Theme 2: Network Models and Linear Systems

Martin Berggren

November 12, 2010

Linear Systems

- Mathematical models often gives rise to very large linear systems of equations, e.g. the network model in present theme, to be solved by the computer
- The execution time is often dominated by the solution of the linear systems!
- Today's lecture:
  - Algorithms: the basic algorithm for **Gaussian elimination** and **back substitution**
  - The basic algorithm is numerically unstable!
  - Stabilization through row pivoting
  - Computational complexity, execution time
  - LU factorization (a variation of Gaussian elimination)
  - Accuracy w.r.t. roundoff errors
  - Norms on vectors and matrices
  - Condition numbers (sensitivity w.r.t. perturbations)
- *Acknowledgement*: these notes are based on material from Stefan Pålsson, Department of Information Technology, Uppsala University

Aim

Compare with previous linear algebra course:

- The mathematics course:
  - General understanding the properties of vectors, matrices, and linear systems of equations;
  - Learn how to solve small systems of equations by hand
- Here:
  - Understand the computer adapted algorithms and their properties
  - Learn how to solve large systems by computer

Linear systems

Recall from the linear-algebra course:

- A: square n-by-n matrix
- The linear system $Ax = b$ has a unique solution for any column vector b if the matrix is **nonsingular**
- The matrix is nonsingular if and only if any of following conditions holds:
  - The rows are linearly independent
  - The columns are linearly independent
  - $Ax = 0 \Rightarrow x = 0$
  - The determinant of A is nonzero

Algorithms

- Matlab's backslash operator (\) solves the system $Ax = b$

```
>> x = A\b
```

  when A is a square matrix and b, x column vectors
- The standard algorithm: **Gaussian elimination based on LU factorization** (today's subject!)
- Matlab's \ is an "intelligent" operator: chooses different methods depending on the properties of the matrix! (See lab!)

Basic algorithm for Gaussian elimination

Gaussian elimination carried out in two steps:

- **Factorization**: *Elementary row operations* transforms the system $Ax = b$ to the form $Ux = d$, where U is an **upper triangular matrix** matrix
- **Back substitution**: Solving the system $Ux = d$

"Naive" version of factorization step (as when solving by hand):

- Indata: A, b, n (matrix order)

1. Form the total matrix $\hat{A} = [A\ b]$
2. For column $k = 1, 2, \ldots, n-1$

   Zero out the elements in column k for all rows $i > k$ (below column k) by adding the right multiple of row k to row $i = k+1, k+2, \ldots, n$

Code for naive factorization step

Indata: `A` , `b` , `n` (matrix order)
Form the total matrix `Aug = [A  b]`

```
for k = 1: n-1
   for i = k+1:n
      Lik = Aug(i,k)/Aug(k,k);
      for j = k:n+1
         Aug(i,j) = Aug(i,j) - Lik*Aug(k,j);
      end
   end
end
```

- Note that the first n columns of `Aug` is overwritten with a matrix U, and that column $n+1$ of `Aug` (containing the right-hand side `b`) is overwritten with a vector d
- This overwriting strategy saves memory, which is important when the matrix is large!
- The last for loop can in Matlab be shortened:
  `Aug(i,k:n+1) = Aug(i,k:n+1) - Lik*Aug(k,k:n+1);`

Code for back substitution

For $i = n, n-1, \ldots, 1$:

$$U_{ii}x_i + \sum_{j=i+1}^{n} U_{ij}x_j = d_j$$

Indata: `U`, `d`, `n`

```
x(n) = d(n)/U(n,n)
for i = n-1:-1:1
   x(i) = ( d(i) - U(i,i+1:n)*x(i+1:n) )/(U(i,i);
end
```

- Note that `U(i,i+1:n)*x(i+1:n)` denotes the (inner) product of the row vector `U(i,i+1:n)` with the column vector `x(i+1:n)`
- No extra matrix `U` is needed; the factorization step stores `U` in `Aug(1:n,1:n)`
- Usually `Aug(1:n,n+1)` is overwritten with `x`; that is, no separate variable is needed for `x`

The naive factorization algorithm is numerically unstable!

