中国科学院大学

2026

数值代数

Numerical Algebra

第三章、矩阵计算基础

Basics of Matrix Computations

3

Chensong Zhang, AMSS

https://lsec.cc.ac.cn/~zhangcs

课程内容

第一章、绪论与应用背景

第二章、线性代数基础

第三章、矩阵计算基础

第四章、线性方程组的直接方法

第五章、线性方程组的迭代方法

第六章、最小二乘问题的数值方法

第七章、Krylov子空间方法

第八章、特征值问题的数值方法

本章主要内容

第三章、矩阵计算基础

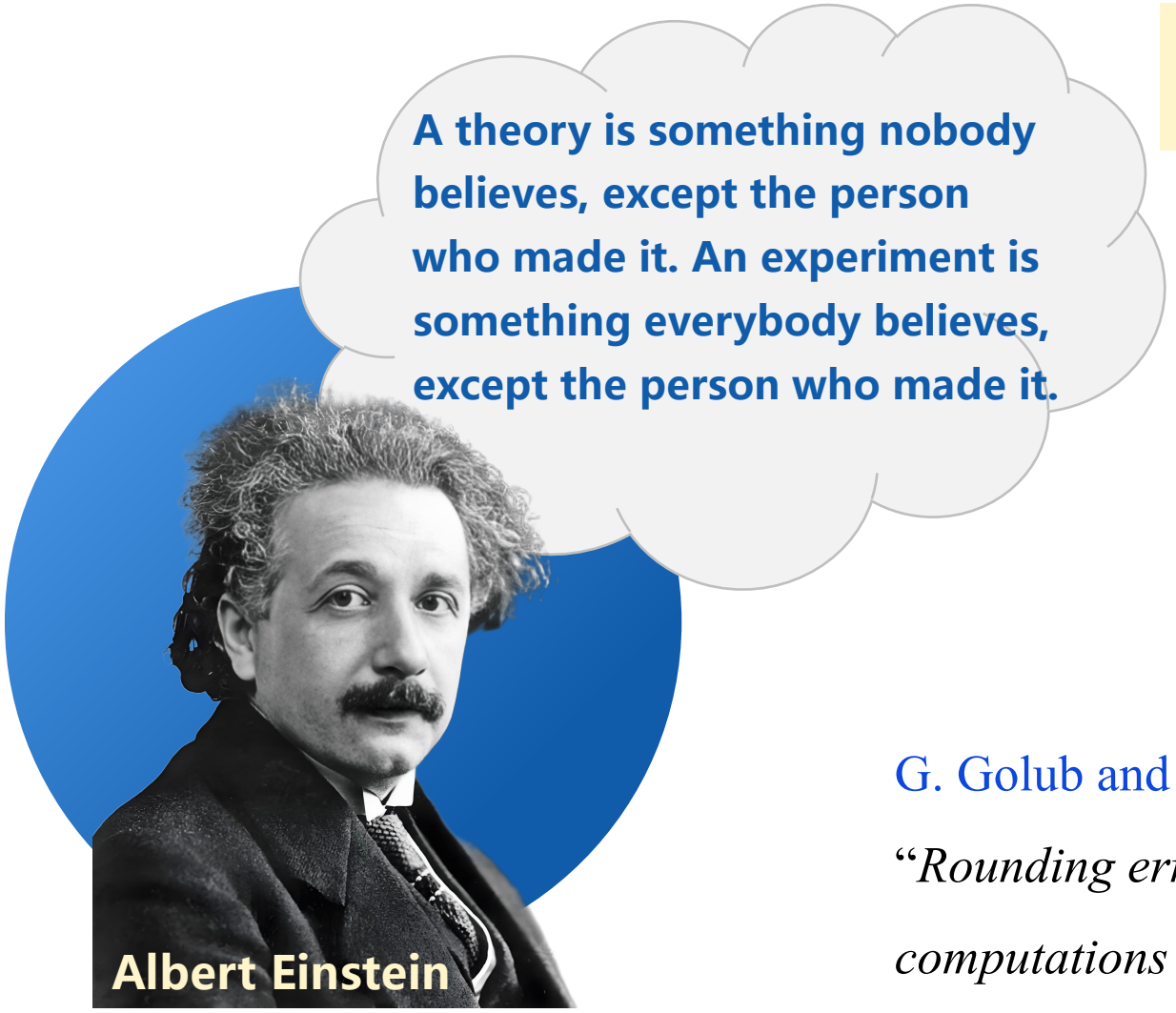
本章聚焦于数值线性代数中的基本问题与工具，重点讲解浮点数表示与误差分析、病态问题的识别与条件数估计、算法的数值稳定性评估。同时，本章将介绍矩阵计算方法中的几个核心工具，如Householder变换、Givens旋转与QR分解，为学生后续学习线性方程组、最小二乘、特征值问题的常用数值方法提供必要的基础。

浮点数与浮点误差

Floating-point numbers and arithmetic error

/3.1

Sources of Error in Simulation



$$u(x) = U_h(x) + \mathcal{E}_{\text{dis}} + \mathcal{E}_{\text{alg}} + \mathcal{E}_{\text{round}}$$

Due to floating-point computations

- **Rounding error**: Error caused by storing an exact value with finite precision in FP storage formats or doing FP computations
- Balance b/w **Accuracy** vs **Cost**

G. Golub and C. F. van Loan, Matrix Computations, 4th Ed, 2013:

"*Rounding errors are part of what makes the field of matrix computations so challenging*"

Why My Code Gives Different Output

- When?
  - ✓ From one run to the other
  - ✓ From one compiler to another
  - ✓ From one option to another
  - ✓ From one computer to another

- How?
  - ✓ Compiler
  - ✓ Compiler options
  - ✓ Libraries linked
  - ✓ Multi-threading

Reproducibility

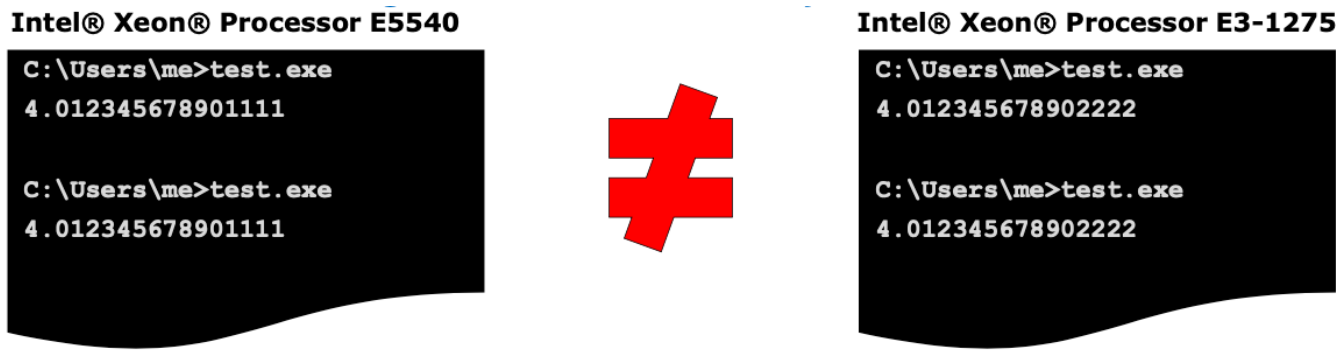
- Produce the same result at any time
- Expect deterministic behaviors
- Give up some possible optimizations

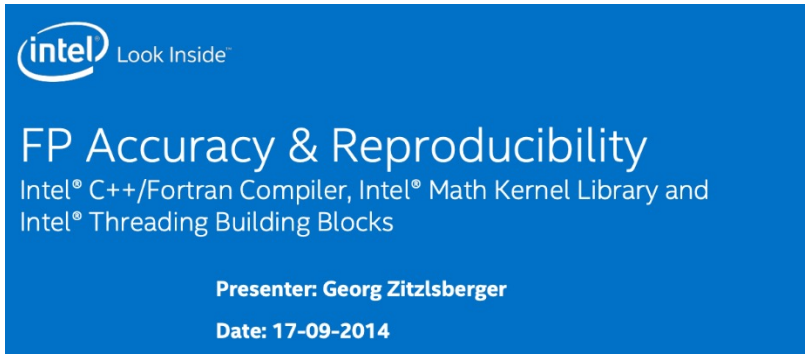
VS

Performance

- Produce a result as fast as possible
- Do whatever is good for speed
- Usually, the results are close

Wake-Up Calls





FP Model	Year = 0	Year = 1	Year = 5	Year = 10
Precise	3525.136841724400	3445.016272216141	3034.082337708471	2608.065832970213
Strict	3525.136841724400	3445.016272216141	3034.082337708471	2608.065832970213
Fast	**3525.136841724400**	**3445.0**50433013021	**3034**.789033440240	**260**9.016046486710

Pressure (psia) at one grid cell for SPE3 benchmark (compiled using Intel® oneAPI 2022)

Q: Can you trust what was produced by your code? Or even by anyone's code?

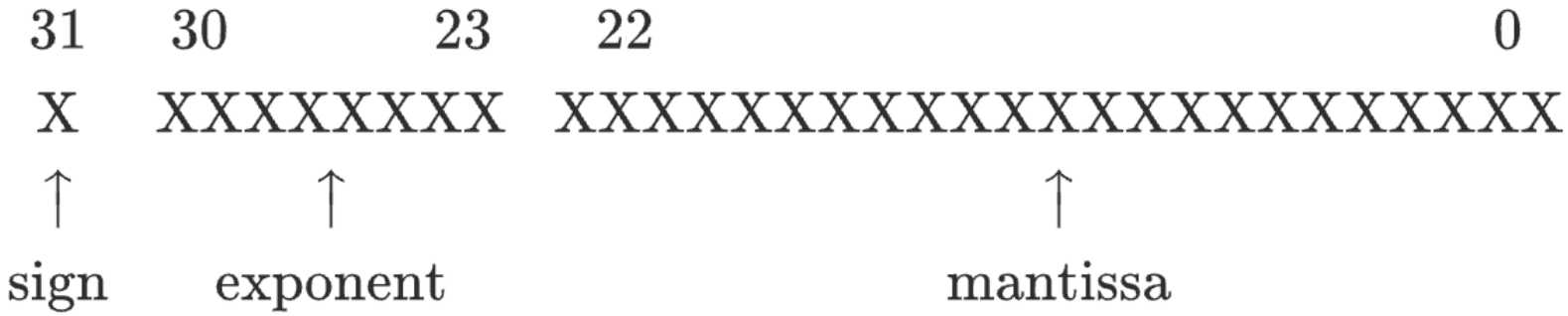
Floating-Point Number Storage

Standard Committee	C/MSC - Microprocessor Standards Committee >	
Status	Active Standard	
PAR Approval	2015-09-03 >	
Superseding	754-2008 >	
Board Approval	2019-06-13	
History	**Published:**	2019-07-22

WORKING GROUP DETAILS

Society	IEEE Computer Society >
Standard Committee	C/MSC - Microprocessor Standards Committee >
Working Group	754_WG - Working Group for Floating-Point Arithmetic
IEEE Program Manager	Malia Zaman Contact Malia Zaman >
Working Group Chair	Leonard Tsai

IEEE 754 single-precision floating-point number



Fraction field, with hidden leading 1 (normalized)

$$\text{Number} = (-1)^s 2^{e-127} 1 \cdot f,$$

where $s = 0$ for positive numbers, 1 for negative numbers,

e = exponent (between 0 and 255), and

f = mantissa.

Normal SP Floating-Point Numbers

Reserved for INF/NAN

1.f * 2 **0111,1111**

Reserved for 0.0 (all bits are 0)

Exponent	Decimal Value	Shift-127 Value
1111,1111	255	**128**
0111,1111	127	0
0000,0001	1	**-126**
0000,0000	0	N/A

Overflow
2^128 ≈
3.40E38

Min Normal
2^(-126) ≈
1.18E-38

Max number
1.1……1 * 2 **1111,1110**

1.f * 2 **0000,0000** ?
No! Need special treatment …

Exponent Value	Mantissa	Purpose
1-254	anything	normal
255	**zero**	**INF**
255	nonzero	NaN
0	**nonzero**	**subnormal**

Range:
[-126,127]

Q: What do we do if a number falls out of this range?

Subnormal SP Floating-Point Numbers

- Subnormal numbers (or denormals) are useful in filling gaps of floating-point system near zero
- Numbers that are smaller than the smallest normal positive number (consider positive part only)

Subnormal numbers have all zero bits for the exponent

$$(-1)^s \times 2^{0-127+1} \times 0.f$$

Q: Why +1? Shift by -126 instead of -127!

Reminder: the exponent of the smallest normal

Cannot add leading-1 anymore: fewer bits ➔ less accurate

- The largest possible subnormal number is $0.111\ldots11 \times 2^{-126} < 1.0\ldots0 \times 2^{-126} \approx 1.18E\text{-}38$
- The smallest possible subnormal number is $0.000\ldots01 \times 2^{-126} \approx 1.40E\text{-}45$
- Some compilers support "**flush subnormals to zero**" (FTZ)
- **Caution**: Floating-point operations in the subnormal region can be very slow!

Range of FP Representation

	signif.	exp.	u	$x_{\min}^{(s)}$	$x_{\min}$	$x_{\max}$
bfloat16	8	8	3.91×10^{-3}	9.18×10^{-41}	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}
fp128	113	15	9.63×10^{-35}	6.48×10^{-4966}	3.36×10^{-4932}	1.19×10^{4932}

t = Number of bits in the fractional field + 1

- fp16 is likely to **overflow** / **underflow**; bfloat16 has low precision.
- We can ignore the small numbers (underflow); but the real problem is: how to handle the large numbers? For low-precision FP systems, need to scale the problem first.
- IEEE-754 rounding rules: round to nearest, ties to even.

Precision of Floating-Point System

Precision describes the "**accuracy**" or "**resolution**" of numbers that a floating-point system is able to represent. It is typically expressed in terms of the number of significant digits, namely t.

- **Machine epsilon** is the gap between 1 and the next larger representable floating-point number

$$1.0 + \varepsilon > 1.0 \quad \blacktriangleright \quad \varepsilon_{\text{double}} := 2^{-52} \approx 2.2204 \times 10^{-16}, \quad \varepsilon_{\text{single}} := 2^{-23} \approx 1.1921 \times 10^{-7}$$

- **Unit round-off** (often denoted as u) refers to the upper bound on the relative error caused by rounding when performing a basic floating-point operation:

  - Floating-point storage: $fl(x) = x(1+\delta), \quad |\delta| \le u$

  - Floating-point arithmetic: $\dfrac{|fl(x \text{ op } y) - (x \text{ op } y)|}{|x \text{ op } y|} \le u$

Rounding Errors

- **Q: How big is the unit round-off / machine epsilon in a floating-point system?**
- Let β be the base (2 for binary numbers) and let t be the number of digits (bits) in the significand.

$$\Rightarrow \quad u = \frac{1}{2}\beta^{1-t} = 2^{-t} \quad \text{for binary numbers}$$

$$\Rightarrow \quad u = \varepsilon/2 \quad \Rightarrow \quad \varepsilon = \beta^{1-t}$$

- **Some Remarks:**
  - There is usually **no essential meaning** to figure out the exact constant in front of machine epsilon and/or unit round-off.
  - In practice and in many publications, the terms "machine epsilon ε" and "unit round-off u" are often used interchangeably.

