An Integer Arithmetic-Based Sparse Linear Solver Using GMRES and Iterative Refinement

Giulia Lionetti   Alberto Taddei

AFAE - Floating-Point Arithmetic and Error Analysis

Jan 27, 2026

Paper

Takeshi Iwashita, Kengo Suzuki, Takeshi Fukaya
An Integer Arithmetic-Based Sparse Linear Solver Using a GMRES Method and Iterative Refinement

Roadmap

1. Motivation: Why integer arithmetic?
2. Problem: What breaks without floating-point?
3. Solution: Three-layer architecture
4. Results: Does it actually work?
5. Discussion: Limitations and implications

Is Moore's Law dead?

Dying? We don't know. But hardware is changing.

Three converging trends:

- Energy efficiency gains are slowing
- Novel architectures emerging (SFQ circuits, neuromorphic chips)
- These new technologies may only support integer arithmetic
  - Integer circuits are simpler and more energy-efficient
  - FP units are complex and power-hungry

Can we do *real* scientific computing without floating-point?

The problem

Solve $\hat{A}\hat{x} = \hat{b}$ to double precision...

...using only integer arithmetic in the main loop.

Challenges:

- Fixed range $\rightarrow$ overflow
- Bit shifts $\rightarrow$ lost precision
- No dynamic exponent

Requirements:

- GMRES-like convergence
- Double-precision accuracy
- Robust to ill-conditioning

What we lose without FP

Floating-Point

$\pm m \times 2^e$

Exponent handles range automatically

$\rightarrow$

Fixed-Point Integer

$\pm$ fixed bits

Algorithm must manage range

	Floating-Point	**Integer/Fixed**
Dynamic range	Automatic (exponent)	Manual (shifts)
Overflow risk	Low	High
Energy per op	High ($\sim$50 pJ)	Low ($\sim$5 pJ)
Hardware complexity	Complex	Simple

This is both the **challenge** and the **opportunity**.

State of the art

Mixed-precision iterative refinement:

- Compute in FP16/FP32, correct in FP64
- Well-studied (Göddeke 2007, Carson & Higham 2018, etc.)
- Works great... but still needs FP hardware

The gap

Nobody's built Krylov solvers that work *without* floating-point kernels.

Three-layer architecture

Layer 1: Iterative Refinement
Compute residual in FP, scale for integer solver, accumulate corrections

↓

Layer 2: Matrix Decomposition
$A = \bar{A}_0 + 2^{-\alpha_1}\bar{A}_1 + \cdots$ (integers + power-of-2 weights)

↓

Layer 3: int-GMRES
Arnoldi, Givens, matvec $\rightarrow$ all in fixed-point

Integer loop stays bounded; FP recovers accuracy.

Layer 1: Iterative refinement

Standard idea: $x \leftarrow x + \gamma^{(k)} x^{(k)}$ where $x^{(k)}$ solves a scaled residual system.

Why it matters here:
Residual $b' = b - Ax$ shrinks each iteration $\rightarrow$ scale by $1/\gamma = \max |b'_i|$ before sending to integer solver.

Result: integer solver always sees magnitudes near 1.

Refinement = automatic range control

Layer 2: Matrix decomposition

Represent A as:

$$A = \bar{A}_0 + 2^{-\alpha_1}\bar{A}_1 + 2^{-\alpha_2}\bar{A}_2 + \cdots$$

$\times 1$	$\times 2^{-\alpha_1}$	$\times 2^{-\alpha_2}$	
$\boxed{\bar{A}_0 \text{ int}}$	$\boxed{\bar{A}_1 \text{ int}}$	$\boxed{\bar{A}_2 \text{ int}}$	$\ldots$
most significant		fine detail	

- Power-of-2 scaling = cheap bit shifts
- Progressive accuracy: use $\bar{A}_0$ early, add terms as needed

Layer 3: int-GMRES

Standard GMRES structure (Arnoldi orthogonalization, Givens rotations, least squares).

Key difference: All inner kernels use fixed-point arithmetic.