Exempel:

$$(A\,|\,b) = \left(\begin{array}{ccc|c} 3 & -1 & 2 & 8 \\ 1 & 0 & -1 & -1 \\ 4 & 2 & -3 & -4 \end{array}\right) \text{ with the exact solution } x = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}$$

Let L_{ik} be the factor used to zero out a_{ik}. For simplicity, assume rounding to 3 decimal digits (instead of rounding to 52 binary!)

$$fl(L_{21}) = fl(1/3) = 0.333$$
$$fl(L_{31}) = fl(4/3) = 1.33$$

$$\Rightarrow \left(\begin{array}{ccc|c} 3 & -1 & 2 & 8 \\ 0 & 0.333 & -1.67 & -3.67 \\ 0 & 3.33 & -5.67 & -14.6 \end{array}\right)$$

$$fl(L_{32}) = fl(3.33/0.333) = 10$$

The naive factorization algorithm is numerically unstable!

$$\Rightarrow \left(\begin{array}{ccc|c} 3 & -1 & 2 & 8 \\ 0 & 0.333 & -1.67 & -3.67 \\ 0 & 0 & 11.0 & 22.1 \end{array}\right) \Rightarrow \text{fl}(x) = \begin{pmatrix} 2.01 \\ -0.848 \\ 1.61 \end{pmatrix}$$

which is far from the real solution $x = (1, -1, 2)^T$

Numerical unstable algorithm: the algorithm successively amplifies the rounding errors. Causes a large error in the solution.

Remember:
`Aug(i,k:n+1) = Aug(i,k:n+1) - Lik*Aug(k,k:n+1)`

The problem: whenever $|L_{ik}| > 1$!, the multiplication will amplify the rounding error in `Aug(k,k:n+1)`

The rounding errors will successively become larger and larger

Row pivoting stabilization

- Cure: **row pivoting**
- Recall: `Lik = Aug(i,k)/Aug(k,k)`
- For each k, find a row m for which it holds that `|Aug(m,k)|` $\geq$ `|Aug(i,k)|, i = k, k+1, ..., n`
- In words: in column k, find the element of the largest magnitude on and below the diagonal
- Swap the content of row m and k
- Then `|Aug(k,k)|` $\geq$ `|Aug(i,k)|`, so $|L_{ik}| \leq 1$, which prevents amplification of the error when multiplying with L_{ik}

Row pivoting

Earlier example

$$(A\,|\,b) = \left(\begin{array}{ccc|c} 3 & -1 & 2 & 8 \\ 1 & 0 & -1 & -1 \\ 4 & 2 & -3 & -4 \end{array}\right) \quad \text{Exchange rows 1 and 3}$$

$$\left(\begin{array}{ccc|c} 4 & 2 & -3 & -4 \\ 1 & 0 & -1 & -1 \\ 3 & -1 & 2 & 8 \end{array}\right)$$

$$fl(L_{21}) = fl(1/4) = 0.25$$
$$fl(L_{31}) = fl(3/4) = 0.75$$

$$\Rightarrow \left(\begin{array}{ccc|c} 4 & 2 & -3 & -4 \\ 0 & -0.5 & -0.25 & 0 \\ 0 & -2.5 & 4.25 & 11 \end{array}\right) \quad \text{Exchange rows 2 and 3}$$

Row pivoting

$$\Rightarrow \left(\begin{array}{ccc|c} 4 & 2 & -3 & -4 \\ 0 & -2.5 & 4.25 & 11 \\ 0 & -0.5 & -0.25 & 0 \end{array}\right)$$

$$fl(L_{32}) = fl(-0.5/-2.5) = 0.2$$

$$\Rightarrow \left(\begin{array}{ccc|c} 4 & 2 & -3 & -4 \\ 0 & -2.5 & 4.25 & 11 \\ 0 & 0 & -1.1 & -2.2 \end{array}\right) \quad \Rightarrow \quad \begin{array}{l} x_1 = 1 \\ x_2 = -1 \\ x_3 = 2 \end{array}$$

Execution time

- It can take very long time to perform Gaussian elimination on large matrices
- A central question: how does the number of floating point operations depend on the order of the matrix?

