On improving the refinement of inaccurate linear solvers

Bastien Vieublé

Paris-Saclay Master HPC

09/02/2026

Section 1

Background on solving linear systems

Linear systems

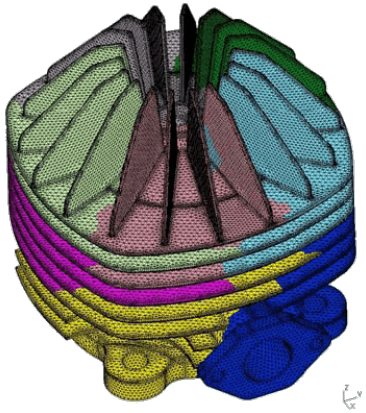
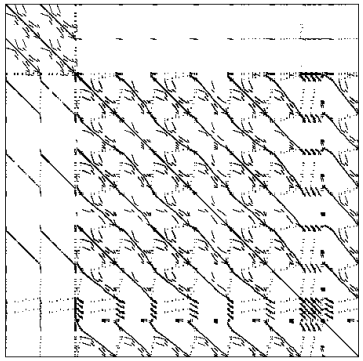
$$Ax = b,$$
$$A \in \mathbb{R}^{n \times n}, \quad b \in \mathbb{R}^n, \quad x \in \mathbb{R}^n$$

Linear systems

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

Various challenges: numerical difficulties, different algorithms, parallelism, computer hardware, etc.

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**!

Examples: low precision, randomization/sketching, low-rank approximations, tensor compression, etc.

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**!

Examples: low precision, randomization/sketching, low-rank approximations, tensor compression, etc.

$\Rightarrow$ **Iterative refinement can correct cheaply the inaccuracies introduced by approximate computing.**

Section 2

Low precision arithmetics and mixed precision algorithms

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 notations

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

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

- Consider the linear system $Ax = b$, we note $\hat{x}$ **the computed solution** in floating point.

- We define the **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}

"*Floating-point arithmetic*" by **Boldo et al.**, 2023, Acta Numerica.

Why using low precision arithmetics ?

Low precision arithmetics = **less accurate** + **narrower range**

BUT **3 main advantages**:

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

As reducing time, memory, and energy consumption are all challenging objectives, low precisions have become the **Wild West of HPC**!

"*Low Precision Floating-Point Formats: The Wild West of Computer Arithmetic*" by **Pranesh**, 2019, SIAM news.

The fundamental dilemma of low precision arithmetics

Problem

- Low precisions greatly **improve performance** of linear solvers...
- BUT they **degrade their accuracy** at the same time.
- Application experts generally **require "high accuracy"** on the solution (i.e., most commonly single or 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 precisions greatly **improve performance** of linear solvers...
- BUT they **degrade their accuracy** at the same time.
- Application experts generally **require "high accuracy"** on the solution (i.e., most commonly single or 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**!

Section 3

Introduction to iterative refinement

Newton's method for correcting 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 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** with eventually **numerical approximations**, which lead to the presence of computing errors in every of these steps.

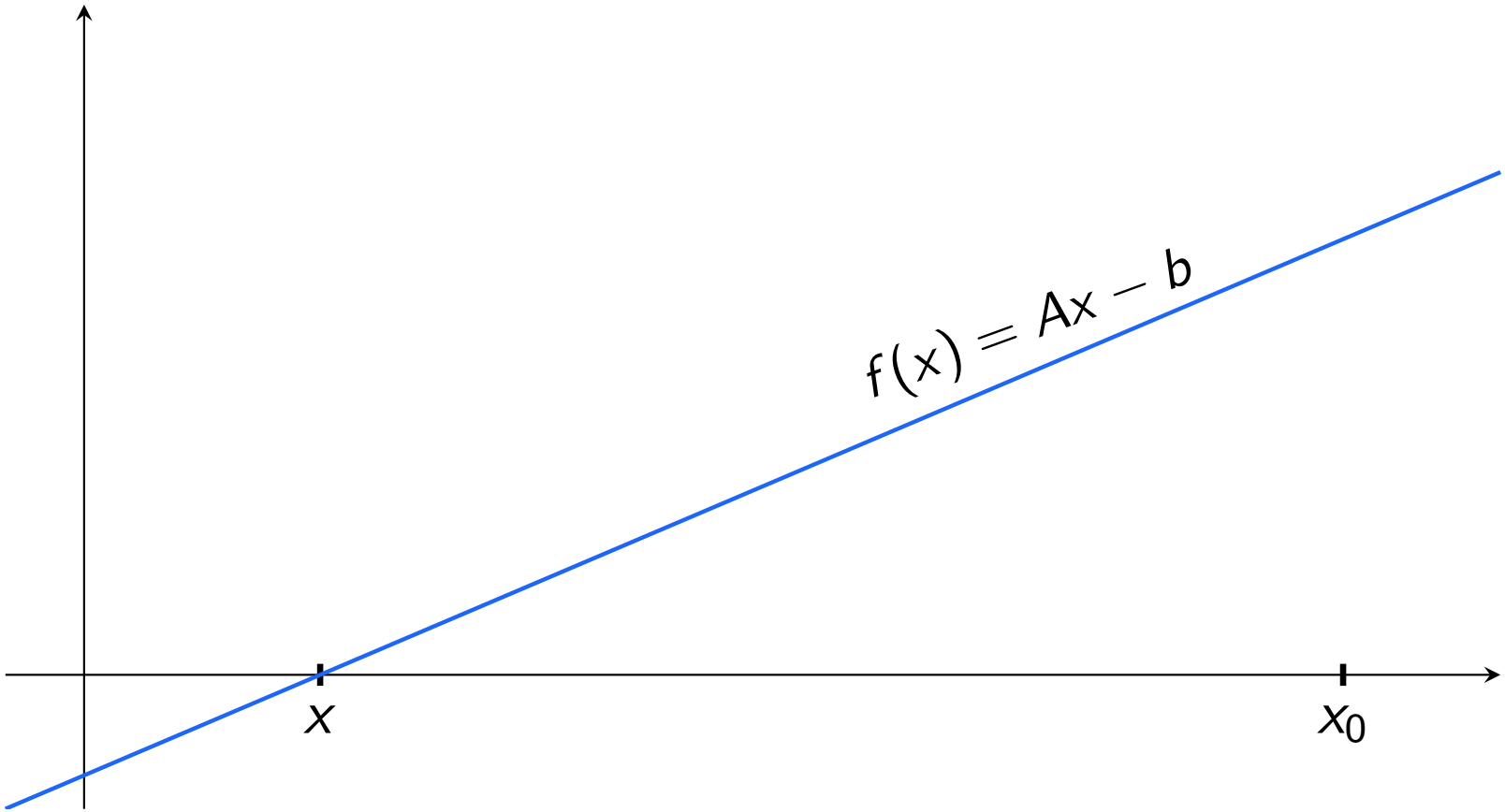
$\Rightarrow$ **What is the impact of these errors in the procedure ?**

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: $x_{i+1} = x_i + d_i$.

