Mixed precision iterative refinement

Bastien Vieublé

29/01/2024

AMSS, Chinese Academy of Sciences

Who am I ?

2015
to 2019

Engineering school, ENSEEIHT

Computer science and applied math

Who am I ?

2015 to 2019

Engineering school, ENSEEIHT

Computer science and applied math

Apr. 2019 to Sep. 2019

Internship, Chinese Academy of Sciences (AMSS)

Machine learning

Who am I ?

2015 to 2019 — **Engineering school**, ENSEEIHT

Computer science and applied math

Apr. 2019 to Sep. 2019 — **Internship**, Chinese Academy of Sciences (AMSS)

Machine learning

Oct. 2019 to Oct. 2022 — **PhD student**, IRIT

Numerical analysis and high performance computing

Who am I ?

2015 to 2019 — **Engineering school**, ENSEEIHT
Computer science and applied math

Apr. 2019 to Sep. 2019 — **Internship**, Chinese Academy of Sciences (AMSS)
Machine learning

Oct. 2019 to Oct. 2022 — **PhD student**, IRIT
Numerical analysis and high performance computing

Nov. 2022 to Nov. 2023 — **Post-doc**, University of Manchester
Numerical analysis

Who am I ?

2015 to 2019 — **Engineering school**, ENSEEIHT
Computer science and applied math

Apr. 2019 to Sep. 2019 — **Internship**, Chinese Academy of Sciences (AMSS)
Machine learning

Oct. 2019 to Oct. 2022 — **PhD student**, IRIT
Numerical analysis and high performance computing

Nov. 2022 to Nov. 2023 — **Post-doc**, University of Manchester
Numerical analysis

Dec. 2023 to Now — **Post-doc**, Chinese Academy of Sciences (AMSS)

On solving linear systems

Linear systems and physical applications

$$Ax = b,$$
$$A \in \mathbb{R}^{n \times n}, \quad b \in \mathbb{R}^n, \quad x \in \mathbb{R}^n$$

Linear systems and physical applications

$$Ax = b,$$
$$A \in \mathbb{R}^{n \times n}, \quad b \in \mathbb{R}^n, \quad x \in \mathbb{R}^n$$

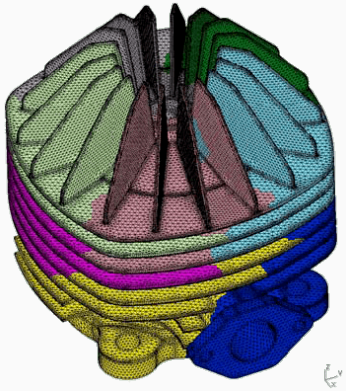
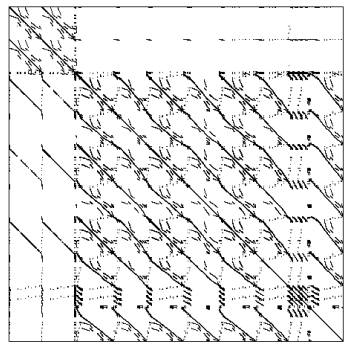
Physicists are extremely good clients of large scale parallel linear solvers!

This is because they describe many of their problems in term of differential equations. Generally, an analytic solution is only available for the simplest cases. Therefore, solving their differential equations often involve the discretization of the problem to transform it into a linear system.

They are generally not much aware of the numerical difficulties, the parallelism, the computer hardware, or the different algorithms to solve their linear systems.

$\Rightarrow$ **This is our job!**

Fat problems require fat computers!


Large-scale linear systems...

Up to billions of unknowns, applications demanding enormous amount of memory and flops of computation.

For a dense problem of size $n = 10^7$, storing the matrix requires TBytes of memory, factorizing the matrix requires Exaflops of computation!

...require large-scale computers.

Increasingly large numbers of cores available, high heterogeneity in the computation (CPU, GPU, FPGA, TPU, etc), and high heterogeneity in data motions (RAM to cache, disk to RAM, node to node, etc).

Two main kinds of solvers

What are the ways to solve a sparse $Ax = b \in \mathbb{R}^n$ on computers?

Iterative solvers

Compute a sequence of x_k converging towards x.

Examples: Gauss-Seidel, SOR, Krylov subspace methods, etc.

- ➤ Low computational cost and memory consumption if the convergence is quick...
- ➤ BUT convergence depends on the matrix properties.

Direct solvers

Based on a factorization of A.

Examples: LDL^T, LU, QR, etc.

- ➤ High computational cost and memory consumption...
- ➤ BUT they are robust and easy to use.

Two main kinds of solvers

What are the ways to solve a sparse $Ax = b \in \mathbb{R}^n$ on computers?

Iterative solvers

Compute a sequence of x_k converging towards x.

Examples: Gauss-Seidel, SOR, Krylov subspace methods, etc.

- ➤ Low computational cost and memory consumption if the convergence is quick...
- ➤ BUT convergence depends on the matrix properties.

Direct solvers

Based on a factorization of A.

Examples: LDL^T, LU, QR, etc.

- ➤ High computational cost and memory consumption...
- ➤ BUT they are robust and easy to use.

$\Rightarrow$ For both, the reduction of the computational cost is the focus of much research.

Reduce the cost by reducing the complexity

Approximate computing: deliberately approximate the computations in order to improve the performance at the cost of introducing a perturbation.

- The perturbed problem should be close to the original one and should reduce time and/or memory!
- In general the larger the perturbations the larger the savings...
- BUT large perturbations = low accuracy!

In this course, we focus on one particular kind of approximate computing techniques: the employment low precision arithmetics.

Low precision arithmetics

Introduction to floating point numbers

Floating point format

Main format for representing real numbers in computers. A number is of the form:

$$x = \pm m \times \beta^{e}$$

Base β (usually 2).

$\pm$ is the bit of sign.

The **mantissa** m is an integer represented in base β.

The **exponent** e is an integer represented in base β.

- The mantissa carries the **significant digits** (i.e., how accurate can be the numbers).
- The exponent carries the **range** (i.e., how far is the lowest and highest representable number).

Examples: $1.4723 = + \underbrace{14723}_{\text{mantissa}} \times \underbrace{10}_{\beta}^{\overbrace{-4}^{\text{exponent}}}$, $-92 = - \underbrace{010111}_{23} \times \underbrace{2}_{\beta}^{\overbrace{0010}^{2}}$

Introduction to floating point numbers

Floating point format

Main format for representing real numbers in computers. A number is of the form:

$$x = \pm m \times \beta^{e}$$

Base β (usually 2).

$\pm$ is the bit of sign.

The **mantissa** m is an integer represented in base β.

The **exponent** e is an integer represented in base β.

The **unit roundoff** u determines the relative accuracy any number in the representable range $[e_{\min}, e_{\max}]$ can be approximated with:

$$\forall x \in [e_{\min}, e_{\max}] \subset \mathbb{R}, \quad \mathrm{fl}(x) = x(1+\delta), \quad |\delta| \leq u,$$

where $\mathrm{fl}(x)$ is the floating point representation of a real number x, and δ the relative difference between this representation and x.

More notation

The condition number of a matrix is a measure of its *"numerical difficulty"*. We will use the two quantities

$$\kappa(A) = \|A\|\|A^{-1}\|, \qquad \mathbf{cond}(A) = \||A||A^{-1}|\|.$$

Suppose we are solving a linear system $Ax = b$ with a computer, due to the finite representation of numbers and accumulation of errors we cannot provide the true x, we then call the computed solution $\hat{x}$.

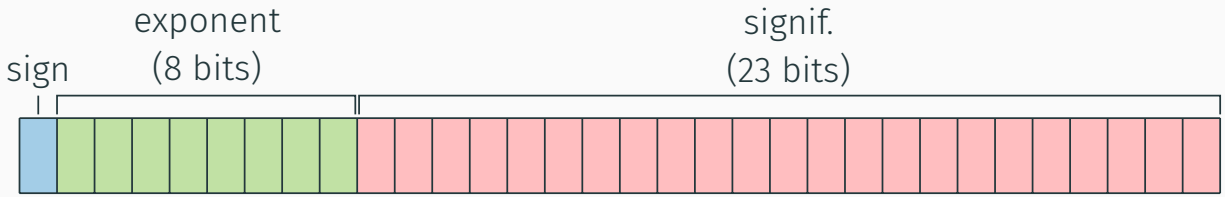
We define the (relative) forward error of our computed solution as

$$fwd = \frac{\|\hat{x} - x\|}{\|x\|}.$$

Commonly available floating point arithmetics

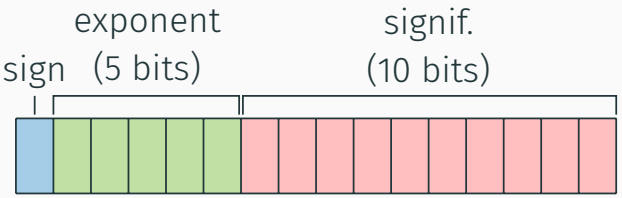
	ID	Signif. bits	Exp. bits	Range	Unit roundoff u
fp128	Q	113	15	$10^{\pm 4932}$	1×10^{-34}
double-fp64	DD	107	11	$10^{\pm 308}$	6×10^{-33}
fp64	D	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	S	24	8	$10^{\pm 38}$	6×10^{-8}
tfloat32	T	11	8	$10^{\pm 38}$	5×10^{-4}
fp16	H	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16	B	8	8	$10^{\pm 38}$	4×10^{-3}
fp8 (E4M3)	R	4	4	$10^{\pm 2}$	6.3×10^{-2}
fp8 (E5M2)	R*	3	5	$10^{\pm 5}$	1.3×10^{-1}

Low precision arithmetics



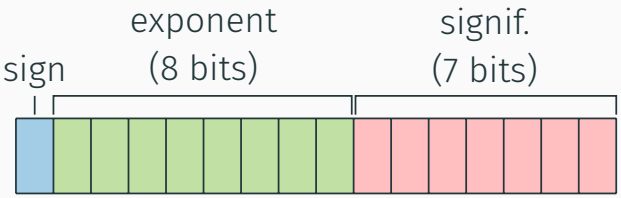
fp32

Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



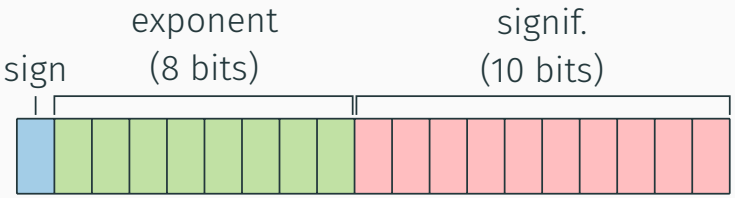
fp16

Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



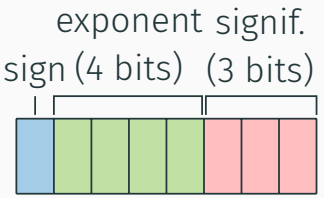
bfloat16

Range $10^{\pm 38}$, $u = 4 \times 10^{-3}$



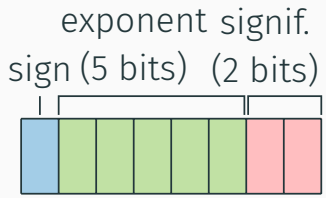
tfloat32

Range $10^{\pm 38}$, $u = 5 \times 10^{-4}$



fp8 (E4M3)

Range $10^{\pm 2}$, $u = 6 \times 10^{-2}$



fp8 (E5M2)

Range $10^{\pm 5}$, $u = 1 \times 10^{-1}$

Why using low precision arithmetics ?

Low precision arithmetics are less accurate and present a narrower range. BUT they have 3 main advantages:

- Storage, data movement and communications are all proportional to the total number of bits. $\Rightarrow$ Time and memory savings!
- Speed of computation is also at least proportional to the total number of bits. $\Rightarrow$ Time savings!
- Power consumption is dependent on the number of bits. $\Rightarrow$ Energy savings!

As reducing time, memory, and energy consumption are all challenging objectives for the ease of high performance computing, low arithmetic precisions has become the Wild West of HPC!

The fundamental dilemma of low precision arithmetics

Problem

➤ Low precision arithmetics can greatly improve performance of linear solvers...

➤ BUT they degrade their accuracy at the same time.

➤ Unfortunately application experts generally require high accuracy on the solution (i.e., most commonly double precision accuracy).

Idea: What if we could use low precisions to accelerate the most expensive parts of the computation, and use higher precision only on some strategic operations to recover the lost accuracy at low cost?

The fundamental dilemma of low precision arithmetics

Problem

➤ Low precision arithmetics can greatly improve performance of linear solvers...

➤ BUT they degrade their accuracy at the same time.

➤ Unfortunately application experts generally require high accuracy on the solution (i.e., most commonly double precision accuracy).

Idea: What if we could use low precisions to accelerate the most expensive parts of the computation, and use higher precision only on some strategic operations to recover the lost accuracy at low cost?

$\Rightarrow$ This is the goal of mixed precision algorithms!

Introduction to iterative refinement

Newton's method for correcting the solution of linear systems

Newton's method consists in building approximations $x_i \in \mathbb{R}^n$ converging toward a zero x of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$:

$$x_{i+1} = x_i - (\nabla f(x_i))^{-1} f(x_i).$$

Newton's method for correcting the solution of linear systems

Newton's method consists in building approximations $x_i \in \mathbb{R}^n$ converging toward a zero x of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$:

$$x_{i+1} = x_i - (\nabla f(x_i))^{-1} f(x_i).$$

We can correct the solution of linear systems by applying Newton's method to the residual $f(x) = Ax - b$. The procedure becomes

$$x_{i+1} = x_i + A^{-1}(b - Ax_i),$$

and can be decomposed into three steps

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Newton's method for correcting the solution of linear systems

Newton's method consists in building approximations $x_i \in \mathbb{R}^n$ converging toward a zero x of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$:

$$x_{i+1} = x_i - (\nabla f(x_i))^{-1} f(x_i).$$

We can correct the solution of linear systems by applying Newton's method to the residual $f(x) = Ax - b$. The procedure becomes

$$x_{i+1} = x_i + A^{-1}(b - Ax_i),$$

and can be decomposed into three steps

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Newton's method for the correction of linear systems is called iterative refinement.

On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

In exact arithmetic and without errors, the iterative refinement procedure gives the solution $x = A^{-1}b$ in one iteration!

On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

In exact arithmetic and without errors, the iterative refinement procedure gives the solution $x = A^{-1}b$ in one iteration!

However, on computers, every step is computed in inexact arithmetic, which lead to the presence of computing errors in every of these steps.