The number of floating point operations

- Consider the second for-loop in the factorization step, which is performed for $k = 1, \ldots, n-1$

`for i = k+1:n`	executed $n-k$ times
`   Lik = Aug(i,k)/Aug(k,k)`	1 op
`   for j = k:n+1`	executed $n-k+2$ times
`      Aug(i,j) = Aug(i,j) - Lik*Aug(k,j)`	2 op

  Number of floating point operations (flops):
  $(n-k)\big[1 + (n-k+2)2\big] \approx 2(n-k)^2$ (plus linear terms in k and n)
- Summing for all k:

$$\sum_{k=1}^{n-1} 2(n-k)^2 = \frac{2}{3}n^3 + O(n^2) \qquad \text{(Lemma 8.3.1 in book)}$$

- Conclusion: the factorization step of Gaussian elimination, applied to an n-by-n system, requires $\frac{2}{3}n^3 + O(n^2)$ flops
- A similar analysis: The backward substitution step requires n^2 flops

Execution time

- The analysis says that that Gaussian elimination is of **complexity** n^3 (factorization) and n^2 (back substitution)
- What does the flop count mean in actual times?
- Assume $t_f = 10^{-9}$ s/flop; a realistic number

	factorization	back substitution
n	$\frac{2}{3}n^3\,t_f$	$n^2\,t_f$
10^3	0.67 s	10^{-3} s
10^6	0.67×10^9 s $\approx$ 21 years	10^3 s $\approx$ 17 min

Execution time

How big system can be solved in one hour if the computer performs at 1 Gflop/s? (Gflop = 10^9 flops)

Answer: $\frac{2}{3}n^3 \cdot 10^{-9} = 3\,600 \Rightarrow n \approx 18\,000$

How big system can be solved in a minute?

Answer: $\frac{2}{3}n^3 \cdot 10^{-9} = 60 \Rightarrow n \approx 4\,500$

Limitation in memory access can cause additional significant delays!

The need for efficient algorithms

- The n^3 complexity limits the usefulness of Gaussian elimination for very large matrices
- Alternative:
  - Exploit any particular *structure* of the matrix, if possible. There are versions of Gaussian elimination for banded or very sparse matrices.
  - A completely different type of algorithms, **iterative methods**, becomes necessary for very large, sparse matrices.
    - These type of matrices are often obtained when the matrix is obtained from the discretization of **partial differential equations**
    - Matrix order up to $n = 10^8$ can appear for such problems!
    - Such problems require large parallel computers (e.g. Akka in Umeå) and specially developed algorithms.

LU factorization

- Common case: a sequence of linear equation using the same matrix but with different right-hand sides:
$$Ax^{(k)} = b^{(k)}, \quad k = 1, \dots, m$$
- Idea: factor A only once:
  - Store U
  - Store the factors L_{ik} in a lower triangular matrix L (that has 1s on the diagonal)
  - Store information about the pivoting (row swaps) in a matrix P
- Called **LU factorization** of A
- Can show that $LU = PA$ (Theorem 8.6.1 in book)

Theorem

A square matrix A is nonsingular if and only if there exists a permutation matrix P, an lower triangular matrix L with ones on the diagonal, and an upper triangular matrix U such that $PA = LU$

LU factorization

- Given A, compute L, U, P, so that $LU = PA$
$$Ax = b \Rightarrow PAx = Pb \Rightarrow LUx = Pb \quad \text{[factorization, } O(n^3) \text{ flops]}$$
- For each right-hand side $b^{(k)}$, do
  - Solve problem
$$Ld = Pb \qquad \text{[forward substitution, } O(n^2)\text{]}$$
    to determine d;
  - Solve problem
$$Ux = d \qquad \text{[back substitution, } O(n^2)\text{]}$$
    to determine the solution $x^{(k)}$

LU factorization

What is the benefit of LU factorization compared to the “usual” Gaussian elimination?