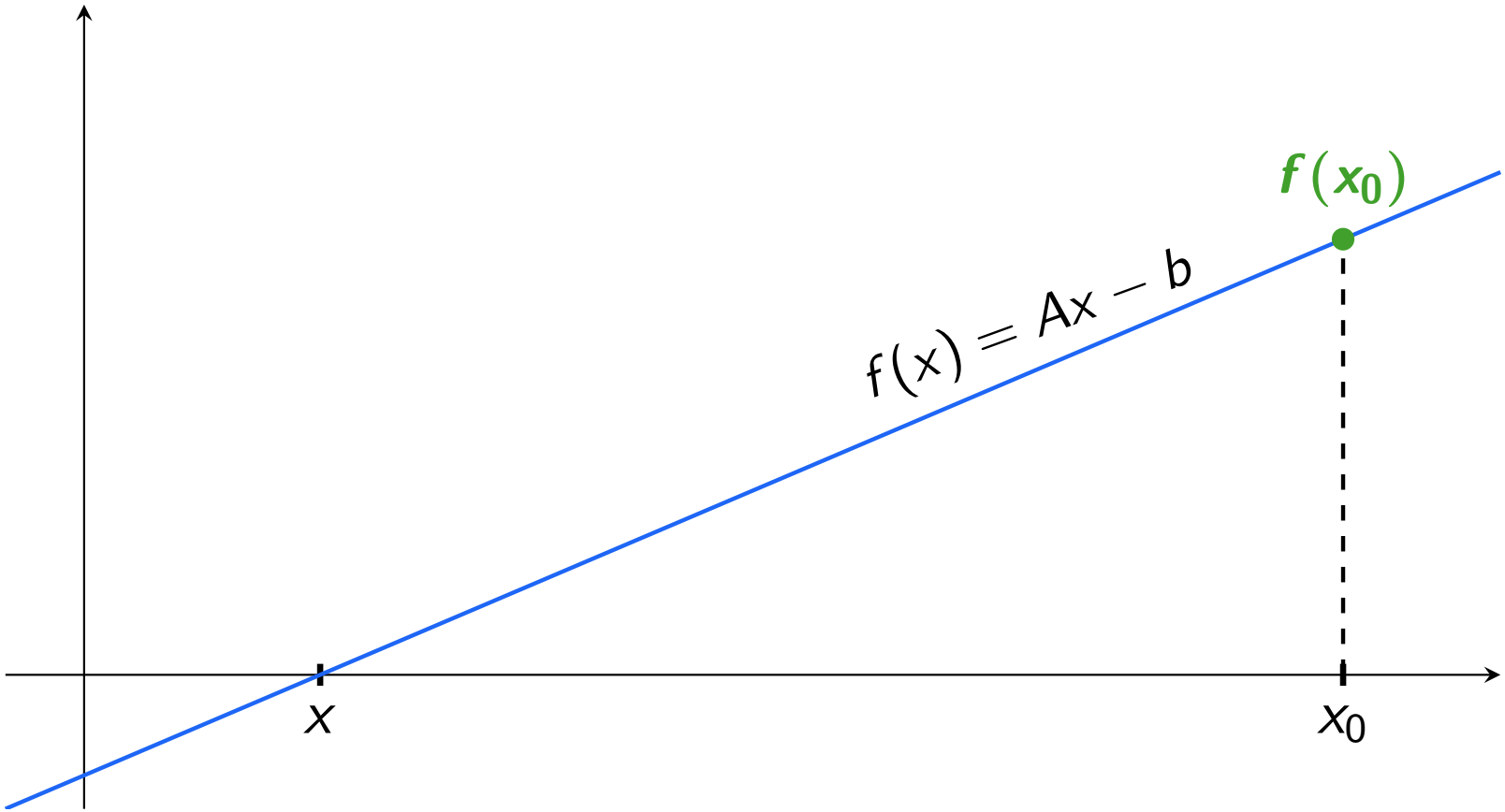


On the effect of rounding errors

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

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

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

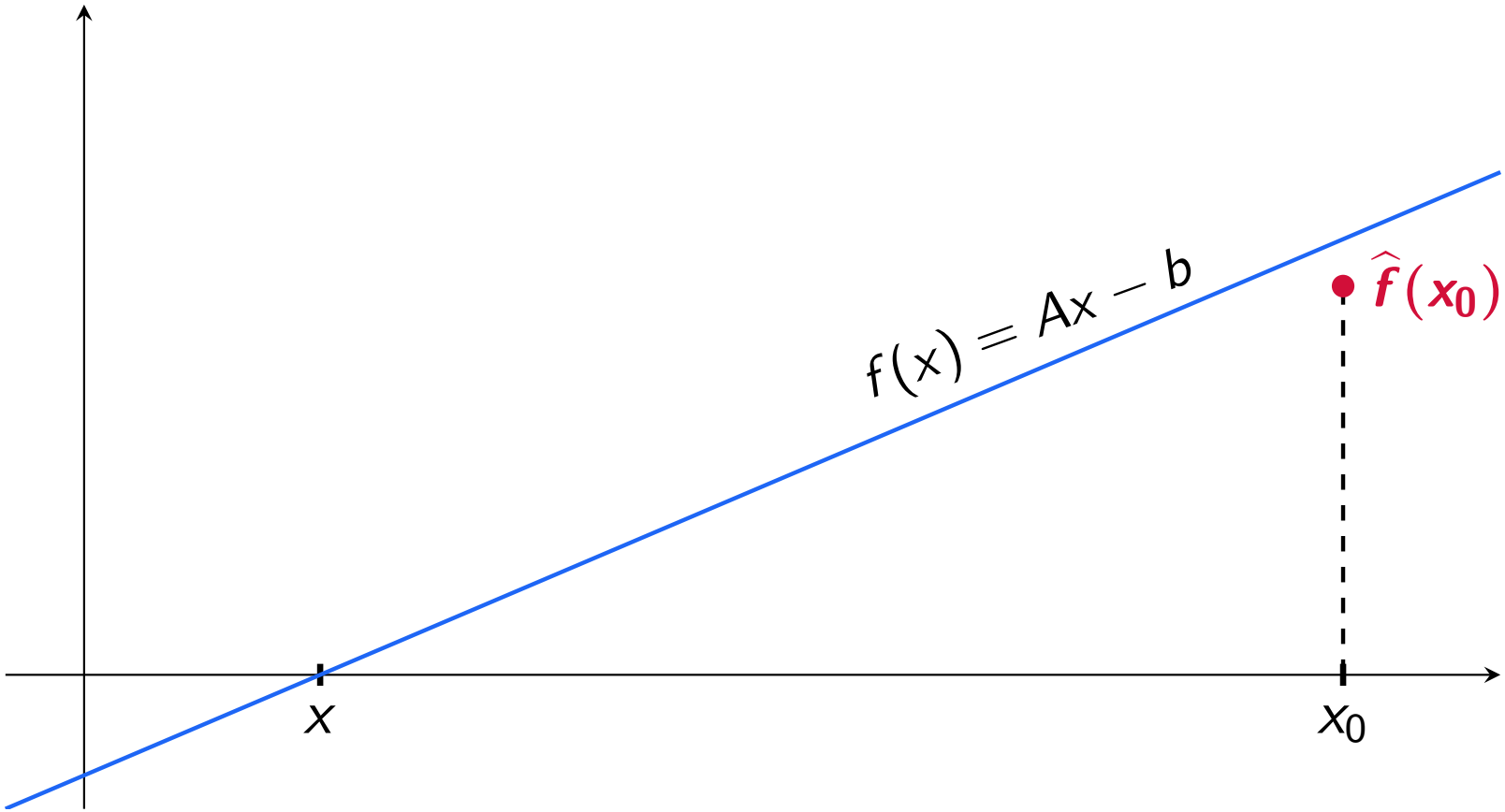


On the effect of rounding errors

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

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

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

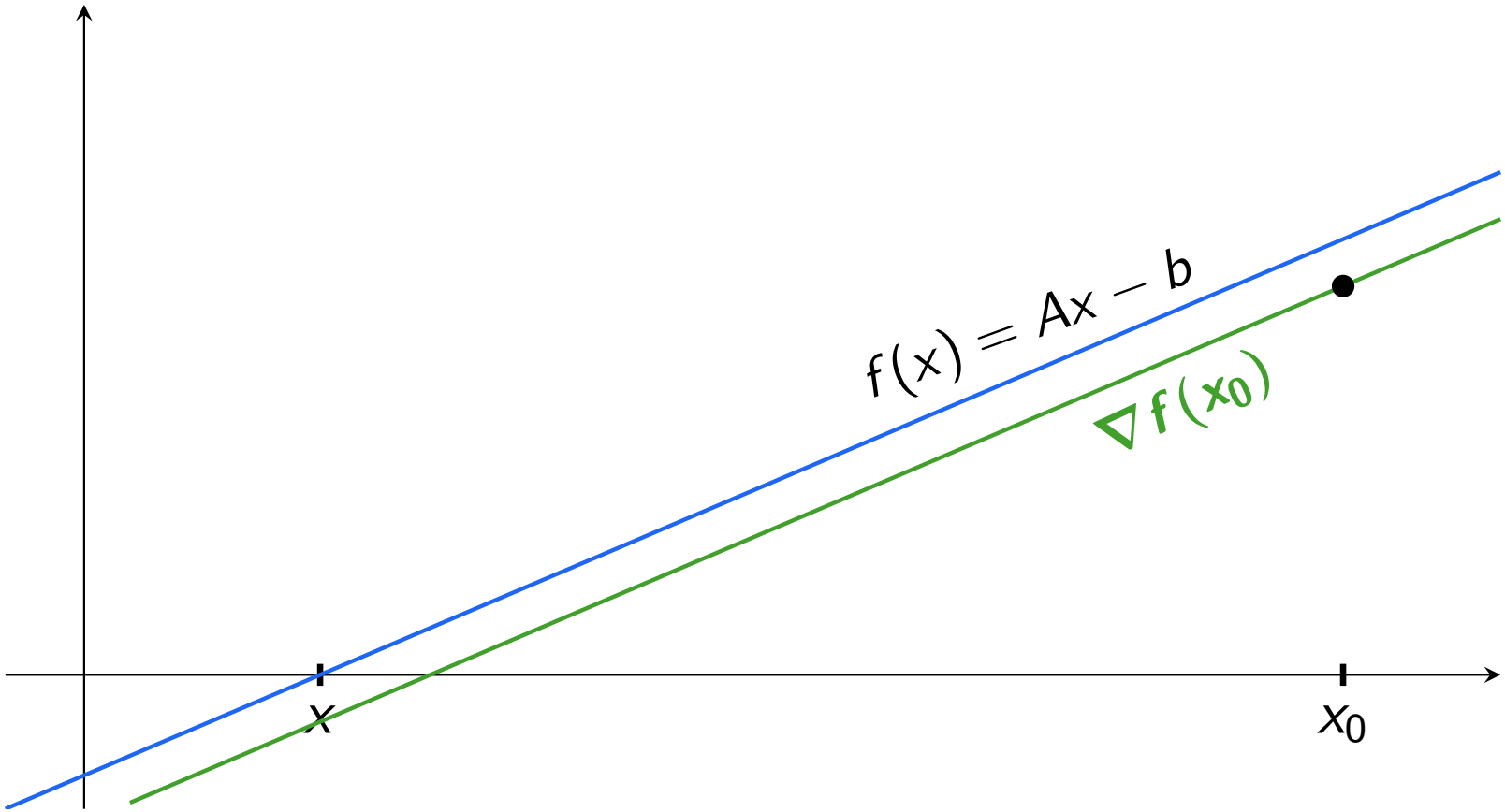


On the effect of rounding errors

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

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

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

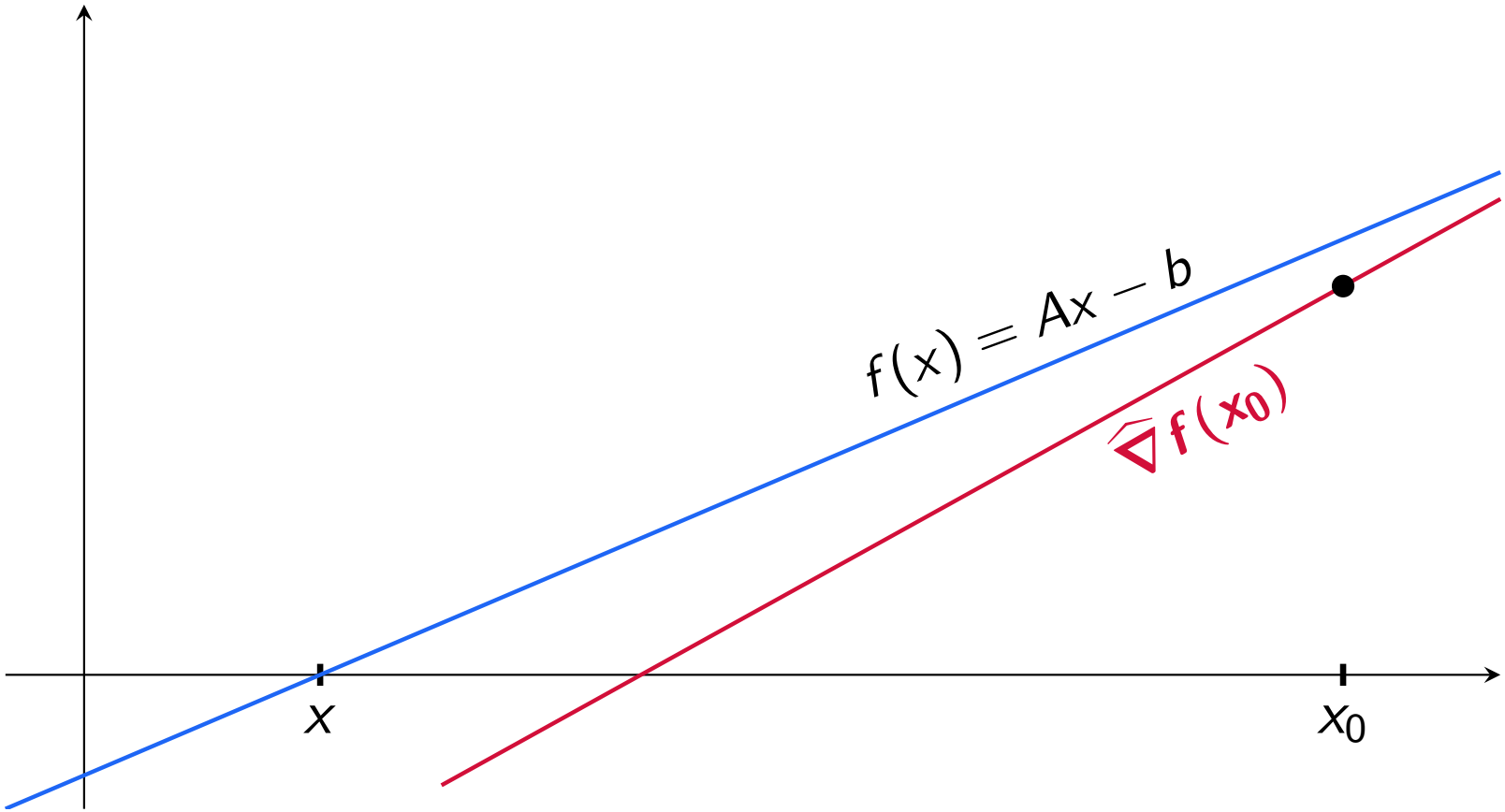


On the effect of rounding errors

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

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

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

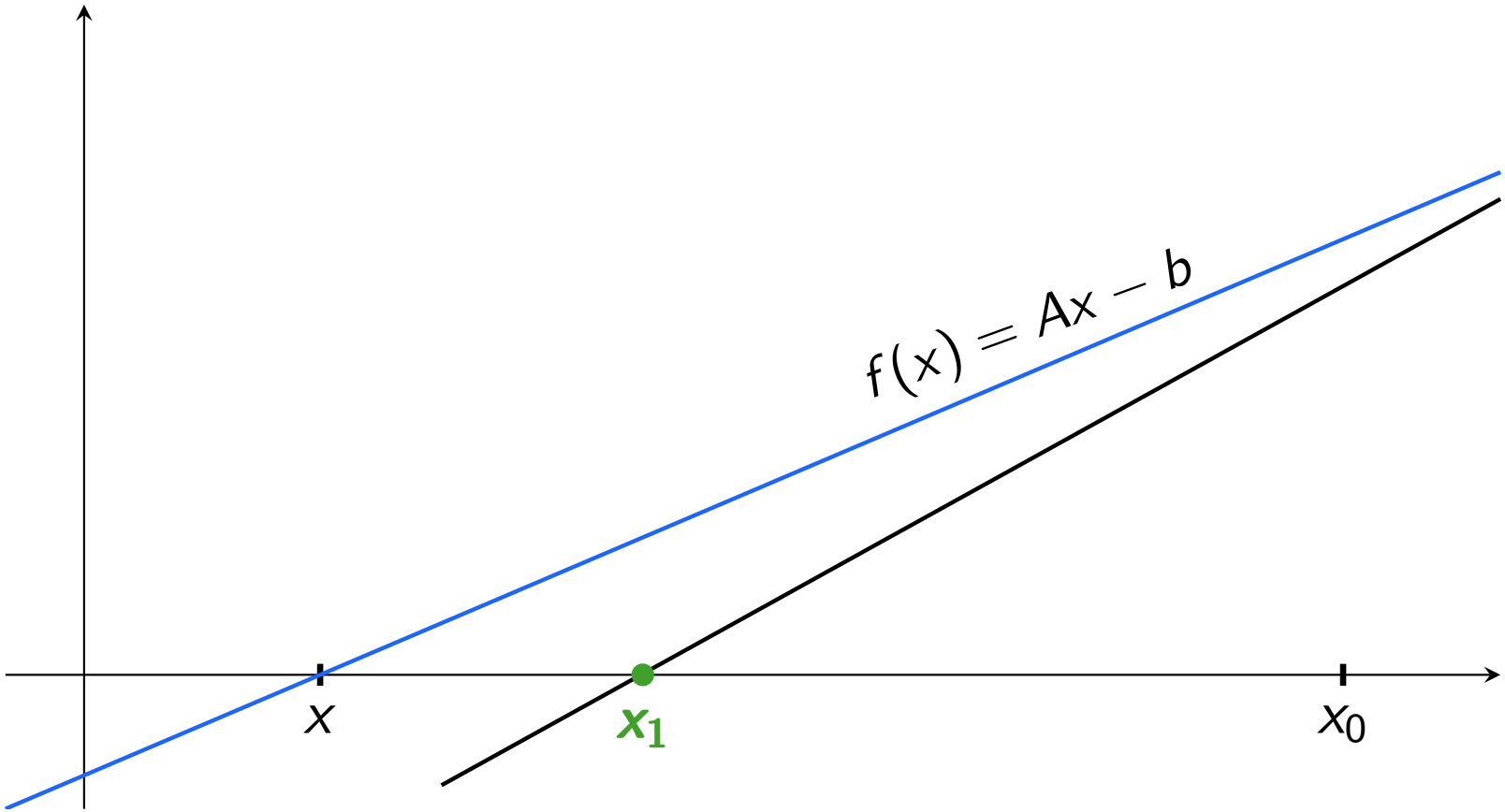


On the effect of rounding errors