Examples of FP Arithmetic

- Note: Associative property of addition does **NOT** hold in FP!

$$\beta = 2,\ t = 24,\ u = 2^{-24}$$

$$\Rightarrow \quad fl\left(fl(1+2^{-25})-1\right) = 0 \qquad \text{and} \qquad fl\left(fl(1-1)+2^{-25}\right) = 2^{-25}$$

- **Catastrophic cancellation** in FP:

$$\beta = 10,\ t = 4 \qquad \text{Round half up (四舍五入)}$$

$$\Rightarrow \begin{cases} fl(1.2345) = 1.235, \quad fl(1.2343) = 1.234 \\ fl(1.2345) - fl(1.2343) = 0.001 \end{cases}$$

	Relative error
Storage:	$< 10^{-3}$
Computation:	**400%**

- Be careful in numerical methods, for example: $f'(x) \approx \dfrac{f(x+h)-f(x)}{h}$

Comparison: Double vs Single Precisions

Compute: $f(x) = \dfrac{1-\cos(x)}{x^2}$

```
>>> x = 5e-4
>>> print((1 - np.cos(x))/np.pow(x,2))
0.49999998941174795
>>> xs = np.float32(5e-4)
>>> print((1 - np.cos(xs))/np.pow(xs,2))
0.4768371

>>> x = 1e-4
>>> print((1 - np.cos(x))/np.pow(x,2))
0.4999999969612645
>>> xs = np.float32(1e-4)
>>> print((1 - np.cos(xs))/np.pow(xs,2))
0.0
```

Approximate: $\cos'(x) \approx \dfrac{\cos(x+h)-\cos(x)}{h}$

```
>>> h = 1e-6
>>> print((np.cos(np.pi/2+h)-np.cos(np.pi/2))/h)
-0.9999999999175667
>>> h = 1e-8
>>> print((np.cos(np.pi/2+h)-np.cos(np.pi/2))/h)
-0.999999993922529

>>> hs = np.float32(1e-6)
>>> print((np.cos(np.pi/2+hs)-np.cos(np.pi/2))/hs)
-0.9973857712212236
>>> hs = np.float32(1e-8)
>>> print((np.cos(np.pi/2+hs)-np.cos(np.pi/2))/hs)
-4.371138861362497
```

Effects of Floating-Point Arithmetic

- We are doing floating-point calculations ➔ Nothing is "exact"

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \le u$$

- Compiler options (FP model) might be doing something for you, under the hood!
  - Reassociation: (a + b) + c ➔ a + (b + c)
  - Division to multiplication: a / b ➔ a * (1 / b)
  - Zero folding: a + 0 ➔ a, a * 0 ➔ 0 (Attention: But "a" might be **INF** or **NaN**)
  - Math functions: sin(a), log(a), … (No international standard on these functions)
  - Handel subnormal FP numbers and underflow (flush to zero or not)
  - Utilize fused multiply-add (FMA) units or not
  - Different code paths (SIMD or non-SIMD, AVX2 v.s. SSE)
- **Takeaway Message:** FP error analysis should be better done ahead of time!

Error Analysis for Inner Product

Define partial sum: $s_p := fl\left(\sum_{k=1}^{p} x_k y_k\right), \quad p = 1, \ldots, n$

$$s_1 = x_1 y_1 (1 + \delta_1), \quad |\delta_1| \le u$$

$$\vdots$$

$$s_p = fl\big(s_{p-1} + fl(x_p y_p)\big) = \big(s_{p-1} + x_p y_p (1 + \delta_p)\big)(1 + \varepsilon_p)$$

$$\vdots \qquad\qquad \varepsilon_1 = 0, \quad |\delta_p| \le u, \ |\varepsilon_p| \le u, \quad p = 2, \ldots, n$$

$$s_p = \sum_{k=1}^{p} x_k y_k (1 + \theta_{k,p}), \qquad 1 + \theta_{k,p} = (1 + \delta_k) \prod_{j=k}^{p} (1 + \varepsilon_j)$$

$\theta_{k,p}$ is the accumulated perturbation of the k-th product in s_p

Taking $p = n$ ⇒ $fl(\boldsymbol{x}^T \boldsymbol{y}) = s_n = \sum_{k=1}^{n} x_k y_k (1 + \theta_{k,n})$

Higham's Lemma

$$\prod_{i=1}^{n}(1+\delta_i) =: 1+\theta_n$$

If $|\delta_i| \le u,\ i = 1:n \ \text{ and } \ nu < 1$

$$\Rightarrow \quad |\theta_n| \le \frac{nu}{1-nu} =: \gamma_n$$

and

Another useful and simple bound can be written roughly as

$$|\theta_n| \le nu + O(u^2)$$

if we ignore the dependence on n for higher order terms.

Error bounds for inner product: **Exercise**

$$\left|fl(\boldsymbol{x}^T\boldsymbol{y}) - \boldsymbol{x}^T\boldsymbol{y}\right| \le \sum_{k=1}^{n} |x_k y_k|\,|\theta_{k,n}| \le \gamma_n |\boldsymbol{x}|^T|\boldsymbol{y}| \Rightarrow \begin{cases} \left|fl(\boldsymbol{x}^T\boldsymbol{y}) - \boldsymbol{x}^T\boldsymbol{y}\right| \le nu|\boldsymbol{x}|^T|\boldsymbol{y}| + O(u^2) \\ \left|fl(\boldsymbol{x}^T\boldsymbol{y}) - \boldsymbol{x}^T\boldsymbol{y}\right| \le Cnu|\boldsymbol{x}|^T|\boldsymbol{y}| \end{cases}$$

Running Error Analysis

- Standard inner product

$s = 0$

for $i = 1 : n$

$s = s + x_i y_i$

end

- Inner product with running error

$s = \mu = 0$

for $i = 1 : n$

$z = x_i y_i$

$s = s + z$

$\mu = \mu + |s| + |z|$

end

$e = u\,\mu$

$z_i = x_i y_i, \quad s_i = \sum_{j=1}^{i} z_j$

$\hat{z}_i = fl(x_i y_i), \quad \hat{s}_i = fl(\hat{s}_{i-1} + \hat{z}_i)$

$\hat{s}_{i-1} + \hat{z}_i = \hat{s}_i(1 + \varepsilon_i), \quad |\varepsilon_i| \le u$

$\rho_i := \hat{s}_i - (\hat{s}_{i-1} + \hat{z}_i) = -\varepsilon_i \hat{s}_i, \quad |\rho_i| \le u|\hat{s}_i|$

$\pi_i := \hat{z}_i - z_i = -\eta_i\, \hat{z}_i, \quad |\pi_i| \le u|\hat{z}_i|$

$e_i = e_{i-1} + \rho_i + \pi_i$

$|e_i| \le |e_{i-1}| + u|\hat{s}_i| + u|\hat{z}_i|$

$|e_i| \le u\,\mu_i, \quad \mu_i = \mu_{i-1} + |\hat{s}_i| + |\hat{z}_i|$

$|e_n| \le u\,\mu_n$

Blocked Algorithms

- Simple summation (forward error analysis)

$$\gamma_n = nu + O(u^2)$$

$$s = x^T y = \sum_{i=1}^{n} x_i\, y_i \quad \Longrightarrow \quad |s - \hat{s}| \le n\,u\,|x|^T\,|y|$$

Algorithm 4: Blocked summation method

```
1  %% Given two vectors x, y ∈ R^n, n = k × b, and s = 0.0;
2  for i = 1 : k
3      iend ← i × b;
4      istart ← iend − b + 1;
5      s_i ← x(istart:iend)^T y(istart:iend);
6      s ← s + s_i;
7  end
```

Note: The algorithm does NOT reduce the number of calculations; but it reduces error accumulation.

- Blocked summation improves accuracy $|s - \hat{s}| \le \left(\frac{n}{k} + k\right) u\,|x|^T|y|$

Further Improvement on Inner Product

- Pairwise or blocked summation does not reduce the total number of additions. The idea is to reduce the length of each **dependent accumulation chain** and hence the error-growth depth.
- Using hardware: Fused Multiply-Add (FMA)

No FMA: $fl(x+yz) = \big(x + yz(1+\delta_1)\big)(1+\delta_2), \quad |\delta_1|, |\delta_2| \le u$

With FMA: $fl(x+yz) = (x+yz)(1+\delta), \quad |\delta| \le u$

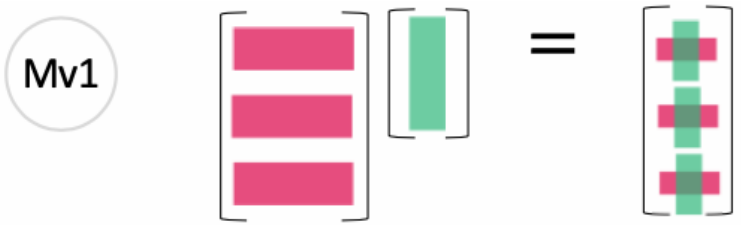
$$\begin{cases} \text{Not only faster in computation} \\ \text{Local first-order bound changes from approximately } 2u \text{ to } u \end{cases}$$

- A more advanced algorithm (FABsum) can give error bound independent of n to first order.

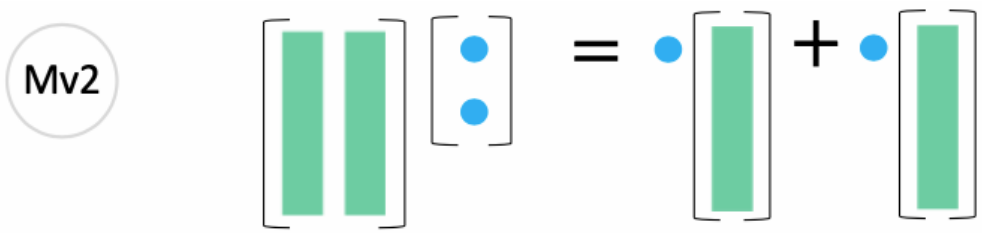
Ref: Blanchard, Pierre, Nicholas John Higham and Théo Mary. "A Class of Fast and Accurate Summation Algorithms." SIAM J. Sci. Comput. 42 (2020): A1541-A1557.

Matrix-Vector Multiplication

- Matrix-vector multiplication ➔ vector-vector multiplication



The row vectors of A are multiplied by a vector $\boldsymbol{x}$ and become the three dot-product elements of $A\boldsymbol{x}$.



The product $A\boldsymbol{x}$ is a linear combination of the column vectors of A.

$$A\boldsymbol{x} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (x_1 + 2x_2) \\ (3x_1 + 4x_2) \\ (5x_1 + 6x_2) \end{bmatrix}$$

$$A\boldsymbol{x} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Architecture ➔ Ordering ➔ Data Structure ➔ Algorithm

Source: Kenji Hiranabe, https://github.com/kenjihiranabe/The-Art-of-Linear-Algebra/, 2022

Challenges in Low-Precision FP

- Smaller range of representable numbers
  - Limited range of lower precision FP might cause overflow
  - Rounding to lower precision may lose important numerical properties (e.g., positive definiteness)
  - Matrix balancing for preprocessing
- Larger unit round-off
  - Lose precision when storing: $fl(x) = x(1+\delta),\ |\delta| \le u$
  - Lose precision when computing: $fl(x \text{ op } y) = (x \text{ op } y)(1+\delta),\ |\delta| \le u$
- Error bounds still apply?
  - Error bound in terms of nu provides **no useful information** if $nu > 1$
  - Developing new techniques and error estimates:
    - ✓ Scaling and shifting approach [Higham, Pranesh, 2019]
    - ✓ Probabilistic approach [Higham, Mary, 2019; Higham, Mary, 2020]

Working With Mixed Precisions

- Compute matrix-vector multiplications in finite precision (forward error):

$$fl(A\boldsymbol{x}) = A\boldsymbol{x} + \delta, \quad |\delta| \le \gamma_n |A||\boldsymbol{x}|$$

- Compute residual in finite precision (forward error):

$$fl(\boldsymbol{b} - A\boldsymbol{x}) = \boldsymbol{b} - A\boldsymbol{x} + \delta, \quad |\delta| \le \gamma_{n+1}\big(|\boldsymbol{b}| + |A||\boldsymbol{x}|\big)$$

- Pessimistic bounds! Improvement using probabilistic error bounds, ...

- **Q: How about working on two-precision computation?**

First compute residual in one precision $\overline{u}$ and then store it in another precision u

$$fl(\boldsymbol{b} - A\boldsymbol{x}) = \boldsymbol{b} - A\boldsymbol{x} + \delta, \quad |\delta| \le u|\boldsymbol{b} - A\boldsymbol{x}| + (1+u)\overline{\gamma}_{n+1}\big(|\boldsymbol{b}| + |A||\boldsymbol{x}|\big)$$

Why? $\hat{r} = fl_u(\tilde{r}) = \tilde{r}(1+\theta), \quad |\theta| \le u$ ▶ $\hat{r} - r = \tilde{r}(1+\theta) - r(1+\theta) + \theta r = \theta r + (1+\theta)e$

Matrix Balancing

$$\forall\, A \in \mathbb{C}^{n\times n}$$

$$D^{-1}AD = [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n] = \begin{bmatrix} \boldsymbol{r}_1 \\ \vdots \\ \boldsymbol{r}_n \end{bmatrix}$$

$$\|\boldsymbol{c}_i\| = \|\boldsymbol{r}_i\|, \quad i = 1:n$$

- Matrix balancing is a crucial preprocessing technique with significant practical importance
- Bit-shifting: powers-of-two scaling is exact in binary FP arithmetic (within the normal exponent range)

```
for i in range(n):  # Process each row/column pair
    # Calculate off-diagonal norms for row i and column i
    row_norm = sum(abs(A_work[i,j]) for j in range(n) if j != i)
    col_norm = sum(abs(A_work[j,i]) for j in range(n) if j != i)

    # Avoid division by zero
    if row_norm > 1e-12 and col_norm > 1e-12:
        # Compute ideal scaling factor
        theoretical_factor = np.sqrt(row_norm / col_norm)

        # Bound the factor to prevent extreme scaling
        bounded_factor = max(0.5, min(2.0, theoretical_factor))

        # Quantize to nearest power of 2 for numerical stability
        exponent = np.round(np.log2(bounded_factor))
        actual_factor = 2.0 ** exponent
    ... ...
```

Intel® Compilers

Packages

Intel® oneAPI Base Toolkit | Intel® C++ Essentials | Intel® Deep Learning Essentials

Operating System

Linux* | Windows*

Installer Type

Offline Installer | Online Installer

What's Included

- Intel® oneAPI DPC++/C++ Compiler
- Intel® DPC++ Compatibility Tool
- Intel® Distribution for GDB*
- Intel® oneAPI DPC++ Library (oneDPL)
- Intel® oneAPI Threading Building Blocks (oneTBB)
- Intel® oneAPI Math Kernel Library (oneMKL)
- Intel® oneAPI Deep Neural Network Library (oneDNN)
- Intel® oneAPI Data Analytics Library (oneDAL)
- Intel® oneAPI Collective Communications Library (oneCCL)
- Intel® Integrated Performance Primitives
- Intel® Cryptography Primitives Library
- Intel® Advisor
- Intel® VTune™ Profiler

Intel® oneAPI Base Toolkit

The Intel® oneAPI Base Toolkit (Base Kit) is a core set of tools and libraries for developing high-performance, data-centric applications across diverse architectures. It features an industry-leading C++ compiler that implements SYCL*, an evolution of C++ for heterogeneous computing.