- **Inefficient strategy:** SSolve each system with `xi=A\bi`
  - A will be factored from scratch for each new right-hand side `bi`!
  - Number of flops: $m(\frac{2}{3}n^3 + n^2)$ (m systems that are factored and back substituted)
- **Efficient strategy:** LU factorize A and solve
  - `d = L\b`
  - `x = U\d`
  - Number of flops: $\frac{2}{3}n^3 + 2mn^2$ (factored only once, m forward- and back substitutions)

LU factorization in Matlab

```
>> A = [3  -1  2; 1  0  -1; 4  2  -3];
>> b = [8; -1; -4];
>> [L, U, P] = lu(A);
L =
    1.0000         0         0
    0.7500    1.0000         0
    0.2500    0.2000    1.0000
U =
    4.0000    2.0000   -3.0000
    0   -2.5000    4.2500
    0         0   -1.1000
P =
    0     0     1
    1     0     0
    0     1     0
```


LU factorization in Matlab

Checking:

```
>> P*A
ans =
     4     2    -3
     3    -1     2
     1     0    -1
>> L*U
ans =
     4     2    -3
     3    -1     2
     1     0    -1
```

Solution with LU factorization

```
>> d = L\(P*b)
d =
   -4.0000
   11.0000
   -2.2000
>> x = U\d
x =
     1
    -1
     2
```

Note: The backslash operator is “smart”; when the matrices are upper or lower triangular, the algorithms for forward and backward substitutions are used instead of full Gaussian elimination.

LU factorization in Matlab

Testing whether backslash is smart enough to employ LU factorization!

```
>> n = 2000;
>> A = rand(n,n);
>> B40 = rand(n, 40); b1 = rand(n,1);
>> tic; X = A\B40; toc
Elapsed time is 1.883686 seconds.
>> tic; x = A\b1; toc
Elapsed time is 1.481570 seconds.
```

- Matrix $B_{40} = [b_1 b_2 \dots b_{40}]$ stores 40 right hand sides
- Matrix $X = [x_1 x_2 \dots x_{40}]$ contains the solutions to the linear systems for the right-hand sides in B_{40}
- 40 systems with the same matrix is solved almost as quickly as only 1 system!
- Indicates that Matlab indeed uses the LU factorization!

LU factorization: example

Mathematical object

$$\begin{pmatrix} 2 & 2 & -2 \\ -4 & -2 & 2 \\ -2 & 3 & 9 \end{pmatrix}$$

Data structures

```
 2   2  -2        1
-4  -2   2        2
-2   3   9        3
```

matrix permutation vector

Row swap:

$$\begin{pmatrix} -4 & -2 & 2 \\ 2 & 2 & -2 \\ -2 & 3 & 9 \end{pmatrix}$$

```
-4  -2   2        2
 2   2  -2        1
-2   3   9        3
```

matrix permutation vector

LU factorization: example

Elimination, column 1, with factors $L_{21} = -1/2$, $L_{31} = 1/2$:

$$\begin{pmatrix} -4 & -2 & 2 \\ 0 & 1 & -1 \\ 0 & 4 & 8 \end{pmatrix}$$

```
-4    -2   2      2
-1/2   1  -1      1
 1/2   4   8      3
```

matrix permutation vektor

Row swap:

$$\begin{pmatrix} -4 & -2 & 2 \\ 0 & 4 & 8 \\ 0 & 1 & -1 \end{pmatrix}$$

```
-4    -2   2      2
 1/2   4   8      3
-1/2   1  -1      1
```

matrix permutation vector

LU factorization: example

Elimination, column 2, with factor $L_{32} = 1/4$:

$$\begin{pmatrix} -4 & -2 & -2 \\ 0 & 4 & 8 \\ 0 & 0 & -3 \end{pmatrix}$$

```
-4    -2    2      2
 1/2   4    8      3
-1/2  1/4  -3      1
```

matrix permutation vector

Done! Matrix interpretation of the data structures:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/2 & 1/4 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} -4 & -2 & 2 \\ 0 & 4 & 8 \\ 0 & 0 & -3 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