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

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

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

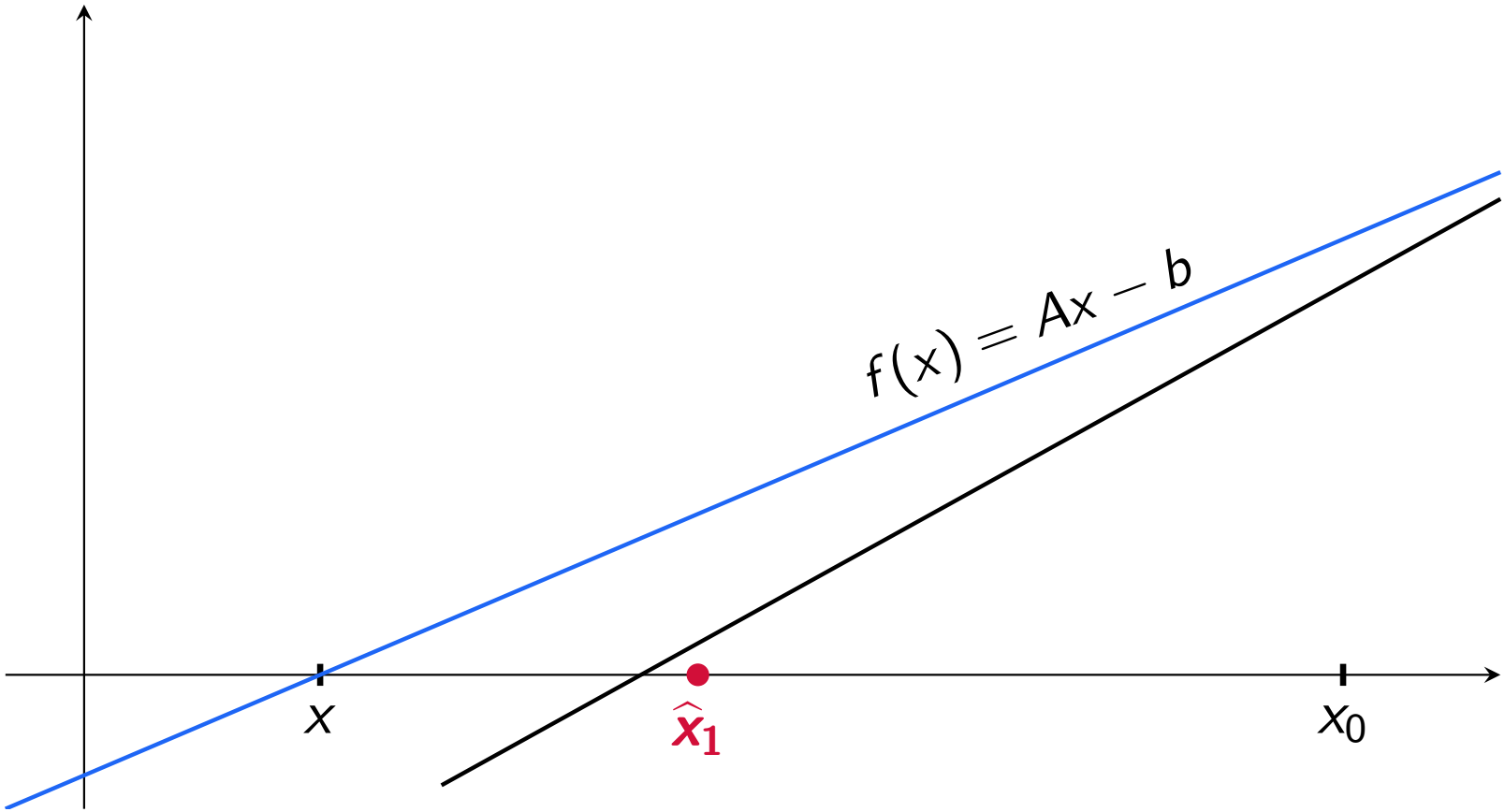


On the effect of rounding errors

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

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

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

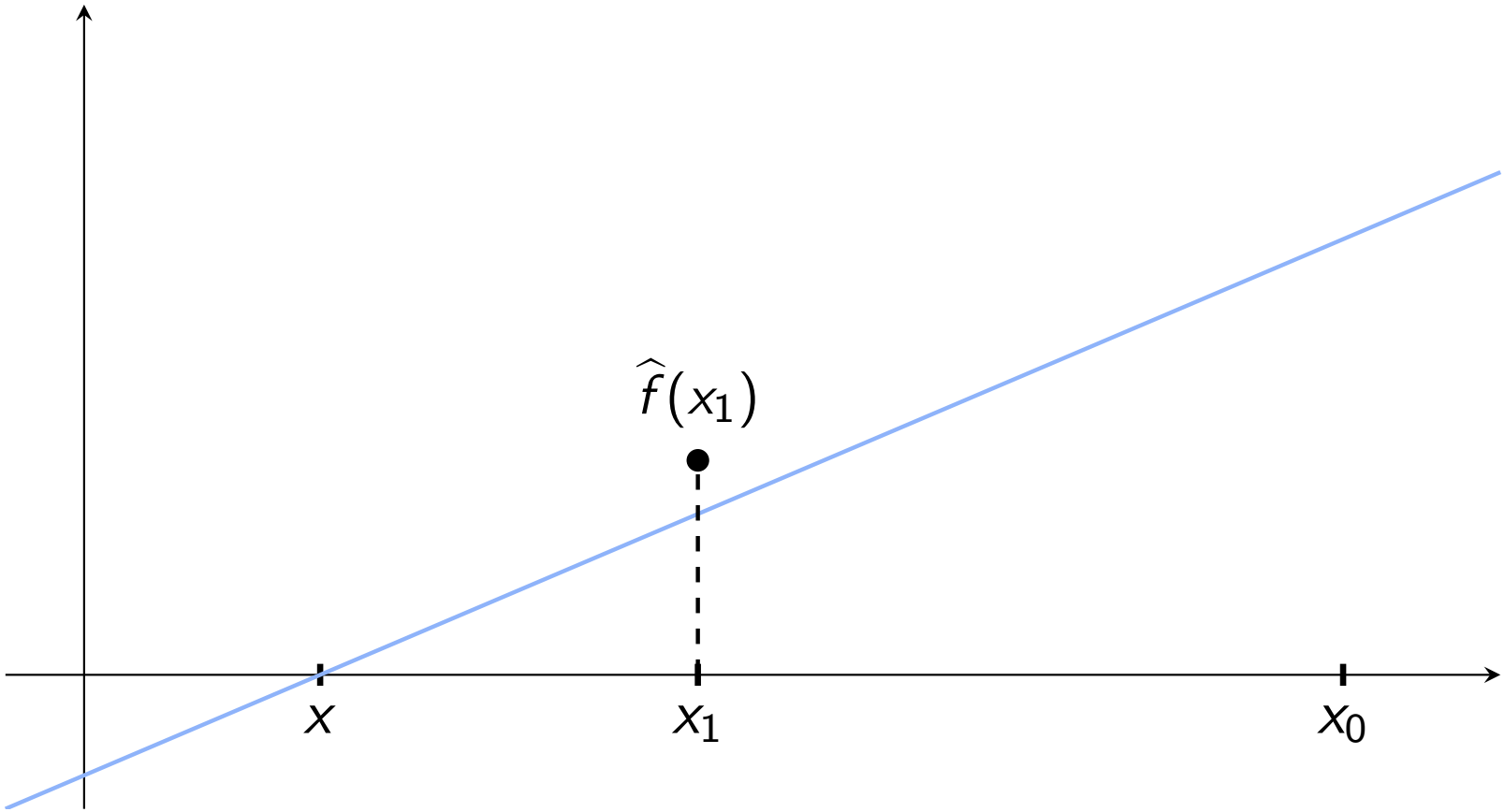


On the effect of rounding errors

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

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

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

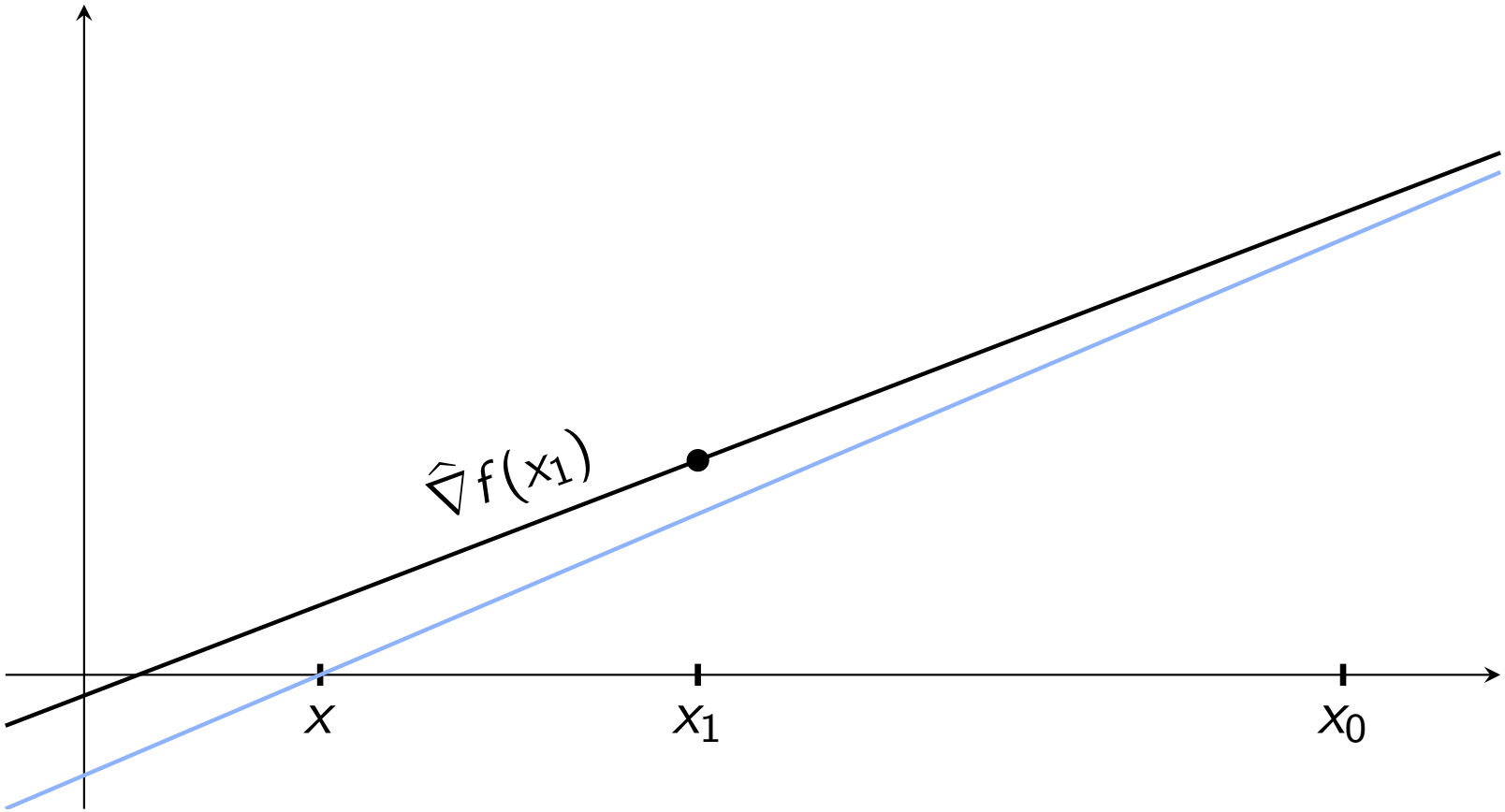


On the effect of rounding errors

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

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

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

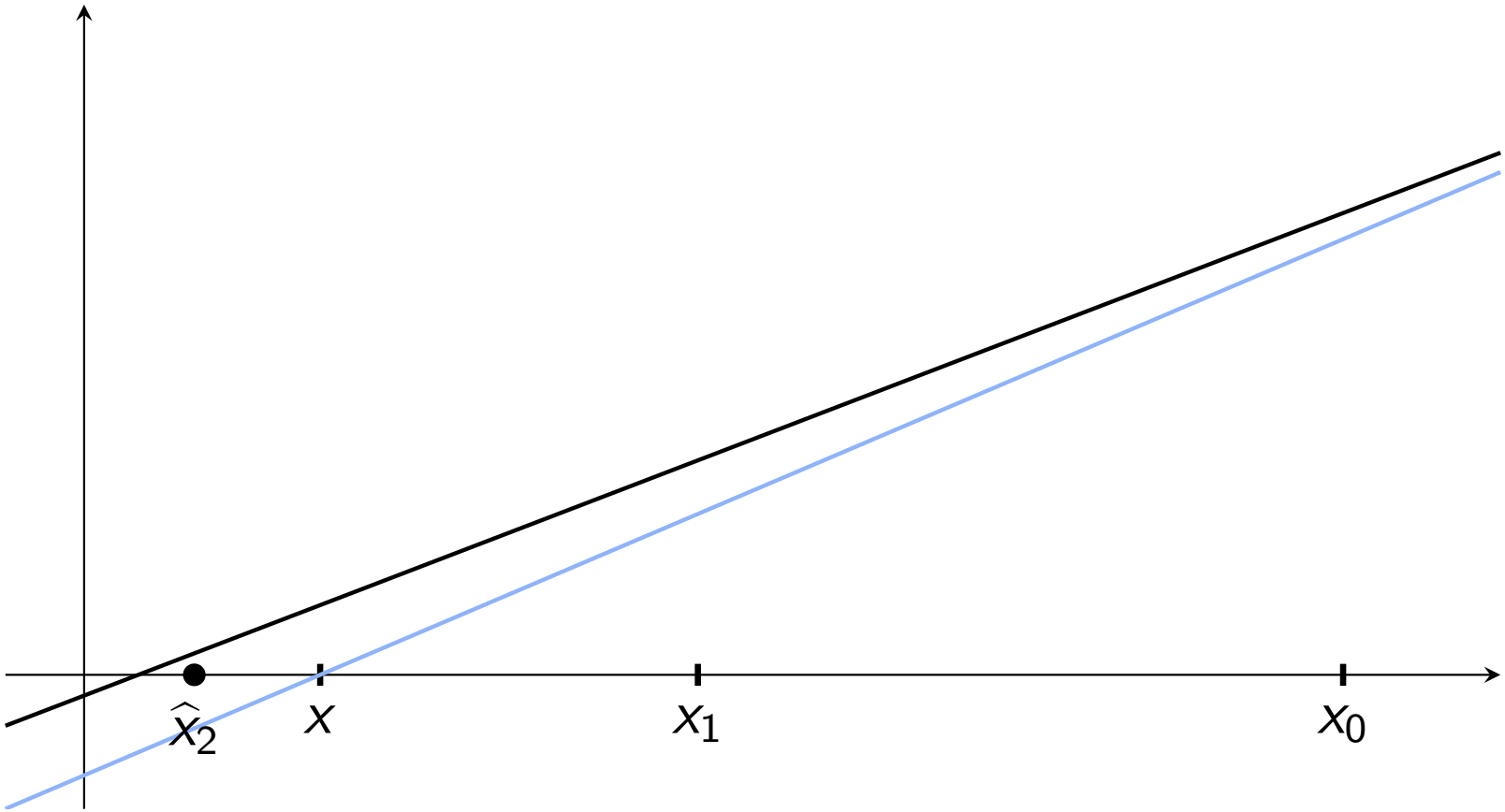


On the effect of rounding errors

(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: $x_{i+1} = x_i + d_i$.

