Using direct solvers

Scientific Computing Winter 2016/2017

Lecture 7

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

With material from from `http://www.cplusplus.com/` and from "Expression templates revisited" by K. Iglberger

Recap from last time

Gaussian elimination expressed in matrix operations: LU factorization

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

$$L_2 L_1 A x = \begin{pmatrix} 6 & -2 & 2 \\ 0 & 4 & -2 \\ 0 & -0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix} = L_2 L_1 b, \qquad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

- Let $L = L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 3 & 1 \end{pmatrix}$, $U = L_2 L_1 A$. 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.

Problem example

Consider

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2 \end{pmatrix}$$

with solution $x = (1,1)^t$

Ordinary elimination:

$$\begin{pmatrix} \epsilon & 1 \\ 0 & (1-\frac{1}{\epsilon}) \end{pmatrix} x = \begin{pmatrix} 1 \\ 2-\frac{1}{\epsilon} \end{pmatrix}$$

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

If $\epsilon < \epsilon_{\mathrm{mach}}$, then $2 - 1/\epsilon = -1/\epsilon$ and $1 - 1/\epsilon = -1/\epsilon$, so

$$x_2 = \frac{2-\frac{1}{\epsilon}}{1-\frac{1}{\epsilon}} = 1, \Rightarrow x_1 = \frac{1-x_2}{\epsilon} = 0$$

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

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} x = \begin{pmatrix} 2 \\ 1-2\epsilon \end{pmatrix}$$

If ϵ very small:

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

- Factorization: $PA = LU$, where P is a permutation matrix which can be encoded usin 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
- Standard routines from LAPACK: `dgetrf`, (factorization) `dgetrs` (solve) used in overwhelming number of codes (e.g. matlab, scipy etc.). Also, C++ matrix libraries use them. Unless there is special need, they should be used.
- Complexity of LU-Factorization: $O(n^3)$, some theoretically better algorithms are known with e.g. $O(n^{2.736})$

Cholesky factorization

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

Matrices from PDE: a first example

- "Drosophila": Poisson boundary value problem in rectangular domain

Given:

- Domain $\Omega = (0, X) \times (0, Y) \subset \mathbb{R}^2$ with boundary $\Gamma = \partial\Omega$, outer normal $\mathbf{n}$
- Right hand side $f : \Omega \to \mathbb{R}$
- "Conductivity" λ
- Boundary value $v : \Gamma \to \mathbb{R}$
- Transfer coefficient α

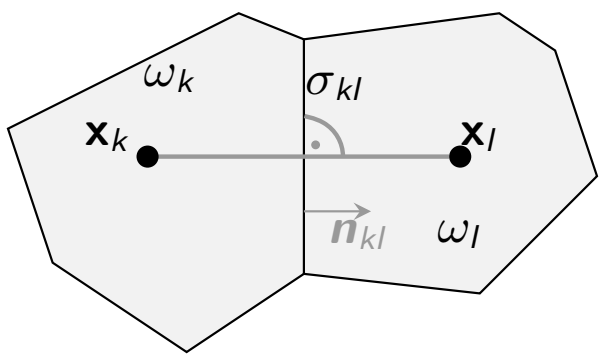
Search function $u : \Omega \to \mathbb{R}$ such that

$$\begin{aligned} -\nabla \cdot \lambda \nabla u &= f \quad \text{in} \Omega \\ -\lambda \nabla u \cdot \mathbf{n} + \alpha(u - v) &= 0 \quad \text{on} \Gamma \end{aligned}$$

- Example: heat conduction:
  - u: temperature
  - f: volume heat source
  - λ: heat conduction coefficient
  - v: Ambient temperature
  - α: Heat transfer coefficient