LU factorization: example

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/2 & 1/4 & 1 \end{pmatrix} \begin{pmatrix} -4 & -2 & 2 \\ 0 & 4 & 8 \\ 0 & 0 & -3 \end{pmatrix} = \begin{pmatrix} -4 & -2 & 2 \\ -2 & 3 & 9 \\ 2 & 2 & -2 \end{pmatrix}$$

$$PA = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 & 2 & -2 \\ -4 & -2 & 2 \\ -2 & 3 & 9 \end{pmatrix} = \begin{pmatrix} -4 & -2 & 2 \\ -2 & 3 & 9 \\ 2 & 2 & -2 \end{pmatrix}$$

- Conclusion: $LU = PA$
- No extra storage: L (except the diagonal) and U are stored in the memory location of A
- The pivoting information is stored in the integer permutation vector (a full matrix P with mostly zeros would be a waste of storage)

Accuracy

$$Ax = b$$

- Exact solution x (usually unknown)
- Rounding errors accumulated during Gaussian elimination yields computed solution $\tilde{x}$
- How accurate is the computed solution?
- "Natural" test: check whether the equations are satisfied!
- The **residual**

$$b - A\tilde{x}$$

should be close to zeros!

Residual and accuracy: example

$$A = \begin{pmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{pmatrix}, \quad b = \begin{pmatrix} 0.8642 \\ 0.1440 \end{pmatrix}$$

```
>> xe = single(A)\single(b)
xe =
    1.3332
   -1.0000
>> xd = A\b
xd =
    2.0000
   -2.0000
```

`xe` computed with A and b in single precision

`xd` computed with A and b in the "usual" double precision

Residual and accuracy: example

```
>> res = b - A*xe
res =
   1.0e-06 *
   -0.1152
   -0.0087
```

The residual is small: "exact up to roundoff", i.e. around ϵ_M in single precision

```
>> xd - xe
ans =
    0.6668
   -1.0000
```

But the error is large!

```
>> cond(A)
ans =
   2.4973e+08
```

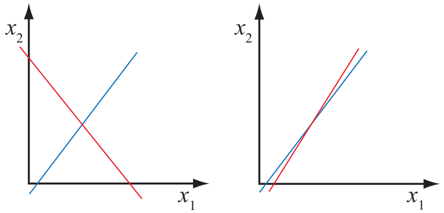
Note that the so-called **condition number** is high!

Condition number and residual

- Conclusion: the size of the residual is **not** a reliable measure of accuracy!
- Why?
- The example problem is **ill conditioned** (=sensible to perturbations)
- We need a better way of estimating the error than measuring the residual!

Well conditioned and ill conditioned problems

- The concepts of **well conditioned** and **ill conditioned** problems can be illustrated graphically for linear systems in two unknowns:



- The two lines depict which x_1 and x_2 values that satisfy the two equations
- The solution to the system of equations is at the intersection between the lines
- When the equations almost describe the same line, the lines are close together even far away from the intersection point. Then the residual is small even far away from the solution.

Norms

- In order to measure errors, we need to be able to measure "sizes" of vectors and matrices in a way that generalizes the concept of **absolute number** for real numbers
- We will use **norms**, using the notation $\|x\|$ for the norm of a vector x
- There are both **vector norms** and **matrix norms**

Vector norms

The most common vector norms for $x = (x_1, \dots, x_n)^T$:

- 2-norm, Euclidian norm:

$$\|x\|_2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$$

- 1-norm

$$\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$

- ∞-norm, max norm

$$\|x\|_\infty = \max(|x_1|, |x_2|, \dots, |x_n|)$$

Vector norms

- Why are there different norms?
- One particular norm is sometimes more appropriate than another:
  - The 2 norm gives the direct route ("fågelvägen")
  - The 1 norm gives the distance for the shortest distance **along the streets**!

