

RSLAB

A Pure-Rust Sparse Direct Solver with Preconditioned and Recycled Krylov Refinement:
Algorithms, A-Priori Predictors, and a Learned Configuration Tuner

Milan Rother

July 4, 2026

Abstract

RSLAB is a pure-Rust sparse direct solver. It provides a supernodal complex-symmetric $\mathbf{LDL}^\top$ factorization with Bunch–Kaufman pivoting and an unsymmetric multifrontal LU, is generic over the scalar type (`f64`, `f32`, `Complex<f64>`, `Complex<f32>`), and links no external numeric library (no BLAS, LAPACK, or METIS): the SIMD dense kernels, the fill-reducing orderings, and the factorization are all Rust. The analyze phase computes, in addition to the symbolic factorization, a set of a-priori predictors that are exact functions of the matrix structure: a per-path peak-memory bound (separate for the left-looking and the multifrontal schedule), a geometric-flop work estimate, a runtime estimate from a hardware calibration, and a thread-count recommendation. The solver exposes its analyze, factor, and kernel parameters through one flat settings interface, and a learned configuration tuner selects a setting from the matrix structure, constrained by a deterministic guard stack that keeps its predicted memory below the untuned default and falls back to the default outside the training range. The numeric factorization is bit-identical regardless of the worker count, and the resource planner is a pure function of its inputs, which makes batched execution reproducible and its memory and runtime cost computable in advance. When the factor is made approximate — static-pivoted for an indefinite operator, or thresholded to save memory — it becomes a preconditioner, and RSLAB closes the accuracy gap with a matching Krylov layer: complex-symmetric short recurrences (COCG/COCR), flexible GMRES (FGMRES) with a documented backward-stable orthogonalization split, a batched multi-RHS block GMRES with within-cycle deflation, and GCRO-DR Krylov-subspace recycling for sequences of related solves. This report documents the algorithms, the predictors, the tunable stages, the determinism properties, the solver-in-the-loop execution model, and the preconditioned and recycled iterative layer, and reports the solver against `faer`, MKL PARDISO, and SuperLU over the SuiteSparse collection.

1 Scope

RSLAB factorizes and solves $Ax = b$ for sparse A that is either complex-symmetric (routed to $\mathbf{LDL}^\top$) or general unsymmetric (routed to LU). The primary target is the complex-symmetric systems of frequency-domain electromagnetic and FEM analysis, including curl–curl (Nédélec edge-element) Maxwell problems with lossy media or PML, which are complex-symmetric rather than Hermitian. The implementation is stable Rust with no C/Fortran dependency and no BLAS/LAPACK/METIS linkage, and is exposed to Python through a PyO3 layer. The default factorization is exact and the solve is a direct back-substitution; when the factor is deliberately made approximate it serves as a preconditioner for the built-in Krylov solvers (Sec. 11), which are the intended path for the large indefinite complex-symmetric systems and the matrix-free MoM/FEM operators the direct factor cannot hold whole.

2 Architecture

The solver follows the standard three-phase sparse-direct flow. *Analyze* orders the matrix, builds the elimination tree, detects and amalgamates supernodes, computes column counts, and produces

the a-priori predictors (Sec. 6) and the structural feature vector (Sec. 6). *Factor* performs the numeric $\mathbf{LDL}^\top$ or LU on the ordered, equilibrated matrix. *Solve* runs the triangular substitutions. Analyze is separated from factor at the API level: a symbolic object (`LdlSymbolic` or `LuSymbolic`) is produced once and can factor many matrices that share its sparsity pattern, which is the basis of the solver-in-the-loop model (Sec. 10). Across the corpus the numeric factor dominates the wall-clock, with analyze and solve small by comparison (Fig. 1), so the tuning and the parallelism target the factor.