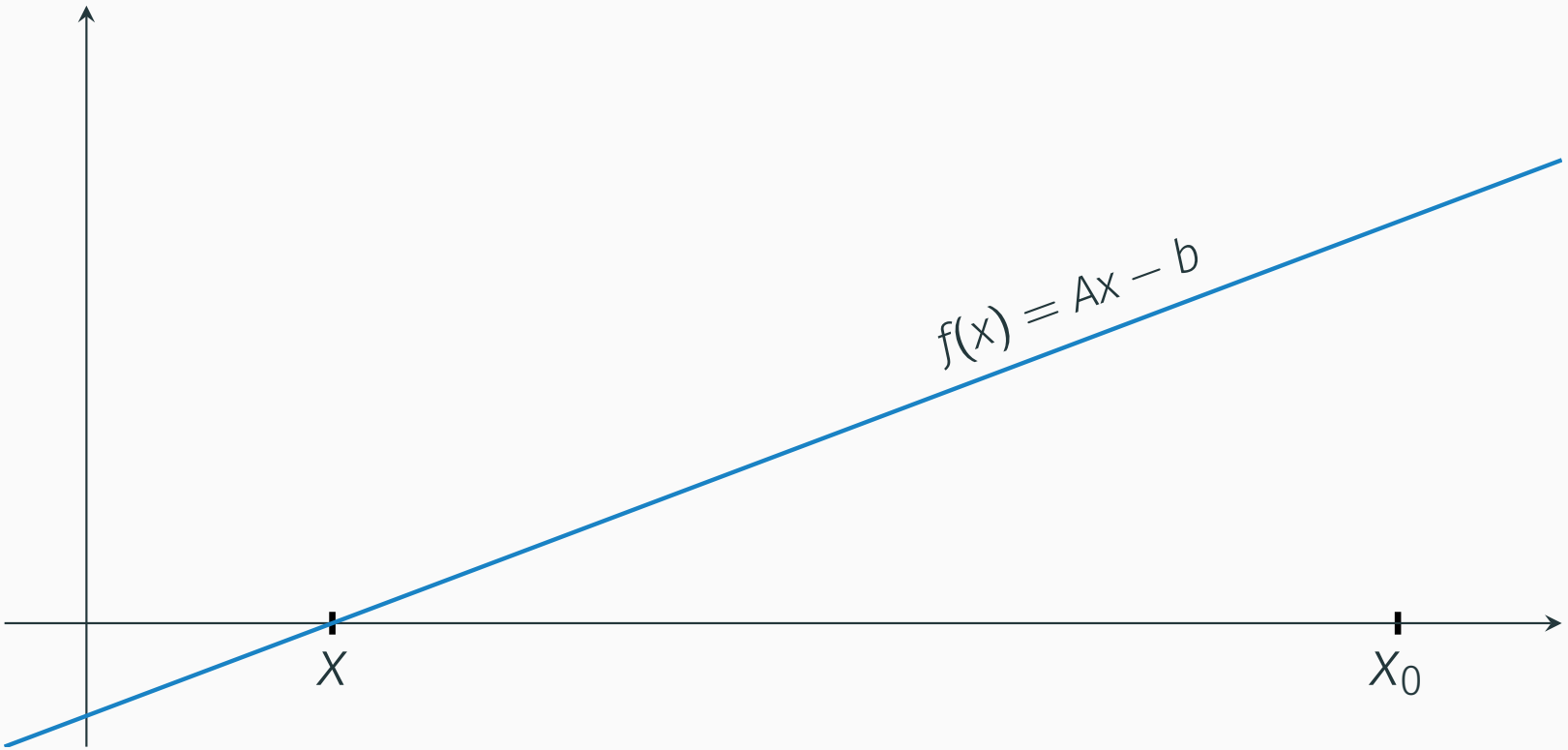
$\Rightarrow$ **What is the impact of these errors in the procedure ?**