Choose a Version: 2025.3.1

Size	2432.34 MB
Version	2025.3.1
Date	January 22, 2026
SHA384	8c9c64f68b3810f709b0bb159fc563c697cdc1eea704c6260e7009384e0859ad45be639e8883ffbff8d4219f585e0e35

Intel® oneAPI Base Toolkit (version 2025.3.1) has been updated to include functional and security updates. Users should update to the latest version.

Continue as a Guest (download starts immediately) →

By downloading, you agree to our Privacy and Terms of use

- Install with an Offline GUI Installer
- Install through a Command Line
- Configure a Windows System After Installation
- Run Sample Code to Verify Installation

1

Get Your System Ready

Check that you have the supported hardware and that your system meets the minimum requirements to install the HPC Kit.

- System Requirements
- Supported Hardware

2

Download the HPC Kit

After downloading, follow the Get Started Guide to configure your system and run your first sample.

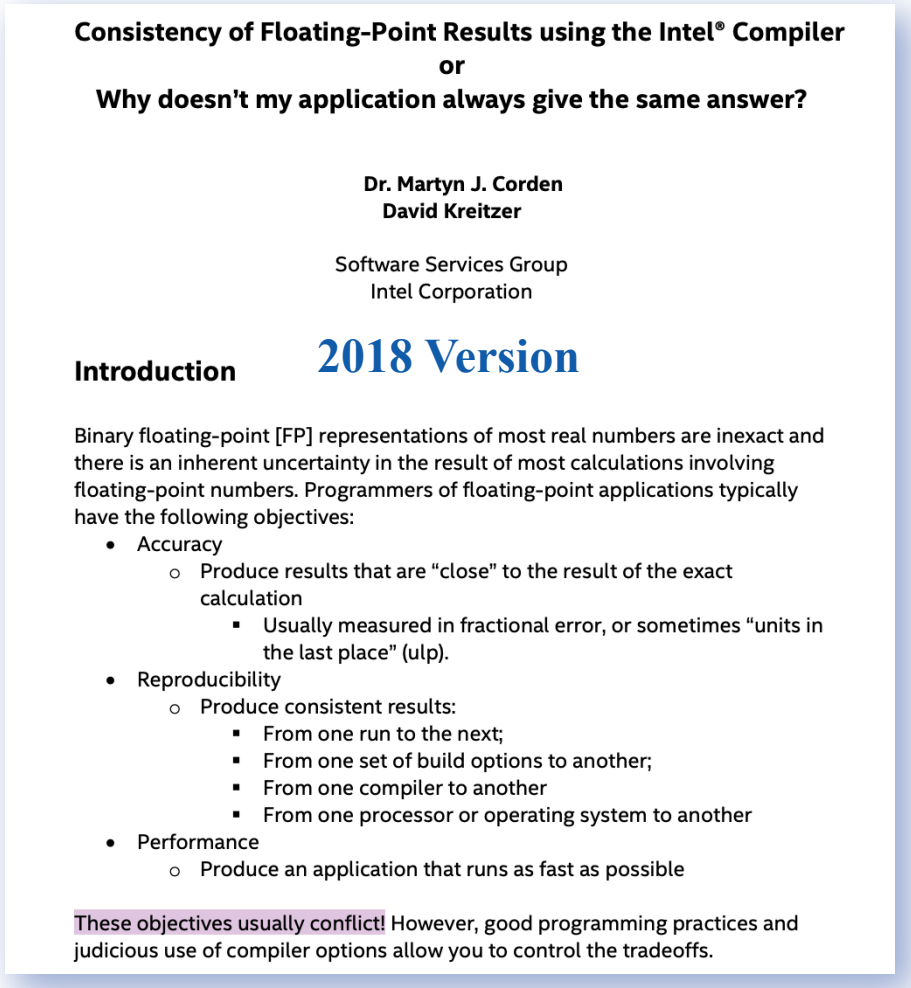
3

Next Steps

Access samples or run your own workloads with the help of tutorials and training.

- Run HPC Code with OpenMP
- Essentials of SYCL*
- Code Samples in GitHub*
- Intel oneAPI Programming Guide

FP Accuracy and Reproducibility



Consistency of Floating-Point Results using the Intel® Compiler
or
Why doesn't my application always give the same answer?

Dr. Martyn J. Corden
David Kreitzer

Software Services Group
Intel Corporation

Introduction **2018 Version**

Binary floating-point [FP] representations of most real numbers are inexact and there is an inherent uncertainty in the result of most calculations involving floating-point numbers. Programmers of floating-point applications typically have the following objectives:

- Accuracy
  - Produce results that are "close" to the result of the exact calculation
    - Usually measured in fractional error, or sometimes "units in the last place" (ulp).
- Reproducibility
  - Produce consistent results:
    - From one run to the next;
    - From one set of build options to another;
    - From one compiler to another
    - From one processor or operating system to another
- Performance
  - Produce an application that runs as fast as possible

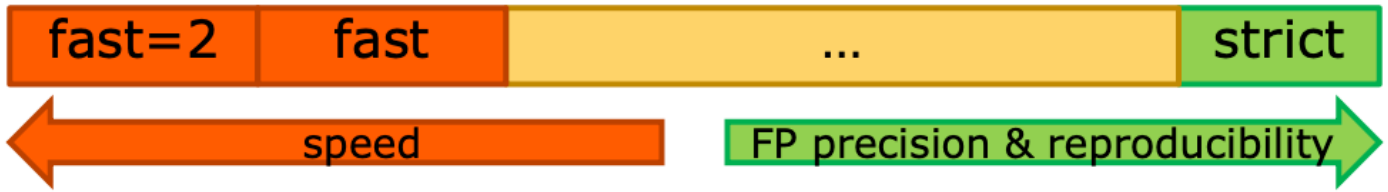
These objectives usually conflict! However, good programming practices and judicious use of compiler options allow you to control the tradeoffs.

- Use **/fp:precise /fp:source** (Windows*) or **-fp-model precise -fp-model source** (Linux*) to improve the consistency and reproducibility of FP results while limiting the impact on performance.
- If reproducibility between different processor types of the same architecture is important, include **/Qimf-arch-consistency:true** (Windows) or **-fimf-arch-consistency=true** (Linux).
- For best reproducibility between processors that support fused multiply-add (FMA) instructions and processors that do not, include **/Qfma-** (Windows) or **-no-fma** (Linux).
- For version 17 or later of the compiler, the best reproducibility may be obtained with the single switch **/fp:consistent** (Windows) or **-fp-model consistent** (Linux), which sets the above options.

Ref: https://www.intel.com/content/www/us/en/developer/articles/technical/consistency-of-floating-point-results-using-the-intel-compiler.html

FP Model for Intel® Compilers

FP model	Description
precise	• Allow value-safe optimizations only
source	• Specify the intermediate precision used for FP expression evaluation
except	• Enable strict floating point exception semantics
strict	• Enable access to the FPU environment • Disable floating point contractions such as fused multiply-add instructions • Imply “precise” and “except”
fast[=1] (default)	• Allow value-unsafe optimizations compiler chooses precision for expression evaluation • Floating-point exception semantics not enforced • Access to the FPU environment not allowed • Floating-point contractions are allowed
fast=2	• Additional approximations allowed



Source: Georg Zitzlsberger, Intel, 2014

An Example of FP Option

- A loop can be auto-vectorized (requires partial sum – reordering is done under the hood)

```
for (int i = 0; i < n; i++ ) sum = sum + x[i]*x[i];
```

- If you compile with **–fp-model precise** or **–fp-model consistent**, the loops are not auto-vectorized

```
remark #15331: loop was not vectorized: precise FP model implied by the
command line or a directive prevents vectorization.
```

- But you can do it by hand by adding **OpenMP** directives (for instance)

```
#pragma omp simd reduction(+:sum)
    for (int i = 0; i < n; i++ ) sum = sum + x[i]*x[i];
```