$\Rightarrow$ **Iterative refinement can improve and correct the computed solutions of linear solvers under computing errors.**

Historical perspectives

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

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

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

$\Rightarrow$ **Iterative refinement can improve and correct the computed solutions of linear solvers under computing errors.**

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: $x_{i+1} = x_i + d_i$.

$\Rightarrow$ **Iterative refinement can improve and correct the computed solutions of linear solvers under computing errors.**

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!**

Section 4

From the 40s to the 70s

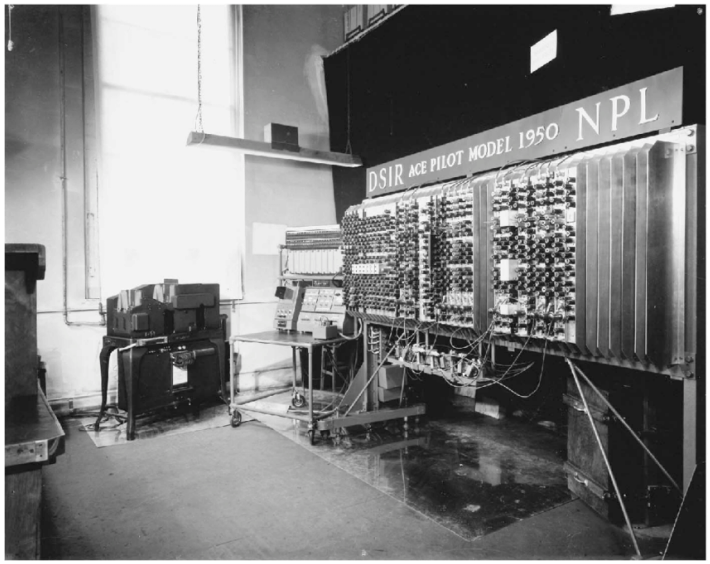
Origin

(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 **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 **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.

> "*In multiplying two s-place numbers, most computing machines do actually form the true 2s-place product, and the rounding off to s-places is a separate operation [...]*" **von Neumann and Goldstine**, 1947.

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**.

$$\frac{\|\widehat{x} - x\|_2}{\|x\|_2} \leq \kappa(A)\boldsymbol{u}_{\text{working}} \quad \Rightarrow \quad \frac{\|\widehat{x} - x\|_2}{\|x\|_2} \leq \boldsymbol{u}_{\text{working}}$$

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**.

$$\frac{\|\widehat{x} - x\|_2}{\|x\|_2} \leq \kappa(A)\boldsymbol{u}_{\text{working}} \quad \Rightarrow \quad \frac{\|\widehat{x} - x\|_2}{\|x\|_2} \leq \boldsymbol{u}_{\text{working}}$$

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., Gaussian elimination with partial pivoting):

Two main pieces of context

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

(2) Solving the **correction equation**: $\boldsymbol{d_i = U \backslash L \backslash r_i}$ (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., Gaussian elimination 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., Gaussian elimination with partial pivoting):

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

> "*Until recently, direct solution methods were often preferred to iterative methods in real applications because of their robustness and predictable behavior.*" **Saad**, 2000.

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., Gaussian elimination with partial pivoting):

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

➤ The factorization $A = LU$ ($\boldsymbol{O(n^3)}$) can be computed once, and the solve $U\backslash L\backslash r_i$ ($\boldsymbol{O(n^2)}$) are applied multiple times ⇒ **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 **Fox et al.**, 1948, The Quarterly Journal of Mechanics and Applied Mathematics.
- "*On the improvement of the solutions to a set of simultaneous linear equations using the ILLIAC*" by **Snyder**, 1955, Mathematical Tables and Other Aids to Computation.
- "*Solution of real and complex systems of linear equations*" by **Bowdler et al.**, 1966, Numerische Mathematik.
- "*Iterative refinement of the solution of a positive definite system of equations*" by **Martin et al.**, 1971, Linear Algebra.
- "*Introduction to matrix computations*" by **Stewart**, 1973, Academic Press.

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)

> "*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 [...]*" **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)

> "*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 [...]*" **Gauss**, 1809.

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

Founding article of rounding error analysis:

"*Numerical inverting of matrices of high order*" by **von Neumann and Goldstine**, 1947, Bulletin of the American Mathematical Society.

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 analyses of iterative refinement aim at both:

➤ Determine the **rate of convergence** of the refinement. The rate of convergence also define the **convergence condition**.

➤ Determine to which accuracy the solution will be refined. This is referred to as **attainable or 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 analyses of iterative refinement aim at both:

➤ Determine the **rate of convergence** of the refinement. The rate of convergence also define the **convergence condition**.

➤ Determine to which accuracy the solution will be refined. This is referred to as **attainable or limiting accuracies**.

Fixed point/block floating point:

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

Floating point:

"*Iterative refinement in floating point*" by **Moler**, 1967, Journal of the ACM.

Least squares problem: $\min_x \|b - Ax\|_2$

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

(2) Solving the **correction equation**: $\boldsymbol{d_i = R^{-1}Q^T r_i}$ (working)

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

➤ Consistent or nearly-consistent system approach:

"*Linear least squares solutions by Householder transformations*" by **Businger and Golub**, Numerische Mathematik.

"*Note on the iterative refinement of least squares solution*" by **Golub and Wilkinson**, 1966, Numerische Mathematik.

Least squares problem: $\min_x \|b - Ax\|_2$

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

(2) Solving the **correction equation**: $\boldsymbol{A^T A d_i = A^T r_i}$ (working)

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

➤ Consistent or nearly-consistent system approach:

"*Linear least squares solutions by Householder transformations*" by **Businger and Golub**, Numerische Mathematik.

"*Note on the iterative refinement of least squares solution*" by **Golub and Wilkinson**, 1966, Numerische Mathematik.

➤ Normal equation approach:

"*Note on the iterative refinement of least squares solution*" by **Golub and Wilkinson**, 1966, Numerische Mathematik.

Least squares problem: $\min_x \|b - Ax\|_2$

(1) Computing the **residual**: $\begin{bmatrix} h_i \\ g_i \end{bmatrix} = \begin{bmatrix} b - r_i - Ax_i \\ -A^T r_i \end{bmatrix}$ (extra)

(2) Solving the **correction equation**: $\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} d_{i,1} \\ d_{i,2} \end{bmatrix} = \begin{bmatrix} h_i \\ g_i \end{bmatrix}$ (working)

(3) **Updating** the solution: $\begin{bmatrix} r_{i+1} \\ x_{i+1} \end{bmatrix} = \begin{bmatrix} r_i \\ x_i \end{bmatrix} + \begin{bmatrix} d_{i,1} \\ d_{i,2} \end{bmatrix}.$ (working)

- Augmented system approach:

"*Iterative refinement of linear least squares solutions I*" by **Björck**, 1967, BIT.

"*On the Iterative Refinement of Least Squares Solutions*" by **Fletcher**, 1975, Journal of the American Statistical Association.

Section 5

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.

> "*The primary drawback of mixed precision iterative improvement is that its implementation is somewhat machine-dependent. This discourages its use in software that is intended for wide distribution.*" **Golub and van Loan**, 1996.

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:

➤ Double precision circuitry was optimized and not much slower than single precision. Data transfer was fast and not a concern. $\Rightarrow$ **Single not faster than double** precision.

> "*In the past, [...] the load was on the floating point processing units rather than the memory subsystem, and so the single precision data motion advantages were for the most part irrelevant.*" **Buttari et al.**, 2007.

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:

➤ **Iterative methods** (CG, GMRES, LSQR, etc.) were on the rise, and **unstable** direct methods were more and more considered to target parallel computing and sparse data structures.

> "*[...] the increased need for solving very large linear systems triggered a noticeable and rapid shift toward iterative techniques in many applications. This trend can be traced back to the 1960s and 1970s [...]*" **Saad**, 2000.

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:

$\Rightarrow$ **A new form of IR emerged where every operations are run 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.*" **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.*" **Moler**, 1967.

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

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

"*Iterative refinement implies numerical stability for gaussian elimination*" by **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 **Jankowski and 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 **Turner and 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 **Zlatev**, 1982, SIAM Journal on Numerical Analysis.

"*Solving Sparse Linear Systems with Sparse Backward Error*" by **Arioli et al.**, 1989, SIAM Journal on Matrix Analysis and Applications.

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 **Li and Demmel**, 1998.

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 $Ad_i = r_i$!**

Without numerical pivoting:

"*On the Stability of Cholesky Factorization for Symmetric Quasidefinite Systems*" by **Gill et al.**, 1996, SIAM Journal on Matrix Analysis and Applications.

A software standard

(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)

Fixed precision iterative refinement is **embedded in many major software**:

- Dense linear system: **Lapack** (with routines whose names end in -rfs and called by the expert drivers whose names end in -svx).
- Sparse linear system: **MUMPS**, **PaStiX**, or **SuperLU**.

"*Analysis and comparison of two general sparse solvers for distributed memory computers*" by **Amestoy et al.**, 2001, ACM Transactions on Mathematical Software.

Other applications

(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)

➤ For computation of eigenpairs:

"*Improving the Accuracy of Computed Eigenvalues and Eigenvectors*" by **Dongarra et al.**, 1983, SIAM Journal on Numerical Analysis.

Other applications

(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)

- For computation of eigenpairs:

  "*Improving the Accuracy of Computed Eigenvalues and Eigenvectors*" by **Dongarra et al.**, 1983, SIAM Journal on Numerical Analysis.

- In block elimination methods:

  "*Block elimination with one refinement solves bordered linear systems accurately*" by **Govaerts and Pryce**, 1990, BIT.

Other applications

(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)

- For computation of eigenpairs:

  "*Improving the Accuracy of Computed Eigenvalues and Eigenvectors*" by **Dongarra et al.**, 1983, SIAM Journal on Numerical Analysis.

- In block elimination methods:

  "*Block elimination with one refinement solves bordered linear systems accurately*" by **Govaerts and Pryce**, 1990, BIT.

- For fault-tolerant computing:

  "*Floating point fault tolerance with backward error assertions*" by **Boley et al.**, 1995, IEEE Transactions on Computers.

Other applications

(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)

- For computation of eigenpairs:

  "*Improving the Accuracy of Computed Eigenvalues and Eigenvectors*" by **Dongarra et al.**, 1983, SIAM Journal on Numerical Analysis.

- In block elimination methods:

  "*Block elimination with one refinement solves bordered linear systems accurately*" by **Govaerts and Pryce**, 1990, BIT.

- For fault-tolerant computing:

  "*Floating point fault tolerance with backward error assertions*" by **Boley et al.**, 1995, IEEE Transactions on Computers.

- For Vandermonde-like systems:

  "*Fast Solution of Vandermonde-Like Systems Involving Orthogonal Polynomials*" by **Higham**, 1988, IMA Journal of Numerical Analysis.

(Too) Early (Too) pioneering studies

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

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

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

Often forgotten, these work were the first to solve the **correction equation in lower precision!**

- "*Iterative refinement for linear systems in variable-precision arithmetic*" by **Kiełbasiński**, 1981, BIT.
- "*Fast Hybrid Solution of Algebraic Systems*" by **Douglas et al.**, 1990, SIAM Journal on Scientific and Statistical Computing.
- "*Efficient High Accuracy Solutions with GMRES(m)*" by **Turner and Walker**, 1992, SIAM Journal on Scientific and Statistical Computing.

(Too) Early (Too) pioneering studies

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

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

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

Often forgotten, these work were the first to solve the **correction equation in lower precision!**

"*Iterative refinement for linear systems in variable-precision arithmetic*" by **Kiełbasiński**, 1981, BIT.

"*Fast Hybrid Solution of Algebraic Systems*" by **Douglas et al.**, 1990, SIAM Journal on Scientific and Statistical Computing.

"*Efficient High Accuracy Solutions with GMRES(m)*" by **Turner and Walker**, 1992, SIAM Journal on Scientific and Statistical Computing.

This approach will become very important...

Section 6

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.*" **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**!

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)

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 $\boldsymbol{Ax} = \boldsymbol{b}$ **twice faster** while providing **double precision accuracy**.

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

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)

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 $\boldsymbol{Ax} = \boldsymbol{b}$ **twice faster** while providing **double precision accuracy**.

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

$\Rightarrow$ **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 **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 **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.

Examples of application

(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)

➤ Accelerate direct solvers:

"*Exploiting fast hardware floating point in high precision computation*" by **Geddes and Zheng**, 2003, ISSAC '03.

"*Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy*" by **Buttari et al.**, 2008, ACM TOMS.

Examples of application

(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)

- Accelerate direct solvers:

  "*Exploiting fast hardware floating point in high precision computation*" by **Geddes and Zheng**, 2003, ISSAC '03.

  "*Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy*" by **Buttari et al.**, 2008, ACM TOMS.

- Accelerate iterative solvers:

  "*Mixed Precision Iterative Refinement Methods for Linear Systems: Convergence Analysis Based on Krylov Subspace Methods.*" by **Anzt et al.**, 2012.