Matrix norms

- Most commonly defined with the help of a given vector norm:

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

- Yields the maximal "amplification factor" that the matrix causes when it is applied to a vector
- From the definition follows that for each $x \neq 0$,

$$\frac{\|Ax\|}{\|x\|} \leq \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \|A\| \tag{1}$$

that is,

$$\|Ax\| \leq \|A\|\|x\| \qquad \forall x$$

- Simpler formulas than the definition itself can be derived for the 1, ∞, and 2 norms

Matrix norms

- It can be shown that:

$$\|A\|_1 = \max_j \left(\sum_i |A_{ij}| \right) \quad \text{(the largest 1 norm of the column vectors)}$$

$$\|A\|_\infty = \max_i \left(\sum_j |A_{ij}| \right) \quad \text{(the largest 1 norm of the row vectors)}$$

$$\|A\|_2 = \sqrt{\max_i(\lambda_i(A^T A))} \quad \text{(the square root of the largest eigenvalue of } A^T A\text{)}$$

- The 1 and ∞ norms are much simpler and faster to compute!

Norms in Matlab

`norm(x)`	the 2 norm of the row or column vector `x`
`norm(A)`	the 2 norm of the matrix `A`
`norm(A,1)`	the 1 norm of matrix `A`
`norm(A,Inf)`	the ∞ norm of matrix `A`

Errors and condition number

Let b a right-hand side, $\tilde{b}$ a disturbed RHS (from rounding e.g.), and $x \neq 0$, $\tilde{x}$ corresponding solutions of the linear system

$$Ax = b, \quad A\tilde{x} = \tilde{b} \quad \Rightarrow A(x - \tilde{x}) = b - \tilde{b} \quad \Leftrightarrow$$
$$x - \tilde{x} = A^{-1}(b - \tilde{b}) \quad \Rightarrow \|x - \tilde{x}\| = \|A^{-1}(b - \tilde{b})\| \leq \|A^{-1}\|\|b - \tilde{b}\|$$

The last inequality uses property (1). Dividing with $\|x\|$ yields

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \frac{\|A^{-1}\|}{\|x\|}\|b - \tilde{b}\| \tag{2}$$

Since $\|b\| = \|Ax\| \leq \|A\|\|x\|$ (again using (1)), we have

$$\frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|} \tag{3}$$

By substituting (3) into (2), we obtain following bound for the relative error in the solution:

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \|A^{-1}\|\|A\|\frac{\|b - \tilde{b}\|}{\|b\|}$$

Errors and condition number

- We have thus proved following estimate of the relative error
$$\frac{\|x-\tilde{x}\|}{\|x\|} \le \kappa(A)\frac{\|b-\tilde{b}\|}{\|b\|}$$
where $\kappa(A) = \|A^{-1}\|\|A\|$ is the **condition number** of matrix A
- In words: the relative error in x is bounded by the condition number times the relative error in the right hand side
- Errors in b can thus be amplified with a factor $\kappa(A)$ when solving the linear system!
- Note that we have not made any assumptions on the nature of the disturbances or on the method to solve the system

Errors and condition number

- Note that the condition number of a matrix depends on the choice of matrix norm!
- For our example

$$A = \begin{pmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{pmatrix} \qquad \begin{aligned} \kappa_2(A) &= 2.5 \times 10^8 \\ \kappa_1(A) &= 3.3 \times 10^8 \\ \kappa_\infty(A) &= 3.3 \times 10^8 \end{aligned}$$

- The relative error in b is in the best of cases bounded by machine epsilon, i.e. about 10^{-16} (in single precision about 10^{-8})
- Thus, all accuracy can be lost in single precision (the relative error 1, i.e. 100 %), in double precision "half" of the accuracy

Condition numbers

- Mathematically, a matrix is either singular or not singular. For computational purposes, it is useful to talk also of "near singularity"
- A high condition number (an ill conditioned problem) indicates that the matrix "almost" is singular
- A high condition number is a property of the underlying linear system!
- The condition number and the potential for sensitivity of disturbances cannot be changed by choice of solution algorithm for the linear system!

Condition numbers

- Condition numbers act like a warning sign: the error **may** become large when solving linear systems
- The error estimate involving the condition number consider the worst case scenario: it may happen that the error will not become as large as the estimate indicates
- It holds that
$$\kappa(A) = \|A^{-1}\|\|A\| \ge \|A^{-1}A\| = \|I\| = 1$$
- The condition number is thus in the best case 1, which means no amplification of the error in the right-hand side