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
```

条件数与病态问题

Condition number and ill-conditioned problems

/3.2

Condition of Function Evaluation

- Function Evaluation (FE): Evaluating a function at x, namely, $f(x)$
- **Q**: How sensitive is function evaluation w.r.t. perturbations? ➔ Condition of FE
- Error Analysis:
  - **Forward error:**

$$|\delta y| := \left|y - \hat{y}\right| = \left|f(x) - \hat{y}\right|$$

  - **Backward error:**

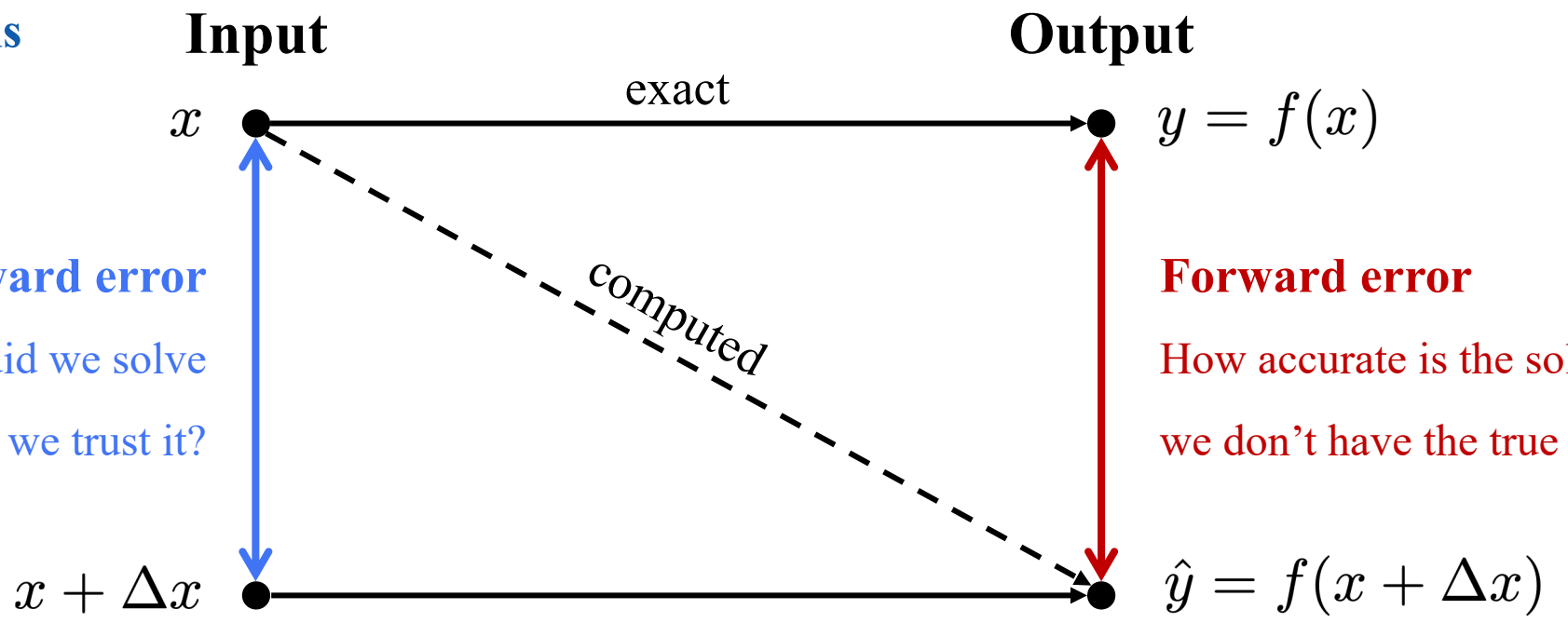
$$|\delta x| := \left|x - \hat{x}\right|, \quad \text{s.t. } y = f(\hat{x}) = f(x + \delta x)$$

  - $f(x + \delta x) \approx f(x) + f'(x)\,\delta x \Rightarrow \hat{f} - f \approx f'(x)\,\delta x \Rightarrow |\delta y| \approx |f'(x)|\,|\delta x|$

Error amplifier

Backward and Forward Error Analysis

Error Analysis



Backward error

What problem did we solve actually? Can we trust it?

Forward error

How accurate is the solution? But we don't have the true solution.

Remark (Mixed forward-backward error)

In some cases, we might be interested in a more involved relation:

$$\hat{y} + \Delta y = f(x + \Delta x), \quad |\Delta y| \le \varepsilon |y|, \quad |\Delta x| \le \eta |x|$$

Backward and Forward Errors

Forward Error $\leq$ **Condition Number** $\times$ **Backward Error**

James Wilkinson, 1970 Turing Award for his research in NA to facilitate the use of the digital computer, work in computations in LA and backward error analysis.

Some remarks:

- Large condition number usually indicates that the problem is ill-conditioned.
- For well-conditioned problems, backward stability of numerical methods implies forward stability.
- Even if the numerical method is backward stable, it could still yield a large forward error; In this case, we can tell that the trouble is caused by the problem itself, instead of the numerical method.

Measuring Sensitivity, Revisited

- Now consider **relative errors** instead of absolute errors

$$\delta f := f - \hat{f},\ \delta x := x - \hat{x} \quad \Rightarrow \quad \lim_{\epsilon \to 0} \sup_{\|\delta x\| \le \epsilon} \frac{\|\delta f(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|f(x)\|}$$

Condition number

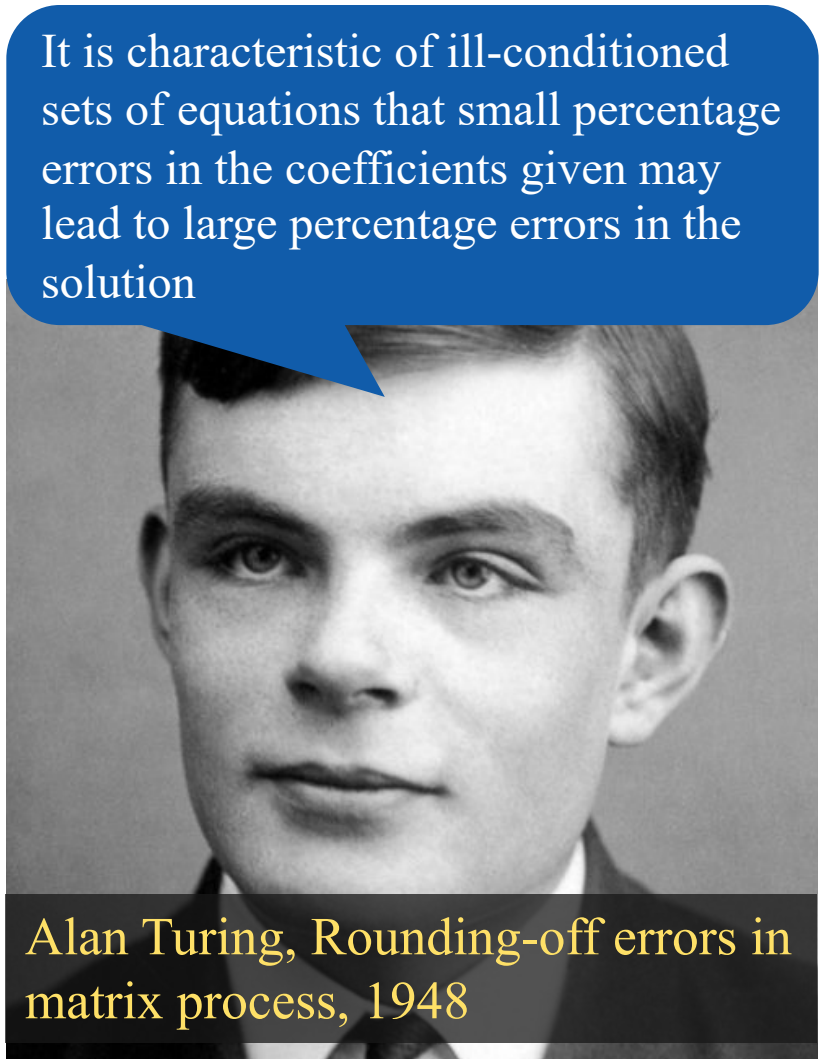
- **Special Example**: condition number of a linear system $Ax = b$

$$\max_{b, \delta b \neq 0} \frac{\|A^{-1}\delta b\|}{\|\delta b\|} \cdot \frac{\|b\|}{\|A^{-1}b\|} = \max_{\delta b \neq 0} \frac{\|A^{-1}\delta b\|}{\|\delta b\|} \max_{b \neq 0} \frac{\|b\|}{\|A^{-1}b\|}$$

$$= \max_{\delta b \neq 0} \frac{\|A^{-1}\delta b\|}{\|\delta b\|} \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \|A^{-1}\|\|A\| \quad =: \kappa(A)$$

- Condition number is crucial to the accuracy of finite-precision linear solvers; understanding it is essential in numerical linear algebra -- it tells you whether computational results can be trusted!

Condition Number, Revisited



- For an invertible square matrix, the condition number is

$$\kappa_p(A) := \|A\|_p \, \|A^{-1}\|_p$$

- Usually, we use the spectral norm

$$\|A\|_2 := \left(\rho(A^T A)\right)^{\frac{1}{2}}$$

$$\kappa_2(A) := \|A\|_2 \, \|A^{-1}\|_2 = \frac{\sigma_1(A)}{\sigma_n(A)} \qquad \text{if } A \text{ nonsingular}$$

$$\kappa_2(A) = \frac{\max_i |\lambda_i(A)|}{\min_i |\lambda_i(A)|} \qquad \text{if } A \text{ normal}$$

$$\kappa_2(Q) \equiv 1 \qquad \text{if } Q \text{ unitary}$$

$$\kappa_2(A) = \kappa_2(UA) = \kappa_2(AU) \qquad \text{if } U \text{ is unitary}$$

Backward Error of Linear System Solver

Theorem 3.1 (Normwise Backward Error of A Linear System)

Let $\eta(\boldsymbol{y}) := \min\{\varepsilon : (A+\Delta A)\boldsymbol{y} = \boldsymbol{b}+\Delta\boldsymbol{b},\ \ \|\Delta A\| \le \varepsilon\|F\|,\ \ \|\Delta\boldsymbol{b}\| \le \varepsilon\|\boldsymbol{f}\|\}$

Then we have

$$\eta(\boldsymbol{y}) = \frac{\|\boldsymbol{r}\|}{\|F\|\,\|\boldsymbol{y}\| + \|\boldsymbol{f}\|}, \quad \boldsymbol{r} := \boldsymbol{b} - A\boldsymbol{y}$$

Sketch of proof:

1. Show that the RHS is always less than or equal to $\eta(\boldsymbol{y})$.

2. Let $\boldsymbol{z} := \dfrac{\boldsymbol{y}}{\|\boldsymbol{y}\|^2}$ ➔ $\boldsymbol{z}^H\boldsymbol{y} = 1$

   Set $\Delta A_{\min} := \dfrac{\|F\|\|\boldsymbol{y}\|}{\|F\|\|\boldsymbol{y}\| + \|\boldsymbol{f}\|}\boldsymbol{r}\boldsymbol{z}^H$

   $$\Delta\boldsymbol{b}_{\min} := -\frac{\|\boldsymbol{f}\|}{\|F\|\|\boldsymbol{y}\| + \|\boldsymbol{f}\|}\boldsymbol{r}$$

   ➡

$$\begin{aligned}
&(A+\Delta A_{\min})\boldsymbol{y} \\
&= A\boldsymbol{y} + \frac{\|F\|\,\|\boldsymbol{y}\|}{\|F\|\,\|\boldsymbol{y}\| + \|\boldsymbol{f}\|}\boldsymbol{r} \\
&= \boldsymbol{b} - \boldsymbol{r} + \frac{\|F\|\,\|\boldsymbol{y}\|}{\|F\|\,\|\boldsymbol{y} + \|\boldsymbol{f}\|}\boldsymbol{r} \\
&= \boldsymbol{b} + \Delta\boldsymbol{b}_{\min}
\end{aligned}$$

Forward Error of Linear System Solver

Theorem 3.2 (Sensitivity Analysis of Linear Solver)

Consider linear systems: $A\boldsymbol{x} = \boldsymbol{b}, \quad A \in \mathbb{R}^{n\times n}, \quad \boldsymbol{0} \neq \boldsymbol{b} \in \mathbb{R}^n$

$$(A + \Delta A)\boldsymbol{y} = \boldsymbol{b} + \Delta\boldsymbol{b}, \quad \Delta A \in \mathbb{R}^{n\times n}, \quad \Delta\boldsymbol{b} \in \mathbb{R}^n$$

If $\dfrac{\|\Delta A\|}{\|A\|} \leq \varepsilon, \quad \dfrac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|} \leq \varepsilon, \quad \delta := \varepsilon\kappa(A) < 1$, then we have:

$$A + \Delta A \text{ is non-singular and } \frac{\|\boldsymbol{y}\|}{\|\boldsymbol{x}\|} \leq \frac{1+\delta}{1-\delta} \quad \Rightarrow \quad \frac{\|\boldsymbol{y}-\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \frac{2\varepsilon}{1-\delta}\kappa(A)$$

Example (Perturbation on RHS):

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 10^{-6} \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} 1 \\ 10^{-6} \end{bmatrix}, \ \Delta\boldsymbol{b} = \begin{bmatrix} 0 \\ 10^{-6} \end{bmatrix} \quad \blacktriangleright \quad 1 = \frac{\|\boldsymbol{y}-\boldsymbol{x}\|_\infty}{\|\boldsymbol{x}\|_\infty} \leq \varepsilon\kappa_\infty(A) = 1$$

Proof of Sensitivity Analysis

1 $\|A^{-1}\Delta A\| \le \|A^{-1}\|\,\|\Delta A\| \le \varepsilon\|A^{-1}\|\,\|A\| = \delta < 1$

$\Rightarrow$ $A + \Delta A$ is nonsingular Inverse of Perturbed Matrix (Theorem 2.11)

2 $(I + A^{-1}\Delta A)\boldsymbol{y} = \boldsymbol{x} + A^{-1}\Delta\boldsymbol{b}$

$$\Rightarrow \quad \|\boldsymbol{y}\| \le \|(I + A^{-1}\Delta A)^{-1}\|\big(\|\boldsymbol{x}\| + \|A^{-1}\|\|\Delta\boldsymbol{b}\|\big)$$
$$\le \frac{1}{1-\delta}\big(\|\boldsymbol{x}\| + \|A^{-1}\|\|\Delta\boldsymbol{b}\|\big) \le \frac{1+\delta}{1-\delta}\|\boldsymbol{x}\|$$

Inverse of Perturbed Matrix

3 $\boldsymbol{y} - \boldsymbol{x} = A^{-1}\Delta\boldsymbol{b} - A^{-1}\Delta A\boldsymbol{y}$ $\Rightarrow$ $\|\boldsymbol{y} - \boldsymbol{x}\| \le \varepsilon\|A^{-1}\|\,\|\boldsymbol{b}\| + \varepsilon\|A^{-1}\|\,\|A\|\,\|\boldsymbol{y}\|$

$$\Rightarrow \quad \frac{\|\boldsymbol{y}-\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \le \varepsilon\kappa(A)\left(\frac{\|\boldsymbol{b}\|}{\|A\|\|\boldsymbol{x}\|} + \frac{\|\boldsymbol{y}\|}{\|\boldsymbol{x}\|}\right) \le \frac{2\varepsilon}{1-\delta}\kappa(A)$$

Exercise

Ill-Conditioned Systems

- Suppose that the computed solution is the exact solution of the following perturbed equation:

$$(A+E)\hat{\boldsymbol{x}} = \boldsymbol{b} + \boldsymbol{e}, \quad \|E\|_\infty \le u\,\|A\|_\infty, \quad \|\boldsymbol{e}\|_\infty \le u\,\|\boldsymbol{b}\|_\infty$$

- Using Theorem 3.2, we can estimate the error of linear solution

$$u\,\kappa_\infty(A) \le \frac{1}{2} \quad \Rightarrow \quad \frac{\|\boldsymbol{x}-\hat{\boldsymbol{x}}\|_\infty}{\|\boldsymbol{x}\|_\infty} \le 4u\kappa_\infty(A)$$

- If the system is **ill-conditioned**, $u\kappa_\infty(A) \approx 1$, then the relative error could be very large.

- In such cases, solving the linear system is **sensitive to floating-point error**; so we should not blame numerical methods alone if the quality of the solution is not satisfactory.

- We will further discuss stability of numerical methods in later chapters.

Some Remarks on Condition Numbers

- Some properties of condition number for nonsingular matrix $A \in \mathbb{R}^{n\times n}$
  - Lower bound: $\kappa(A) = \|A\|\,\|A^{-1}\| \geq \|A\,A^{-1}\| = 1$ for induced norms
  - Inversion & scaling: $\kappa(A) = \kappa(A^{-1})$ $\kappa(\alpha A) = \kappa(A), \quad \alpha \neq 0$
- Some other useful condition-number-like terms:

$$\mathrm{cond}(A) := \big\||A^{-1}|\,|A|\big\|_\infty \qquad \mathrm{cond}(A, \boldsymbol{x}) := \frac{\big\||A^{-1}|\,|A|\,|\boldsymbol{x}|\big\|_\infty}{\|\boldsymbol{x}\|_\infty}$$

$$\mathrm{cond}(A, \boldsymbol{x}) \leq \mathrm{cond}(A) \leq \kappa_\infty(A)$$

- **Key Takeaways:** (1) A small residual does not guarantee a small error when the condition number is large; (2) Even a stable algorithm can produce inaccurate results for an ill-conditioned problem; (3) Improving conditioning through preconditioning can accelerate iterative convergence.

A Simple Example

- Five-point finite difference stencil for the Poisson's equation on a unit square (Dirichlet boundary)

```
import numpy as np
from scipy.sparse import diags, eye, kron

n = 100                          # 每个方向的内部网格点数
h = 1.0 / (n + 1)

T = diags([-np.ones(n-1), 2*np.ones(n), -np.ones(n-1)],
          offsets=[-1, 0, 1], format="csr")
I = eye(n, format="csr")

A = (kron(I, T, format="csr") + kron(T, I, format="csr")) / h**2
```

$$A = \frac{1}{h^2}\left(I_n \otimes T + T \otimes I_n\right).$$

- Condition number will increase when h is getting smaller
- **Note:** Diagonal preconditioning does not work in this case

迭代精校方法

Iterative refinement (IR) methods

/3.3

Iterative Refinement Methods

- Sometimes, the condition number of a linear system could be very large.
- Solvers might not be able to give a solution as accurate as we need them to provide; This has been noticed long ago (Wilkinson 1948), but using high-precision is costly.
- One can improve accuracy iteratively using the 3-precision iterative refinement (IR) method:

Algorithm 1: General iterative refinement algorithm