On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

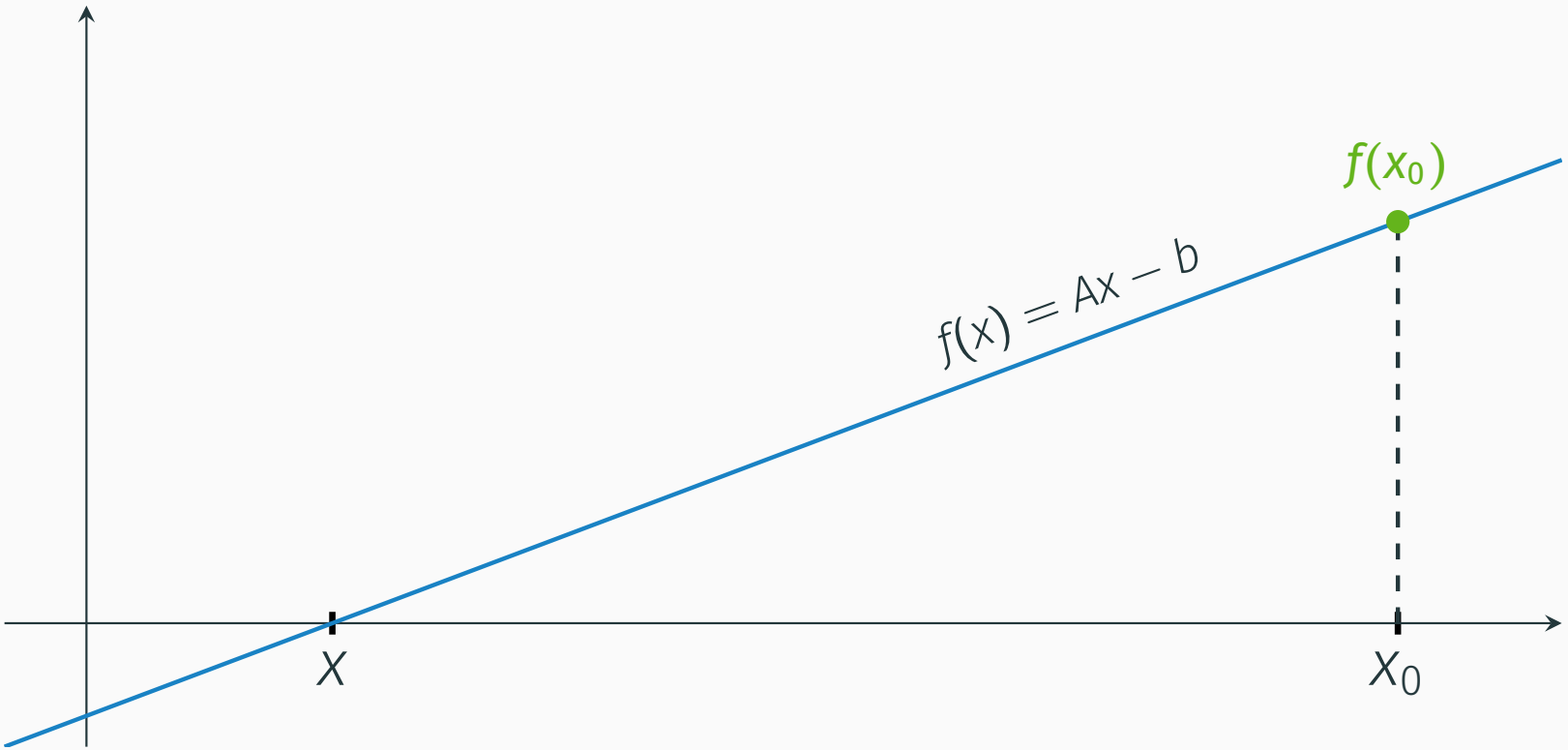


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

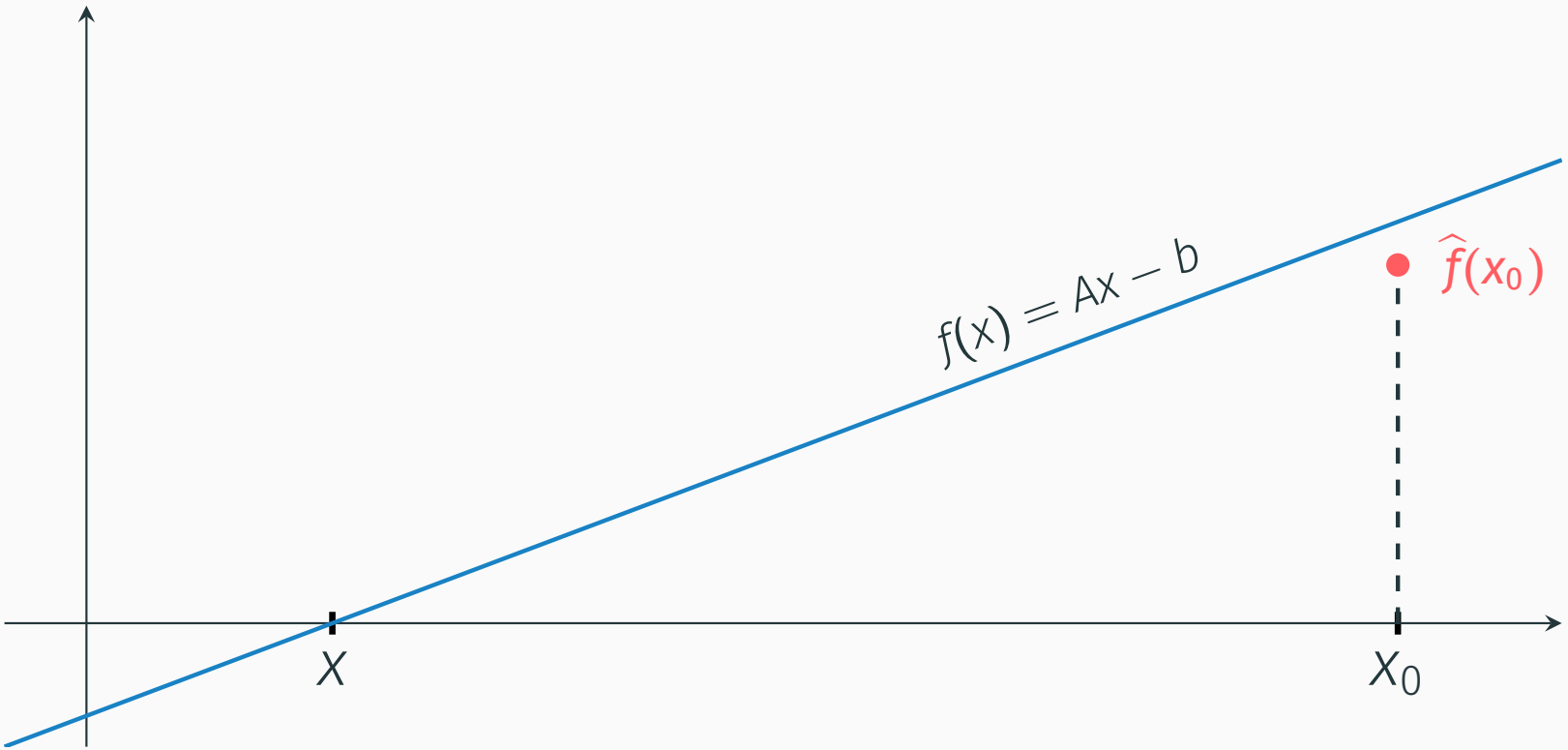


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

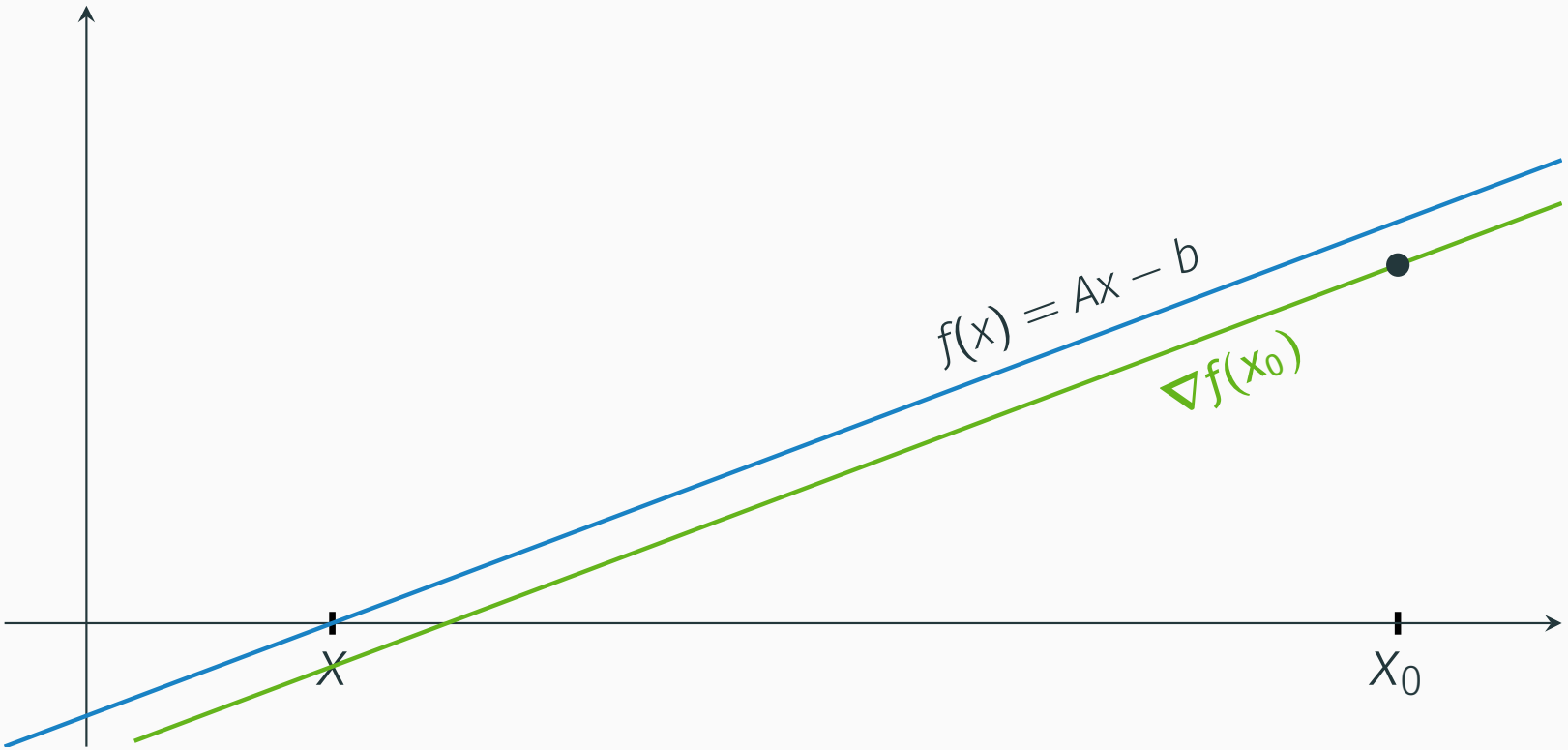


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

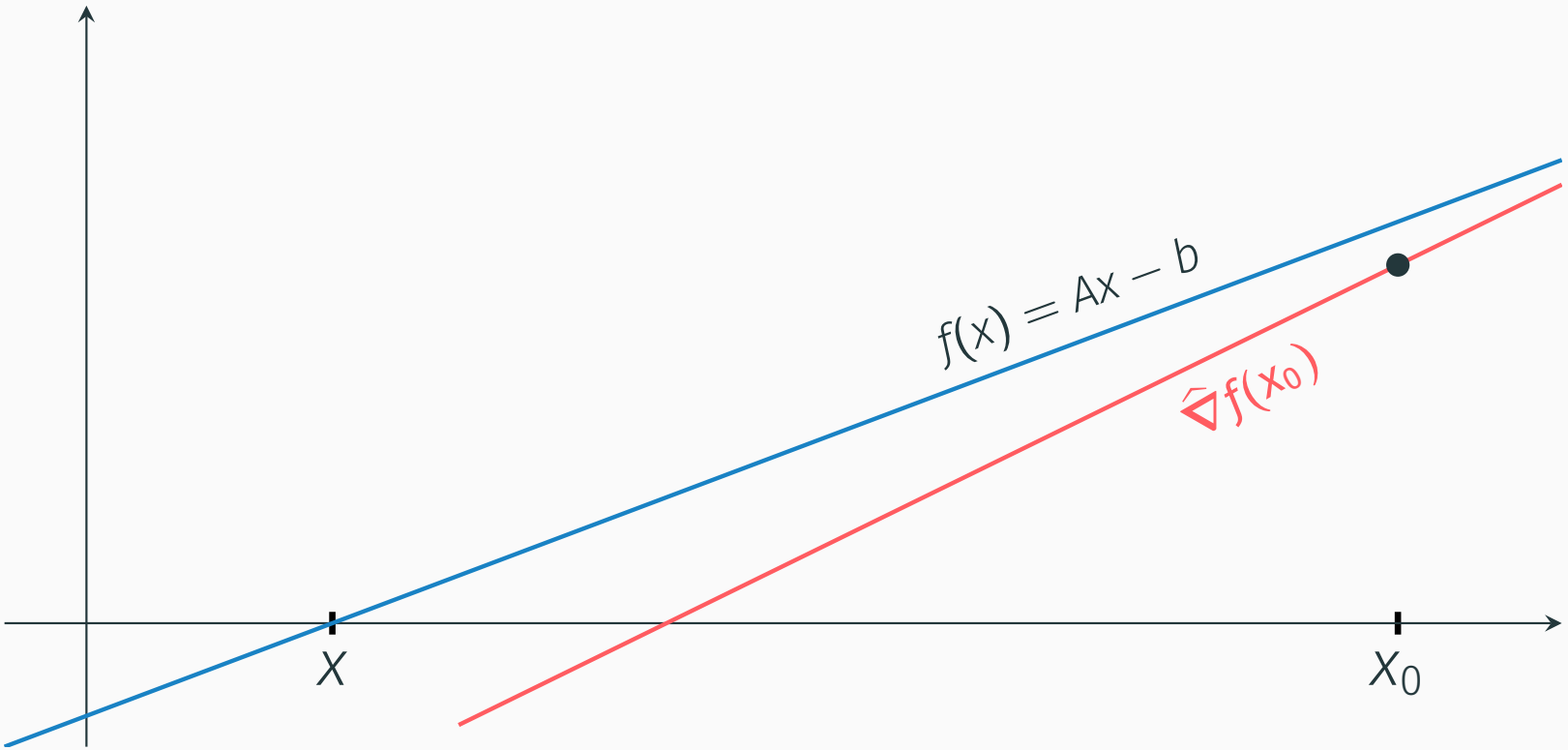


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

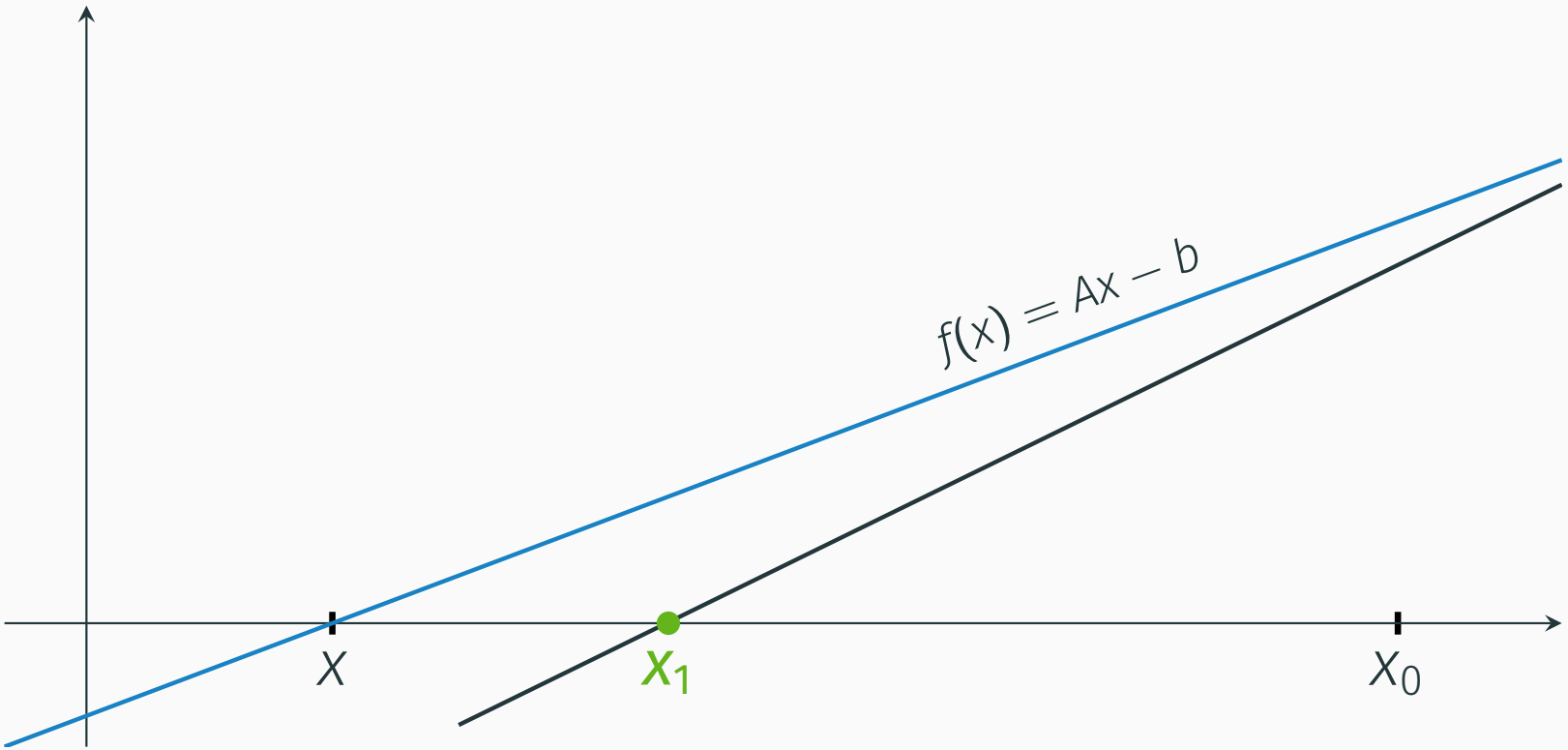


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

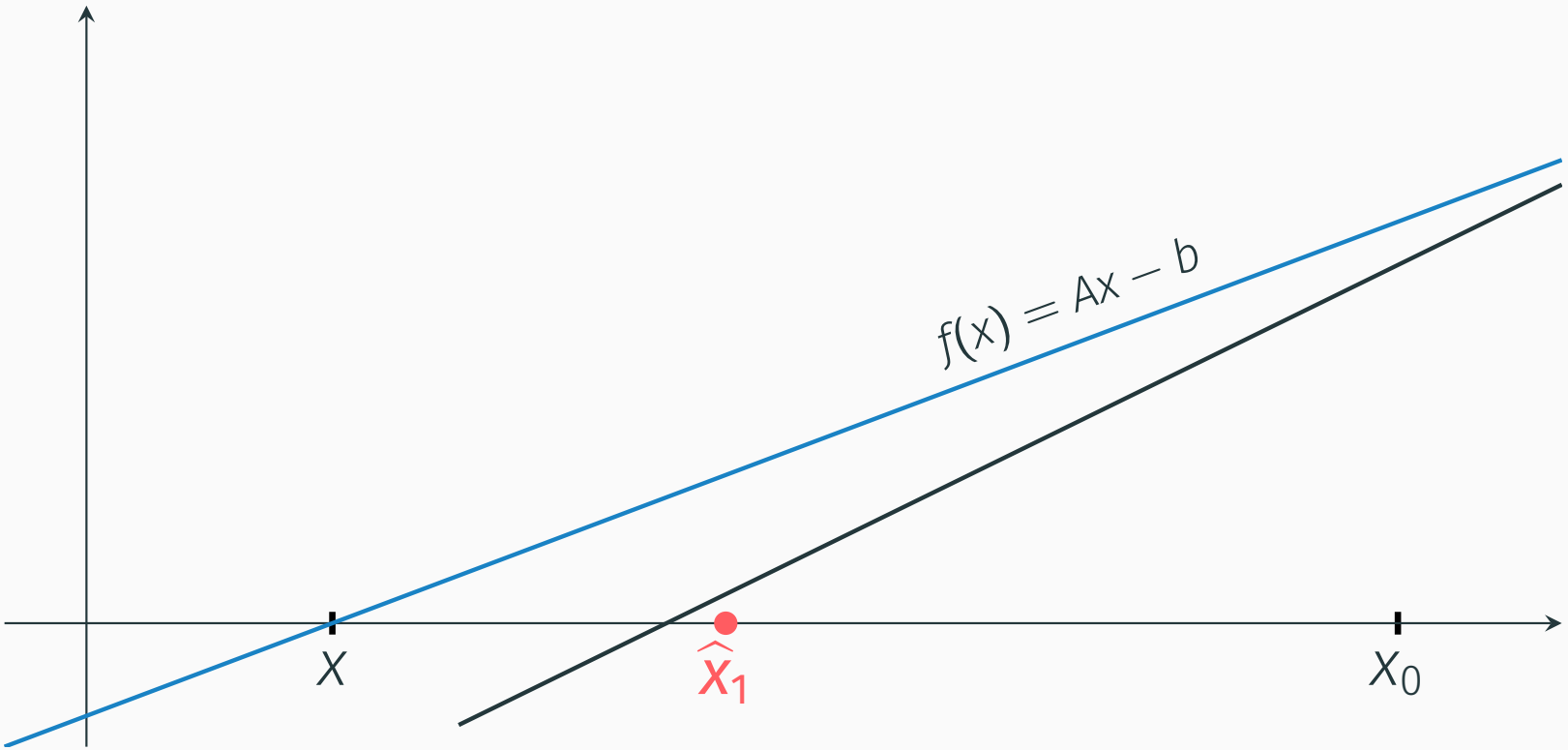


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

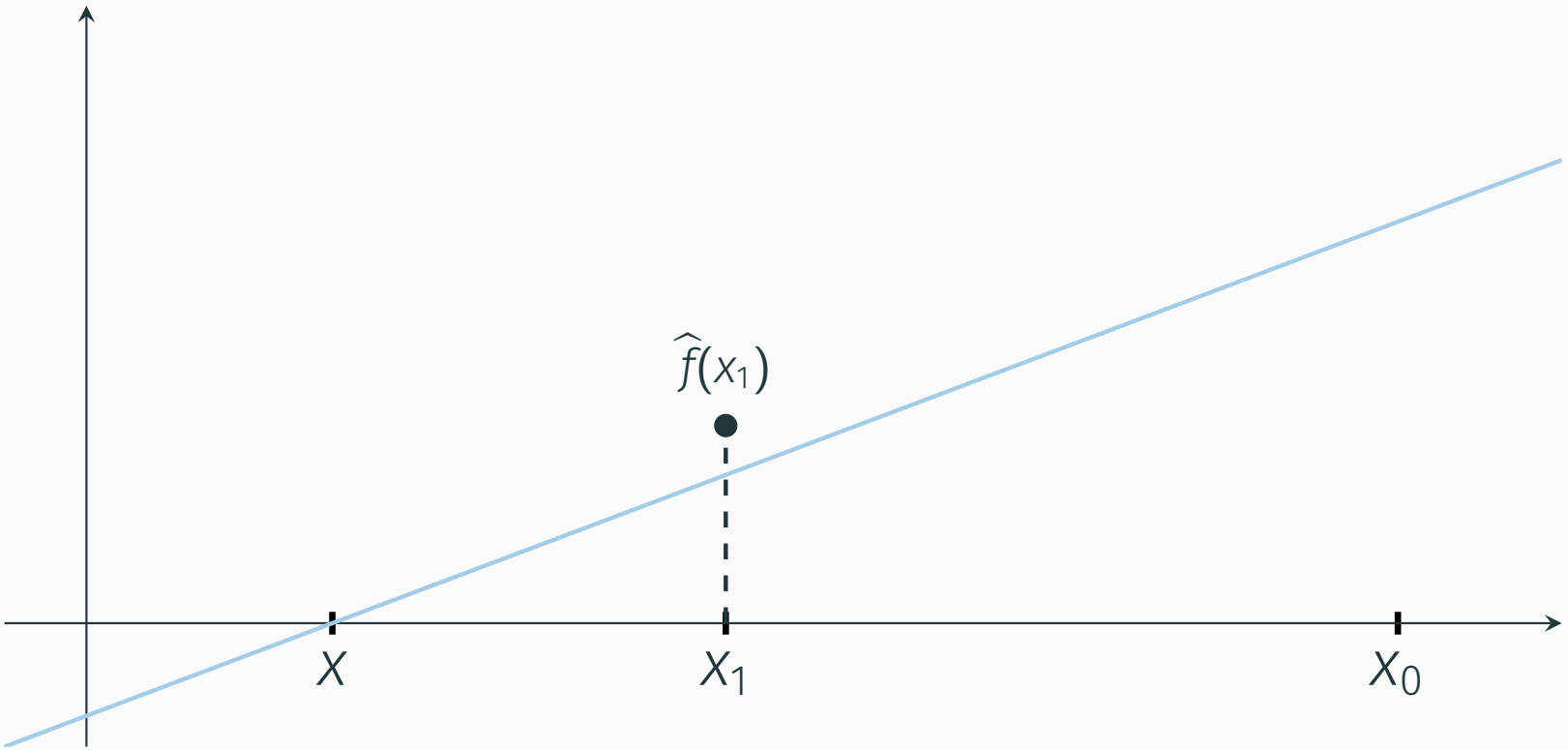


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

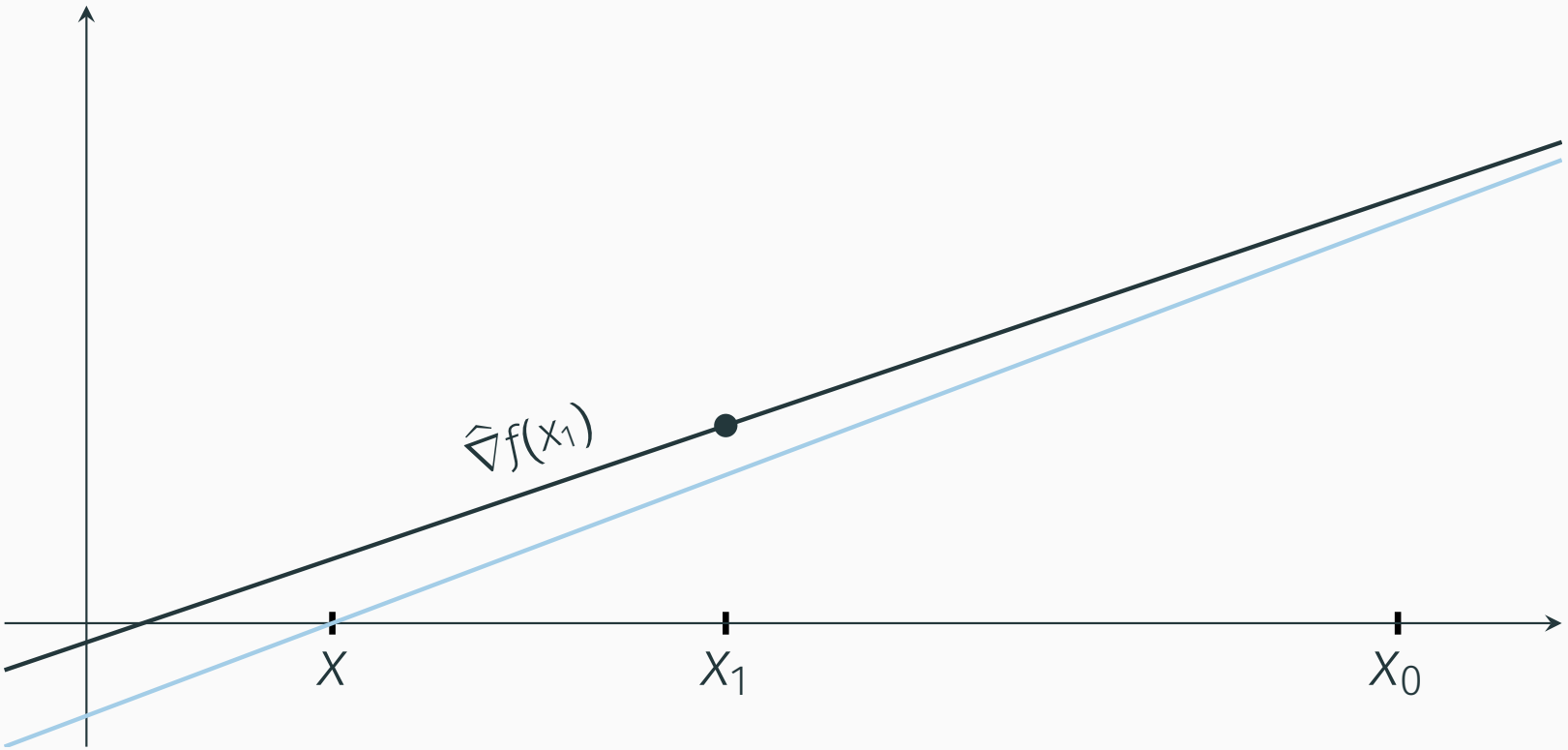


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

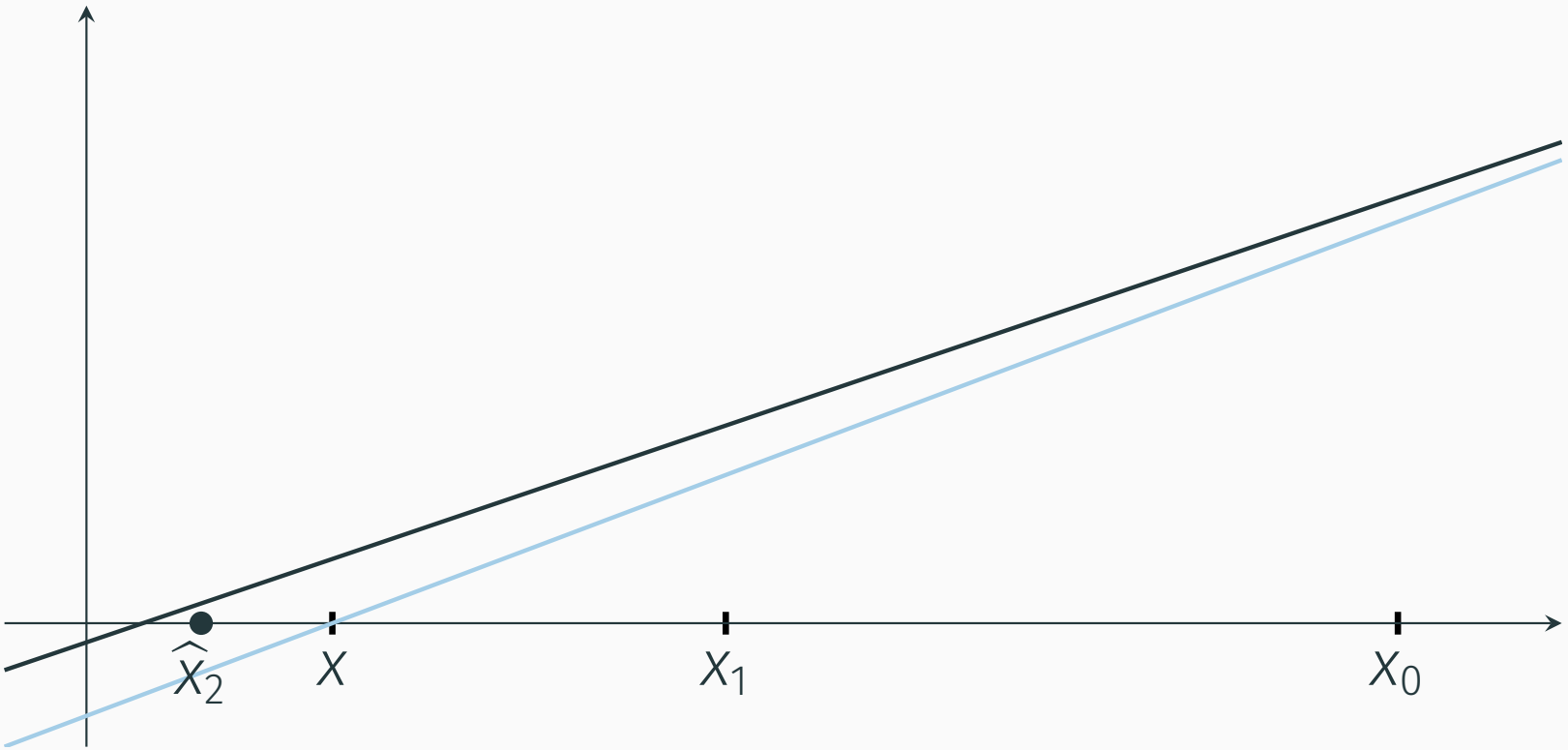


On the effect of rounding errors

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.