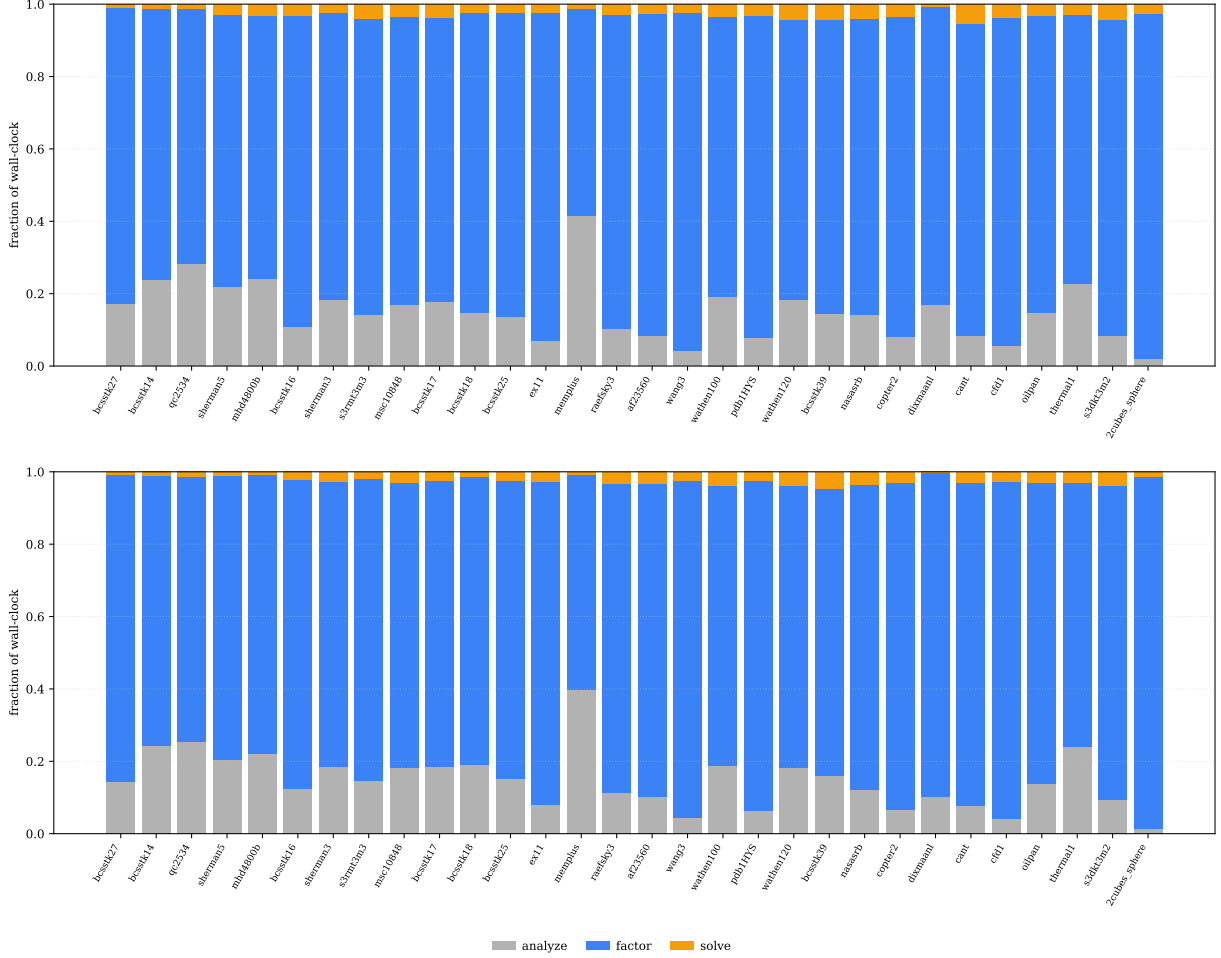


Figure 1: Analyze / factor / solve wall-clock split per matrix, for the left-looking and multifrontal paths. The numeric factor dominates.