```
%% Given a nonsingular matrix A ∈ ℝ^{n×n}, b ∈ ℝ^n, and an initial guess x_0 ∈ ℝ^n;
%% Set three precisions u_r ⩽ u ⩽ u_ℓ for residual, work, and solve, respectively;
for i = 0 : MaxIter or converged
    Compute r_i ← b − A x_i in precision u_r;
    Solve the error equation A e_i = r_i in precision u_ℓ;
    Update x_{i+1} ← x_i + e_i in precision u;
end
```

Accuracy of IR Methods

Theorem 3.3 (Carson and Higham, 2018)

The limiting accuracy depends on the **residual and work precisions**, but does not depend on the **solve precision**, the initial guess, or how the system (Line 5) is solved.

$$\frac{\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|_\infty}{\|\boldsymbol{x}\|_\infty} \lesssim u + 4p\,\mathrm{cond}(A, \boldsymbol{x})\, u_r$$

Convergence factor

$\phi_i := \xi_i + 2\min\left(\mathrm{cond}(A), \kappa_\infty(A)\mu_i\right)u_\ell$

is sufficiently small

$$\begin{cases} \xi_i := \dfrac{\|\hat{\boldsymbol{e}}_i - \boldsymbol{e}_i\|_\infty}{\|\boldsymbol{e}_i\|_\infty} \\ \mu_i := \dfrac{\|A(\hat{\boldsymbol{x}}_i - \boldsymbol{x}_i)\|_\infty}{\|A\|_\infty \|\hat{\boldsymbol{x}}_i - \boldsymbol{x}_i\|_\infty} \le 1 \end{cases}$$

Maximum nnz in any row of the equation

Condition number

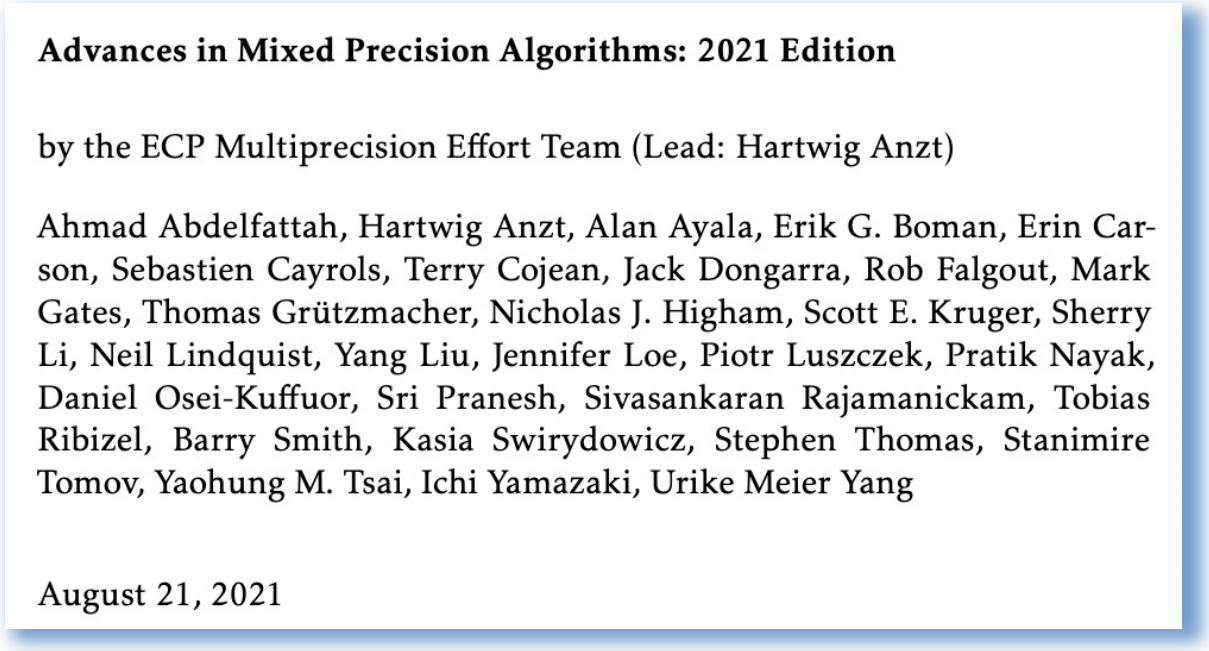
$$\mathrm{cond}(A) := \left\| |A^{-1}|\,|A| \right\|_\infty$$

$$\mathrm{cond}(A, \boldsymbol{x}) := \frac{\left\| |A^{-1}|\,|A|\,|\boldsymbol{x}| \right\|_\infty}{\|\boldsymbol{x}\|_\infty}$$

$$\mathrm{cond}(A, \boldsymbol{x}) \le \mathrm{cond}(A) \le \kappa_\infty(A)$$

Different Versions of IR

- **Traditional IR methods** for old hardware (some old computers can compute residual using higher precision than working precision without extra work)
- **Mono-precision IR methods**: focus on improving stability
- **IR methods with low-precision solvers**: emphasize efficiency and energy consumption

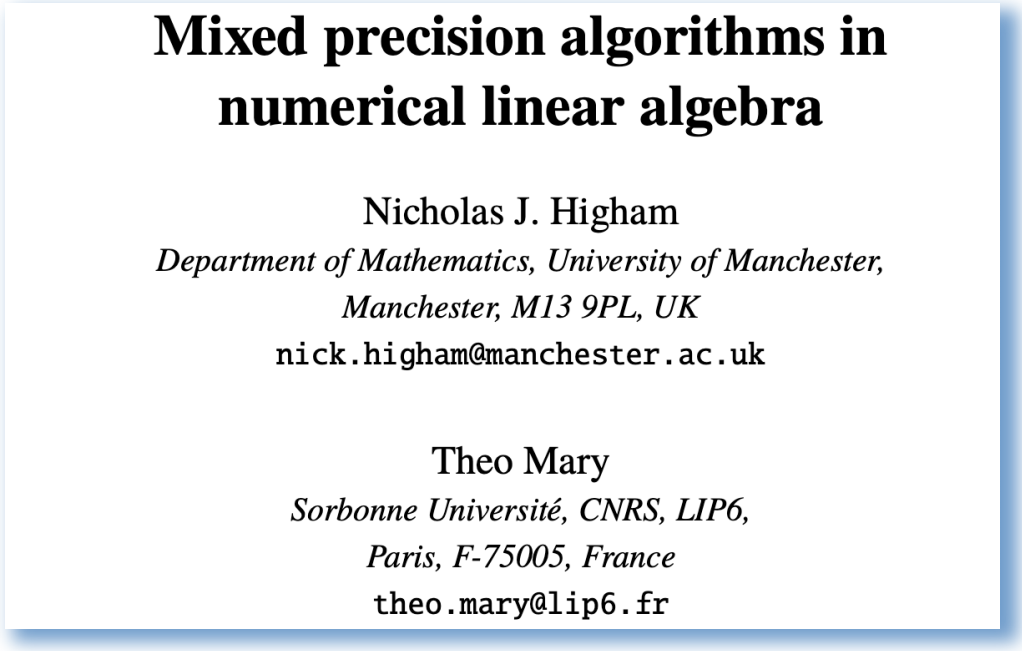


Advances in Mixed Precision Algorithms: 2021 Edition

by the ECP Multiprecision Effort Team (Lead: Hartwig Anzt)

Ahmad Abdelfattah, Hartwig Anzt, Alan Ayala, Erik G. Boman, Erin Carson, Sebastien Cayrols, Terry Cojean, Jack Dongarra, Rob Falgout, Mark Gates, Thomas Grützmacher, Nicholas J. Higham, Scott E. Kruger, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Daniel Osei-Kuffuor, Sri Pranesh, Sivasankaran Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M. Tsai, Ichi Yamazaki, Urike Meier Yang

August 21, 2021



Mixed precision algorithms in numerical linear algebra

Nicholas J. Higham
Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK
nick.higham@manchester.ac.uk

Theo Mary
Sorbonne Université, CNRS, LIP6, Paris, F-75005, France
theo.mary@lip6.fr

Example: LU-IR Method

Algorithm 2: LU-IR method