Historical perspectives

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Iterative refinement has been used for more than 70 years. It has constantly been evolving over time, repeatedly reconsidered according to trends, researcher's interests, and hardware specifications, as well as the computing challenges of each computing era.

Historical perspectives

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Iterative refinement has been used for more than 70 years. It has constantly been evolving over time, repeatedly reconsidered according to trends, researcher's interests, and hardware specifications, as well as the computing challenges of each computing era.

$\Rightarrow$ History can give us a better understanding of *why* and *how* we use this algorithm today!

From the 40s to the 70s

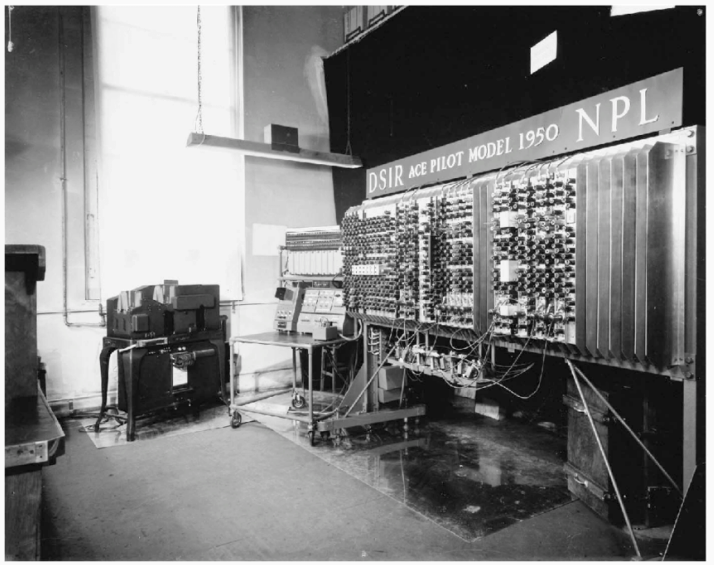
Origin

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Iterative refinement was implemented on the first computers!



Pilot ACE - One of the first computers, designed by Alan Turing.

"*Progress report on the Automatic Computing Engine*" by **James H. Wilkinson**, 1948.

Origin

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Iterative refinement was implemented on the first computers!

The fatherhood is generally attributed to James H. Wilkinson who first described, implemented, and reported the algorithm into a document.

"*Progress report on the Automatic Computing Engine*" by **James H. Wilkinson**, 1948.

James H. Wilkinson

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

1st element of context - Special hardware design of that time allowed costless accumulation of inner products in extra precision.

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

1st element of context - Special hardware design of that time allowed costless accumulation of inner products in extra precision.

$\Rightarrow$ The residual could be computed in **extra precision**, the correction equation and the update are computed in **working precision**. It removes the effect of the conditionning of the problem (i.e., $\kappa(A)$) on the error.

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

1st element of context - Special hardware design of that time allowed costless accumulation of inner products in extra precision.

$\Rightarrow$ The residual could be computed in **extra precision**, the correction equation and the update are computed in **working precision**. It removes the effect of the conditionning of the problem (i.e., $\kappa(A)$) on the error.

Mixed precision is not a new idea!

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

2nd element of context - Iterative refinement was focused on improving stable **direct solvers** (e.g., LU with partial pivoting)

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

2nd element of context - Iterative refinement was focused on improving stable **direct solvers** (e.g., LU with partial pivoting)

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

2nd element of context - Iterative refinement was focused on improving stable **direct solvers** (e.g., LU with partial pivoting) because

➤ Iterative solvers were not so trendy (less robust and reliable, and not particularly better in performance on low dimensional dense problems of this time).

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U\backslash L\backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

2nd element of context - Iterative refinement was focused on improving stable **direct solvers** (e.g., LU with partial pivoting) because

- ➤ Iterative solvers were not so trendy (less robust and reliable, and not particularly better in performance on low dimensional dense problems of this time).
- ➤ It is computationaly interesting: the factorization $A = LU$ ($O(n^3)$) can be computed once, and the solve $U\backslash L\backslash r_i$ ($O(n^2)$) can be applied multiple times $\Rightarrow$ **Refinement iterations are cheap!**

Two main pieces of context

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

All the studies of this period match this form of iterative refinement.

- "*Notes on the solution of algebraic linear simultaneous equations*" by **Leslie Fox et al.**, 1948.
- "*On the improvement of the solutions to a set of simultaneous linear equations using the ILLIAC*" by **James N. Snyder**, 1955.
- "*Note on the iterative refinement of least squares solution*" by **Gene H. Golub and James H. Wilkinson**, 1966.
- "*Solution of real and complex systems of linear equations*" by **Hilary J. Bowdler et al.**, 1966.
- "*Iterative refinement of linear least squares solutions I*" by **Åke Björck**, 1967.

Rounding error analyses

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

We have strong rounding error analysis results on iterative refinement from this period.

> *"Since [...] the numbers [...] are all to a certain extent approximate only, the results of all calculations performed by the aid of these numbers can only be approximately true ..."* **C. F. Gauss**, 1809.

Rounding error analyses

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

We have strong rounding error analysis results on iterative refinement from this period.

> *"Since [...] the numbers [...] are all to a certain extent approximate only, the results of all calculations performed by the aid of these numbers can only be approximately true ..."* **C. F. Gauss**, 1809.

⇒ **The role of rounding error analyses is to discover to what extent these calculations are approximately true!**

Rounding error analyses

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Rounding error analysis on iterative refinement aim at both:

- Highlight under **which conditions** the solution can be improved by refinement. This is referred to as **convergence conditions**.
- Demonstrate to **what accuracy** the solution will be refined. This is referred to as **limiting accuracies**.

Rounding error analyses

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

Rounding error analysis on iterative refinement aim at both:

- Highlight under **which conditions** the solution can be improved by refinement. This is referred to as **convergence conditions**.
- Demonstrate to **what accuracy** the solution will be refined. This is referred to as **limiting accuracies**.

The first two error analyses of iterative refinement were conducted by Wilkinson and Moler for, resp., fixed point and floating point arithmetics.

"*Rounding Errors in Algebraic Processes*" by **James H. Wilkinson**, 1963.

"*Iterative refinement in floating point*" by **Cleve B. Moler**, 1967.

From the 70s to the 2000s

Hardware and software changes

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Hardware and software changes reshaped the way we use iterative refinement:

Hardware and software changes

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Hardware and software changes reshaped the way we use iterative refinement:

➤ Accumulation of the inner products in extra precision was not widely available across machines anymore.

Hardware and software changes

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Hardware and software changes reshaped the way we use iterative refinement:

- Accumulation of the inner products in extra precision was not widely available across machines anymore.
- Single precision (fp32) was not substantially faster than double precision (fp64).

Hardware and software changes

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

Hardware and software changes reshaped the way we use iterative refinement:

- Accumulation of the inner products in extra precision was not widely available across machines anymore.
- Single precision (fp32) was not substantially faster than double precision (fp64).
- Iterative methods (CG, GMRES, etc.) were on the rise, and unstable direct methods were more and more considered to target parallel computing and sparse data structures.

Hardware and software changes

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Hardware and software changes reshaped the way we use iterative refinement:

- Accumulation of the inner products in **extra precision was not** widely **available** across machines anymore.
- **Single precision** (fp32) was **not** substantially **faster** than double precision (fp64).
- **Iterative methods** (CG, GMRES, etc.) were on the rise, and **unstable** direct methods were more and more considered to target parallel computing and sparse data structures.

$\Rightarrow$ For these reasons, a new form emerged where every operations are in the same precision: fixed precision iterative refinement!

Why is it relevant?

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

Common thought was that fixed precision iterative refinement is useless!

> *"In this case, x_m is often no more accurate than x_1."* **C. Moler**, 1967.

Why is it relevant?

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

Common thought was that fixed precision iterative refinement is useless!

> "*In this case, x_m is often no more accurate than x_1.*" **C. Moler**, 1967.

As the context has changed, this idea has been challenged by

"*Iterative refinement implies numerical stability*" by **Michal Jankowski and Henryk Woźniakowski**, 1977.

"*Iterative refinement implies numerical stability for gaussian elimination*" by **Robert D. Skeel**, 1980.

They showed that while fixed precision iterative refinement cannot correct (much) stable solvers, it can **transform an unstable solver into a stable one**.

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

$\Rightarrow$ **Any kind of linear solver can solve the correction equation $Ad_i = r_i$!**

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

$\Rightarrow$ **Any kind of linear solver can solve the correction equation $Ad_i = r_i$!**

With **Chebyshev iterations**:

"*Iterative refinement implies numerical stability*" by **Michal Jankowski and Henryk Woźniakowski**, 1977.

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

$\Rightarrow$ **Any kind of linear solver can solve the correction equation $Ad_i = r_i$!**

With **GMRES**:

"*Efficient High Accuracy Solutions with GMRES(m)*" by **Kathryn Turner and Homer F. Walker**, 1992.

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

$\Rightarrow$ **Any kind of linear solver can solve the correction equation $Ad_i = r_i$!**

With **LU and drop strategies**:

"*Use of Iterative Refinement in the Solution of Sparse Linear Systems*" by **Zahari Zlatev**, 1982.

Widen the range of solvers

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

From the 40s to the 70s, stable direct solvers were the standard. BUT from the 70s to the 2000s, a wider variety of linear solvers emerged.

$\Rightarrow$ **Any kind of linear solver can solve the correction equation $Ad_i = r_i$!**

With **LU and static pivoting**:

"*Making Sparse Gaussian Elimination Scalable by Static Pivoting*" by **Xiaoye S. Li and James W. Demmel**, 1998.

From the 2000s to the 2010s

The advent of low precision

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2000s, single precision became effectively $2\times$ **faster** in hardware than double precision. This major hardware change will **start the advent of low precision**!

PROBLEM:

> *"[...] the advantages of single precision performance in scientific computing did not come to fruition until recently, due to the fact that most scientific computing problems require double precision accuracy."* **A. Buttari et al.**, 2007.

The advent of low precision

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2000s, single precision became effectively $2\times$ **faster** in hardware than double precision. This major hardware change will **start the advent of low precision**!

PROBLEM:

> *"[...] the advantages of single precision performance in scientific computing did not come to fruition until recently, due to the fact that most scientific computing problems require double precision accuracy."* **A. Buttari et al.**, 2007.

SOLUTION: Iterative refinement to leverage the power of single precision while delivering double precision accuracy!

Get back the accuracy

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $d_i = U \backslash L \backslash r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Strategy to accelerate LU direct solver with single precision proposed in

"*Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)*" by **J. Langou et al.**, 2006.

Compute the **factorization** ($O(n^3)$) in single precision, and compute the **residual** and **update** ($O(n^2)$) in double precision to improve the accuracy of the solution.

As the **refinement** steps are asymptotically **negligible**, we can solve $Ax = b$ **twice faster** while providing **double precision accuracy**.

$\Rightarrow$ **Iterative refinement is used for performance!**

The irruption of accelerators

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

In the 2000s, accelerators[1] were more and more considered in linear algebra computations.

[1] Specialized hardware made to perform specific tasks more efficiently than if it was run on general-purpose CPUs.

The irruption of accelerators

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

In the 2000s, accelerators[1] were more and more considered in linear algebra computations.

➤ Iterative refinement with FPGA (Field-Programmable Gate Array).

📖 *"High-Performance Mixed- Precision Linear Solver for FPGAs"* by **Junqing Sun et al.**, 2008.

Spartan FPGA from Xilinx

[1] Specialized hardware made to perform specific tasks more efficiently than if it was run on general-purpose CPUs.

The irruption of accelerators

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

In the 2000s, accelerators[1] were more and more considered in linear algebra computations.

➤ Iterative refinement with GPU (Graphics Processing Unit).

📄 *"Accelerating double precision fem simulations with gpus"* by **Dominik Göddeke et al.**, 2005.

Nvidia Geforce RTX 3090

[1] Specialized hardware made to perform specific tasks more efficiently than if it was run on general-purpose CPUs.

From the mid 2010s to now

The rise of half precision(s)

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2010s, the increasing availability of half precision arithmetics motivated by deep learning fed the need for mixed precision algorithms:

The rise of half precision(s)

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2010s, the increasing availability of half precision arithmetics motivated by deep learning fed the need for mixed precision algorithms:

➤ Accounting for extra precision (e.g., fp128), which has been made available again through software developments, we can generally access four different arithmetics: fp16, fp32, fp64, fp128 (say).
⇒ **Optimizing the computer performance pass through exploiting and combining efficiently each of these arithmetics!**

The rise of half precision(s)

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2010s, the increasing availability of half precision arithmetics motivated by deep learning fed the need for mixed precision algorithms:

➤ Accounting for extra precision (e.g., fp128), which has been made available again through software developments, we can generally access four different arithmetics: fp16, fp32, fp64, fp128 (say).

⇒ **Optimizing the computer performance pass through exploiting and combining efficiently each of these arithmetics!**

➤ While still many applications can handle single precision accuracy, a full half precision solution is not enough!

⇒ **Half precision cannot be used alone!**

The rise of half precision(s)

(1) Computing the **residual**: $r_i = b - Ax_i$

(2) Solving the **correction equation**: $Ad_i = r_i$

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$.

From the 2010s, the increasing availability of half precision arithmetics motivated by deep learning fed the need for mixed precision algorithms:

➤ Accounting for extra precision (e.g., fp128), which has been made available again through software developments, we can generally access four different arithmetics: fp16, fp32, fp64, fp128 (say).

⇒ **Optimizing the computer performance pass through exploiting and combining efficiently each of these arithmetics!**

➤ While still many applications can handle single precision accuracy, a full half precision solution is not enough!

⇒ **Half precision cannot be used alone!**

⇒ **Interest over mixed precision algorithms skyrocketed!**

Towards more versatility

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Review of the past iterative refinement uses:

➤ Extra precision on the residual for a better accuracy.

Towards more versatility

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (working)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i$. (working)

Review of the past iterative refinement uses:

- Extra precision on the residual for a better accuracy.
- Fixed precision for stability.

Towards more versatility

(1) Computing the **residual**: $r_i = b - Ax_i$ (working)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Review of the past iterative refinement uses:

- Extra precision on the residual for a better accuracy.
- Fixed precision for stability.
- Low precision on the solver for improved performance.

Towards more versatility

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Review of the past iterative refinement uses:

- Extra precision on the residual for a better accuracy.
- Fixed precision for stability.
- Low precision on the solver for improved performance.

All can be achieved at once!

> *"[...] by using three precisions instead of two in iterative refinement, it is possible to accelerate the solution process and to obtain more accurate results for a wider class of problems."* **Erin Carson and Nicholas J. Higham**, 2018.

Towards more versatility

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

Review of the past iterative refinement uses:

- Extra precision on the residual for a better accuracy.
- Fixed precision for stability.
- Low precision on the solver for improved performance.

All can be achieved at once!

"*A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems*" by **Erin Carson and Nicholas J. Higham**, 2017.

"*Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions*" by **Erin Carson and Nicholas J. Higham**, 2018.

Ongoing topic!