The code is a Cargo workspace (Table 1). The ordering crates share a common quotient-graph elimination engine; the numeric core holds the two factorization paths and the dense SIMD kernels; the diagnostics, analysis, tuning, and `auto_tune` modules hold the predictors, the feature extraction, the resource planner, and the learned tuner respectively.

Crate / module	Responsibility
<code>rslab-ordering-core</code>	Shared quotient-graph elimination engine (workspace, sliding index allocator with garbage collection, supervariable detection)
<code>rslab-amd</code>	Approximate minimum degree
<code>rslab-amf</code>	Approximate minimum fill (HAMF4)
<code>rslab-metis</code>	Multilevel nested dissection (iterative driver, König separators)
<code>symbolic</code>	Elimination tree, supernode amalgamation, column counts, ordering dispatch
<code>numeric</code>	Supernodal $\mathbf{LDL}^\top$, multifrontal LU, panel-freeing, dense SIMD kernels, scoped pools; the Krylov layer (COCG/COCR, FGMRES, block GMRES, GCRO-DR) and its dense harmonic-Ritz eigensolver
<code>diagnostics</code>	Per-path a-priori memory and flop estimates
<code>analysis</code>	Structural feature extraction, thread-count predictor
<code>tuning</code>	Hardware calibration, runtime estimate, budget-aware resource planner
<code>auto_tune</code>	Embedded performance model and guard stack (pure-Rust inference)
<code>python</code>	PyO3 bindings

Table 1: Workspace layout.

3 Fill-reducing orderings

The ordering determines fill and therefore both factor time and memory. RSLAB implements three ordering families behind an `OrderingMethod` enum and an adaptive dispatcher.

3.1 Approximate minimum degree

AMD [1] is a quotient-graph elimination that selects the pivot of least approximate external degree. The workspace holds the element and variable lists in a single sliding index array `iw`, with element pointers `pe`, list lengths `len/elen`, supervariable counts `nv`, running degrees `degree`, a generation-counter mark array `w`, and degree-indexed LIFO bucket lists `head/next/last`; when `iw` fills, an in-place garbage collection compacts the live lists. The elimination loop repeatedly selects the minimum-degree pivot, forms a new element by merging its element list with its variable tail (in place when the pivot has no prior elements, out of place otherwise), and updates the affected variables’ approximate degrees. During the update it performs aggressive element absorption, mass elimination of variables that become interchangeable with the pivot, and supervariable detection by hashing variables with identical adjacency and merging the indistinguishable ones. Variables of degree exceeding $\max(16, \lfloor \alpha \sqrt{n} \rfloor)$ with $\alpha = 10$ are deferred to the tail as dense rows. AMD is the default for very large, very sparse patterns, where the separator overhead of nested dissection does not pay off.

3.2 Approximate minimum fill

AMF [2, 3] replaces the degree score with an approximate *fill* score. Each supervariable i carries

$$\text{RMF}(i) = \frac{\deg(i) (\deg(i) - 1 + 2 \deg_{me}) - \text{WF}(i)}{nv(i) + 1}, \quad (1)$$

where $\text{WF}(i)$ is the accumulated working fill, updated per eliminated element by a surface term $2 \deg - \text{dext} - 1$ and combined across supervariables as $\text{wf}_4 + 2 nv_i \text{wf}_3$. The working fill is accumulated in 64-bit integers because the product is $\mathcal{O}(n^2)$ and overflows 32-bit for $n \gtrsim 46\,000$. Scores are placed in a bucket array of length $2n + 2$ with exact fine buckets $[0, n]$ and coarse buckets of stride $\max(n/8, 1)$ above; the pivot is the minimum exact score. Supervariable merges take the larger of the two working-fill values. AMF is the HAMF4 variant and runs aggressive absorption unconditionally; it is the default for $n \leq 10\,000$, where it stays within $1.10\times$ of the reference HAMF4 fill on a 183 277-matrix oracle suite.