```
%% Given a nonsingular matrix A ∈ ℝ^{n×n} and b ∈ ℝ^n;
%% Set three precisions u_r ⩽ u ⩽ u_ℓ for residual, work, and solve, respectively;
Compute the factorization A = LU in precision u_ℓ;
Solve LUx_0 = b by substitution in precision u_ℓ;
for i = 0 : MaxIter or converged
    Compute r_i ← b − Ax_i in precision u_r;
    Solve LUe_i = r_i by substitution in precision u_ℓ;
    Update x_{i+1} ← x_i + e_i, in precision u;
end
```

3-precision

- The Carson–Higham theorem can be applied to this method
- LU factorization only needs to be computed once for each coefficient matrix
- Less stable but faster factorization methods can be used to improve efficiency (e.g. in SuperLU and PARDISO) [Li and Demmel 1998; Arioli, Duff, Gratton, and Pralet 2007]

Example: GMRES-IR Method

Algorithm 3: Preconditioned GMRES-IR method

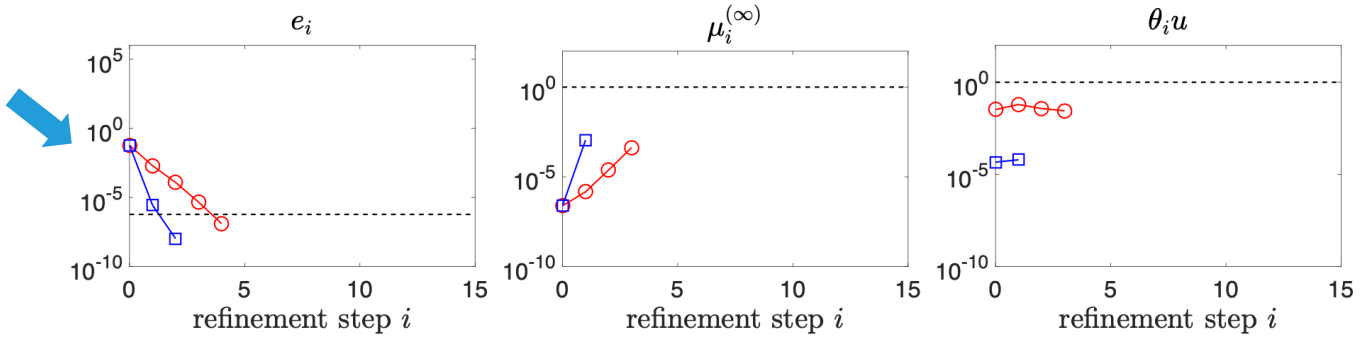
```
%% Given a nonsingular matrix A ∈ R^{n×n} and b ∈ R^n;
%% Set five precisions u_r, u_g, u_p, u, u_ℓ for different purposes;
Compute the factorization A = LU in precision u_ℓ;
Solve LU x_0 = b by substitution in precision u_ℓ;
for i = 0 : MaxIter or converged
    Compute r_i ← b − A x_i in precision u_r;
    Solve A e_i = r_i by PGMRES in precision u_g;
    In PGMRES, preconditioner U^{-1} L^{-1} applied in precision u_p;
    Update x_{i+1} ← x_i + e_i in precision u;
end
```

5-precision

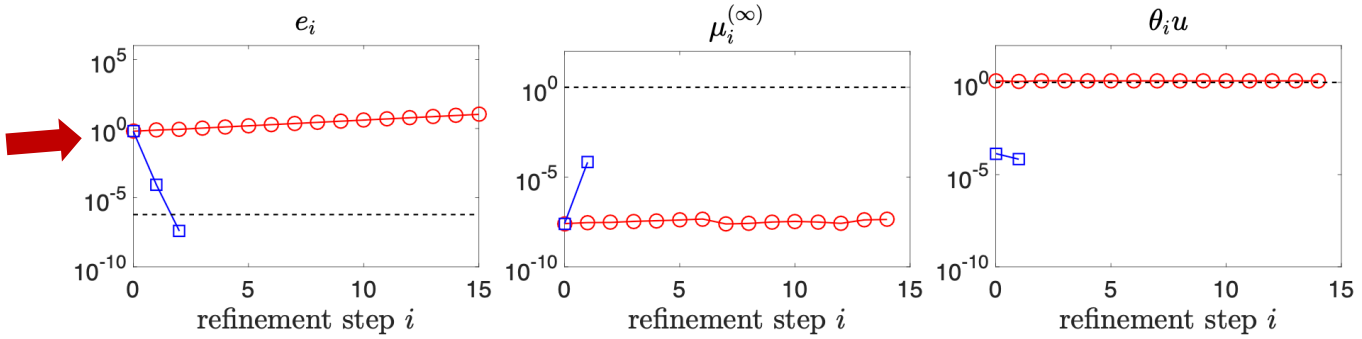
If $\kappa(A)^2 u_\ell^2 (u_g + \kappa(A) u_p)$ is small, GMRES-IR will converge till

$$A\boldsymbol{e}_i = \boldsymbol{r}_i, \quad \frac{\|\boldsymbol{e}_i - \hat{\boldsymbol{e}}_i\|_\infty}{\|\boldsymbol{e}_i\|_\infty} =: \theta_i u \equiv \xi_i \le 1 \quad \Rightarrow \quad \frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \lesssim u + 4\,p\,\mathrm{cond}(A, x) u_r$$

Accuracy of IR Methods



Test 1: `gallery('randsvd',100,1e7,3)`.



Test 2: `gallery('randsvd',100,1e8,3)`.

$$(u_r, u_g, u_p, u, u_\ell) = (2^{-53}, 2^{-24}, 2^{-53}, 2^{-24}, 2^{-24})$$

- GMRES-IR shows advantages compared with LU-IR in practice
- GMRES-IR is used in HPL-AI
- One can also use GMRES for the original equation instead of the error equation
- One can use low precision for GMRES iterations

Source: Erin Carson and Nicholas J. Higham. "A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems". SIAM Journal on Scientific Computing 2017 39:6, A2834-A2856

MP Methods for Sparse Problems

- More difficult to predict speedups of MP iterative methods for sparse matrices.
- LU factorization of a sparse matrix:
  - Sparsity-based reordering and symbolic analysis mainly operate on graph and index data, so reducing floating-point precision offers little direct speedup in these phases.
  - Numerical factorization can generate tiny fill entries that propagate through subsequent elimination steps.
  - Even if all nonzero matrix entries are scaled into the normal range of single precision, subnormal numbers may still arise during factorization, potentially affecting performance depending on the arithmetic precision and hardware.
- Reminder: On some hardware, operations involving subnormal numbers can be much slower than those involving normal numbers.

Ref: Zounon M, Higham NJ, Lucas C, Tisseur F. "Performance impact of precision reduction in sparse linear systems solvers". Peer J Computer Science 2022, 8:e778

Reading and Thinking

Acta Numerica (2022), pp. 347–414
doi:10.1017/S0962492922000022

Printed in the United Kingdom

Mixed precision algorithms in numerical linear algebra

Nicholas J. Higham
Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK
E-mail: nick.higham@manchester.ac.uk

Theo Mary
Sorbonne Université, CNRS, LIP6, Paris, F-75005, France
E-mail: theo.mary@lip6.fr

- Did you have trouble with floating-point precision related problems?
- Did you have trouble reaching the desired accuracy?
- What was the reason? How did you handle them?
- Are the solvers or other algorithms you use numerically stable?
- Can you use mixed precision somewhere in your code?

矩阵计算基本工具

Basic tools of matrix computation

/3.4

Orthogonalization

- The goal of **orthogonalization**:

  To transform a set of linearly independent vectors (which may not be orthogonal and could have very small angles between them) into a new set of vectors that are mutually orthogonal, while ensuring this new set can span the same vector space as the original set.

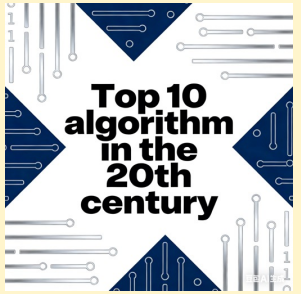
- Orthogonality offers several important advantages:
  - Simplified computations: Orthogonal bases simplify orthogonal projections (see §2.2) and the computation of expansion coefficients, for example.
  - Numerical stability: Orthogonal transformations preserve the 2-norm and therefore do not amplify existing perturbations in that norm, a property that underpins the numerical stability of many algorithms.
  - Broad applicability: Orthogonal bases and transformations play central roles in QR factorization, eigenvalue algorithms, signal processing, and principal component analysis.

QR Decomposition

QR decomposition (QR factorization) factors a matrix into the product of an orthogonal matrix and an upper triangular matrix:

For $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$

$\Rightarrow$ $A = QR, \quad Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper trapezoidal



- Three standard tools for the QR decomposition:
  - **Gram–Schmidt orthogonalization:** Construct orthonormal vectors column by column. Simple concept, but numerically unstable in floating-point computation;
  - **Householder reflections:** Use successive orthogonal reflection transformations to zero out subdiagonal entries (**industry standard**, good numerical stability);
  - **Givens rotations:** Apply plane rotations to eliminate individual non-zero entries (convenient for sparse matrices and updating QR after small matrix changes).

Properties of QR Decomposition

Theorem 3.4

If $A \in \mathbb{R}^{m\times n}$ has full column-rank and $A = QR$ is a QR decomposition, then

$$\text{span}\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\} = \text{span}\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k\}, \quad k = 1 : n$$

Let $Q_1 := Q(1 : m, 1 : n)$ and $Q_2 = Q(1 : m, n + 1 : m)$. Then

$$\text{range}(A) = \text{range}(Q_1), \quad \text{range}(A)^{\perp} = \text{range}(Q_2)$$

Remarks:

1. $\boldsymbol{a}_k = \sum_{i=1}^{k} r_{ik}\boldsymbol{q}_i \in \text{span}\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k\}, \quad k = 1 : n$ ⇒ $\text{span}\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\} \subseteq \text{span}\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k\}$
2. A has full column-rank, so $\dim\big(\text{span}\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\}\big) = k$
3. $A = Q_1 R_1, \quad R_1 := R(1 : n, 1 : n)$ **Reduced QR decomposition**

Reduced QR decomposition

Theorem 3.5 (Reduced QR)

If $A \in \mathbb{R}^{m\times n}$ has full column-rank, the reduced QR decomposition $A = Q_1 R_1$ is unique, where

$Q_1 \in \mathbb{R}^{m\times n}$ has orthonormal columns, $R_1 \in \mathbb{R}^{n\times n}$ is upper triangular with positive diagonals

Moreover, we have

$$R_1 = G \quad \text{and} \quad G^T G = A^T A\ ,\ G \text{ is the upper Cholesky factor of } A^T A$$

Remarks:

1. $A^T A = (Q_1 R_1)^T (Q_1 R_1) = R_1^T R_1$
2. For SPD, the Cholesky factorization is unique and diagonal entries of G are positive. Hence, $Q_1 = AR_1^{-1}$ is also unique.

Q: Can we find the reduced/thin QR decomposition without knowing the full decomposition?

Gram–Schmidt Orthogonalization

- Assume $\boldsymbol{q}_1 = \boldsymbol{a}_1/\|\boldsymbol{a}_1\|_2$ and that $\boldsymbol{a}_2$ is linearly independent of $\boldsymbol{q}_1$ ⇒ $\boldsymbol{w}_2 = \boldsymbol{a}_2 - r_{12}\boldsymbol{q}_1$
- Enforce the orthogonality condition

$$0 = (\boldsymbol{q}_1, \boldsymbol{w}_2) = (\boldsymbol{q}_1, \boldsymbol{a}_2) - r_{12} \quad \Rightarrow \quad r_{12} = (\boldsymbol{q}_1, \boldsymbol{a}_2)$$

- Normalize the remaining component:

$$r_{22} = \|\boldsymbol{w}_2\|_2, \quad \boldsymbol{q}_2 = \boldsymbol{w}_2/r_{22}$$

- Applying this procedure successively to the columns of a full-column-rank matrix $A \in \mathbb{R}^{m\times n}$, with $m \geq n$, yields the thin QR factorization

$$A = QR, \quad Q \in \mathbb{R}^{m\times n}, \quad R \in \mathbb{R}^{n\times n}$$

where $Q^TQ = I_n$ and R is upper triangular. Only n orthonormal vectors are constructed; no extension to a full basis of is needed.

Classical Gram–Schmidt Method

- Orthogonalization

$$\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{j-1}\} \Longrightarrow \{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_{j-1}\} \quad \text{Orthonormal}$$

$$\text{Add a new vector } \boldsymbol{a}_j \Longrightarrow \boldsymbol{q}_j = ?$$

- Classical G-S:

$$\left.\begin{aligned} \boldsymbol{w}_j &= \boldsymbol{a}_j \\ r_{ij} &= \boldsymbol{q}_i^H \boldsymbol{a}_j \\ \boldsymbol{w}_j &= \boldsymbol{w}_j - r_{ij}\boldsymbol{q}_i \end{aligned}\right\} \; i = 1 : j-1 \Longrightarrow \left\{\begin{aligned} \boldsymbol{w}_j &= \boldsymbol{a}_j - \sum_{i=1}^{j-1} r_{ij}\boldsymbol{q}_i \\ \boldsymbol{q}_i^H \boldsymbol{w}_j &= \boldsymbol{q}_i^H \boldsymbol{a}_j - r_{ij} = 0 \\ & \qquad i = 1 : j-1 \end{aligned}\right.$$

$$r_{jj} = \|\boldsymbol{w}_j\|_2$$

$$\boldsymbol{q}_j = \frac{\boldsymbol{w}_j}{r_{jj}}$$

Known to be NOT stable!

Modified Gram–Schmidt Method

- Orthogonalization

$$\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{j-1}\} \Longrightarrow \{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_{j-1}\} \quad \text{Orthonormal}$$

$$\text{Add a new vector } \boldsymbol{a}_j \Longrightarrow \boldsymbol{q}_j = ?$$

- Modified G-S:

$$\left.\begin{aligned} \boldsymbol{w}_j &= \boldsymbol{a}_j \\ r_{ij} &= \boldsymbol{q}_i^H \boldsymbol{w}_j \\ \boldsymbol{w}_j &= \boldsymbol{w}_j - r_{ij}\boldsymbol{q}_i \end{aligned}\right\} \; i = 1 : j-1 \Longrightarrow \left\{\begin{aligned} \boldsymbol{w}_j &= \boldsymbol{a}_j - \sum_{i=1}^{j-1} r_{ij}\boldsymbol{q}_i \\ \boldsymbol{q}_i^H \boldsymbol{w}_j &= \boldsymbol{q}_i^H \boldsymbol{a}_j - r_{ij} = 0 \\ & \qquad i = 1 : j-1 \end{aligned}\right.$$

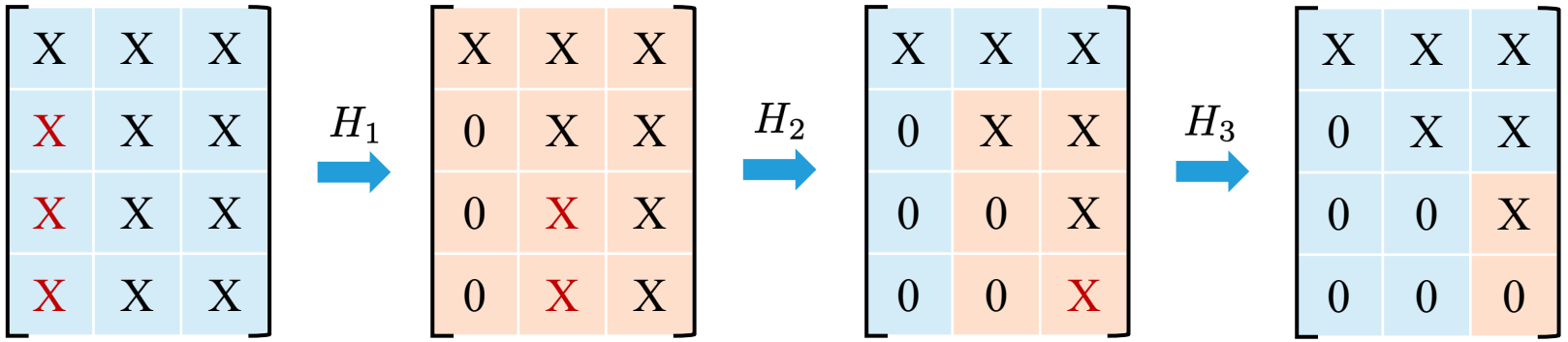
$$r_{jj} = \|\boldsymbol{w}_j\|_2$$

$$\boldsymbol{q}_j = \frac{\boldsymbol{w}_j}{r_{jj}}$$

$$(I - Q_{j-1}Q_{j-1}^H)\boldsymbol{a}_j$$

QR Decomposition via Householder

- Use Householder transforms



$$Q^T = H_3 H_2 H_1$$

Key: Product of orthogonal matrices remains orthogonal

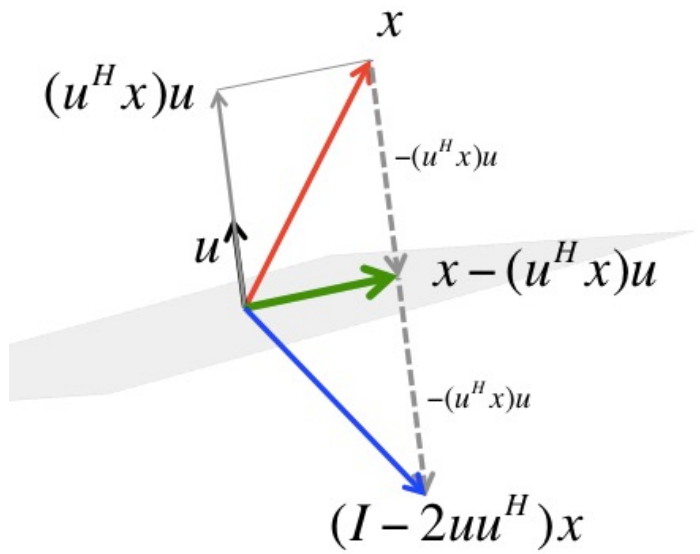
- Householder reflections provide a stable and efficient method to compute the QR decomposition.
- Computational complexity: Full QR $\approx 2m^2n - \frac{2}{3}n^3$; Thin QR $\approx 2mn^2 - \frac{2}{3}n^3$.