The finite volume idea

- Assume Ω is a polygon
- Subdivide the domain Ω into a finite number of **control volumes** :
  $\bar{\Omega} = \bigcup_{k \in \mathcal{N}} \bar{\omega}_k$
  such that
  - ω_k are open (not containing their boundary) convex domains
  - $\omega_k \cap \omega_l = \emptyset$ if $\omega_k \neq \omega_l$
  - $\sigma_{kl} = \bar{\omega}_k \cap \bar{\omega}_l$ are either empty, points or straight lines
    - we will write $|\sigma_{kl}|$ for the length
    - if $|\sigma_{kl}| > 0$ we say that ω_k, ω_l are neigbours
    - neigbours of ω_k: $\mathcal{N}_k = \{l \in \mathcal{N} : |\sigma_{kl}| > 0\}$
- To each control volume ω_k assign a **collocation point**: $\mathbf{x}_k \in \bar{\omega}_k$ such that
  - **admissibility condition**: if $l \in \mathcal{N}_k$ then the line $\mathbf{x}_k\mathbf{x}_l$ is orthogonal to σ_{kl}
  - if ω_k is situated at the boundary, i.e. $\gamma_k = \partial\omega_k \cap \partial\Omega \neq \emptyset$, then $\mathbf{x}_k \in \partial\Omega$



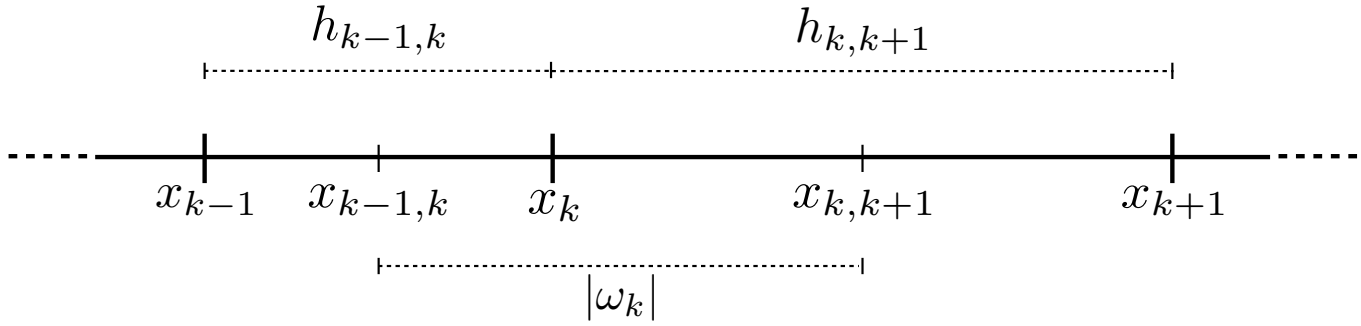
Discretization ansatz

- Given control volume ω_k, integrate equation over control volume

$$
\begin{aligned}
0 &= \int_{\omega_k} \left(-\nabla \cdot \lambda \nabla u - f\right) d\omega \\
&= -\int_{\partial\omega_k} \lambda \nabla u \cdot \mathbf{n}_k d\gamma - \int_{\omega_k} f d\omega && \text{(Gauss)} \\
&= -\sum_{L \in \mathcal{N}_k} \int_{\sigma_{kl}} \lambda \nabla u \cdot \mathbf{n}_{kl} d\gamma - \int_{\gamma_k} \lambda \nabla u \cdot \mathbf{n} d\gamma - \int_{\omega_k} f d\omega \\
&\approx \sum_{L \in \mathcal{N}_k} \frac{\sigma_{kl}}{h_{kl}} (u_k - u_l) + |\gamma_k| \alpha (u_k - v_k) - |\omega_k| f_k
\end{aligned}
$$

- Here,
  - $u_k = u(\mathbf{x}_k)$
  - $v_k = v(\mathbf{x}_k)$
  - $f_k = f(\mathbf{x}_k)$
- $N = |\mathcal{N}|$ equations (one for each control volume)
- $N = |\mathcal{N}|$ unknowns (one in each collocation point $\equiv$ control volume)