Examples of application

(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)

- Accelerate direct solvers:

  "*Exploiting fast hardware floating point in high precision computation*" by **Geddes and Zheng**, 2003, ISSAC '03.

  "*Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy*" by **Buttari et al.**, 2008, ACM TOMS.

- Accelerate iterative solvers:

  "*Mixed Precision Iterative Refinement Methods for Linear Systems: Convergence Analysis Based on Krylov Subspace Methods.*" by **Anzt et al.**, 2012.

- Accelerate multigrid solvers:

  "*Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*" by **Göddeke et al.**, 2007, International Journal of Parallel, Emergent and Distributed Systems.

The comeback of extra precision

(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)

Extra precision iterative refinement was disregarded due to the **lack of portability of extra precision** from hardware to hardware.

The comeback of extra precision

(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)

Extra precision iterative refinement was disregarded due to the **lack of portability of extra precision** from hardware to hardware.

HOWEVER, major **software evolutions** in the 2000s made extra precision accessible and portable again: **XBLAS** standard, **double-double arithmetic**, **Quad-Precision Math Library** (GNU), or **Intel compilers** support for fp128.

> "*However, with the release of the Extended and Mixed Precision BLAS (see §27.10) and the portable reference implementation for IEEE arithmetic, [...] portable mixed precision iterative refinement is now achievable*" **Higham**, 2002.

The comeback of extra precision

(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)

Extra precision iterative refinement was disregarded due to the **lack of portability of extra precision** from hardware to hardware.

HOWEVER, major **software evolutions** in the 2000s made extra precision accessible and portable again: **XBLAS** standard, **double-double arithmetic**, **Quad-Precision Math Library** (GNU), or **Intel compilers** support for fp128.

"*Error bounds from extra-precise iterative refinement*" by **Demmel et al.**, 2006, ACM TOMS.

"*Newton's Method in Floating Point Arithmetic and Iterative Refinement of Generalized Eigenvalue Problems*" by **Tisseur**, 2001, SIAM Journal on Matrix Analysis and Applications.

Section 7

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**:

> "*However, today's dominant market is AI, which does not require the high-precision arithmetic long common in computational modeling and is leading the design of chips with lower-precision arithmetic [...]*" **Deelman et al.**, 2025.

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**:

➤ We can generally access at least four different arithmetics on modern supercomputers: fp16, fp32, fp64, fp128 (say).

$\Rightarrow$ **Optimizing the computer performance pass by 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**:

➤ We can generally access at least four different arithmetics on modern supercomputers: fp16, fp32, fp64, fp128 (say).

$\Rightarrow$ **Optimizing the computer performance pass by exploiting and combining efficiently each of these arithmetics!**

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

$\Rightarrow$ **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**:

➤ We can generally access at least four different arithmetics on modern supercomputers: fp16, fp32, fp64, fp128 (say).

$\Rightarrow$ **Optimizing the computer performance pass by exploiting and combining efficiently each of these arithmetics!**

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

$\Rightarrow$ **Half precision cannot be used alone!**

$\Rightarrow$ **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.*" **Carson and 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 **Carson and Higham**, 2017.

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

Already ubiquitous in our software

(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)

➤ Is used in the **HPL-MxP** benchmark (2025):

"*Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*" by **Haidar at al.**, 2018.

Rank	Computer name	Eflop/s
1	El Capitan	16.680
2	Aurora	11.643
3	Frontier	11.390

Already ubiquitous in our software

(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)

- Is used in the **HPL-MxP** benchmark (2025):

  "*Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*" by **Haidar at al.**, 2018.

Rank	Computer name	Eflop/s
1	El Capitan	16.680
2	Aurora	11.643
3	Frontier	11.390

- Is used with the MUMPS sparse direct solver:

  "*Combining sparse approximate factorization with mixed precision iterative refinement*" by **Amestoy et al.**, 2023, ACM TOMS.

Already ubiquitous in our software

(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)

- Is used in the **HPL-MxP** benchmark (2025):

  "*Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*" by **Haidar at al.**, 2018.

Rank	Computer name	Eflop/s
1	El Capitan	16.680
2	Aurora	11.643
3	Frontier	11.390

- Is used with the MUMPS sparse direct solver:

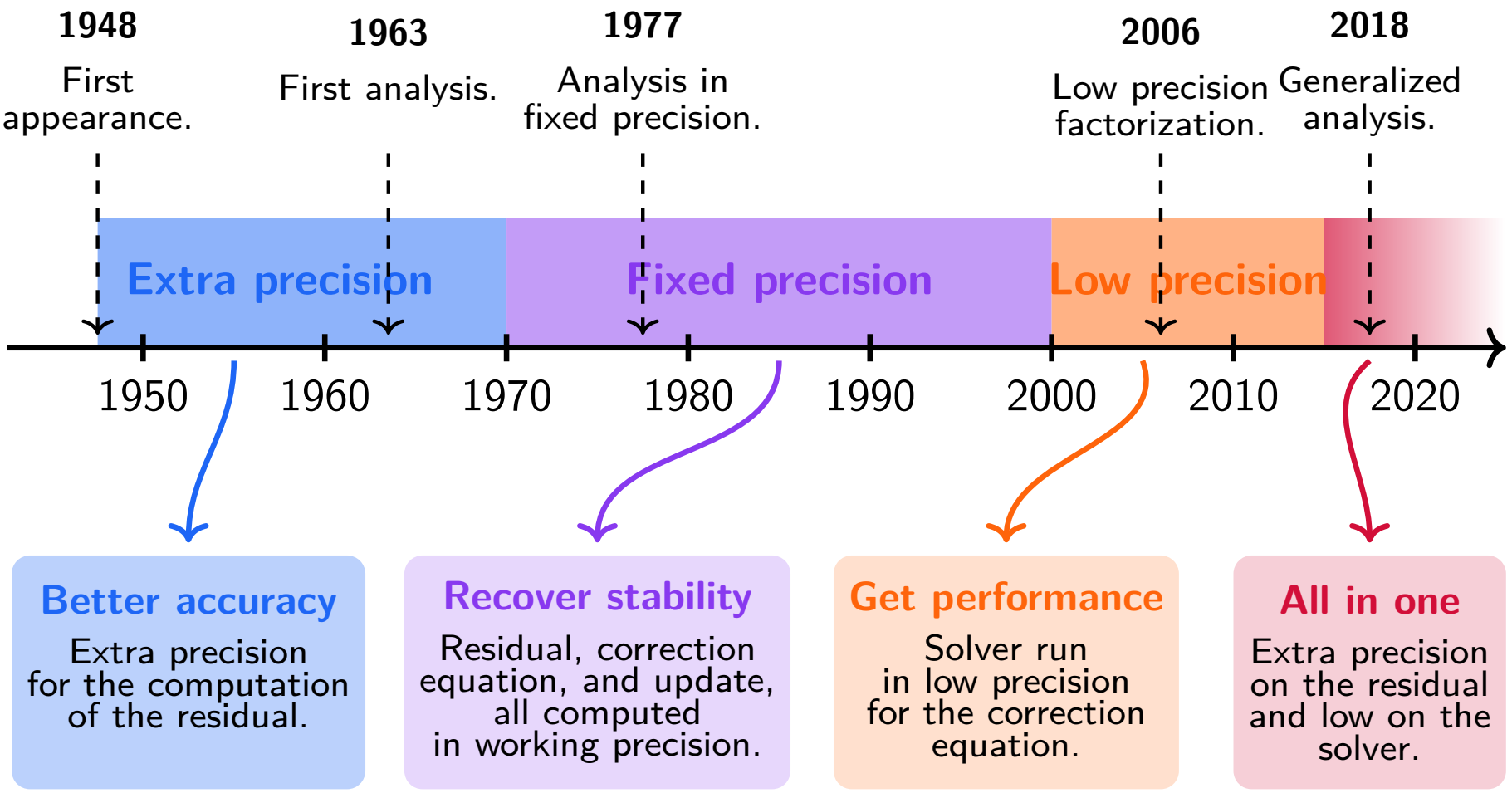
  "*Combining sparse approximate factorization with mixed precision iterative refinement*" by **Amestoy et al.**, 2023, ACM TOMS.

- Available in the **MAGMA**, **NVIDIA cuSolver**, or **SLATE** libraries.

Still an ongoing topic!

- "*Improving the Performance of the GMRES Method Using Mixed-Precision Techniques*" by **Lindquist at al.**, 2020.
- "*Accelerating Geometric Multigrid Preconditioning with Half-Precision Arithmetic on GPUs*" by **Oo and Vogel**, 2020.
- "*Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems.*" by **Higham and Pranesh**, 2021.
- "*Mixed Precision GMRES-based Iterative Refinement with Recycling*" by **Oktay and Carson**, 2022.
- "*A modular framework for the backward error analysis of GMRES*" by **Buttari et al.**, 2025.
- "*A comparison of mixed precision iterative refinement approaches for least-squares problems*" by **Carson and Daužickaitė**, 2025.
- "*AIR: Iterative refinement acceleration using arbitrary dynamic precision*" by **Lee et al.**, 2020.
- "*Mixed precision iterative refinement for least squares with linear equality constraints and generalized least squares problems*" by **Gao et al.**, 2024.

History summary



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

Section 8

Mixed Precision Alternating-Direction Implicit Methods

Background on ADI methods

- **Alternating-Direction Implicit (ADI)** methods are iterative solvers for $Ax = b$ coming from PDEs.

Background on ADI methods

- **Alternating-Direction Implicit (ADI)** methods are iterative solvers for $Ax = b$ coming from PDEs.
- They **split the operator spatially**. E.g., they solve a 2D linear system by reducing the problem to solving a sequence of 1D simpler linear systems.

Background on ADI methods

- **Alternating-Direction Implicit (ADI)** methods are iterative solvers for $Ax = b$ coming from PDEs.
- They **split the operator spatially**. E.g., they solve a 2D linear system by reducing the problem to solving a sequence of 1D simpler linear systems.
- They belong to the class of **matrix splitting stationary iterative methods** (e.g., Jacobi, SOR, Gauss-Seidel).

Background on ADI methods

- **Alternating-Direction Implicit (ADI)** methods are iterative solvers for $Ax = b$ coming from PDEs.
- They **split the operator spatially**. E.g., they solve a 2D linear system by reducing the problem to solving a sequence of 1D simpler linear systems.
- They belong to the class of **matrix splitting stationary iterative methods** (e.g., Jacobi, SOR, Gauss-Seidel).
- We consider a more **general case** where the operators are **not necessarily obtained from spatial splitting**.

GADI: General Alternating-Direction Implicit Framework

GADI iteration

Consider solving iteratively

$$Ax = b$$

Given a splitting $A = M + N$, parameters $\alpha > 0$ and $\omega \in [0, 2)$, a GADI iteration is

$$(\alpha I + M)x_{k+1/2} = (\alpha I - N)x_k + b,$$
$$(\alpha I + N)x_{k+1} = (N - (1 - \omega)\alpha I)x_k + (2 - \omega)\alpha x_{k+1/2}.$$

Covers various ADI methods: Peaceman-Rachford, Douglas-Rachford, Normal/skew-Hermitian splitting, Hermitian/skew-Hermitian splitting for Sylvester and Lyapunov equations.

"*A General Alternating-Direction Implicit Framework with Gaussian Process Regression Parameter Prediction for Large Sparse Linear Systems*" by **Jiang et al.**, 2022.

GADI as a stationary iterative method

A GADI iteration can be rewritten as

$$x_{k+1} = T(\alpha, \omega)x_k + G(\alpha, \omega),$$

with

$$T(\alpha, \omega) = (\alpha I + N)^{-1}(\alpha I + M)^{-1}(\alpha^2 I + MN - (1-\omega)\alpha A),$$
$$G(\alpha, \omega) = (2-\omega)\alpha(\alpha I + N)^{-1}(\alpha I + M)^{-1} b.$$

Proposition

The asymptotic convergence rate of GADI is governed by $\rho(T(\alpha, \omega))$. Additionally, under good condition on the splitting $M + N$, we guarantee $\rho(T(\alpha, \omega)) < 1$ for all $\alpha > 0$ and $\omega \in [0, 2)$.

GADI in residual form and iterative refinement

Algorithm: GADI in residual form

while not converged **do**
    Compute the residual $r_k = b - Ax_k$.
    Solve $(\alpha I + M)z_k = r_k$.
    Solve $(\alpha I + N)d_k = (2 - \omega)\alpha z_k$.
    Compute the next iterate $x_{k+1} = x_k + d_k$.
end while

GADI in residual form and iterative refinement

Algorithm: GADI in residual form

1: **while** not converged **do**
2: Compute the residual $r_k = b - Ax_k$.
3: Solve $(\alpha I + M)z_k = r_k$.
4: Solve $(\alpha I + N)d_k = (2 - \omega)\alpha z_k$.
5: Compute the next iterate $x_{k+1} = x_k + d_k$.
6: **end while**

Algorithm: Mixed precision iterative refinement

1: **while** not converged **do**
2: Compute the residual $r_k = b - Ax_k$. ▷ u_r
3: Solve $Ad_k = r_k$. ▷ u_s
4: Compute the next iterate $x_{k+1} = x_k + d_k$. ▷ u
5: **end while**

GADI in residual form and iterative refinement

Algorithm: Mixed precision GADI

1: **while** not converged **do**
2: Compute the residual $r_k = b - Ax_k$. ▷ u_r
3: Solve $(\alpha I + M)z_k = r_k$. ▷ u_s
4: Solve $(\alpha I + N)d_k = (2 - \omega)\alpha z_k$. ▷ u_s
5: Compute the next iterate $x_{k+1} = x_k + d_k$. ▷ u
6: **end while**

Algorithm: Mixed precision iterative refinement

1: **while** not converged **do**
2: Compute the residual $r_k = b - Ax_k$. ▷ u_r
3: Solve $Ad_k = r_k$. ▷ u_s
4: Compute the next iterate $x_{k+1} = x_k + d_k$. ▷ u
5: **end while**

Rounding error analysis of mixed precision GADI

Assume the linear solvers are backward stable:

$$(\alpha I + M + \Delta M)\widehat{z}_k = \widehat{r}_k, \qquad \|\Delta M\| \leq c(n)\boldsymbol{u_s}\|\alpha I + M\|,$$
$$(\alpha I + N + \Delta N)\widehat{d}_k = (2-\omega)\alpha\widehat{z}_k, \qquad \|\Delta N\| \leq c(n)\boldsymbol{u_s}\|\alpha I + N\|.$$

Theorem

For all GADI iteration $k \geq 1$, the computed iterate $\widehat{x}_{k+1}$ satisfies

$$\|x - \widehat{x}_{k+1}\| \leq \beta\|x - \widehat{x}_k\| + \zeta\|x\|,$$

with

$$\beta \approx \frac{\|T(\alpha,\omega)(x - \widehat{x}_k)\|}{\|x - \widehat{x}_k\|} + c(n)\kappa(\alpha I + M)\kappa(\alpha I + N)\boldsymbol{u_s}^2,$$
$$\zeta \approx c(n)(\boldsymbol{u} + \kappa(A)\boldsymbol{u_r}).$$

[2] Simplified expression using other assumptions listed in the preprint.