- This method is the backbone of several robust numerical libraries (LAPACK, Eigen, SciPy). For production code, use LAPACK's DGEQRF—it handles edge cases and optimizes cache usage.

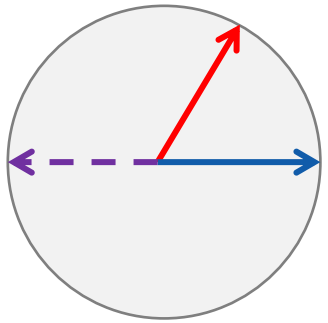
Householder Transformation, Revisited

- **Householder matrix** or transformation
  - $H = I - 2\boldsymbol{u}\boldsymbol{u}^T, \quad \boldsymbol{u} \in \mathbb{R}^n, \ \|\boldsymbol{u}\|_2 = 1$
  - $H(\boldsymbol{y} + a\boldsymbol{u}) = \boldsymbol{y} - a\boldsymbol{u}, \quad \forall\, \boldsymbol{y} \perp \text{span}\{\boldsymbol{u}\}, \ a \in \mathbb{R}$
  - $\forall\, \boldsymbol{0} \neq \boldsymbol{x} \in \mathbb{R}^n, \ \exists\, \boldsymbol{u} \in \mathbb{R}^n, \text{ s.t. } H\boldsymbol{x} = a\boldsymbol{e}_1$

    $$\text{where } a = \pm\|\boldsymbol{x}\|_2, \ \boldsymbol{u} = \frac{\boldsymbol{x} \pm \|\boldsymbol{x}\|_2 \boldsymbol{e}_1}{\|\boldsymbol{x} \pm \|\boldsymbol{x}\|_2 \boldsymbol{e}_1\|_2}$$

    - The Householder transformation reflects a vector $\boldsymbol{x}$ such that only the first entry is nonzero.
    - The vector $\boldsymbol{u}$ is called a Householder vector.





Q: How to implement this in practice?

abbr. **H transformation**

Practical Householder Method

- Try to avoid cancellations and overflows

$$\boldsymbol{u} = \frac{\boldsymbol{x} \pm \|\boldsymbol{x}\|_2 \boldsymbol{e}_1}{\|\boldsymbol{x} \pm \|\boldsymbol{x}\|_2 \boldsymbol{e}_1\|_2}$$

Given $\boldsymbol{x} \in \mathbb{R}^n$. Find $\boldsymbol{v} \in \mathbb{R}^n$, $\beta \in \mathbb{R}$, s.t.

$H = I - \beta \boldsymbol{v}\boldsymbol{v}^T$ is orthogonal and $H\boldsymbol{x} = \pm\|\boldsymbol{x}\|_2 \boldsymbol{e}_1$

- Applying the Householder transformation

$$HA = \left(I - \beta \boldsymbol{v}\boldsymbol{v}^T\right) A = A - \boldsymbol{v}\boldsymbol{w}^T, \quad \boldsymbol{w} = \beta A^T \boldsymbol{v}$$

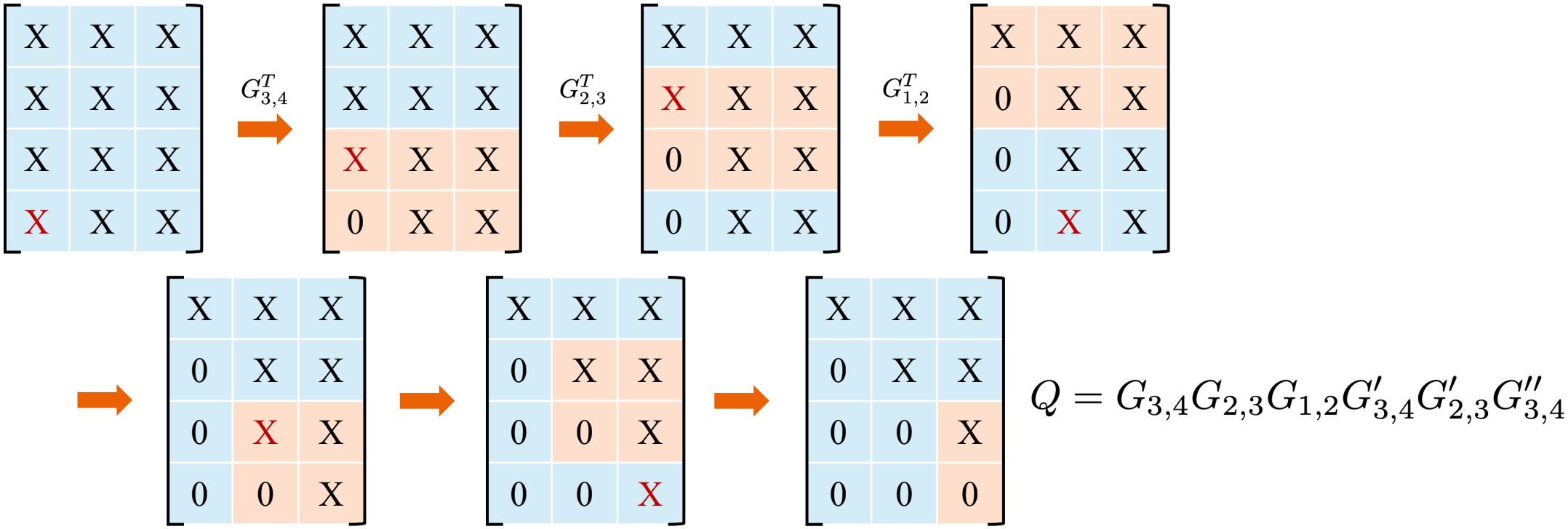
Rank-one update

- There is no need to store a full matrix H; in practice, we can just store a vector and a scalar.

$\boldsymbol{v}, \beta = \text{Householder}(\boldsymbol{x})$
  if $\boldsymbol{x} == \boldsymbol{0}$
    $\beta = 0;\ \boldsymbol{v} = \boldsymbol{e}_1;$ return;
  end
  $\sigma = \max_i |x_i|;$
  $\boldsymbol{x} = \boldsymbol{x}/\sigma;$
  $\mu = \|\boldsymbol{x}\|_2;$
  if $x_1 \le 0$
    $v_1 = x_1 - \mu;$
  else
    $v_1 = x_1 + \mu;$
  end
  $\boldsymbol{v} = \boldsymbol{x}/v_1,\ \boldsymbol{v}(1) = 1;$
  $\beta = 1 + |x_1|/\mu;$

QR Decomposition via Givens

- Use Givens transforms



$$Q = G_{3,4}G_{2,3}G_{1,2}G'_{3,4}G'_{2,3}G''_{3,4}$$

- Givens rotations offer an alternative to Householder reflections for computing QR $\approx 6m^2n - 3mn^2$
- While less efficient for dense matrices, Givens excels in structured, sparse, and dynamic settings

Givens Transformation, Revisited

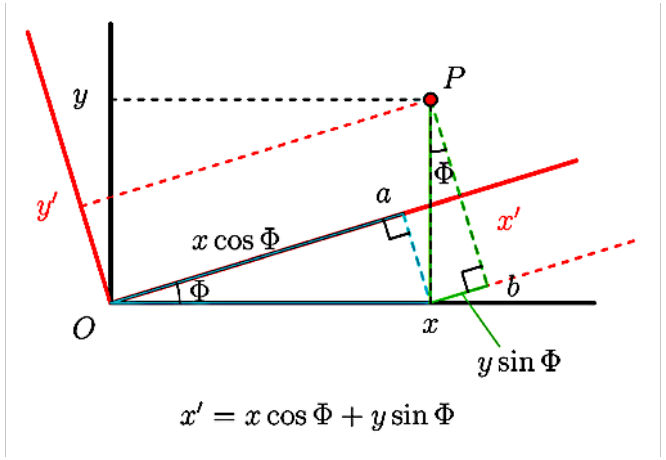
- Givens matrix or transformation

$$G_{i,k}(\Phi) = \begin{bmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c & & & & s & & \\ & & & & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & & & \\ & & & -s & & & & c & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{bmatrix}$$

$$c = \cos(\Phi),\ s = \sin(\Phi),\ \text{and}\ \Phi \in \mathbb{R}$$

Q: How to implement this in practice?

Passive Rotation View:



Geometrically, it represents a plane rotation, and its objective is to annihilate a specific entry in a vector with minimal changes to other entries

abbr. **G transformation**

Representing a Givens Matrix

- **Givens** transformation on a vector:

  Given $\boldsymbol{x} \in \mathbb{R}^n$. Compute $\boldsymbol{y} = G_{i,k}(\Phi)^T \boldsymbol{x} \in \mathbb{R}^n$.

$$y_j = \begin{cases} c\,x_i - s\,x_k, & j = i \\ s\,x_i + c\,x_k, & j = k \\ x_j, & \text{otherwise} \end{cases}$$

$$c = x_i / \sqrt{x_i^2 + x_k^2}, \quad s = -x_k / \sqrt{x_i^2 + x_k^2}$$

$$\Rightarrow \quad y_k = 0$$

$c,\ s = \text{Givens}(x_1,\ x_2)$
  if $x_2 == 0$
    $c = 1,\ s = 0;$
  else
    if $|x_2| > |x_1|$
      $\tau = -x_1/x_2,\ s = 1/\sqrt{1+\tau^2},\ c = s\tau;$
    else
      $\tau = -x_2/x_1,\ c = 1/\sqrt{1+\tau^2},\ s = c\tau;$
    end
  end

- Used when "surgical precision" is needed:
  - Updating the QR decomposition after minor modifications
  - The matrix is sparse or has specific structure (e.g., upper Hessenberg)
  - A small portion of the matrix needs to be zeroed out (e.g., in SVD)

Reducing Cost of Givens Transformation

- Square-root is expensive
  - Modern CPUs have dedicated hardware instructions (such as x86's sqrtsd/sqrtss) to accelerate sqrt, the calculation process often cannot be fully pipelined or requires multiple pipeline stages.
  - For instructions like sqrtsd on Intel processors (Skylake architecture), the latency is between 13 to 19 cycles.
  - Certain compilers offer fast approximate mathematical functions (such as -ffast-math), which may trade accuracy for speed; you must be aware of their impact on accuracy and standard compliance.
- Making Givens transformations **square-root-free**

$$M^T M = \mathrm{diag}(d_i) =: D, \quad d_i > 0 \qquad \text{orthogonal columns}$$

$$Q := MD^{-1/2} \quad \Rightarrow \quad Q^T Q = I \qquad \text{orthonormal (column-wise)}$$

Using this idea, we can represent an orthonormal matrix with M and D!

- **Q: How to eliminate nonzeros in this setting?**

Fast Givens Transformation

- Square-root-free Givens transformations:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad D = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix} \Rightarrow M_1 = \begin{bmatrix} \beta & 1 \\ 1 & \alpha \end{bmatrix} \Rightarrow M_1^T \boldsymbol{x} = \begin{bmatrix} \beta x_1 + x_2 \\ x_1 + \alpha x_2 \end{bmatrix}$$

$$\Rightarrow M_1^T D M_1 = \begin{bmatrix} d_2 + \beta^2 d_1 & d_1\beta + d_2\alpha \\ d_1\beta + d_2\alpha & d_1 + d_2\alpha^2 \end{bmatrix} =: D_1$$

We wish it to be diagonal

$$x_2 \neq 0 \implies \alpha = -x_1/x_2, \ \beta = -\alpha d_2/d_1$$

$$M_1^T \boldsymbol{x} = \begin{bmatrix} x_2(1+\gamma) \\ 0 \end{bmatrix} \Leftarrow \gamma := -\alpha\beta = \frac{d_2}{d_1}\left(\frac{x_1}{x_2}\right)^2$$

$$M_1^T D M_1 = \begin{bmatrix} d_2(1+\gamma) & 0 \\ 0 & d_1(1+\gamma) \end{bmatrix} = D_1 \quad \text{is diagonal}$$

Another form:

$$M_1' = \begin{bmatrix} 1 & \alpha \\ \beta & 1 \end{bmatrix}$$

Such transformations are cheaper to apply: Only need 2 multiplications and 2 additions!

Scaled Transformation for QR

- Define a new scaled transformation:

$$M_1^T D M_1 = D_1 \quad \Rightarrow \quad J := D^{1/2} M_1 D_1^{-1/2} \quad \Rightarrow \quad J^T J = I$$

- The corresponding orthogonal transformation preserves the elimination of the second component:

$$J^T (D^{-1/2} \boldsymbol{x}) = D_1^{-1/2} M_1^T \boldsymbol{x} \quad \text{eliminates } x_2$$

- Let $D_0 = I$. Using a sequence of scaled transforms $M_1, M_2, \ldots, M_L$ for QR:

$$(D_L^{-1/2} M_L^T D_{L-1}^{1/2}) \cdots (D_1^{-1/2} M_1^T D_0^{1/2}) A = D_L^{-1/2} M_L^T \cdots M_2^T M_1^T A = R$$

$$Q = M_1 M_2 \cdots M_L D_L^{-1/2}$$

- The matrix Q is orthogonal because

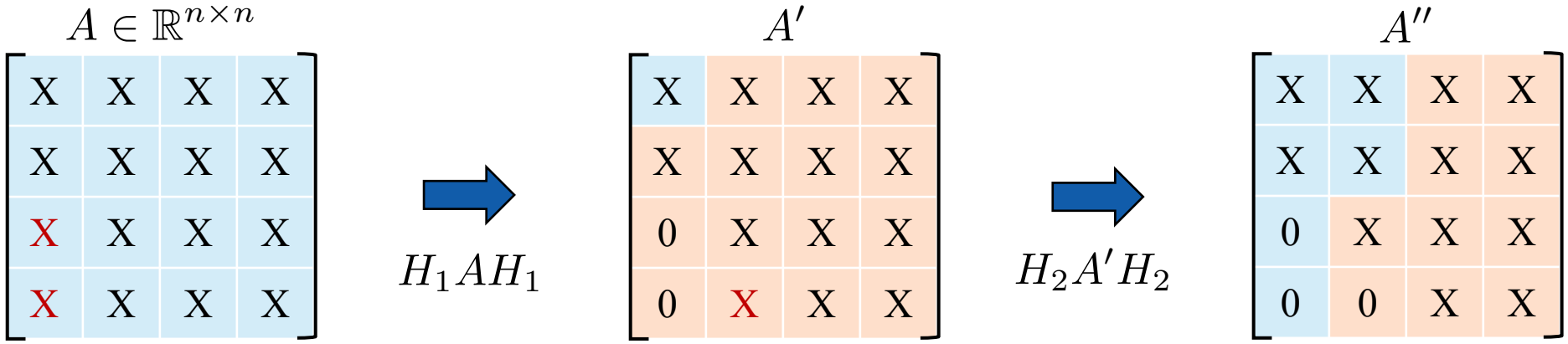
$$P_0 = I, \quad P_k = P_{k-1} M_k \quad \Rightarrow \quad P_k^T P_k = M_k^T D_{k-1} M_k = D_k$$

When applying such scaled form, we don't need to compute sqrt in each step!

- Maintain the transformation in a scaled form during elimination and only compute the actual rotation

Making A Matrix Upper Hessenberg

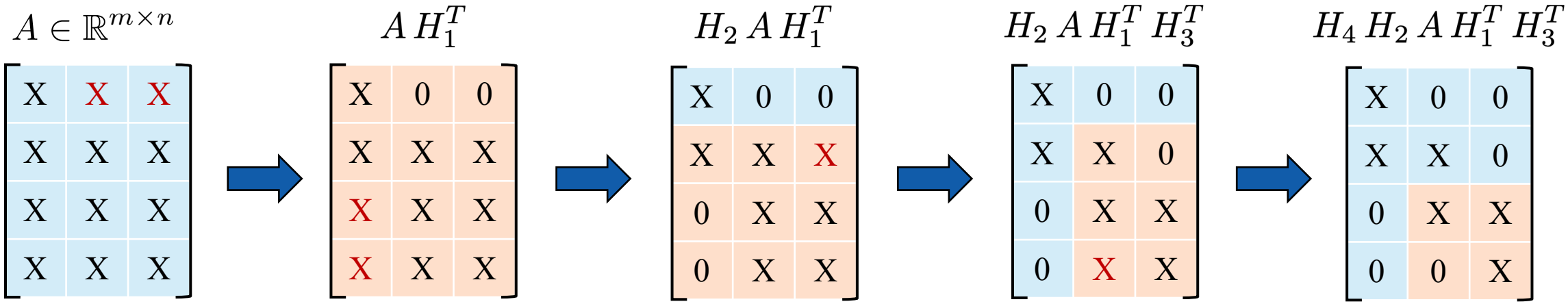
Matrix Hessenbergization is a fundamental preprocessing technique that transforms a general square matrix into an upper Hessenberg form through orthogonal similarity transformations. This creates an almost triangular matrix that preserves the original matrix's eigenvalues.



$$\tilde{H}_1 A(2:n,1) = \begin{bmatrix} \mathrm{X} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \Rightarrow \quad H_1 := \begin{bmatrix} 1 & 0 \\ 0 & \tilde{H}_1 \end{bmatrix}$$

Making A Matrix Bidiagonal

Matrix bidiagonalization is another crucial preprocessing technique that reduces a rectangular matrix to a bidiagonal form through orthogonal transformations. This reduction is achieved through alternating Householder reflections from the left and right, making it numerically stable and efficient.



Remarks:

- This bidiagonalization algorithm is numerically stable by applying Householder transformations.
- It is very useful in various methods, such as LSQR and SVD.

Method of Minimal Iteration

- In 1950, Lanczos introduced a revolutionary approach for the eigenvalue problem:

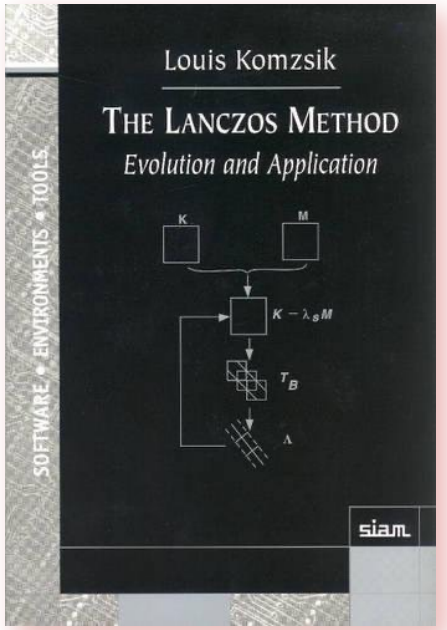
  $A\boldsymbol{v} = \lambda \boldsymbol{v}$, where $A \in \mathbb{R}^{n\times n}$. Assume that A is symmetric for the moment.

  Given an initial guess $\boldsymbol{v}_0 \neq \mathbf{0}$

$$\boldsymbol{v}_1 = A\,\boldsymbol{v}_0 - \alpha_0 \boldsymbol{v}_0 \qquad \textbf{residual for EV}$$

$$\Rightarrow \quad \alpha_0 = \arg\min_{\alpha} \left\| A\,\boldsymbol{v}_0 - \alpha \boldsymbol{v}_0 \right\|_2^2 \qquad \text{minimize residual}$$

$$\Rightarrow \quad \alpha_0 = \frac{(\boldsymbol{v}_0, A\,\boldsymbol{v}_0)}{(\boldsymbol{v}_0, \boldsymbol{v}_0)} = \frac{\boldsymbol{v}_0^T A\,\boldsymbol{v}_0}{\boldsymbol{v}_0^T \boldsymbol{v}_0} \qquad \text{orthogonal projection}$$



- We immediately notice the following orthogonality condition:

$$(\boldsymbol{v}_1, \boldsymbol{v}_0) = (A\,\boldsymbol{v}_0 - \alpha_0 \boldsymbol{v}_0, \boldsymbol{v}_0) = 0 \qquad (1)$$

Ref: Lanczos, Cornelius. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators." Journal of research of the National Bureau of Standards 45.4 (1950): 255-282.

Lanczos Method

- We then continue this procedure by minimizing residuals at each iteration:

$$\boldsymbol{v}_2 = A\boldsymbol{v}_1 - \alpha_1\boldsymbol{v}_1 \quad \Rightarrow \quad \boldsymbol{v}_2 = A\boldsymbol{v}_1 - \alpha_1\boldsymbol{v}_1 - \beta_0\boldsymbol{v}_0 \qquad (2)$$

It is NOT a good idea to forget the previous directions

Using the linear combination of the previous directions

$$\Rightarrow \quad \|A\boldsymbol{v}_1 - \alpha\boldsymbol{v}_1 - \beta\boldsymbol{v}_0\|_2^2 = (A\boldsymbol{v}_1, A\boldsymbol{v}_1) - 2\alpha(\boldsymbol{v}_1, A\boldsymbol{v}_1) + \alpha^2(\boldsymbol{v}_1, \boldsymbol{v}_1)$$

$$- 2\beta(\boldsymbol{v}_0, A\boldsymbol{v}_1) + \beta^2(\boldsymbol{v}_0, \boldsymbol{v}_0) + 2\alpha\beta(\boldsymbol{v}_1, \boldsymbol{v}_0) \qquad (3)$$

orthogonality

$$\Rightarrow \quad \alpha_1 = \frac{(\boldsymbol{v}_1, A\boldsymbol{v}_1)}{(\boldsymbol{v}_1, \boldsymbol{v}_1)} \quad \text{and} \quad \beta_0 = \frac{(\boldsymbol{v}_0, A\boldsymbol{v}_1)}{(\boldsymbol{v}_0, \boldsymbol{v}_0)} = \frac{(\boldsymbol{v}_1 + \alpha_0\boldsymbol{v}_0)^T\boldsymbol{v}_1}{(\boldsymbol{v}_0, \boldsymbol{v}_0)} = \frac{\|\boldsymbol{v}_1\|_2^2}{\|\boldsymbol{v}_0\|_2^2} \geq 0$$

Orthogonality Conditions