3.3 Nested dissection

The nested-dissection ordering [4, 5] is a multilevel recursive bisection driven by an explicit work stack of (subgraph, vertex map, offset) items rather than recursion, so deep dissection trees cannot overflow the call stack. Each subgraph is bisected by coarsening the graph, computing an initial bisection on the coarsest level, refining it with Fiduccia–Mattheyses [6], and uncoarsening with projection and refinement at each level. The vertex separator is extracted from the edge-cut bipartition by a minimum vertex cover, computed by König’s theorem: a maximum matching on the bipartite crossing-edge graph, then the alternating-path reachable set gives the cover. Two guards bound the recursion: subgraphs below `nd_to_amd_switch` = 200 vertices are ordered directly by AMD, and a split where one side holds $\geq 90\%$ of the vertices falls back to an AMD leaf rather than recursing on a graph without a good separator. Nested dissection is the default for $n > 10\,000$.

3.4 Dispatch

`OrderingMethod::Auto` routes by cheap structural predicates. A very large and sparse pattern ($n > 100\,000$, average degree < 5) goes to AMD. An arrow or bordered pattern goes to AMF, which defers the dense border to the trailing block where nested dissection would smear it across separators; the detector flags a pattern where at least 20% of the nonzeros are concentrated in fewer than 5% of the columns above a degree floor of $\max(64, 8\overline{\deg})$. Otherwise the size rule above applies. `OrderingMethod::AutoRace` instead runs the full symbolic factorization for each concrete candidate and keeps the one with the smallest factor nnz, at about $4\times$ the single-pass analysis cost.

4 Symbolic analysis

From the ordered pattern RSLAB builds the elimination tree and detects fundamental supernodes, which are maximal runs of columns whose row structure differs only by the eliminated column. Small supernodes are amalgamated to widen the dense fronts that feed the BLAS-3 kernels. Nodes with fewer than `nemin` = 16 columns [7] are merged with their parent; the merge order uses a column renumbering that re-postorders the tree so a desired parent–child merge becomes structurally adjacent, which is what makes amalgamation effective on bushy assembly trees. A shape predicate selects between the adjacency-only and the renumbering merge strategies, using the renumbering strategy unless the fraction of multi-child internal nodes is below 0.05 (a path-like tree, where renumbering would over-merge). Beyond that, *relaxed* amalgamation [8], applied for $n \geq 1024$, merges an adjacent child into its parent even when not fill-justified as long as the merged supernode stays within a width and extra-row budget, trading a little explicit-zero fill for wider, higher-rank Schur updates. Column counts are computed by a depth-first walk of the elimination tree, and drive the factor-size estimate.

5 Numeric factorization

5.1 Equilibration

Before factoring, A is symmetrically equilibrated as $\hat{A} = DAD$ with a real diagonal $D = \text{diag}(s)$, $s_i = 1/\sqrt{\max_j |A_{ij}|}$, computed in one pass over the lower triangle; an all-zero row is left unscaled. The real, symmetric, positive diagonal preserves complex symmetry and the pivot signs, tolerates zero diagonals (common in saddle-point systems), and improves the conditioning the bounded pivoting sees. The solve unwinds $x = D(\hat{A}^{-1}(Db))$. The LU path applies a two-sided equilibration $\hat{A} = D_r A D_c$.

5.2 Supernodal left-looking $\mathbf{LDL}^\top$

The default path is a supernodal left-looking $\mathbf{LDL}^\top$. Each supernode holds a dense panel of size $\text{nrow} \times \text{ncol}$, where nrow is the number of rows in its factored column structure. The panel is assembled from A , then updated by every previously factored descendant through the `cmod` operation, which pulls the descendant’s columns, applies its block-diagonal D , and subtracts the resulting rank update; the update runs as a scalar triple loop below a flop threshold and as a SIMD GEMM above it. The fully-summed block is then factored in place by `cdiv`, a blocked Bunch–Kaufman partial factorization with panel width `panel_nb` = 64.