Comparison to previous analysis

Other mixed precision ADI: "*Towards a mixed-precision ADI method for Lyapunov equations*" by **Schulze and Saak**, 2025.

Comparison to previous analysis

Other mixed precision ADI: "*Towards a mixed-precision ADI method for Lyapunov equations*" by **Schulze and Saak**, 2025.

Comparison to iterative refinement: "*Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions*" by **Carson and Higham**, 2018.

$=$ We solve the GADI subsytems inaccurately while still computing a highly accurate solution.

$\neq$ Strong incentive to use very low precision $\boldsymbol{u_s}$ since having

$$\kappa(\alpha I + M)\kappa(\alpha I + N)\boldsymbol{u_s} \ll \frac{\|T(\alpha,\omega)(x - \widehat{x}_k)\|}{\|x - \widehat{x}_k\|} \xrightarrow{k\to\infty} \rho(T(\alpha,\omega))$$

is numerically useless.

Comparison to previous analysis

Other mixed precision ADI: "*Towards a mixed-precision ADI method for Lyapunov equations*" by **Schulze and Saak**, 2025.

Comparison to iterative refinement: "*Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions*" by **Carson and Higham**, 2018.

$=$ We solve the GADI subsytems inaccurately while still computing a highly accurate solution.

$\neq$ Strong incentive to use very low precision $\boldsymbol{u_s}$ since having

$$\kappa(\alpha I + M)\kappa(\alpha I + N)\boldsymbol{u_s} \ll \frac{\|T(\alpha,\omega)(x - \widehat{x}_k)\|}{\|x - \widehat{x}_k\|} \xrightarrow{k\to\infty} \rho(T(\alpha,\omega))$$

is numerically useless.

Comparison to stationary iterations: "*On the Numerical Behavior of Matrix Splitting Iteration Methods for Solving Linear Systems*" by **Bai and Rozložník**, 2015.

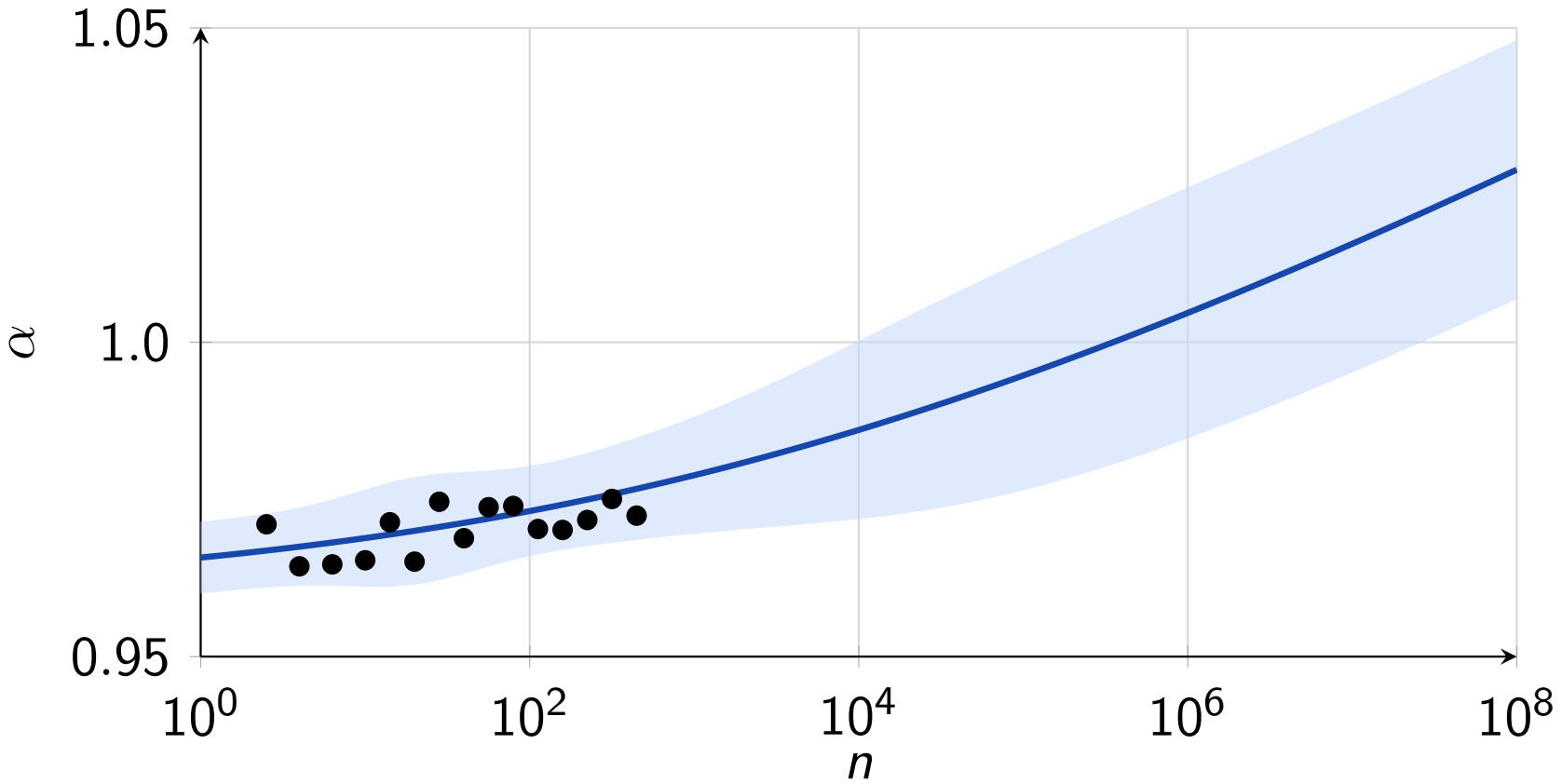
Gaussian Process Regression (GPR) to predict α

Issue: The GADI number of iterations is highly sensitive to the choice of α. Best choice of α depends non-linearly on the precisions u, u_r, and u_s and the dimension n.

Gaussian Process Regression (GPR) to predict α

Issue: The GADI number of iterations is highly sensitive to the choice of α. Best choice of α depends non-linearly on the precisions u, u_r, and u_s and the dimension n.

Solution: GPR prediction



GADI implementation details

- For all problems, we use the HSS splitting:

$$H = \alpha I + \frac{A + A^T}{2}, \quad S = \alpha I + \frac{A - A^T}{2}.$$

- We solve the shifted symmetric and skew-symmetric systems with CG in FP64, FP32, or BF16. The rest of the operations are in FP64.
- We use cuBLAS and cuSPARSE for the implementations.
- With BF16, the matrices and vectors are stored in BF16 but accumulated in FP32 within the chip $\rightarrow$ This benefits memory-bound operations.
- We form and store persistently: A, H, S, and $\alpha I - N$.

Other implementation details

- *Direct solver baseline:* We compare against the NVIDIA's cuDSS sparse direct solver in FP64 or FP32.
- *Iterative solver baseline:* We compare against mixed precision GMRES-based iterative refinement; GMRES iterations are applied in FP32.

  "*Accelerating Restarted GMRES With Mixed Precision Arithmetic*" by **Lindquist et al.**, 2022.
- We do not account for the overhead of the GPR dataset generation and training.
- NVIDIA A100 80GiB SXM GPU.

Summary

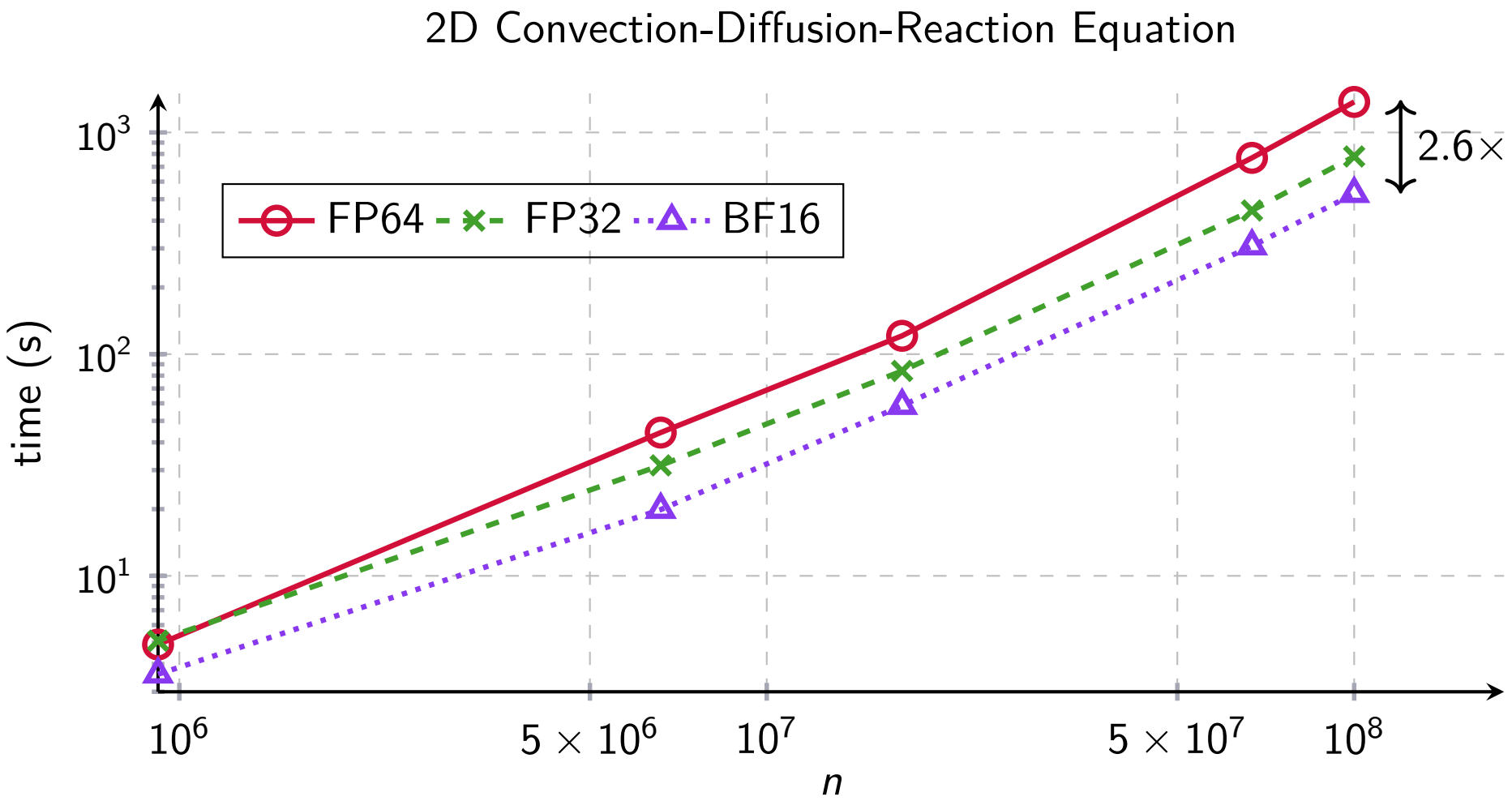
Algs.	u_s	u	u_r
GADI-FP64	FP64	FP64	FP64
GADI-FP32	FP32	FP64	FP64
GADI-BF16	BF16	FP64	FP64
GMRES-FP32	FP32	FP64	FP64
cuDSS	FP64 or FP32		

Performance analysis on GPU

2D Convection-Diffusion-Reaction Equation;
$\|b - A\widehat{x}_k\|_2 / \|r_0\|_2 \leq 10^{-10}$; $n = n_g^2$.

Grid Size n_g	**Runtime Performance (seconds)**				
	GADI			**GMRES**	**CUDSS**
	FP64	FP32	BF16	FP32	FP64
960	4.9	5.1	3.6	**3.3**	4.3
2560	44.3	31.5	**19.9**	23.7	35.5
4096	121.0	84.0	**58.6**	76.9	97.1
8192	768.1	444.6	**307.2**	547.3	436.1
10000	1373.0	779.2	**529.5**	973.8	—

Performance analysis on GPU



Performance analysis on GPU

3D Convection-Diffusion Equation;
$\|b - A\widehat{x}_k\|_2 / \|r_0\|_2 \leq 10^{-6}$; $n = n_g^3$.