- It is easy to notice that the following orthogonality conditions:

$$(\boldsymbol{v}_2, \boldsymbol{v}_1) = (A\,\boldsymbol{v}_1 - \alpha_1 \boldsymbol{v}_1, \boldsymbol{v}_1) = 0$$

$$(\boldsymbol{v}_2, \boldsymbol{v}_0) = (A\,\boldsymbol{v}_1 - \beta_0 \boldsymbol{v}_0, \boldsymbol{v}_0) = 0$$

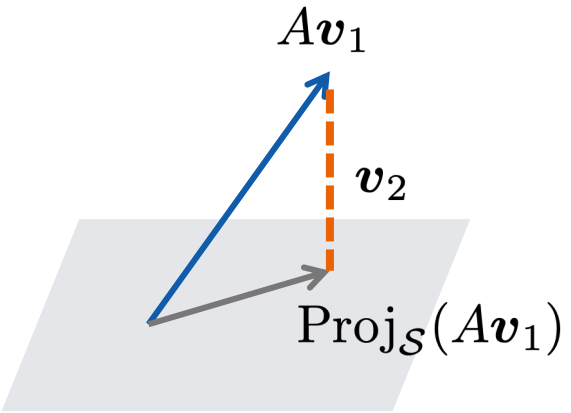
- Geometric interpretation:

$$\mathcal{S} := \mathrm{span}\{\boldsymbol{v}_0, \boldsymbol{v}_1\}$$

$$\Rightarrow \quad \mathrm{Proj}_{\mathcal{S}}(A\,\boldsymbol{v}_1) = \frac{(\boldsymbol{v}_0, A\,\boldsymbol{v}_1)}{(\boldsymbol{v}_0, \boldsymbol{v}_0)}\boldsymbol{v}_0 + \frac{(\boldsymbol{v}_1, A\,\boldsymbol{v}_1)}{(\boldsymbol{v}_1, \boldsymbol{v}_1)}\boldsymbol{v}_1 = \beta_0 \boldsymbol{v}_0 + \alpha_1 \boldsymbol{v}_1$$

$$\Rightarrow \quad \boldsymbol{v}_2 = A\,\boldsymbol{v}_1 - \alpha_1 \boldsymbol{v}_1 - \beta_0 \boldsymbol{v}_0 = A\,\boldsymbol{v}_1 - \mathrm{Proj}_{\mathcal{S}}(A\,\boldsymbol{v}_1) \perp \mathcal{S}$$

$$\Rightarrow \quad \textbf{The updated residual is orthogonal to the previous residuals!}$$



Short Recurrence Property

- **Q: How about the next iteration?**

$$\boldsymbol{v}_3 = A\,\boldsymbol{v}_2 - \alpha_2\boldsymbol{v}_2 - \beta_1\boldsymbol{v}_1 - \gamma_0\,\boldsymbol{v}_0$$

$$\Rightarrow \quad \gamma_0 = \frac{(\boldsymbol{v}_0, A\,\boldsymbol{v}_2)}{(\boldsymbol{v}_0, \boldsymbol{v}_0)}$$

- If we continue the procedure like this, we will need to save more and more vectors during iteration!
- Lanczos noticed that **only need to store the previous two iterations!**

$$(A\,\boldsymbol{v}_1, \boldsymbol{v}_0) = (\boldsymbol{v}_1, A\,\boldsymbol{v}_0) = (\boldsymbol{v}_1, \boldsymbol{v}_1 + \alpha_0\boldsymbol{v}_0) = (\boldsymbol{v}_1, \boldsymbol{v}_1)$$

$$(A\,\boldsymbol{v}_2, \boldsymbol{v}_1) = (\boldsymbol{v}_2, A\,\boldsymbol{v}_1) = (\boldsymbol{v}_2, \boldsymbol{v}_2 + \alpha_1\boldsymbol{v}_1 + \beta_0\boldsymbol{v}_0) = (\boldsymbol{v}_2, \boldsymbol{v}_2)$$

$$\Rightarrow \quad \beta_i \geq 0$$

$$(A\,\boldsymbol{v}_2, \boldsymbol{v}_0) = (\boldsymbol{v}_2, A\,\boldsymbol{v}_0) = (\boldsymbol{v}_2, \boldsymbol{v}_1 + \alpha_0\boldsymbol{v}_0) = 0 \quad \Rightarrow \quad \gamma_0 = 0$$

orthogonality

4

Lanczos and Matrix Polynomials

- Denote the previously generated, unnormalized vectors as follows

$$\begin{aligned}
\boldsymbol{z}_1 &= A\boldsymbol{z}_0 - \alpha_0\boldsymbol{z}_0 = (A-\alpha_0 I)\boldsymbol{z}_0 \\
\boldsymbol{z}_2 &= A\boldsymbol{z}_1 - \alpha_1\boldsymbol{z}_1 - \beta_0\boldsymbol{z}_0 = (A-\alpha_1 I)\boldsymbol{z}_1 - \beta_0\boldsymbol{z}_0 \\
\boldsymbol{z}_3 &= A\boldsymbol{z}_2 - \alpha_2\boldsymbol{z}_2 - \beta_1\boldsymbol{z}_1 = (A-\alpha_2 I)\boldsymbol{z}_2 - \beta_1\boldsymbol{z}_1 \\
&\vdots
\end{aligned}
\qquad
\begin{aligned}
p_1(t) &= t-\alpha_0 \qquad p_0(t) = 1 \\
p_2(t) &= (t-\alpha_1)p_1 - \beta_0 p_0 \\
p_3(t) &= (t-\alpha_2)p_2 - \beta_1 p_1
\end{aligned}$$

- To summarize, we have the matrix polynomial form of the Lanczos iteration:

$$\boldsymbol{z}_n = A\boldsymbol{z}_{n-1} - \alpha_{n-1}\boldsymbol{z}_{n-1} - \beta_{n-2}\boldsymbol{z}_{n-2} = (A-\alpha_{n-1}I)\boldsymbol{z}_{n-1} - \beta_{n-2}\boldsymbol{z}_{n-2}$$

$$\Rightarrow \quad p_n(t) = (t-\alpha_{n-1})p_{n-1}(t) - \beta_{n-2}p_{n-2}(t) \quad \Rightarrow \quad \boldsymbol{z}_n = p_n(A)\boldsymbol{z}_0$$

$$A\boldsymbol{z}_{j-1} = \boldsymbol{z}_j + \alpha_{j-1}\boldsymbol{z}_{j-1} + \beta_{j-2}\boldsymbol{z}_{j-2} \quad \Rightarrow \quad A\boldsymbol{v}_j = \sqrt{\beta_{j-2}}\boldsymbol{v}_{j-1} + \alpha_{j-1}\boldsymbol{v}_j + \sqrt{\beta_{j-1}}\boldsymbol{v}_{j+1}$$

$$\boldsymbol{v}_1 := \boldsymbol{z}_0/\|\boldsymbol{z}_0\|_2, \quad \boldsymbol{v}_{j+1} := \boldsymbol{z}_j/(\|\boldsymbol{z}_0\|_2\sqrt{\beta_0\beta_1\cdots\beta_{j-1}}) \qquad (j \geq 2)$$

Lanczos Iteration

- From now on, we use normalized Lanczos vectors indexed from 1, and redefine the coefficients

Lanczos Algorithm:

Given $\beta_0 = 0,\ \boldsymbol{v}_0 = \boldsymbol{0}, \|\boldsymbol{v}_1\|_2 = 1;$

for $j = 1, 2, \ldots$

$\boldsymbol{w}_j = A\boldsymbol{v}_j - \beta_{j-1}\boldsymbol{v}_{j-1};$

$\alpha_j = \boldsymbol{v}_j^T \boldsymbol{w}_j;$

$\boldsymbol{w}_j = \boldsymbol{w}_j - \alpha_j \boldsymbol{v}_j;$

$\beta_j = \|\boldsymbol{w}_j\|_2;$

if $\beta_j == 0$, stop;

$\boldsymbol{v}_{j+1} = \boldsymbol{w}_j / \beta_j;$

end

$$A\,\boldsymbol{v}_j = \beta_{j-1}\boldsymbol{v}_{j-1} + \alpha_j \boldsymbol{v}_j + \beta_j \boldsymbol{v}_{j+1}$$

$$\Rightarrow \quad A\,V_j = V_j\,T_j + \beta_j\,\boldsymbol{v}_{j+1}\boldsymbol{e}_j^T$$

Q: What happens if $\beta_m = 0$?

$$A\,V_m = V_m\,T_m$$

$$A(V_m \boldsymbol{z}) = V_m(T_m \boldsymbol{z}) \qquad \textbf{Invariant Subspace}$$

On the other hand, if no earlier breakdown occurs, $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ form an orthonormal basis of $\mathbb{R}^n$

$$\boldsymbol{w}_n = A\boldsymbol{v}_n - \alpha_n \boldsymbol{v}_n - \beta_{n-1}\boldsymbol{v}_{n-1}$$

$$(\boldsymbol{w}_n, \boldsymbol{v}_i) = 0, \quad i = 1, \ldots, n \quad \Rightarrow \quad \beta_n \equiv 0$$

Lucky Breakdown and Finite Termination

- Early breakdown is lucky
  - If $\beta_m = 0$, it is not a failure; in this case, every eigenpair of T_m yields an eigenpair of A.
  - This is called a **lucky breakdown**, namely, the iteration terminates successfully.
  - If the desired eigenpairs have already been found, stop; otherwise, just restart with another unit vector orthogonal to all previously generated Lanczos vectors.
- Finite termination (in exact arithmetic)

$$A V_n = V_n T_n, \quad \beta_n = 0 \quad \Rightarrow \quad V_n^T A V_n = T_n = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

If no earlier breakdown occurs, the procedure finishes at step *n*

Approximating Eigenpairs

- Before termination, the Lanczos relation gives

$$AV_m = V_m T_m + \beta_m \boldsymbol{v}_{m+1} \boldsymbol{e}_m^T, \quad \boldsymbol{e}_m \in \mathbb{R}^m \quad \Rightarrow \quad T_m \boldsymbol{y}_i = \theta_i \boldsymbol{y}_i, \quad \|\boldsymbol{y}_i\|_2 = 1$$

Ritz value: Eigenvalues of T_m

- Construct an approximate eigenvector

$$\boldsymbol{u}_i = V_m \boldsymbol{y}_i, \quad \|\boldsymbol{u}_i\|_2 = 1$$

$$\boldsymbol{r}_i := A\boldsymbol{u}_i - \theta_i \boldsymbol{u}_i = \beta_m \boldsymbol{v}_{m+1} (\boldsymbol{e}_m^T \boldsymbol{y}_i) \quad \Rightarrow \quad \|\boldsymbol{r}_i\|_2 = \beta_m |y_{i,m}|$$

- A real symmetric matrix has an orthonormal eigenbasis. Hence we have

$$A\boldsymbol{q}_j = \lambda_j \boldsymbol{q}_j, \quad \boldsymbol{u}_i = \sum_{j=1}^n c_j \boldsymbol{q}_j \quad \Rightarrow \quad \sum_{j=1}^n |c_j|^2 = 1$$

$$\boldsymbol{r}_i = (A - \theta_i I)\boldsymbol{u}_i = \sum_{j=1}^n c_j (\lambda_j - \theta_i) \boldsymbol{q}_j \quad \Rightarrow \quad \min_{\lambda \in \lambda(A)} |\theta_i - \lambda| \le \|\boldsymbol{r}_i\|_2$$

Lanczos and Characteristic Polynomial

$$T_n = V_n^T A V_n$$

$$\det(\lambda I - A) = \det(\lambda I - T_n) \quad \Rightarrow \quad \lambda I_n - T_n = \begin{bmatrix} \lambda - \alpha_1 & -\beta_1 & & & \\ -\beta_1 & \lambda - \alpha_2 & -\beta_2 & & \\ & -\beta_2 & \ddots & \ddots & \\ & & \ddots & \ddots & -\beta_{n-1} \\ & & & -\beta_{n-1} & \lambda - \alpha_n \end{bmatrix}$$

$$p_n(\lambda) = (\lambda - \alpha_n)\det(\lambda I - T_{n-1}) + \beta_{n-1}\det(M_{n-1}) \qquad M_{n-1} := \begin{bmatrix} \lambda I_{n-2} - T_{n-2} & 0 \\ -\beta_{n-2}\boldsymbol{e}_{n-2}^T & -\beta_{n-1} \end{bmatrix}$$

$$\begin{cases} p_0(t) = 1, \quad p_1(t) = t - \alpha_1 \\ p_n(\lambda) = (\lambda - \alpha_n)p_{n-1}(\lambda) - \beta_{n-1}^2 p_{n-2}(\lambda) \end{cases}$$

In exact arithmetic, if no breakdown occurs before step n of the Lanczos method, then p_n is the characteristic polynomial of A.

Extension to Nonsymmetric Matrices

- For a nonsymmetric matrix, consider the right and left eigenvalue problems:

$$A\boldsymbol{x}_k = \lambda_k \boldsymbol{x}_k, \quad \boldsymbol{y}_k^T A = \lambda_k \boldsymbol{y}_k^T \quad (k = 1, 2, \ldots, n)$$

- For nonsymmetric problems, we can replace the previous single orthonormal Lanczos basis with two biorthogonal bases generated using A and A^T, respectively.

- Initialization

$$\boldsymbol{v}_0 = \boldsymbol{w}_0 = \boldsymbol{0} \qquad \text{and} \qquad \boldsymbol{w}_1^T \boldsymbol{v}_1 \neq 0$$

$$\boldsymbol{v}_1 \leftarrow \frac{\boldsymbol{v}_1}{\|\boldsymbol{v}_1\|_2}, \quad \boldsymbol{w}_1 \leftarrow \frac{\boldsymbol{w}_1}{\boldsymbol{w}_1^T \boldsymbol{v}_1}, \qquad \beta_0 = \gamma_0 = 0$$

- Construct two sets of vectors $\{\boldsymbol{v}_i\}$ and $\{\boldsymbol{w}_i\}$, that satisfy the **bi-orthogonality condition**

$$\boldsymbol{w}_i^T \boldsymbol{v}_j = \delta_{ij}, \quad 1 \leq i, j \leq m$$

Lanczos Method for Nonsymmetric Matrices

- General Lanczos method

for $(j = 1 : n)$

$\alpha_j = \boldsymbol{w}_j^T A\, \boldsymbol{v}_j;$

$\boldsymbol{p}_j = A\boldsymbol{v}_j - \alpha_j \boldsymbol{v}_j - \beta_{j-1}\boldsymbol{v}_{j-1};$

$\boldsymbol{q}_j = A^T \boldsymbol{w}_j - \alpha_j \boldsymbol{w}_j - \gamma_{j-1}\boldsymbol{w}_{j-1};$

$\delta_j = \boldsymbol{q}_j^T \boldsymbol{p}_j;$

if $\delta_j == 0$, stop;

$\gamma_j = \sqrt{|\delta_j|};$

$\beta_j = \delta_j / \gamma_j;$

$\boldsymbol{v}_{j+1} = \boldsymbol{p}_j / \gamma_j;$

$\boldsymbol{w}_{j+1} = \boldsymbol{q}_j / \beta_j;$

end

Assuming no breakdown occurs before step n,

prove using induction:

$$W_n^T A V_n = T_n = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \gamma_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{n-1} \\ & & \gamma_{n-1} & \alpha_n \end{bmatrix}$$

Exercise!

Historic Comments on Lanczos Methods

- 1950: The Lanczos iteration was published. Cornelius Lanczos was looking for a way to find the eigenvalues of large matrices (instead of thinking about solving linear systems) by reducing them to a tridiagonal form. This laid the groundwork for modern Krylov subspace methods.
- 1952: Magnus Hestenes and Eduard Stiefel published their landmark paper, "Methods of Conjugate Gradients for Solving Linear Systems." They were looking for an iterative way to solve systems of linear equations that would converge in at most n steps in exact arithmetic; see Chapter 7.
- Even though invented in the early 1950s, both methods were largely ignored for 20 years.
  - Lanczos suffered because, in finite-precision arithmetic, the vectors lost orthogonality quickly, making it seem "unstable".
  - CG initially attracted limited practical interest, because too much emphasis was placed on finite-termination. Interest revived when using it as an iterative method for large sparse SPD systems.

中国科学院大学

2026

数值代数

Numerical Algebra

Chensong Zhang, AMSS

https://lsec.cc.ac.cn/~zhangcs

Release version 2026.10.22