available which carefully work with cache lines etc.

Matrices from PDEs

- So far, we assumed that matrices are stored in a two-dimensional, $n \times n$ array of numbers
- This kind of matrices are also called *dense* matrices
- As we will see, matrices from PDEs (can) have a number of structural properties one can take advantage of when storing a matrix and solving the linear system

1D heat conduction

- v_L, v_R: ambient temperatures, α: heat transfer coefficient
- Second order boundary value problem in $\Omega = [0,1]$:

$$\begin{aligned} -u''(x) &= f(x) \qquad\qquad \text{in}\Omega \\ -u'(0) + \alpha(u(0) - v_L) &= 0 \\ u'(1) + \alpha(u(1) - v_R) &= 0 \end{aligned}$$

- Let $h = \frac{1}{n-1}$, $x_i = x_0 + (i-1)h \; i = 1 \dots n$ be discretization points, let u_i approximations for $u(x_i)$ and $f_i = f(x_i)$
- Finite difference approximation:

$$\begin{aligned} -u'(0) + \alpha(u(0) - v_L) &\approx \frac{1}{h}(u_0 - u_1) + \alpha(u_0 - v_L) \\ -u''(x_i) - f(x_i) &\approx \frac{1}{h^2}(u_{i+1} - 2u_i - u_{i-1}) - f_i \qquad (i = 2 \dots n-1) \\ u'(1) + \alpha(u(1) - v_R) &\approx \frac{1}{h}(u_n - u_{n-1}) + \alpha(u_n - v_R) \end{aligned}$$

1D heat conduction: discretization matrix

- equations $2 \dots n-1$ multiplied by h
- only nonzero entries written

$$\begin{pmatrix}
\alpha+\frac{1}{h} & -\frac{1}{h} & & & & & \\
-\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & & \\
 & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\
 & \ddots & \ddots & \ddots & \ddots & & \\
 & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\
 & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\
 & & & & -\frac{1}{h} & \frac{1}{h}+\alpha
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix}
=
\begin{pmatrix} \alpha v_L \\ hf_2 \\ hf_3 \\ \vdots \\ hf_{N-2} \\ hf_{N-1} \\ \alpha v_R \end{pmatrix}$$

- Each row contains ≤ 3 elements
- Only $3n-2$ of n^2 elements are non-zero

General tridiagonal matrix

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{pmatrix}$$

- To store matrix, it is sufficient to store only nonzero elements in three one-dimensional arrays for a_i, b_i, c_i, respectively

Gaussian elimination for tridiagonal systems

Gaussian elimination using arrays a, b, c as matrix storage ?

From what we have seen, this question arises in a quite natural way, and historically, the answer has been given several times

- TDMA (tridiagonal matrix algorithm)
- “Thomas algorithm” (Llewellyn H. Thomas, 1949 (?))
- “Progonka method” (from Russian ”run through”; Gelfand, Lokutsievski, 1952, published 1960)

Progonka: derivation

- $a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i \quad (i = 1 \dots n)$; $a_1 = 0$, $c_N = 0$
- For $i = 1 \dots n-1$, assume there are coefficients α_i, β_i such that $u_i = \alpha_{i+1} u_{i+1} + \beta_{i+1}$.
- Then, we can express u_{i-1} and u_i via u_{i+1}: $(a_i\alpha_i\alpha_{i+1} + b_i\alpha_{i+1} + c_i)u_{i+1} + a_i\alpha_i\beta_{i+1} + a_i\beta_i + b_i\beta_{i+1} - f_i = 0$
- This is true independently of u if

$$\begin{cases} a_i\alpha_i\alpha_{i+1} + b_i\alpha_{i+1} + c_i & = 0 \\ a_i\alpha_i\beta_{i+1} + a_i\beta_i + b_i\beta_{i+1} - f_i & = 0 \end{cases}$$

- or for $i = 1 \dots n-1$:

$$\begin{cases} \alpha_{i+1} & = -\frac{c_i}{a_i\alpha_i + b_i} \\ \beta_{i+1} & = \frac{f_i - a_i\beta_i}{a_i\alpha_i + b_i} \end{cases}$$

Progonka: realization

- Forward sweep:

$$\begin{cases}\alpha_2 & = -\frac{c_1}{b_1}\\ \beta_2 & = \frac{f_i}{b_1}\end{cases}$$

  for $i = 2 \dots n-1$

$$\begin{cases}\alpha_{i+1} & = -\frac{c_i}{a_i\alpha_i + b_i}\\ \beta_{i+1} & = \frac{f_i - a_i\beta_i}{a_i\alpha_i + b_i}\end{cases}$$

- Backward sweep:

$$u_n = \frac{f_n - a_n\beta_n}{a_n\alpha_n + b_n}$$

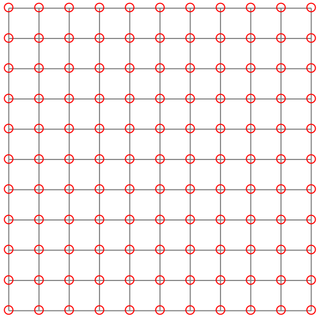
  for $n-1 \dots 1$:

$$u_i = \alpha_{i+1}u_{i+1} + \beta_{i+1}$$

Progonka: properties

- n unknowns, one forward sweep, one backward sweep
  $\Rightarrow O(n)$ operations vs. $O(n^3)$ for algorithm using full matrix
- No pivoting $\Rightarrow$ stability issues
  - Stability for diagonally dominant matrices ($|b_i| > |a_i| + |c_i|$)
  - Stability for symmetric positive definite matrices

2D finite difference grid



- Each discretization point has not more then 4 neighbours
- Matrix can be stored in five diagonals,
  LU factorization not anymore $\equiv$ "fill-in"
- Certain iterative methods can take advantage of the regular and hierachical structure (multigrid) and are able to solve system in $O(n)$ operations
- Another possibility: fast Fourier transform with $O(n \log n)$ operations

Sparse matrices

- Tridiagonal and five-diagonal matrices can be seen as special cases of *sparse matrices*
- Generally they occur in finite element, finite difference and finite volume discretizations of PDEs on structured and unstructured grids
- Definition: Regardless of number of unknowns n, the number of non-zero entries per row remains limited by n_r
- If we find a scheme which allows to store only the non-zero matrix entries, we would need $nn_r = O(n)$ storage locations instead of n^2
- The same would be true for the matrix-vector multiplication if we program it in such a way that we use every nonzero element just once: matrix-vector multiplication would use $O(n)$ instead of $O(n^2)$ operations

Sparse matrix questions

- What is a good storage format for sparse matrices?
- Is there a way to implement Gaussian elimination for general sparse matrices which allows for linear system solution with $O(n)$ operation ?
- Is there a way to implement Gaussian elimination *with pivoting* for general sparse matrices which allows for linear system solution with $O(n)$ operations?
- Is there *any algorithm* for sparse linear system solution with $O(n)$ operations?