Grid Size n_g	Runtime Performance (seconds)				
	GADI			**GMRES**	**CUDSS**
	FP64	FP32	BF16	FP32	FP32
180	3.9	**2.4**	2.8	3.7	—
256	12.2	**7.3**	8.9	15.8	—
320	25.7	**15.3**	18.4	41.5	—
360	40.2	**23.9**	26.4	58.8	—
400	62.7	**36.9**	50.3	107.8	—
450	101.1	**60.6**	137.3	199.9	—
512	—	—	390.7	**328.3**	—

Performance analysis on GPU

Complex Reaction-Diffusion Equation;
$\|b - A\widehat{x}_k\|_2 / \|r_0\|_2 \leq 10^{-6}$; $n = 2n_g^2$.

Grid Size n_g	**Runtime Performance (seconds)**				
	GADI			**GMRES**	**CUDSS**
	FP64	FP32	BF16	FP32	FP32
1024	6.2	4.0	**2.4**	5.1	13.7
2048	21.8	13.1	**8.1**	17.5	56.9
4096	79.6	46.5	**27.7**	59.2	257.1
5120	121.7	72.4	**40.2**	90.4	402.9
8192	309.0	180.2	**98.7**	225.6	—

Section 9

Why you should use Flexible GMRES instead of iterative refinement

What is GMRES?

Consider the Generalized Minimal RESidual (GMRES) algorithm.

Algorithm: GMRES(A, b, x_0, τ)

Require: $A \in \mathbb{R}^{n \times n}$, $b, x_0 \in \mathbb{R}^n$, $\tau \in \mathbb{R}$

$r_0 = b - Ax_0$
$\beta = \|r_0\|$, $v_1 = r_0/\beta$, $k = 1$
repeat
    $w_k = Av_k$
    for $i = 1, \ldots, k$ **do**
        $h_{i,k} = v_i^T w_k$
        $w_k = w_k - h_{i,k} v_i$
    end for
    $h_{k+1,k} = \|w_k\|$, $v_{k+1} = w_k / h_{k+1,k}$
    $V_k = [v_1, \ldots, v_k]$
    $H_k = \{h_{i,j}\}_{1 \leq i \leq j+1; 1 \leq j \leq k}$
    $y_k = \text{argmin}_y \|\beta e_1 - H_k y\|$
    $k = k + 1$
until $\|\beta e_1 - H_k y_k\| \leq \tau$
$x_k = x_0 + V_k y_k$

What is GMRES?

Consider the Generalized Minimal RESidual (GMRES) algorithm.

➤ GMRES = Krylov-based iterative solver for the solution of **general square linear systems $Ax = b$**.

Algorithm: GMRES(A, b, x_0, τ)

Require: $A \in \mathbb{R}^{n \times n}$, $b, x_0 \in \mathbb{R}^n$, $\tau \in \mathbb{R}$

$r_0 = b - Ax_0$
$\beta = \|r_0\|$, $v_1 = r_0/\beta$, $k = 1$
repeat
  $w_k = Av_k$
  for $i = 1, \ldots, k$ **do**
    $h_{i,k} = v_i^T w_k$
    $w_k = w_k - h_{i,k} v_i$
  end for
  $h_{k+1,k} = \|w_k\|$, $v_{k+1} = w_k / h_{k+1,k}$
  $V_k = [v_1, \ldots, v_k]$
  $H_k = \{h_{i,j}\}_{1 \leq i \leq j+1; 1 \leq j \leq k}$
  $y_k = \text{argmin}_y \|\beta e_1 - H_k y\|$
  $k = k + 1$
until $\|\beta e_1 - H_k y_k\| \leq \tau$
$x_k = x_0 + V_k y_k$

What is GMRES?

Consider the Generalized Minimal RESidual (GMRES) algorithm.

- GMRES = Krylov-based iterative solver for the solution of **general square linear systems $Ax = b$**.
- Computes iteratively an orthonormal **Krylov basis V_k** through an Arnoldi process.

Algorithm: GMRES(A, b, x_0, τ)

Require: $A \in \mathbb{R}^{n \times n}$, $b, x_0 \in \mathbb{R}^n$, $\tau \in \mathbb{R}$

$r_0 = b - Ax_0$
$\beta = \|r_0\|, \ v_1 = r_0/\beta, \ k = 1$
repeat
  $w_k = Av_k$
  for $i = 1, \ldots, k$ **do**
    $h_{i,k} = v_i^T w_k$
    $w_k = w_k - h_{i,k} v_i$
  end for
  $h_{k+1,k} = \|w_k\|, v_{k+1} = w_k / h_{k+1,k}$
  $V_k = [v_1, \ldots, v_k]$
  $H_k = \{h_{i,j}\}_{1 \leq i \leq j+1; 1 \leq j \leq k}$
  $y_k = \text{argmin}_y \|\beta e_1 - H_k y\|$
  $k = k + 1$
until $\|\beta e_1 - H_k y_k\| \leq \tau$
$x_k = x_0 + V_k y_k$

What is GMRES?

Consider the Generalized Minimal RESidual (GMRES) algorithm.

- GMRES = Krylov-based iterative solver for the solution of **general square linear systems $Ax = b$**.
- Computes iteratively an orthonormal **Krylov basis V_k** through an Arnoldi process.
- Chooses the vector x_k in $span\{V_k\}$ that **minimizes $\|Ax_k - b\|$**.

Algorithm: GMRES(A, b, x_0, τ)

Require: $A \in \mathbb{R}^{n\times n}$, $b, x_0 \in \mathbb{R}^n$, $\tau \in \mathbb{R}$

$r_0 = b - Ax_0$
$\beta = \|r_0\|,\ v_1 = r_0/\beta,\ k = 1$
repeat
  $w_k = Av_k$
  for $i = 1, \ldots, k$ **do**
    $h_{i,k} = v_i^T w_k$
    $w_k = w_k - h_{i,k} v_i$
  end for
  $h_{k+1,k} = \|w_k\|, v_{k+1} = w_k/h_{k+1,k}$
  $V_k = [v_1, \ldots, v_k]$
  $H_k = \{h_{i,j}\}_{1\leq i\leq j+1; 1\leq j\leq k}$
  $y_k = \text{argmin}_y \|\beta e_1 - H_k y\|$
  $k = k + 1$
until $\|\beta e_1 - H_k y_k\| \leq \tau$
$x_k = x_0 + V_k y_k$

What is GMRES?

Consider the Generalized Minimal RESidual (GMRES) algorithm.

- GMRES = Krylov-based iterative solver for the solution of **general square linear systems $Ax = b$**.
- Computes iteratively an orthonormal **Krylov basis V_k** through an Arnoldi process.
- Chooses the vector x_k in $span\{V_k\}$ that **minimizes $\|Ax_k - b\|$**.
- **Reiterate** until x_k is a satisfactory approximant of x.

Algorithm: GMRES(A, b, x_0, τ)

Require: $A \in \mathbb{R}^{n\times n}$, $b, x_0 \in \mathbb{R}^n$, $\tau \in \mathbb{R}$

$r_0 = b - Ax_0$
$\beta = \|r_0\|,\ v_1 = r_0/\beta,\ k = 1$
repeat
  $w_k = Av_k$
  for $i = 1, \ldots, k$ **do**
    $h_{i,k} = v_i^T w_k$
    $w_k = w_k - h_{i,k} v_i$
  end for
  $h_{k+1,k} = \|w_k\|, v_{k+1} = w_k / h_{k+1,k}$
  $V_k = [v_1, \ldots, v_k]$
  $H_k = \{h_{i,j}\}_{1\le i\le j+1; 1\le j\le k}$
  $y_k = \text{argmin}_y \|\beta e_1 - H_k y\|$
  $k = k + 1$
until $\|\beta e_1 - H_k y_k\| \le \tau$
$x_k = x_0 + V_k y_k$

Preconditioned GMRES as a refinement

Preconditioner are ubiquitous to accelerate Krylov-based iterative solvers. Here are three ways to preconditioned GMRES with a preconditioner M:

- **Left-preconditioning (LGMRES):**

$$M^{-1}Ax = M^{-1}b.$$

- **Right-preconditioning (RGMRES):**

$$AM^{-1}u = b, \qquad x = M^{-1}u.$$

- **Flexible-preconditioning (FGMRES):** A variant of RGMRES storing an additional basis and allowing the preconditioner $M^{(k)}$ to vary from an iteration to another.

Preconditioned GMRES as a refinement

Preconditioner are ubiquitous to accelerate Krylov-based iterative solvers. Here are three ways to preconditioned GMRES with a preconditioner M:

- **Left-preconditioning (LGMRES):**

$$M^{-1}Ax = M^{-1}b.$$

- **Right-preconditioning (RGMRES):**

$$AM^{-1}u = b, \qquad x = M^{-1}u.$$

- **Flexible-preconditioning (FGMRES):** A variant of RGMRES storing an additional basis and allowing the preconditioner $M^{(k)}$ to vary from an iteration to another.

$\Rightarrow$ **GMRES can also refine an inaccurate linear solver by using it as a preconditioner.**

Which is the better refinement?

Consider an inaccurate solver that act as a constant operator M^{-1} and that can be refined by both iterative refinement and GMRES.

$\Rightarrow$**Can we say something about which is faster?**

Which is the better refinement?

Consider an inaccurate solver that act as a constant operator M^{-1} and that can be refined by both iterative refinement and GMRES.

$\Rightarrow$**Can we say something about which is faster?**

What we know from the literature:

- Experimental observation that FGMRES needs less iterations:

  "*Using FGMRES to obtain backward stability in mixed precision*" by **Arioli and Duff**, 2008.

- Inner-outer FGMRES-GMRES can be expected faster than restarted GMRES:

  "*Flexible Inner-Outer Krylov Subspace Methods*" by **Simoncini and Szyld**, 2002.

- Stationary iterative methods converge slower than Krylov subspace methods:

  "*An Introduction to Domain Decomposition Meth- ods: Algorithms, Theory, and Parallel Implementation*" by **Dolean et al.**, Chap. 3, 2015.

A misleading argument

➤ Iterative refinement is a stationary iteration:

$$x_{k+1} = x_k + M^{-1}(b - Ax_k).$$

➤ Taking $x_0 = 0$, the kth iterative refinement iterate satisfies

$$\begin{aligned} x_k &\in \mathcal{K}_k(M^{-1}A, M^{-1}b) \\ &\in \mathit{span}\{M^{-1}b, (M^{-1}A)(M^{-1}b), \ldots, (M^{-1}A)^{k-1}(M^{-1}b)\}. \end{aligned}$$

➤ On the other hand, LGMRES looks at each iteration for an optimal iterate x_k in the same space $\mathcal{K}_k(M^{-1}A, M^{-1}b)$:

$$\min_{x_k \in \mathcal{K}_k(M^{-1}A, M^{-1}b)} \|M^{-1}b - M^{-1}Ax_k\|$$

A misleading argument

➤ Iterative refinement is a stationary iteration:

$$x_{k+1} = x_k + M^{-1}(b - Ax_k).$$

➤ Taking $x_0 = 0$, the kth iterative refinement iterate satisfies

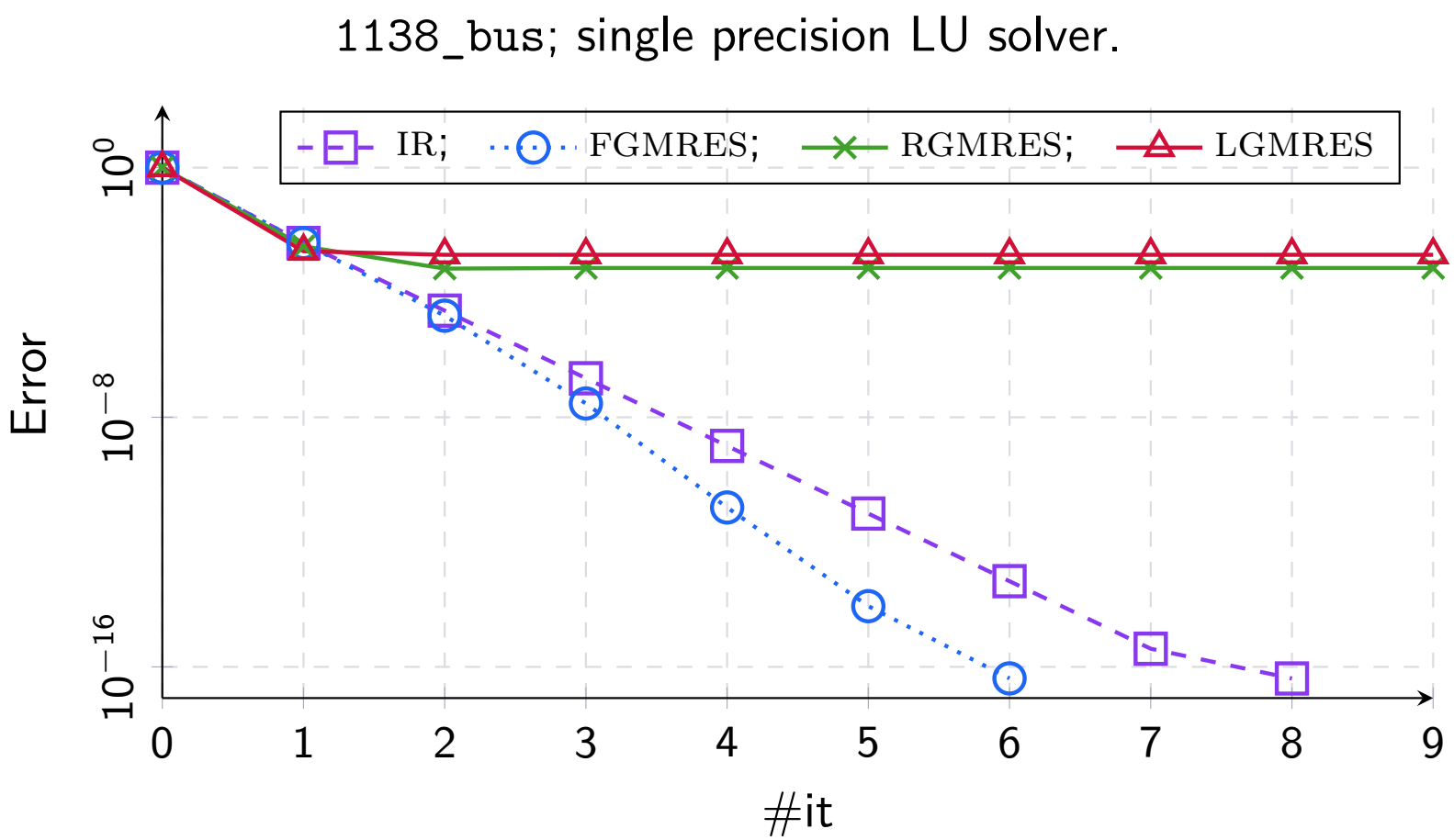
$$\begin{aligned} x_k &\in \mathcal{K}_k(M^{-1}A, M^{-1}b) \\ &\in \mathit{span}\{M^{-1}b, (M^{-1}A)(M^{-1}b), \ldots, (M^{-1}A)^{k-1}(M^{-1}b)\}. \end{aligned}$$