(1) Computing the **residual**: $r_i = b - Ax_i$ (extra)

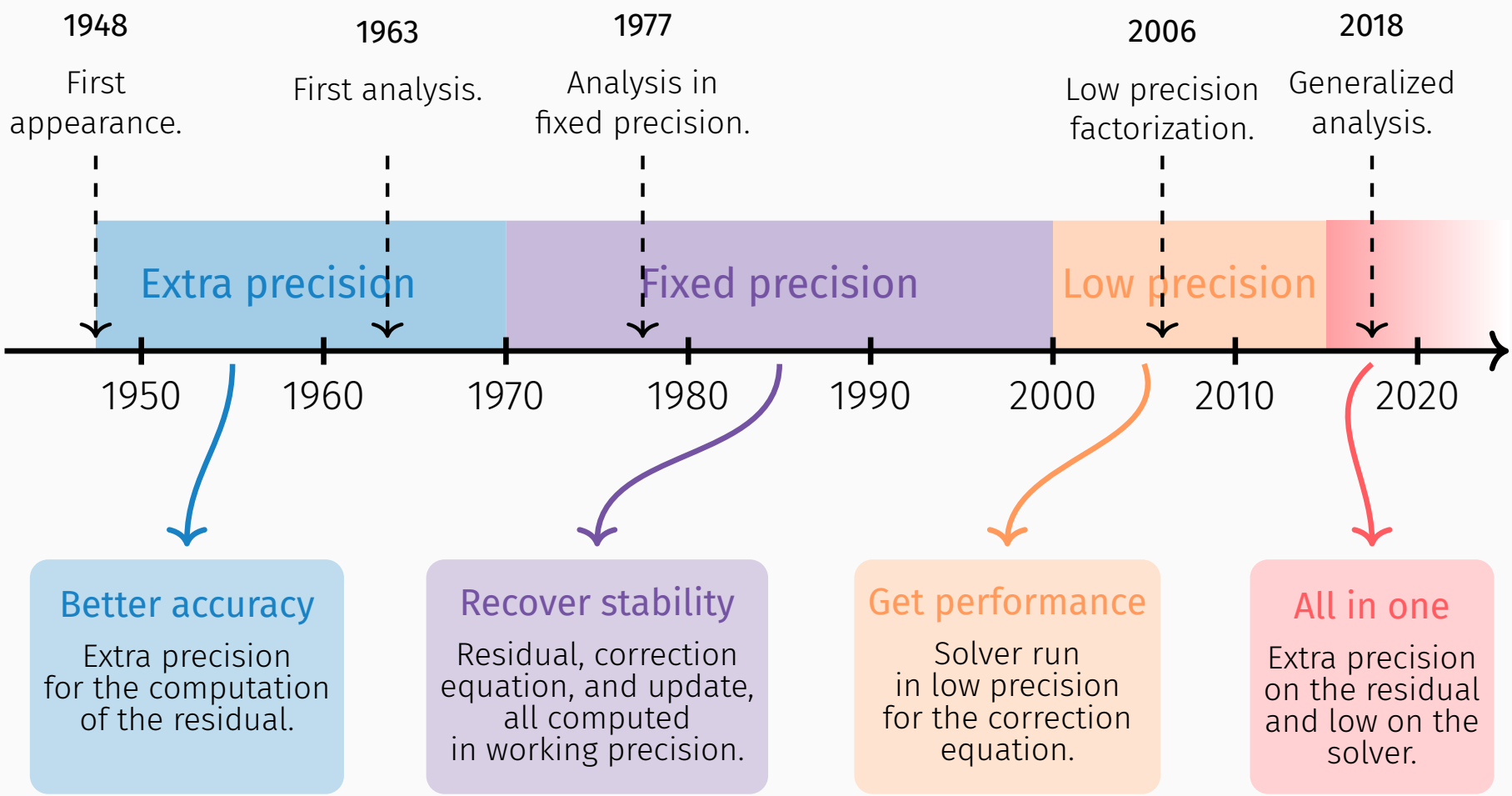
(2) Solving the **correction equation**: $Ad_i = r_i$ (low)

(3) **Updating** the solution: $x_{i+1} = x_i + d_i.$ (working)

It is still a hot topic!

- "*Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*" by **Azzam Haidar at al.**, 2018.
- "*Improving the Performance of the GMRES Method Using Mixed-Precision Techniques*" by **Neil Lindquist at al.**, 2020.
- "*Accelerating Geometric Multigrid Preconditioning with Half-Precision Arithmetic on GPUs*" by **Kyaw L. Oo and Andreas Vogel**, 2020.
- "*Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems.*" by **Nicholas J. Higham and Srikara Pranesh**, 2021.
- "*Mixed Precision GMRES-based Iterative Refinement with Recycling*" by **Eda Oktay and Erin Carson**, 2022.

History summary



"Mixed precision iterative refinement for the solution of large sparse linear systems" by **Bastien Vieublé**, 2022.

State-of-the-art mixed precision iterative refinement (all-in-one)

Generalized iterative refinement

Algorithm Generalized iterative refinement

1: Initialize x_0
2: **while not** converged **do**
3: Compute $r_i = b - Ax_i$ (u_r)
4: Solve $Ad_i = r_i$ (u_s)
5: Compute $x_{i+1} = x_i + d_i$ (u)
6: **end while**

- The precisions u_r, u_s, and u are arbitrary. Depending on the context, they refer to a floating point arithmetic, its unit roundoff, or the accuracy of the operation.
- The linear solver at step 4 is arbitrary.

Any analysis on generalized iterative refinement holds for any specialization of this algorithm. That is, for a given solver and a given set of precisions.

Generalized iterative refinement

Algorithm Generalized iterative refinement

1: Initialize x_0
2: **while not** converged **do**
3: Compute $r_i = b - Ax_i$ (u_r)
4: Solve $Ad_i = r_i$ (u_s)
5: Compute $x_{i+1} = x_i + d_i$ (u)
6: **end while**

Conditions on the solver at step 4:

$$\widehat{d}_i = (I + u_s E_i) d_i, \quad u_s \|E_i\|_\infty < 1,$$
$$\|\widehat{r}_i - A\widehat{d}_i\|_\infty \leq u_s (c_1 \|A\|_\infty \|\widehat{d}_i\|_\infty + c_2 \|\widehat{r}_i\|_\infty),$$

where E_i, c_1, and c_2 are functions of n, A, $\widehat{r}_i$, and u_s and have nonnegative entries. We recall, the quantities with an "hat" are computed quantities.

Convergence of the forward error

Theorem (Forward error convergence)

Let generalized iterative refinement be applied to $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ is nonsingular, and assume the solver used meets the previous conditions. As long as

$$\phi_i \equiv 2u_s \min(\mathrm{cond}(A), \kappa_\infty(A)\mu_i) + u_s\|E_i\|_\infty \ll 1,$$

the forward error is reduced on the ith iteration by a factor approximatively ϕ_i until an iterate $\widehat{x}_i$ is produced for which

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \lesssim 4nu_r\,\mathrm{cond}(A, x) + u.$$

Convergence of the forward error

Theorem (Forward error convergence)

Let generalized iterative refinement be applied to $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ is nonsingular, and assume the solver used meets the previous conditions. As long as

$$\phi_i \equiv 2u_s \min(\mathbf{cond}(A), \kappa_\infty(A)\mu_i) + u_s\|E_i\|_\infty \ll 1,$$

the forward error is reduced on the ith iteration by a factor approximatively ϕ_i until an iterate $\widehat{x}_i$ is produced for which

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \lesssim 4nu_r\,\mathbf{cond}(A, x) + u.$$

- $\phi_i \ll 1$ is the convergence condition. It only depends on the precision u_s of the solver. It is dominated by the term $u_s\|E_i\|_\infty$, such that we simplify the condition by $\phi_i \approx u_s\|E_i\|_\infty \ll 1$.
- $4nu_r\,\mathbf{cond}(A, x) + u$ is the limiting accuracy. It only depends on u and u_r. If u_r is chosen accurate enough, we can remove the dependence on $\mathbf{cond}(A, x)$.

LU-IR3: Specialization to LU direct solver

Algorithm LU-based iterative refinement in three precisions

1: Compute the LU factorization $A = \hat{L}\hat{U}$ (u_f)
2: Solve $Ax_0 = b$ (u_f)
3: **while not** converged **do**
4: Compute $r_i = b - Ax_i$ (u_r)
5: Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$ (u_f)
6: Compute $x_{i+1} = x_i + d_i$ (u)
7: **end while**

LU-IR3: Specialization to LU direct solver

Algorithm LU-based iterative refinement in three precisions

1:	Compute the LU factorization $A = \hat{L}\hat{U}$	$\mathcal{O}(n^3)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	(u_f)
3:	**while not** converged **do**		
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	(u_r)
5:	Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$	$\mathcal{O}(n^2)$	(u_f)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	(u)
7:	**end while**		

The strategy is to accelerate with low precisions the factorization $O(n^3)$ and recover a good accuracy by using higher precisions for the residual and update $O(n^2)$.

$\Rightarrow$ **We are faster than an LU direct solver in precision u and as accurate!**

LU-IR3: Specialization to LU direct solver

Algorithm LU-based iterative refinement in three precisions

1:	Compute the LU factorization $A = \hat{L}\hat{U}$	$\mathcal{O}(n^3)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	(u_f)
3:	**while not** converged **do**		
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	(u_r)
5:	Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$	$\mathcal{O}(n^2)$	(u_f)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	(u)
7:	**end while**		

From the previous theorem we know that the limiting accuracy is of order $u_r\,\mathbf{cond}(A) + u$.

AND the convergence condition is $u_s\|E_i\|_\infty \ll 1 \Rightarrow$ **We need to determine** u_s **and** $\|E_i\|_\infty$ for the case of the LU solver!

LU-IR3: Specialization to LU direct solver

Algorithm LU-based iterative refinement in three precisions

1:	Compute the LU factorization $A = \hat{L}\hat{U}$	$\mathcal{O}(n^3)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	(u_f)
3:	**while not** converged **do**		
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	(u_r)
5:	Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$	$\mathcal{O}(n^2)$	(u_f)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	(u)
7:	**end while**		

We know that the LU solver provide a solution satisfying

$$\frac{\|d_i - \widehat{d}_i\|_\infty}{\|d_i\|_\infty} \ll u_f \kappa_\infty(A).$$

We identify $u_s\|E_i\|_\infty \equiv u_f\kappa_\infty(A)$ knowing that by definition we have

$$\frac{\|d_i - \widehat{d}_i\|_\infty}{\|d_i\|_\infty} \leq u_s\|E_i\|_\infty.$$

LU-IR3: Specialization to LU direct solver

Algorithm LU-based iterative refinement in three precisions

1:	Compute the LU factorization $A = \hat{L}\hat{U}$	$\mathcal{O}(n^3)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	(u_f)
3:	**while not** converged **do**		
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	(u_r)
5:	Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$	$\mathcal{O}(n^2)$	(u_f)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	(u)
7:	**end while**		

	Convergence condition	Forward error
LU-IR3	$u_f\kappa(A) \ll 1$	$u_r\kappa(A) + u$

Limit: Very low precision factorization leads to a very restrictive convergence condition for LU-IR3 (e.g., with $u_f =$ fp16 we have $\kappa(A) \ll 2 \times 10^3$).

LU-GMRES-IR5: Get more robust

Algorithm GMRES-based iterative refinement in five precisions

1: Compute the LU factorization $A = \hat{L}\hat{U}$ (u_f)
2: Solve $Ax_0 = b$ (u_f)
3: **while not** converged **do**
4: Compute $r_i = b - Ax_i$ (u_r)
5: Solve $\hat{U}^{-1}\hat{L}^{-1}Ad_i = \hat{U}^{-1}\hat{L}^{-1}r_i$ by GMRES at precision (u_g) with matrix vector products with $\widetilde{A} = \hat{U}^{-1}\hat{L}^{-1}A$ at precision (u_p)
6: Compute $x_{i+1} = x_i + d_i$ (u)
7: **end while**

"*A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems*" by **Erin Carson and Nicholas J. Higham**, 2017.

"*Five-precicion GMRES-based iterative refinement*" by **Amestoy, Buttari, L'Excellent, Higham, Mary, Vieublé**, 2023.

LU-GMRES-IR5: Get more robust

Algorithm GMRES-based iterative refinement in five precisions

1: Compute the LU factorization $A = \hat{L}\hat{U}$ (u_f)
2: Solve $Ax_0 = b$ (u_f)
3: **while not** converged **do**
4: Compute $r_i = b - Ax_i$ (u_r)
5: Solve $\hat{U}^{-1}\hat{L}^{-1}Ad_i = \hat{U}^{-1}\hat{L}^{-1}r_i$ by GMRES at precision (u_g) with matrix vector products with $\widetilde{A} = \hat{U}^{-1}\hat{L}^{-1}A$ at precision (u_p)
6: Compute $x_{i+1} = x_i + d_i$ (u)
7: **end while**

➤ Based on GMRES solver which is a well-known Krylov subspace based iterative solver.

➤ LU-GMRES-IR5 is a more robust form of iterative refinement capable of tackling higher condition numbers $\kappa(A)$ than LU-IR3.

LU-GMRES-IR5: Get more robust

Algorithm GMRES-based iterative refinement in five precisions

1: Compute the LU factorization $A = \hat{L}\hat{U}$ (u_f)
2: Solve $Ax_0 = b$ (u_f)
3: **while not** converged **do**
4: Compute $r_i = b - Ax_i$ (u_r)
5: Solve $\hat{U}^{-1}\hat{L}^{-1}Ad_i = \hat{U}^{-1}\hat{L}^{-1}r_i$ by GMRES at precision (u_g) with matrix vector products with $\widetilde{A} = \hat{U}^{-1}\hat{L}^{-1}A$ at precision (u_p)
6: Compute $x_{i+1} = x_i + d_i$ (u)
7: **end while**

	Convergence condition	Forward error
LU-IR3	$\kappa(A)u_f \ll 1$	$u_r\kappa(A) + u$
LU-GMRES-IR5	$(u_g + u_p\kappa(A))(1 + u_f^2\kappa(A)^2) \ll 1$[a]	$u_r\kappa(A) + u$

[a] The proof for the convergence condition of LU-GMRES-IR5 still relies on identifying u_s and E_i, but is a bit more technical. We will not attempt to do it here.

LU-GMRES-IR5: Get more robust

Algorithm GMRES-based iterative refinement in five precisions

1: Compute the LU factorization $A = \hat{L}\hat{U}$ (u_f)
2: Solve $Ax_0 = b$ (u_f)
3: **while not** converged **do**
4: Compute $r_i = b - Ax_i$ (u_r)
5: Solve $\hat{U}^{-1}\hat{L}^{-1}Ad_i = \hat{U}^{-1}\hat{L}^{-1}r_i$ by GMRES at precision (u_g) with matrix vector products with $\widetilde{A} = \hat{U}^{-1}\hat{L}^{-1}A$ at precision (u_p)
6: Compute $x_{i+1} = x_i + d_i$ (u)
7: **end while**

	Convergence condition	Forward error
LU-IR3	$\kappa(A)u_f \ll 1$	$u_r\kappa(A) + u$
LU-GMRES-IR5	$(u_g + u_p\kappa(A))(1 + u_f^2\kappa(A)^2) \ll 1$	$u_r\kappa(A) + u$

Example: If $u_f =$ fp16, the condition on LU-IR3 is 2×10^3, on LU-GMRES-IR5 with $u_g =$ fp64 and $u_p =$ fp128 it is 2×10^{11}!