Integer kernels:	matvec, dot products, norms, Givens
FP only:	initial residual, final least squares, solution update

Problem: Dot products and norms overflow easily.

Fixed-point arithmetic

Numbers stored as $Q_{d_m.d_f}$ (sign + d_m integer + d_f fractional bits).

The overflow problem:

Bits: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Without shift: Overflow risk!

With shift ($\gg \beta$): Lost precision!

$$t_r = \big((t_1 \gg \beta_1) \cdot (t_2 \gg \beta_2)\big) \gg (d_f - \beta_1 - \beta_2)$$

More shift
$\Rightarrow$ safer, loses bits

Less shift
$\Rightarrow$ risky, keeps bits

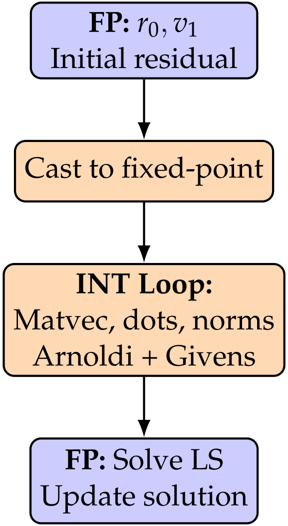
Smart shifting

Exploit GMRES structure:

- Krylov vectors are normalized $\rightarrow$ many leading zeros $\rightarrow$ safe
- Givens coefficients satisfy $|c|, |s| \leq 1 \rightarrow$ safe
- Dot products between normalized vectors $\rightarrow$ mostly safe

Use algorithm invariants to minimize shifts
= maximize effective precision

Core algorithm: Visual overview



Integer kernels:

- Matrix-vector
- Dot products
- Norms
- Givens rotations

Arithmetic stays bounded in the loop; FP ensures accuracy at boundaries.

Core algorithm: Pseudocode

$r_0 \leftarrow b^{(k)} - A^{(k)}x^{(k)}$, $v_1 \leftarrow r_0/\|r_0\|$ ▷ FP

$\bar{v}_1 \leftarrow \texttt{cast}(v_1)$ ▷ Convert to fixed-point

for $j = 1, \ldots, m$ **do**

  $\bar{w} \leftarrow \bar{A}^{(k)}\bar{v}_j$ ▷ INT: matvec

  for $i = 1, \ldots, j$ **do**

    $\bar{h}_{i,j} \leftarrow \langle \bar{w}, \bar{v}_i \rangle$ ▷ INT: dot with shifts

    $\bar{w} \leftarrow \bar{w} - \bar{h}_{i,j}\bar{v}_i$ ▷ INT: orthogonalize

  end for

  $\bar{h}_{j+1,j} \leftarrow \|\bar{w}\|$, $\bar{v}_{j+1} \leftarrow \bar{w}/\bar{h}_{j+1,j}$ ▷ INT

  Apply Givens rotations ▷ INT

end for

Solve least squares, update $x^{(k)}$ ▷ FP

Preconditioning is critical

Standard reason: faster convergence.

Integer arithmetic reason: reduces overflow risk.

Better conditioning → Smaller values → Fewer shifts → Higher precision

They use **ILU(0)**, implemented entirely in integer arithmetic.

Experimental setup

- 10 sparse matrices from SuiteSparse
- Target: relative residual $< 10^{-8}$ (measured in FP64)
- Fixed-point: $WL = 64$, $d_f = 30$
- Compare iteration counts: double-GMRES vs int-GMRES

Note

This paper measures **convergence**, not performance.
(Speed/power measurements need target hardware.)

Without preconditioning

Matrix	Double	int-GMRES	Ratio
atmosmodj	2100	2100	1.0×
atmosmodl	420	420	1.0×
wang3	510	630	1.24×
cage14	30	60	2.0×

Mixed results. Some identical, some significantly slower.

Why? Aggressive shifts needed to prevent overflow → accuracy loss → slower convergence.