➤ On the other hand, LGMRES looks at each iteration for an optimal iterate x_k in the same space $\mathcal{K}_k(M^{-1}A, M^{-1}b)$:

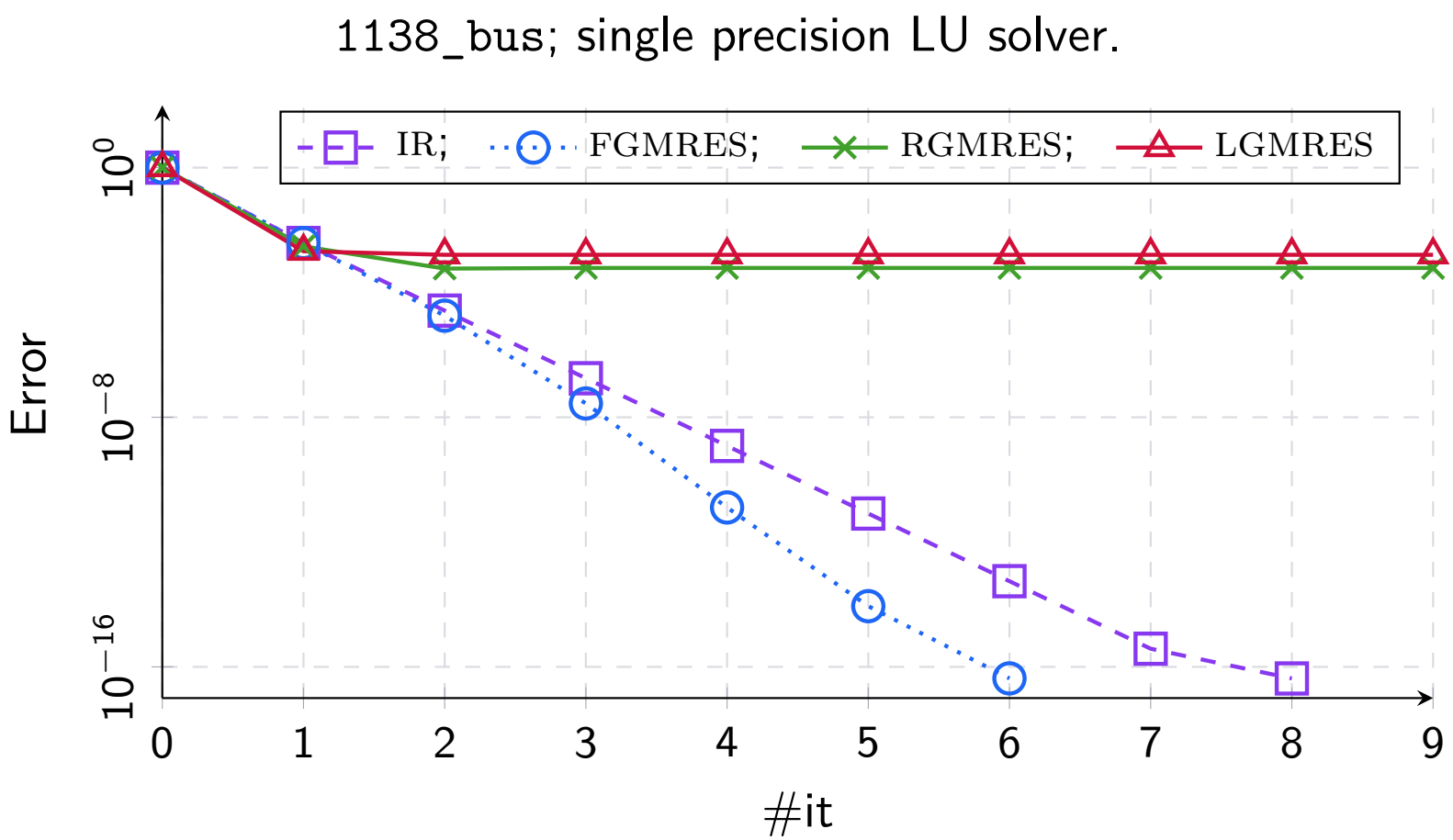
$$\min_{x_k \in \mathcal{K}_k(M^{-1}A, M^{-1}b)} \|M^{-1}b - M^{-1}Ax_k\|$$

$\Rightarrow$ We should expect LGMRES (and RGMRES/FGMRES) to converge faster than iterative refinement.

A counterexample



A counterexample



Why: Attainable accuracies of LGMRES and RGMRES are sensitive to the accuracy at which M^{-1} is applied.

"*Mixed precision strategies for preconditioned GMRES: a comprehensive analysis*" by **Buttari et al.**, 2025.

About comparing FGMRES and iterative refinement

Can we guarantee that FGMRES always refine faster than iterative refinement?

About comparing FGMRES and iterative refinement

Can we guarantee that FGMRES always refine faster than iterative refinement?

Sharp convergence rate for GMRES in the general case is an ongoing difficult problem:

- "*Any nonincreasing convergence curve is possible for GMRES*" by **Greenbaum et al.**, 1996.
- "*Any nonincreasing convergence curves are simultaneously possible for GMRES and weighted GMRES, as well as for left and right preconditioned GMRES*" by **Matalon and Spillane**, 2025.
- "*GMRES convergence bounds that depend on the right-hand-side vector*" by **Titley-Peloquin et al.**, 2013.

About comparing FGMRES and iterative refinement

Can we guarantee that FGMRES always refine faster than iterative refinement?

Sharp convergence rate for GMRES in the general case is an ongoing difficult problem:

- "*Any nonincreasing convergence curve is possible for GMRES*" by **Greenbaum et al.**, 1996.
- "*Any nonincreasing convergence curves are simultaneously possible for GMRES and weighted GMRES, as well as for left and right preconditioned GMRES*" by **Matalon and Spillane**, 2025.
- "*GMRES convergence bounds that depend on the right-hand-side vector*" by **Titley-Peloquin et al.**, 2013.

BUT we have strong assumptions on the preconditioner/inaccurate solver:

$$\widehat{d}_k - d_k = \boldsymbol{u_s} E_k d_k, \qquad \boldsymbol{u_s} \|E_k\| \leq 1.$$

Studying the convergence with a proxy algorithm

The $(k+1)$th iterate of iterative refinement satisfies:

$$x_{k+1} = x_1 + \qquad d_1 + \qquad d_2 + \cdots + \qquad d_k,$$

Studying the convergence with a proxy algorithm

The $(k+1)$th iterate of iterative refinement satisfies:

$$x_{k+1} = x_1 + \alpha_1^{(k)} d_1 + \alpha_2^{(k)} d_2 + \cdots + \alpha_k^{(k)} d_k,$$

with $y_k = [\alpha_1^{(k)}, \ldots, \alpha_k^{(k)}]^T = [1, \ldots, 1]^T$.

Studying the convergence with a proxy algorithm

The $(k+1)$th iterate of iterative refinement satisfies:

$$x_{k+1} = x_1 + \alpha_1^{(k)} d_1 + \alpha_2^{(k)} d_2 + \cdots + \alpha_k^{(k)} d_k,$$

with $y_k = [\alpha_1^{(k)}, \ldots, \alpha_k^{(k)}]^T = [1, \ldots, 1]^T$.

BUT we could choose better:

$$\underset{\alpha_1^{(k)}, \ldots, \alpha_k^{(k)}}{\arg\min} \|b - A(x_1 + \alpha_1^{(k)} d_1 + \cdots + \alpha_k^{(k)} d_k)\|.$$

Studying the convergence with a proxy algorithm

The $(k+1)$th iterate of iterative refinement satisfies:

$$x_{k+1} = x_1 + \alpha_1^{(k)} d_1 + \alpha_2^{(k)} d_2 + \cdots + \alpha_k^{(k)} d_k,$$

with $y_k = [\alpha_1^{(k)}, \ldots, \alpha_k^{(k)}]^T = [1, \ldots, 1]^T$.

BUT we could choose better:

$$\underset{\alpha_1^{(k)}, \ldots, \alpha_k^{(k)}}{\arg\min} \|b - A(x_1 + \alpha_1^{(k)} d_1 + \cdots + \alpha_k^{(k)} d_k)\|.$$

Algorithm: Optimalized iterative refinement

1: **while** not converged **do**
2: Compute the residual $r_k = b - Ax_k$.
3: Solve $Ad_k = r_k$ and store $D_k = [D_{k-1}, d_k]$.
4: Compute the next iterate $x_{k+1} = x_k + D_k y_k$ where $y_k = \text{argmin}_y \|r_1 - AD_k y\|$.
5: **end while**

Convergence analysis

Theorem

The ratio between the (backward error) convergence rates of optimal iterative refinement and iterative refinement for all iteration $k \geq 1$ is

$$\frac{\|(I - Q_k Q_k^T)(A\widehat{d}_k - \widehat{r}_k)\|}{\|A\widehat{d}_k - \widehat{r}_k\|} \leq 1,$$

where $(I - Q_k Q_k^T)$ is an orthogonal projection on the complement of $span\{A[\widehat{d}_1, \ldots, \widehat{d}_k]\}$.

Convergence analysis

Theorem

The ratio between the (backward error) convergence rates of optimal iterative refinement and iterative refinement for all iteration $k \geq 1$ is

$$\frac{\|(I - Q_k Q_k^T)(A\widehat{d}_k - \widehat{r}_k)\|}{\|A\widehat{d}_k - \widehat{r}_k\|} \leq 1,$$

where $(I - Q_k Q_k^T)$ is an orthogonal projection on the complement of $span\{A[\widehat{d}_1, \ldots, \widehat{d}_k]\}$.

Equivalently, and under our assumptions on the solver, we write

$$(I - Q_k Q_k^T)(A\widehat{d}_k - \widehat{r}_k) = (I - Q_k Q_k^T)A(\widehat{d}_k - d_k) = \boldsymbol{u_s}(I - Q_k Q_k^T)AE_k d_k.$$

Convergence analysis

Theorem

The ratio between the (backward error) convergence rates of optimal iterative refinement and iterative refinement for all iteration $k \geq 1$ is

$$\frac{\|(I - Q_k Q_k^T)(A\widehat{d}_k - \widehat{r}_k)\|}{\|A\widehat{d}_k - \widehat{r}_k\|} \leq 1,$$

where $(I - Q_k Q_k^T)$ is an orthogonal projection on the complement of $span\{A[\widehat{d}_1, \ldots, \widehat{d}_k]\}$.

Equivalently, and under our assumptions on the solver, we write

$$(I - Q_k Q_k^T)(A\widehat{d}_k - \widehat{r}_k) = (I - Q_k Q_k^T)A(\widehat{d}_k - d_k) = \boldsymbol{u_s}(I - Q_k Q_k^T)AE_k d_k.$$

$\Rightarrow$ **The more** $span\{\widehat{d}_1, \ldots, \widehat{d}_k\}$ **spans** $E_k d_k$**, the faster is optimal iterative refinement over iterative refinement.**

The link to FGMRES

Algorithm: Optimal IR as Flexible Simpler GMRES

Initialize x_1.
$r_1 = b - Ax_1$
repeat
    Solve $Ad_i = r_i/\|r_i\|$ and form $D_i = [D_{i-1}, d_i]$
    $w_i = Ad_i$
    for $l = 1 : i-1$ **do**
        $h_{l,i} = v_l^T w_i$
        $w_i = w_i - h_{l,i} v_l$
    end for
    $h_{i,i} = \|w_i\|_2$
    Compute $v_i = w_i/h_{i,i}$ and store $V_i = [V_{i-1},\ v_i]$.
    Compute $r_{i+1} = r_i - \beta_i v_i$, where $\beta_i = r_i^T v_i$ and $b_i = [b_{i-1}, \beta_i]$.
until convergence
Solve the triangular system $H_i t_i = b_i$.
$x_i = x_1 + D_i t_i$

Numerical experiments

Number of refinement iterations; LU factors in BFloat16; $n \leq 1000$; $\|x - \widehat{x}_k\|/\|x\| \leq 10^{-14}$.

Matrix	IR	Simpler FGMRES	FGMRES
pores_1	25	21	**19**
fs_680_1	18	**11**	**11**
cz148	32	**15**	**15**
ck104	32	**20**	**20**
cz308	117	**19**	22
fs_680_3	33	**17**	**17**
Si2	51	**19**	**19**
LFAT5	28	**12**	**12**
tub100	73	**31**	33
bcsstk34	37	**21**	**21**

The End!

Any Question?