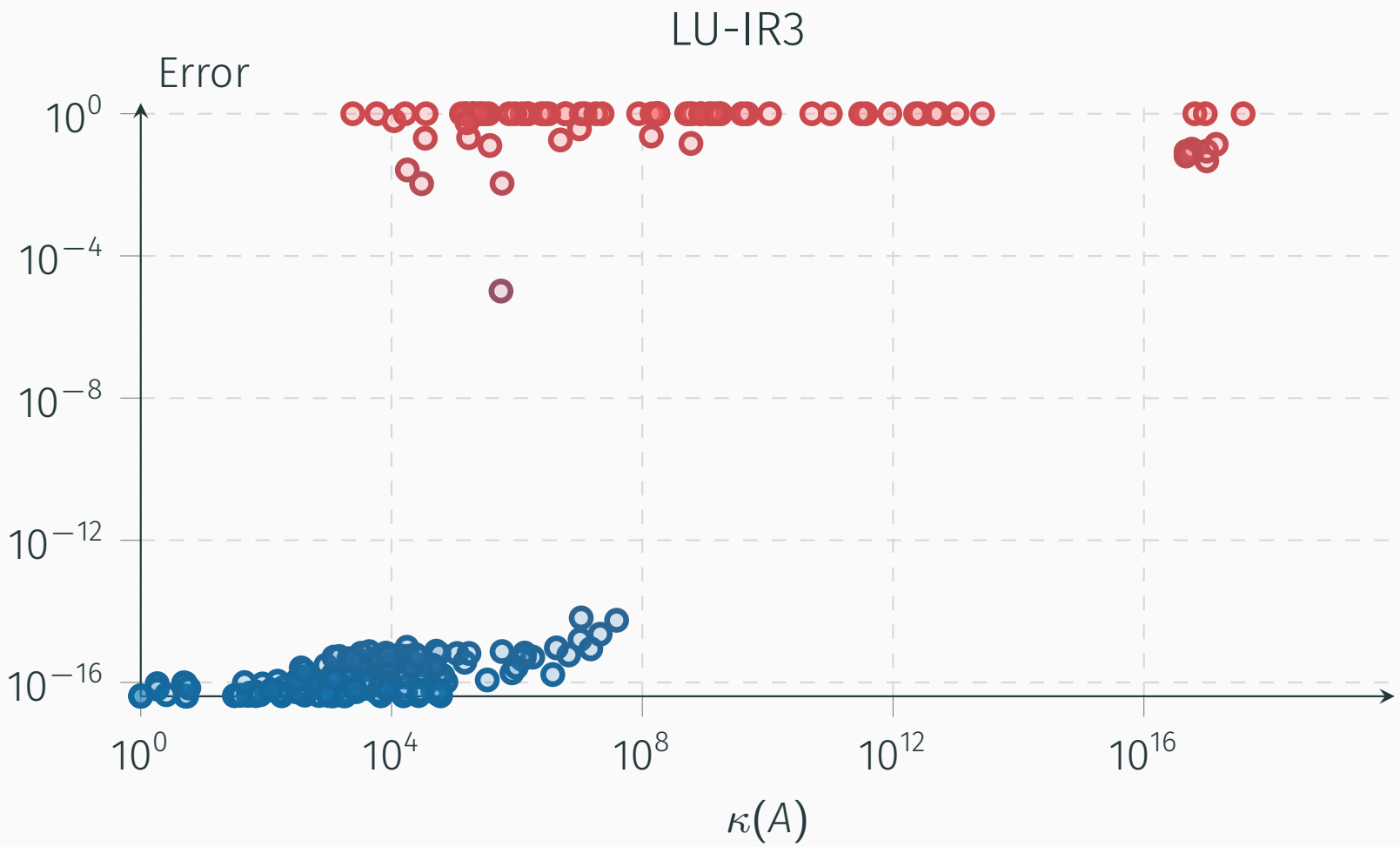
LU-IR3 Vs LU-GMRES-IR5: Theoretical robustness

u_g	u_p	Convergence Condition ($\max(\kappa(A))$)
LU-IR3		2×10^3
R	S	8×10^3
B	S	3×10^4
H	S	4×10^4
H	D	9×10^4
S	D	8×10^6
D	D	3×10^7
D	Q	2×10^{11}

Combinations of LU-GMRES-IR5 for $u_f = H$ and $u = D$.

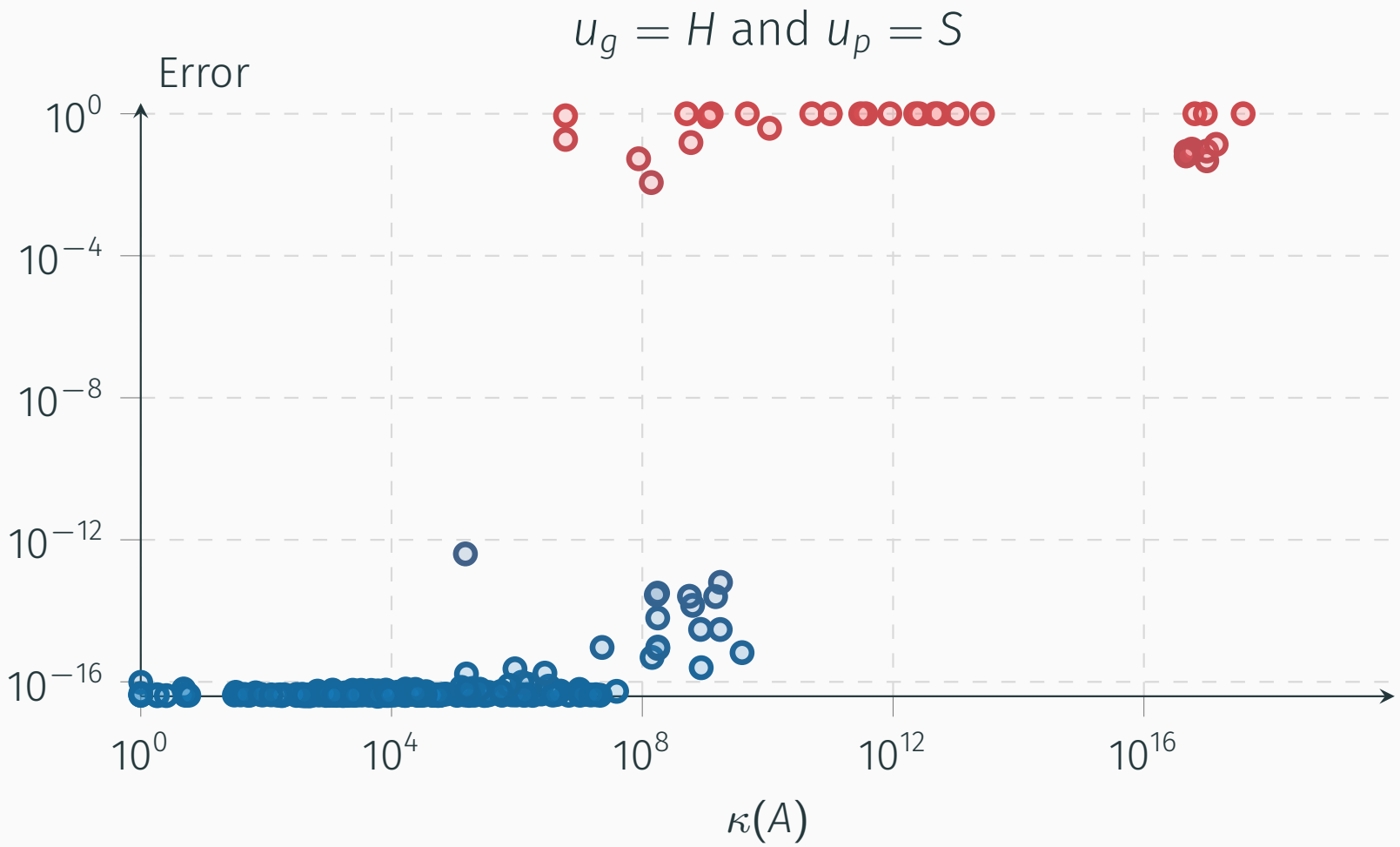
- The more we increase the precisions u_g and u_p, the more robust we are.
- LU-GMRES-IR5 is flexible regarding the conditioning of the problems and the choice of precisions $\Rightarrow$ **It offers finer trade-offs between robustness and performance!**

LU-IR3 Vs LU-GMRES-IR5: Experimental robustness



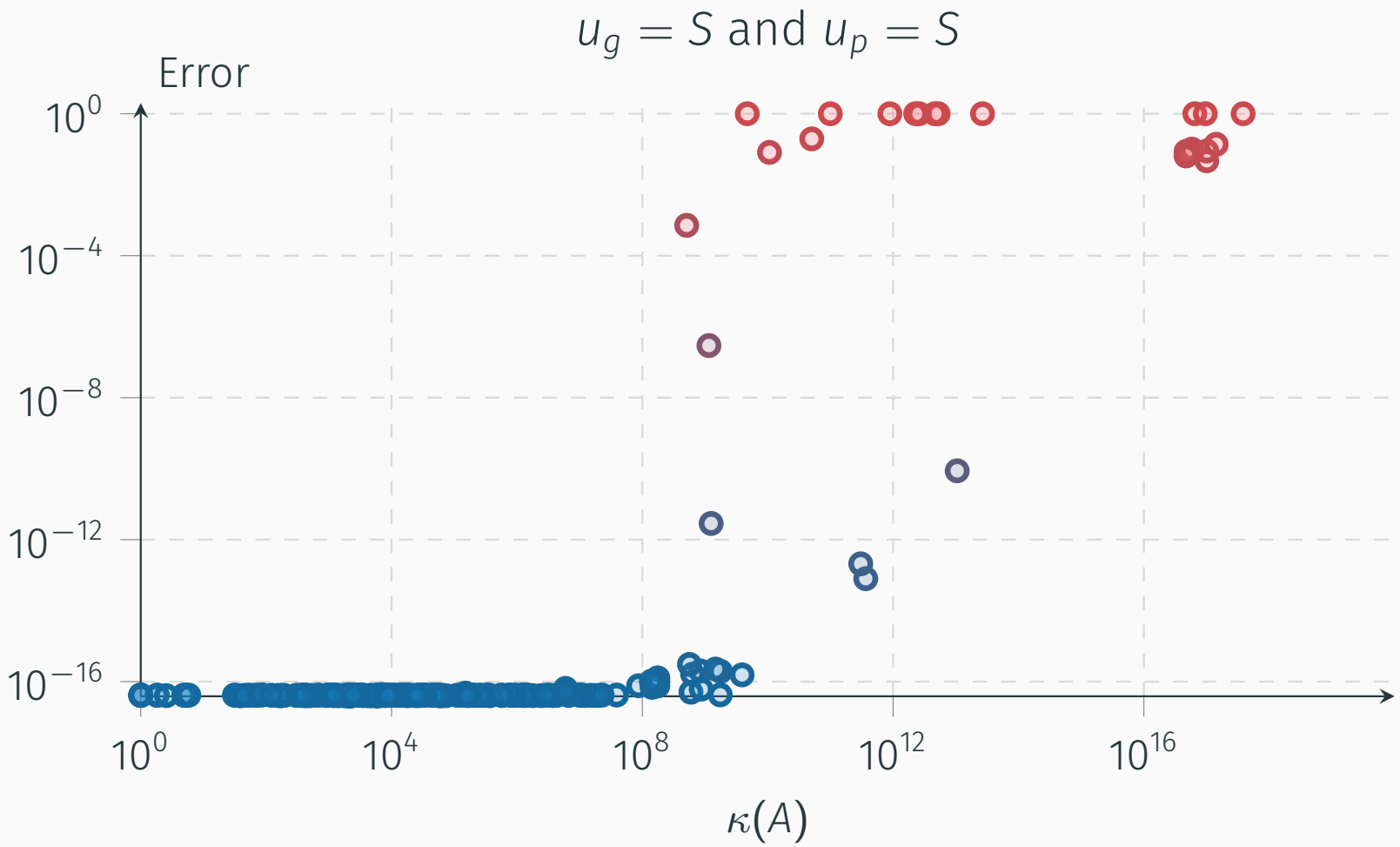
Forward errors of 250 SuiteSparse matrices of various conditioning. We fix $u_f = H$, $u = D$, and $u_r = Q$.

LU-IR3 Vs LU-GMRES-IR5: Experimental robustness



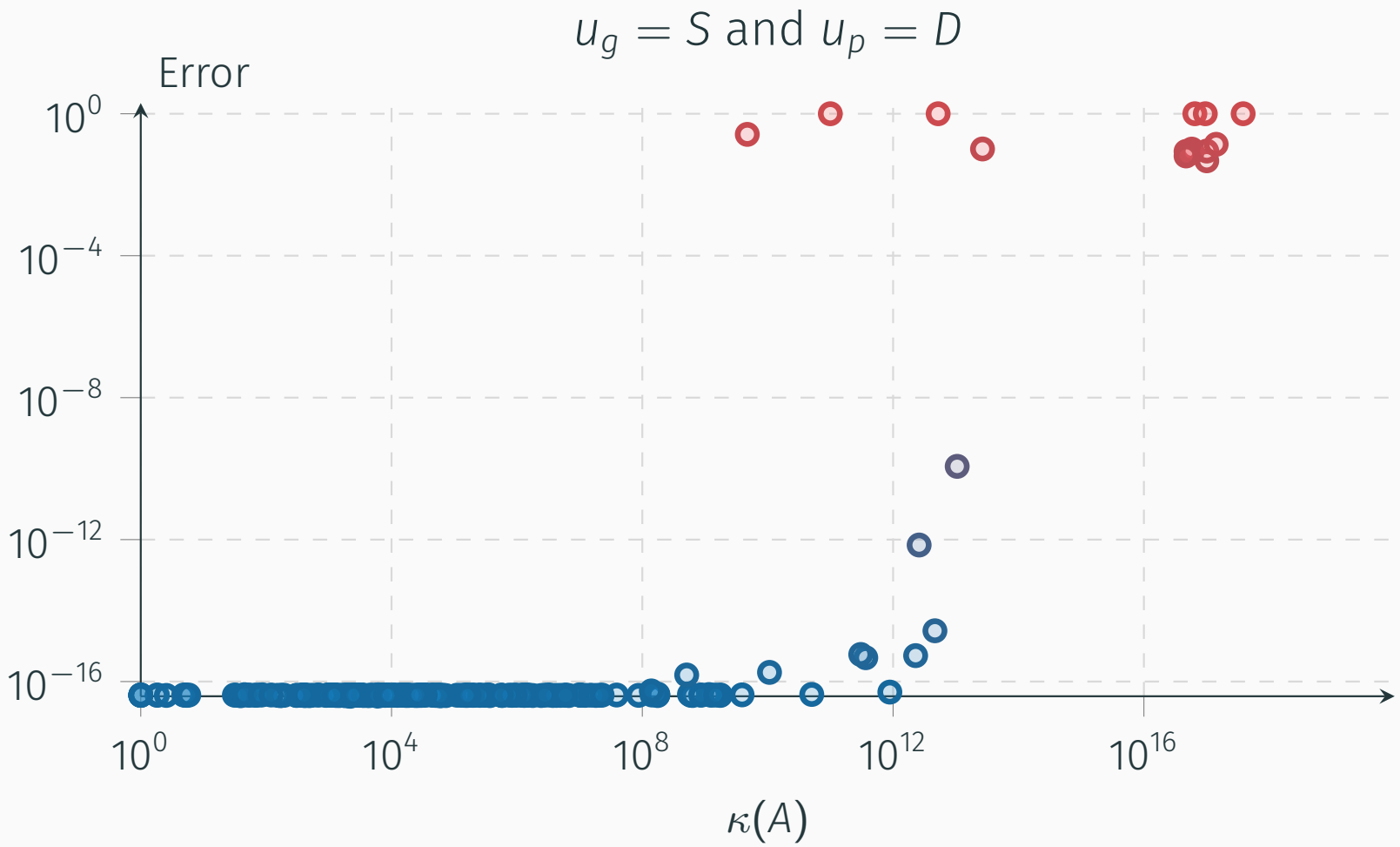
Forward errors of 250 SuiteSparse matrices of various conditioning. We fix $u_f = H$, $u = D$, and $u_r = Q$.

LU-IR3 Vs LU-GMRES-IR5: Experimental robustness



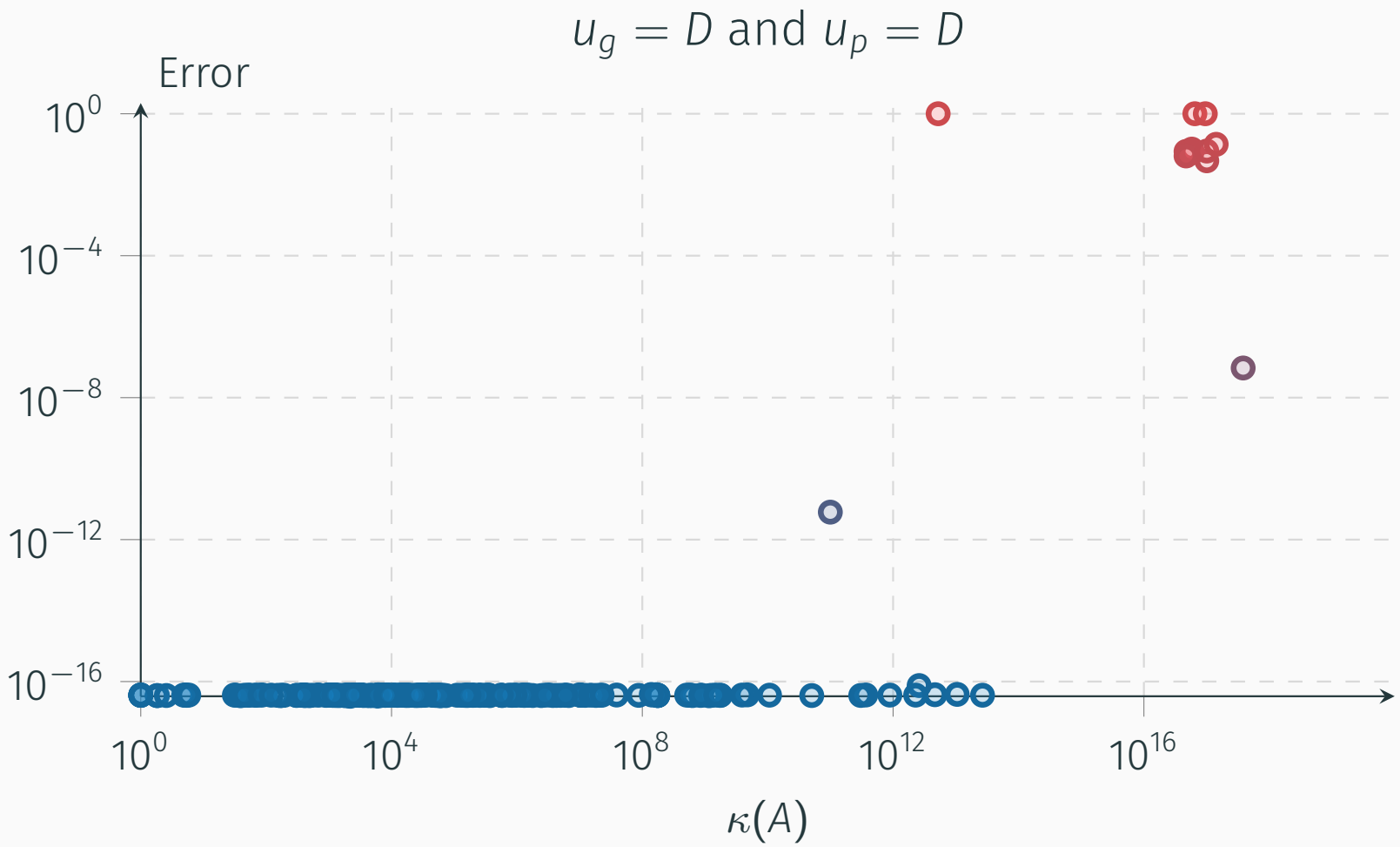
Forward errors of 250 SuiteSparse matrices of various conditioning. We fix $u_f = H$, $u = D$, and $u_r = Q$.