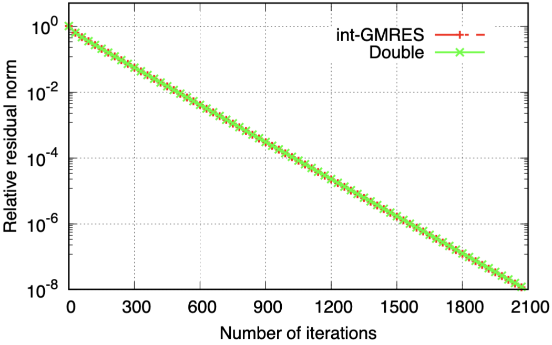
With ILU preconditioning

Matrix	Double+ILU	int+ILU	Ratio
atmosmodj	300	300	1.0×
atmosmodl	120	120	1.0×
wang3	120	120	1.0×
cage14	30	60	2.0×

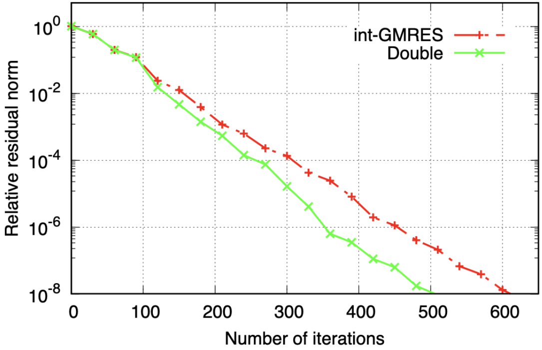
Much better! Most cases identical to double precision.

Preconditioning enables integer arithmetic by stabilizing ranges.

Convergence curves

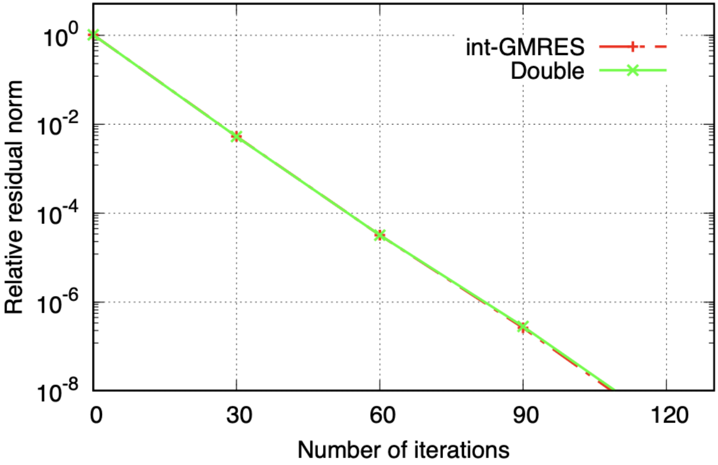


atmosmodj (no precond): identical



wang3 (no precond): slower

With ILU: wang3



With preconditioning, int-GMRES tracks double precision closely.

What they contributed

1. Working integer-GMRES implementation
2. Iterative refinement framework for integer arithmetic
3. Operation-specific shift strategy exploiting GMRES invariants
4. Empirical evidence: ILU preconditioning is essential

Main insight: Precision management moves from hardware to algorithm.

Limitations

- Only tested on moderately conditioned problems
- Several tuning parameters (d_f, shifts, decomposition depth)
- No actual performance/energy measurements
- Theoretical convergence guarantees unclear

But: demonstrates feasibility. Integer Krylov solvers can work.

Conclusions

Integer-only solvers are viable
if you manage range explicitly

- Iterative refinement controls magnitudes
- Smart shifts exploit algorithmic structure
- Preconditioning is crucial

Opens path to ultra-low-power scientific computing.

Thank you!

Questions?

References

- T. Iwashita, K. Suzuki, T. Fukaya,
  An Integer Arithmetic-Based Sparse Linear Solver Using a GMRES Method and Iterative Refinement.
- Y. Saad,
  Iterative Methods for Sparse Linear Systems, SIAM.
- E. Carson, N. J. Higham,
  Iterative refinement in three precisions, SIAM SISC (2018).
- A. Haidar et al.,
  Mixed-precision iterative refinement on GPUs, SC18.