1D finite volume grid



- $\Omega = [0, X]$
- Collocation points:
  $0 = x_1 < x_2 < \cdots < x_{n-1} < x_n = X$
- Control volumes:

$$
\begin{aligned}
\omega_1 &= (x_1, (x_1 + x_2)/2) \\
\omega_2 &= ((x_1 + x_2)/2, (x_2 + x_3)/2) \\
&\vdots \\
\omega_{N-1} &= ((x_{N-2} + x_{N-1})/2, (x_{N-1} + x_N)/2) \\
\omega_N &= ((x_{N-1} + x_N)/2, x_N)
\end{aligned}
$$

- Maximum number of neighbours: 2

Discretization matrix (1D)

Assume $\lambda = 1$, $h_{kl} = h$ and we count collocation points from $1 \dots N$. For $k = 2 \dots N-1$, $\omega_K = h$, and

$$\sum_{L \in \mathcal{N}_k} \frac{\sigma_{kl}}{h_{kl}} (u_k - u_l) = \frac{1}{h}(-u_{k-1} + 2u_k - u_{k+1})$$

The linear system then is (only nonzero entries marked):

$$\begin{pmatrix} \alpha + \frac{1}{h} & -\frac{1}{h} & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\ & & & & -\frac{1}{h} & \frac{1}{h} + \alpha \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} \frac{h}{2} f_1 + \alpha v_1 \\ h f_2 \\ h f_3 \\ \vdots \\ h f_{N-2} \\ h f_{N-1} \\ \frac{h}{2} f_N + \alpha v_n \end{pmatrix}$$

General tridiagonal matrix

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

Gaussian elimination for tridiagonal systems

- TDMA (tridiagonal matrix algorithm)
- "Thomas algorithm" (Llewellyn H. Thomas, 1949 (?))
- "Progonka method" (Gelfand, Lokutsievski, 1952, published 1960)

$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i,\ a_1 = 0,\ c_N = 0$

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}:
$(a_i \alpha_i \alpha_{i+1} + c_i \alpha_{i+1} + b_i) u_{i+1} + a_i \alpha_i \beta_{i+1} + a_i \beta_i + c_i \beta_{i+1} - f_i = 0$

This is true independently of u if

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

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

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

Progonka algorithm

Forward sweep:

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

for $i = 2 \dots n-1$

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

Backward sweep:

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

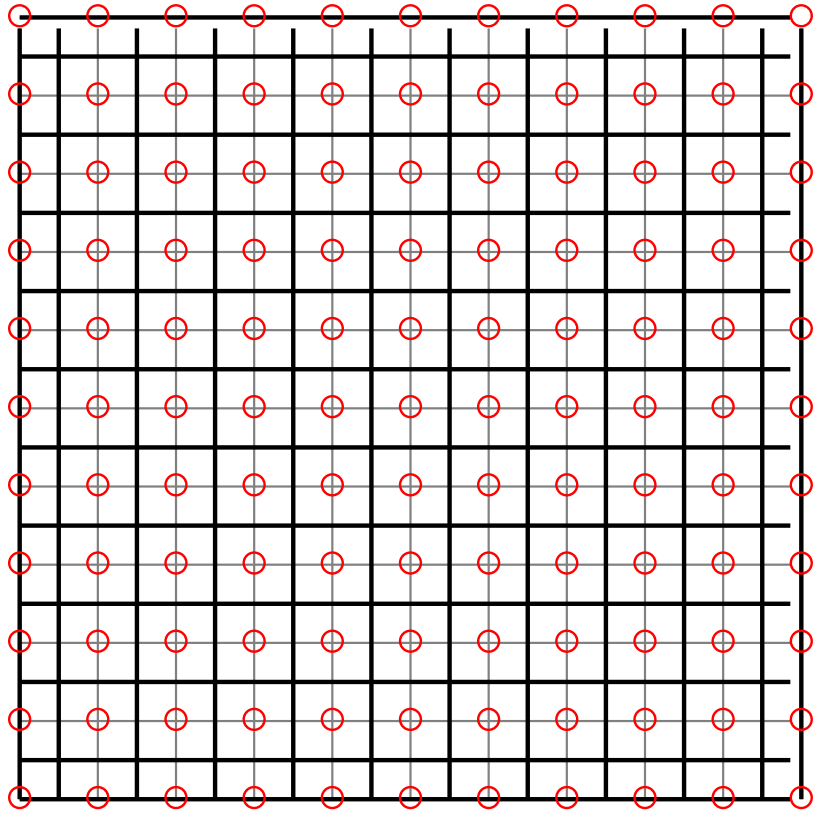
for $n-1 \dots 1$:

