∼

Recap linear algebra + direct solvers

Scientific Computing Winter 2016/2017

Lecture 6

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

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

made wit pandoc

Recap from last time

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,

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

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

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

Solution of linear systems of equations

Approaches to linear system solution

Solve $Ax = b$

- Direct methods:
    - Exact
        - up to machine precision!
    - 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
    - Cheaper in space and (possibly) time
    - Convergence guarantee is problem dependent and can be tricky

Really bad example of direct method

Cramer's rule
write $|A|$ for determinant, then

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

$O(n!)$ operations...

Gaussian elimination

- Essentially the only feasible direct solution method
- Solve $Ax = b$ with square matrix A.

Gauss 1

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

Step 1

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

Step 2

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

Gauss 2

Solve upper triangular system

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

$$\begin{aligned} -4x_3 &= -9 & & & \Rightarrow x_3 &= \frac{9}{4} \\ -4x_2 - 2x_3 &= -6 & \Rightarrow -4x_2 &= \frac{21}{2} & \Rightarrow x_2 &= -\frac{21}{8} \\ 6x_1 - 2x_2 + 2x_3 &= 2 & \Rightarrow 6x_1 &= 2 - \frac{21}{4} - \frac{18}{4} = -\frac{31}{4} & \Rightarrow x_1 &= -\frac{-31}{24} \end{aligned}$$

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 ⇒ 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

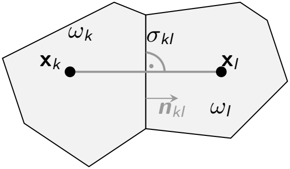
Search function $u : \Omega \to \mathbb{R}$ such that

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

- 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 $\mathbf{x}_k \in \partial\Omega$



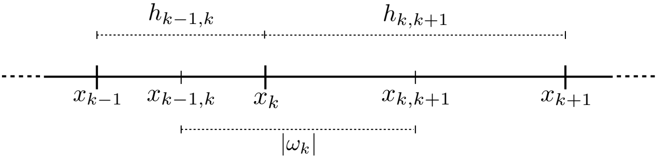
Discretization ansatz

- 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:

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

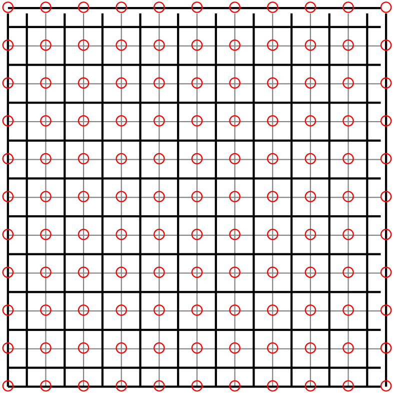
for $n - 1 \dots 1$:

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

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
- In the special case of tridiagonal matrices, progonka gives an algorithm which allows to solve the nonlinear system with $O(n)$ operations

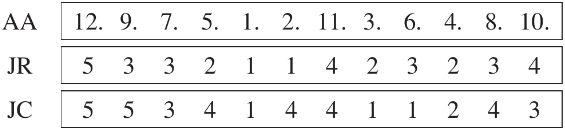
Sparse matrix questions

- What is a good 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?

Coordinate (triplet) format

- store all nonzero elements along with their row and column indices
- one real, two integer arrays, length = nnz= number of nonzero elements

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



AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

Y.Saad, Iterative Methods, p.92

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`

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

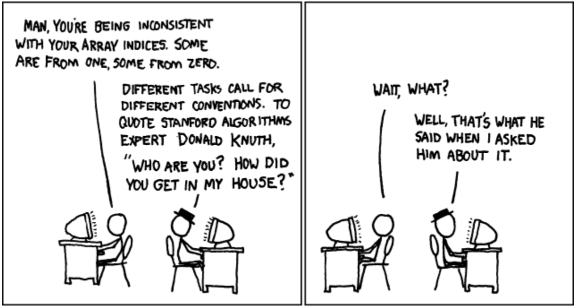
AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Y.Saad, Iterative Methods, p.93

- Used in most sparse matrix packages

The big schism

- Worse than catholics vs. protestants or shia vs. sunni. . .
- Should array indices count from zero or from one ?
- Fortran, Matlab, Julia count from one
- C/C++, python count from zero
- I am siding with the one fraction
- but I am tolerant, so for this course . . .
  - It matters when passing index arrays to sparse matrix packages


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

http://xkcd.com/1739/

CRS again

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

```
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