LU-IR3 Vs LU-GMRES-IR5: Experimental robustness



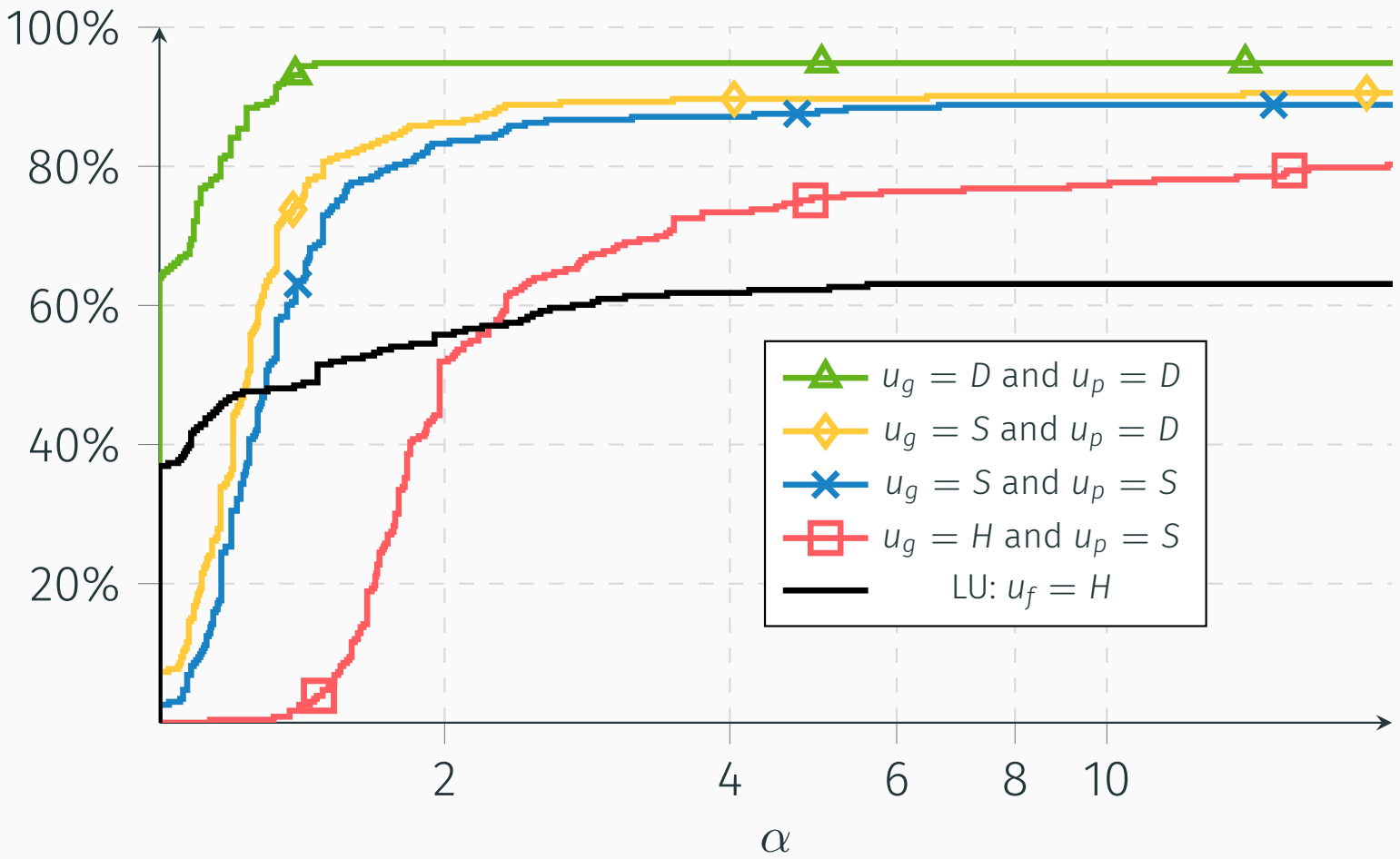
Forward errors of 250 SuiteSparse matrices of various conditioning. We fix $u_f = H$, $u = D$, and $u_r = Q$.

LU-IR3 Vs LU-GMRES-IR5: Experimental robustness



Forward errors of 250 SuiteSparse matrices of various conditioning. We fix $u_f = H$, $u = D$, and $u_r = Q$.

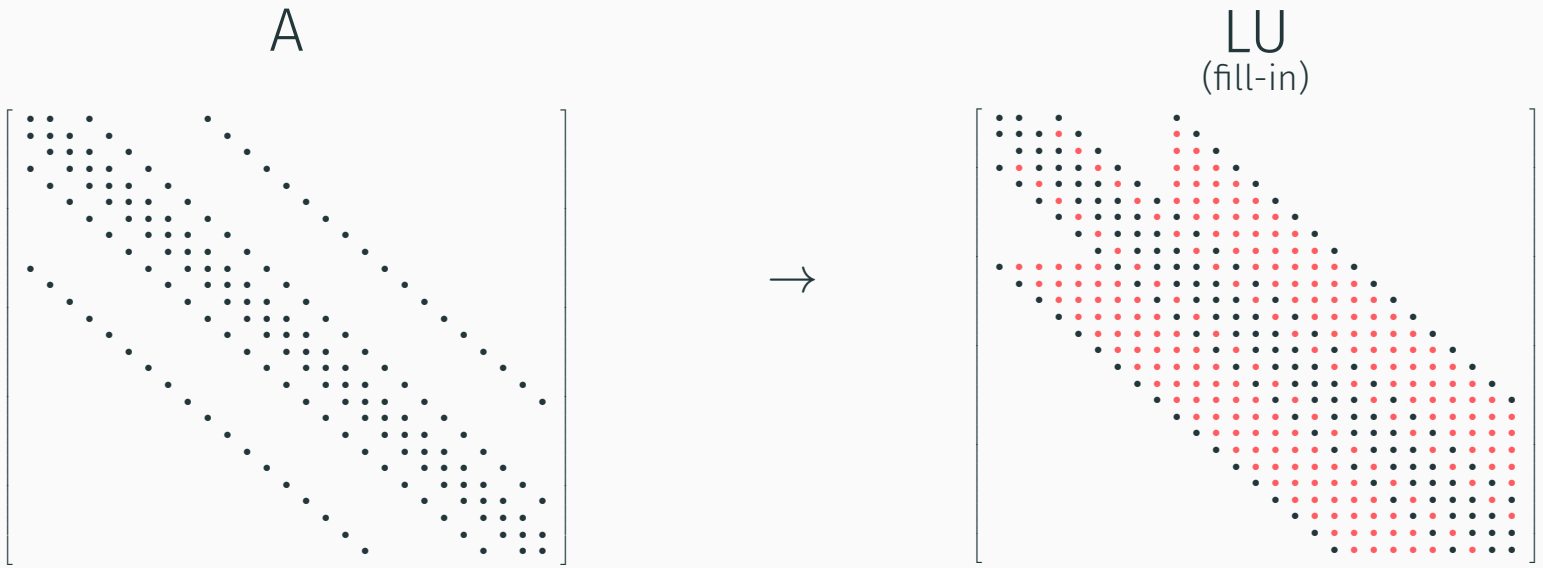
Performance profile: #iterations comparison



Percentage of the 250 SuiteSparse matrices for which a given combination requires less than α times the number of iterations required by the best combination. We set $u_f = H$.

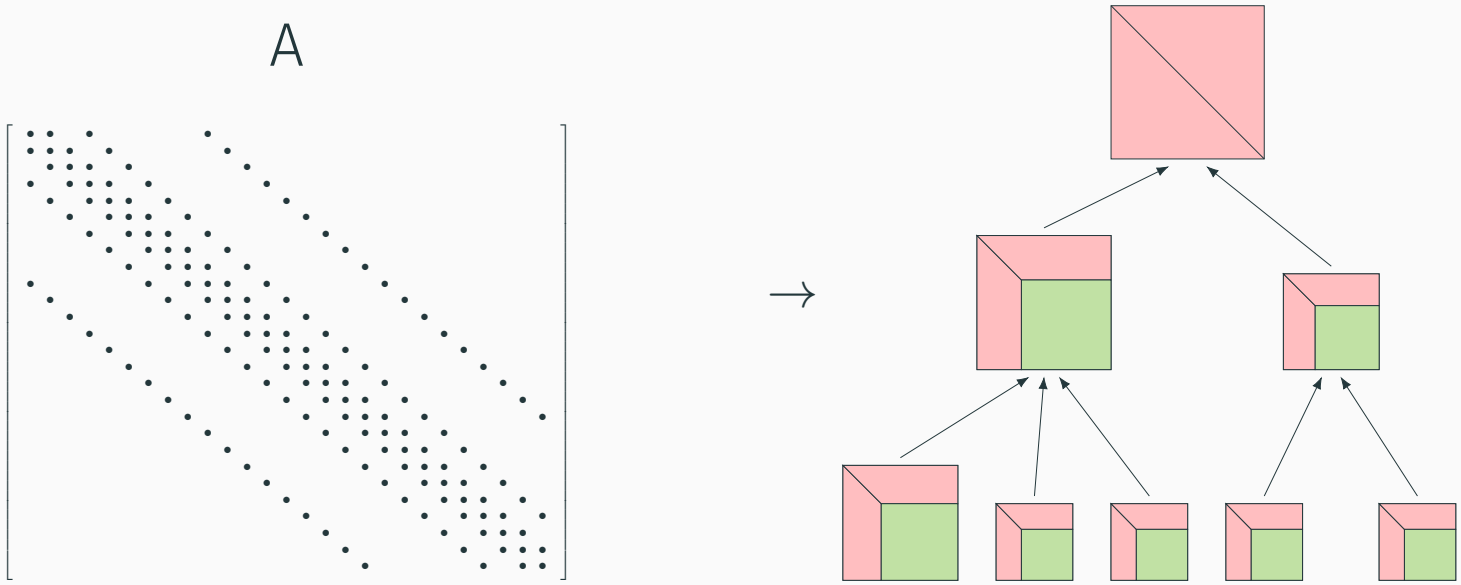
Performance analysis on sparse problems using numerical approximations

LU Sparse Direct Factorization: Fill-in



Multifrontal LU Sparse Direct Factorization

A multifrontal solver decomposes the sparse factorization into a series of partial factorizations of dense matrices whose dependencies are represented by an assembly tree:



➤ The red parts are the LU entries, the green part are temporary data.

➤ In multifrontal factorization the total memory consumption is higher than the factors in memory. The difference is called the active memory overhead.

Specific features of sparse iterative refinement

Algorithm Iterative refinement: complexities Dense VS Sparse

1:	Compute the LU factorization $A = LU$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_f)
3:	**while not** converged **do**			
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	(u_r)
5:	Solve $Ad_i = r_i$.	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_s)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u)
7:	**end while**			

Fill-in in sparse direct solvers, i.e. $\mathrm{NNZ}(A) \ll \mathrm{NNZ}(LU)$!

Specific features of sparse iterative refinement

Algorithm Iterative refinement: complexities Dense VS Sparse

1:	Compute the LU factorization $A = LU$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_f)
3:	**while not** converged **do**			
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	(u_r)
5:	Solve $Ad_i = r_i$.	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_s)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u)
7:	**end while**			

Fill-in in sparse direct solvers, i.e. $\mathrm{NNZ}(A) \ll \mathrm{NNZ}(LU)$!

➤ SpMV much cheaper than solve $\Rightarrow$ $u_r \ll u$ has limited impact on performance (even for $u_r =$ fp128).

Specific features of sparse iterative refinement

Algorithm Iterative refinement: complexities Dense VS Sparse

1:	Compute the LU factorization $A = LU$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_f)
3:	**while not** converged **do**			
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	(u_r)
5:	Solve $Ad_i = r_i$.	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_s)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u)
7:	**end while**			

Fill-in in sparse direct solvers, i.e. $\mathrm{NNZ}(A) \ll \mathrm{NNZ}(LU)$!

➤ Memory space of A in u_r ($\mathcal{O}(n)$ entries) negligible compared with the LU factors in u_f ($\mathcal{O}(n^{4/3})$ entries) $\Rightarrow$ LU-IR3 saves memory over a direct solver in u!

Specific features of sparse iterative refinement

Algorithm Iterative refinement: complexities Dense VS Sparse

1:	Compute the LU factorization $A = LU$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	(u_f)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_f)
3:	**while not** converged **do**			
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	(u_r)
5:	Solve $Ad_i = r_i$.	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{4/3})$	(u_s)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u)
7:	**end while**			

Fill-in in sparse direct solvers, i.e. $\mathrm{NNZ}(A) \ll \mathrm{NNZ}(LU)$!

➤ **(Multifrontal only)** Even if LU-GMRES-IR5 fully stores the factors in $u_p = u$, it does not need to store the active memory in $u_p = u$
$\Rightarrow$ LU-GMRES-IR5 can save memory over a direct solver in u.

Specific features of approximate factorization

Algorithm LU-IR3: complexities Sparse VS Approximations

1:	Compute the LU factorization $A = \hat{L}\hat{U}$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^\alpha)$	(ϵ)
2:	Solve $Ax_0 = b$	$\mathcal{O}(n^{4/3})$	$\mathcal{O}(n^\beta)$	(ϵ)
3:	**while not** converged **do**			
4:	Compute $r_i = b - Ax_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u_r)
5:	Solve $Ad_i = r_i$ by $d_i = \hat{U}^{-1}\hat{L}^{-1}r_i$.	$\mathcal{O}(n^{4/3})$	$\mathcal{O}(n^\beta)$	(ϵ)
6:	Compute $x_{i+1} = x_i + d_i$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	(u)
7:	**end while**			

- Where $2 \geq \alpha$ and $4/3 \geq \beta$.
- Reduce the complexity at the cost of introducing a (controlled) perturbation ϵ in the factorization and solve.
- We use Block Low-Rank and static pivoting approximations in our experiments.

Implemented parallel methods

Solver	u_f	u	u_r	u_g	u_p	$\max(\kappa(A))$ $(\epsilon = 0)$	forward error
DMUMPS	fp64 LU direct solver					—	$\kappa(A) \times 10^{-16}$
LU-IR	S	D	D	—	—	2×10^{7}	$\kappa(A) \times 10^{-16}$
LU-GMRES-IR	S	D	D	D	D	1×10^{10}	$\kappa(A) \times 10^{-16}$

➤ LU-IR and LU-GMRES-IR use single precision (fp32) factorization and numerical approximations to save resources.

➤ We use the multifrontal sparse solver MUMPS. While we expect our conclusions on the execution time to hold for all direct sparse solvers. Our conclusions on the memory consumption related to the active memory are specific to the multifrontal solvers.

Matrix set

Name	N	NNZ	Arith.	Sym.	$\kappa(A)$	Fact. (flops)	Slv. (flops)
ElectroPhys10M	1.02E+07	1.41E+08	R	1	1.10E+01	4E+14	9E+10
DrivAer6M	6.11E+06	4.97E+07	R	1	9.40E+05	6E+13	3E+10
Queen_4147	4.14E+06	3.28E+08	R	1	4.30E+06	3E+14	6E+10
tminlet3M	2.84E+06	1.62E+08	C	0	2.70E+07	1E+14	2E+10
perf009ar	5.41E+06	2.08E+08	R	1	3.70E+08	2E+13	2E+10
elasticity-3d	5.18E+06	1.16E+08	R	1	3.60E+09	2E+14	5E+10
lfm_aug5M	5.52E+06	3.71E+07	C	1	5.80E+11	2E+14	5E+10
CarBody25M	2.44E+07	7.06E+08	R	1	8.60E+12	1E+13	3E+10
thmgas	5.53E+06	3.71E+07	R	0	8.28E+13	1E+14	4E+10

Set of **industrial** and SuiteSparse matrices.