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

Progonka algorithm - 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 volume grid



- Red circles: discretization nodes
- Thin lines: original “grid”
- Thick lines: boundaries of control volumes
- Each discretization point has not more then 4 neighbours

Sparse matrices

- 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:
  martrix-vector multiplication uses $O(n)$ instead of $O(n^2)$ operartions

Compressed Row Storage (CRS) format

(aka Compressed Sparse Row (CSR) or IA-JA etc.)

- real array `AA`, length nnz, containing all nonzero elements row by row
- integer array `JA`, length nnz, containing the column indices of the elements of `AA`
- integer array `IA`, length n+1, containing the start indizes of each row in the arrays `IA` and `JA` and `IA(n+1)=nnz+1`

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Y.Saad, Iterative Methods, p.93

- Used in most sparse matrix packages

CRS again, this time with 0-based indexing

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

```
AA: 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
JA: 0 3 0 1 3 0 2 3 4 2 3 4
IA: 0 2 4 0 11 12
```

- some package APIs provide the possibility to specify array offset
- index shift is not very expensive compared to the rest of the work

Sparse direct solvers

- Sparse direct solvers implement Gaussian elimination with different pivoting strategies
  - UMFPACK
  - Pardiso (omp + MPI parallel)
  - SuperLU
  - MUMPS (MPI parallel)
  - Pastix
- Quite efficient for 1D/2D problems
- They suffer from *fill-in*: $\Rightarrow$ huge memory usage for 3D

Getting started with solver in C++

numcxx

numcxx is a small, header only 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...
- Aims:
  - Python interface
  - Interface to triangulations
  - Sparse matrices + UMFPACK interface
  - Iterative solvers

numcxx classes

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

Expression templates

- This is a C++ technique which allows to implement expressions while avoiding introduction and copies of temporary objects

```
Vector a,b,c;
c=a+b;
```

- Has been realized in numcxx, allowing for more readable code and re-use of template based iterative solver library

Code with temporary objects

```
inline const Vector
operator+( const Vector& a, const Vector& b )
{
   Vector tmp( a.size() );
   for( std::size_t i=0; i<a.size(); ++i )
      tmp[i] = a[i] + b[i];
   return tmp;
}
```

Code with expression templates I

(K. Iglberger, "Expression templates revisited")

Expression template:

```
template< typename A, typename B >
class Sum {
public:
    Sum( const A& a, const B& b ) : a_( a ), b_( b )    {}
    std::size_t size() const { return a_.size(); }
    double operator[]( std::size_t i ) const
    { return a_[i] + b_[i]; }
private:
    const A& a_;    // Reference to the left-hand side operand
    const B& b_;    // Reference to the right-hand side operand
};
```

Overloaded + operator:

```
template< typename A, typename B >
const Sum<A,B> operator+( const A& a, const B& b )
{
   return Sum<A,B>( a, b );
}
```

Code with expression templates II

Method to copy vector data from expression:

```
class Vector
{
   public:
   // ...
   template< typename A >
   Vector& operator=( const A& expr )
   {
    for( std::size_t i=0; i<expr.size(); ++i )
        v_[i] = expr[i];
    return *this;
   }
// ...
};
```

After template instantiation, the compiler will use

```
   for( std::size_t i=0; i<a.size(); ++i )
      c[i] = a[i] + b[i];
```

Obtaining and compiling the examples

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

```
$ cp -r /net/wir/examples/part2 .
```

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

```
$ g++ --std=c++11 -I/net/wir/include -o executable source.cxx -llapack -lblas
```

(or just invoke `make`)

- Run executable

```
$ ./executable
```

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::DArray1<double>::create(n);
auto pZ=numcxx::DArray1<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...
```