➤ The matrices are ordered in increasing $\kappa(A)$, the higher $\kappa(A)$ is, the slower the convergence (if reached at all).

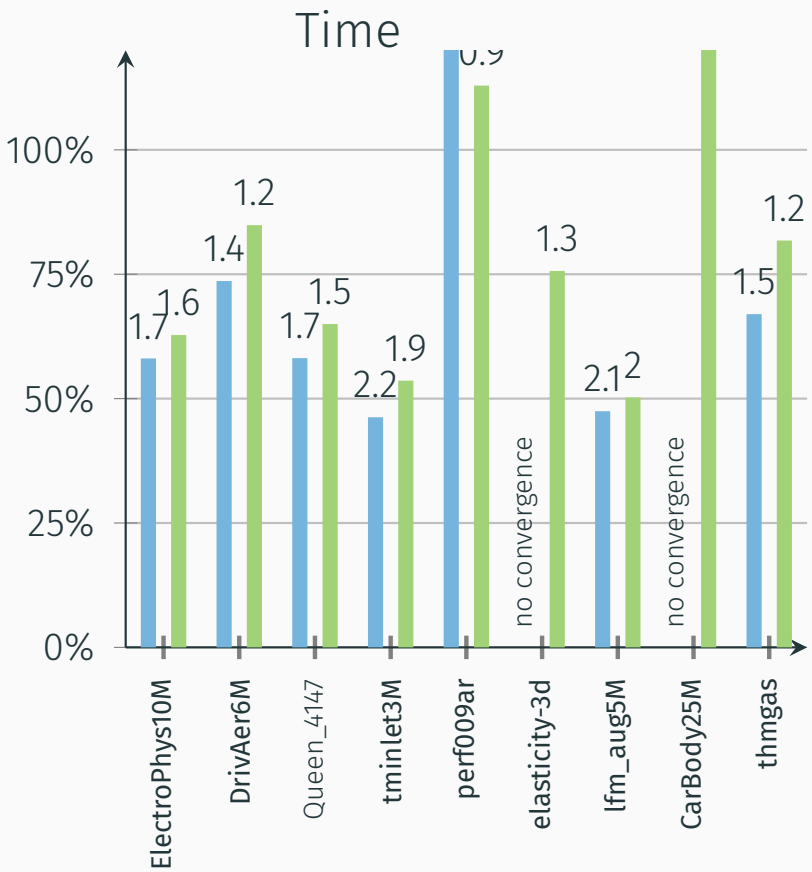
Matrix set

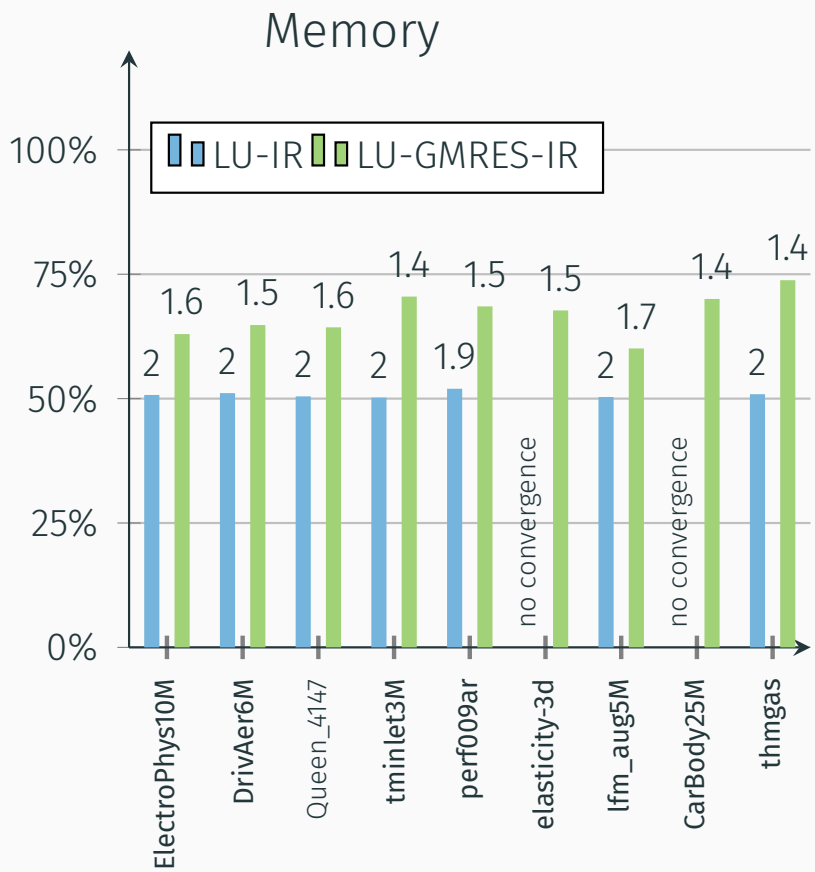
Name	N	NNZ	Arith.	Sym.	$\kappa(A)$	Fact. (flops)	Slv. (flops)
ElectroPhys10M	1.02E+07	1.41E+08	R	1	1.10E+01	4E+14	9E+10
DrivAer6M	6.11E+06	4.97E+07	R	1	9.40E+05	6E+13	3E+10
Queen_4147	4.14E+06	3.28E+08	R	1	4.30E+06	3E+14	6E+10
tminlet3M	2.84E+06	1.62E+08	C	0	2.70E+07	1E+14	2E+10
perf009ar	5.41E+06	2.08E+08	R	1	3.70E+08	2E+13	2E+10
elasticity-3d	5.18E+06	1.16E+08	R	1	3.60E+09	2E+14	5E+10
lfm_aug5M	5.52E+06	3.71E+07	C	1	5.80E+11	2E+14	5E+10
CarBody25M	2.44E+07	7.06E+08	R	1	8.60E+12	1E+13	3E+10
thmgas	5.53E+06	3.71E+07	R	0	8.28E+13	1E+14	4E+10

Set of **industrial** and SuiteSparse matrices.

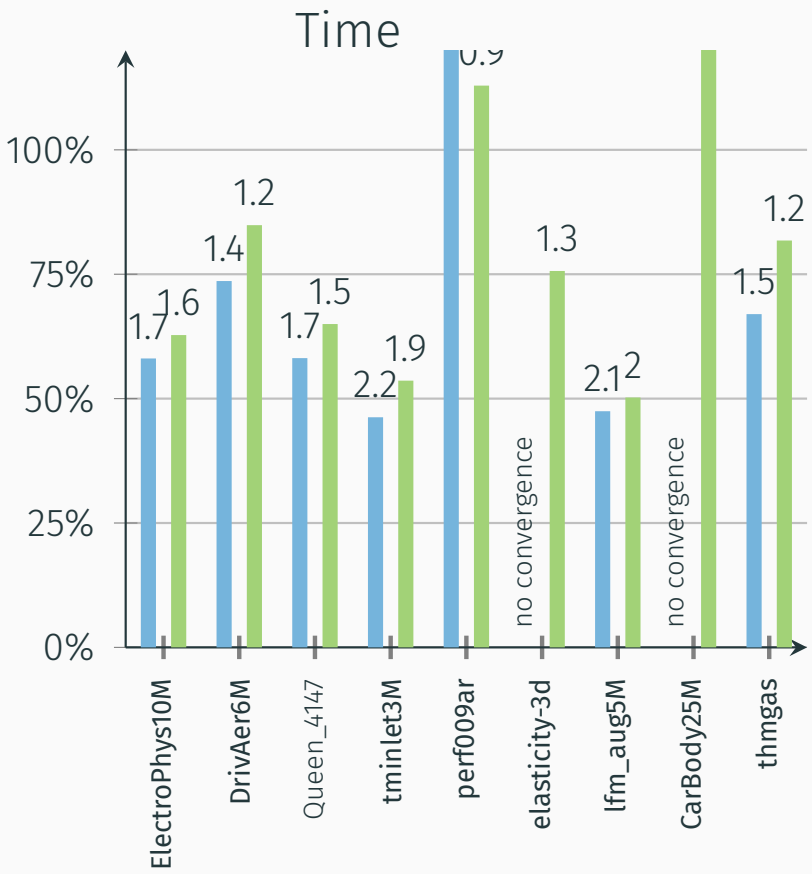
➤ We run on OLYMPE supercomputer nodes (two Intel 18-cores Skylake/node), 1 node (2MPI×18threads) or 2 nodes (4MPI×18threads) depending on the matrix size.

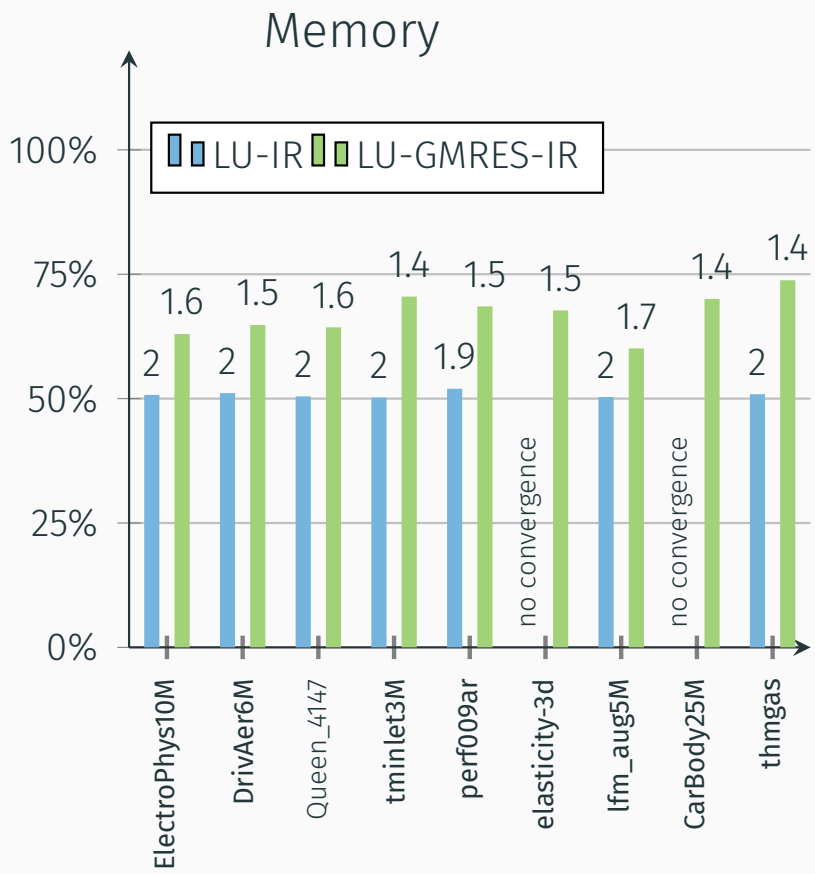
Time and memory performance w.r.t. DMUMPS (no approx)





Time and memory performance w.r.t. DMUMPS (no approx)

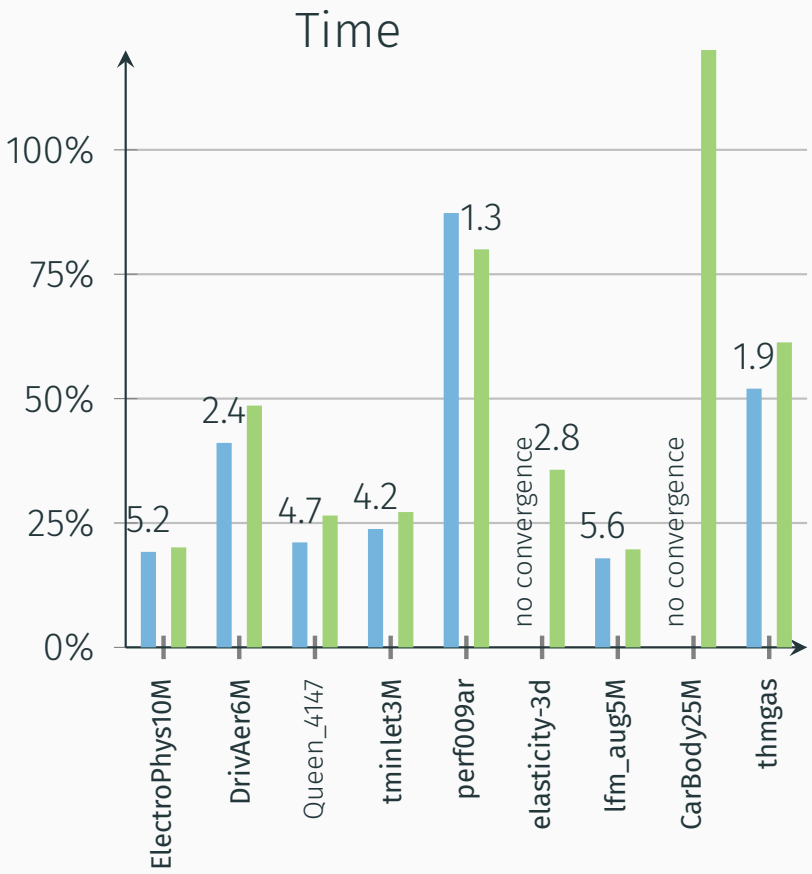


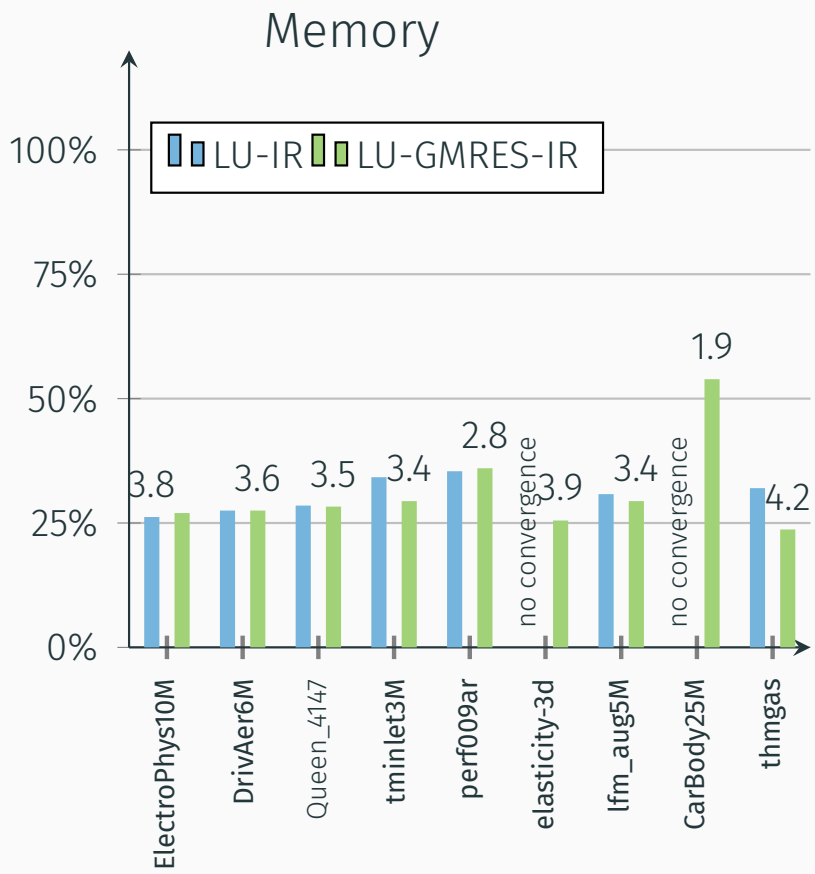


- LU-IR up to 2.2× faster.
- LU-GMRES-IR up to 1.9× faster. Slower than LU-IR, but more robust.

- LU-IR consumes 2× less memory.
- LU-GMRES-IR consumes at best 1.7× less despite factors in double ⇒ save active memory.

Best time and memory achieved w.r.t. DMUMPS (with approx)





$\Rightarrow$ Up to 5.6$\times$ faster and 4.2$\times$ less memory with the same accuracy on the solution than the double direct LU solver!

"Combining sparse approximate factorization with mixed precision iterative refinement" by **Amestoy, Buttari, L'Excellent, Higham, Mary, Vieublé**, 2022.

Conclusion

The lecture in a nutshell

- Low precisions are a potential source of substantial resource savings BUT they degrade the accuracy of the solution.
- Iterative refinement is one of the first mixed precision algorithm. It was implemented on the first computers.
- Iterative refinement can be used to efficiently exploit low precisions while safely improving the accuracy of the solution.
- The ability or not of iterative refinement to improve the solution depends on the condition number of the problem (i.e., $\kappa(A)$).
- Iterative refinement has been proven extremely efficient at computing the solution of large parallel sparse systems coming from real-life and industrial applications.

Question Time !