The Bunch–Kaufman kernel [9] factors the dense block as $P^\top AP = \mathbf{LDL}^\top$ with L unit lower triangular and D block-diagonal of 1×1 and 2×2 pivots. For complex-symmetric A the control flow is that of the real symmetric `sytf2` with magnitudes $|z|$ and no conjugation. The pivot threshold is $\alpha = (1 + \sqrt{17})/8 \approx 0.6404$; pivot selection is bounded to the panel’s fully-summed rows so the factorization stays within the supernode. A 2×2 block is rejected as singular when

$$|d_{11}d_{22} - d_{21}^2| < \varepsilon \max(|d_{11}|, |d_{21}|, |d_{22}|)^2, \quad \varepsilon = 1 \times 10^{-14},$$

which bounds the element growth injected into the trailing update. Each panel’s trailing Schur update is deferred and routed through a single SIMD GEMM.

5.3 Panel freeing

The transient memory of the left-looking path is controlled by freeing each dense panel once its last consumer is done. Every supernode carries a reference count of the ancestors that will still update from it. When a descendant’s `cmod` decrements that count to zero, the panel is compacted into its final CSC factor fragment and the dense buffer is released; the count is manipulated with acquire/release atomics so this is safe under the tree-parallel factorization. The resident data is therefore the compact factor plus only the live panels, not all panels at once. Sibling subtrees factor concurrently, and the per-node GEMMs parallelize above their flop thresholds.

5.4 Multifrontal path

The alternative multifrontal path [10, 11] assembles a dense frontal matrix per supernode from its eliminated columns and the contribution blocks (CBs) of its children, factors the fully-summed part, and passes the Schur complement to the parent as its CB. A work-stealing tree recursion factors children in parallel; each child’s CB is freed the moment it is extend-added into the parent, so only the active contribution frontier is live. Where the child CB stack is large (at least 4×10^6 scalars) children are reordered by Liu’s (peak – *cb*) rule [12] to shrink the transient peak while preserving leaf parallelism; the reordering is a scheduling hint and does not change the numbering, factor, or fill. The multifrontal front is denser and more BLAS-3-rich than a left-looking panel but carries a higher transient peak, which is why the two paths are separate and the choice is per matrix.

5.5 Unsymmetric LU

General matrices reuse the symmetric multifrontal machinery on the symmetrized pattern $A \cup A^\top$, with UMFPACK-style [13] threshold partial pivoting: within each front’s fully-summed block a diagonal candidate is accepted when $|a_{jj}| \geq \tau |a_{\text{colmax}}|$ with $\tau = 0.1$, else the column maximum is swapped up. The panel width is `NB` = 32, narrower than the $\mathbf{LDL}^\top$ default because the serial within-panel factorization is the bottleneck while the SIMD GEMM is fast. L is stored CSC (unit lower), U CSR; the column permutation is the fill-reducing ordering and a separate row permutation records the pivot interchanges.

5.6 Static pivoting

The `ZeroPivotAction` policy governs near-singular pivots. `Fail` (the default) returns a rank-deficiency error on a pivot below the tolerance. `PerturbToEps` lifts any pivot of magnitude below a floor to that floor while preserving its phase, $d \mapsto d(\text{floor}/|d|)$, so the factorization satisfies $\mathbf{LDL}^\top = A + \Delta$ with a bounded Δ and never fails; the count of perturbed pivots is tracked per front. This never-fail factor is the preconditioner the Krylov layer of Sec. 11 consumes.

5.7 Kernel scheduling and scoped pools

Four flop-count thresholds route the dense work, passed as a cheap `Copy KernelTuning` value rather than through global state: `scalar_gate` (≈ 4096 flops) below which an update runs as a scalar triple loop instead of a SIMD GEMM; `par_gemm` ($\approx 1 \times 10^6$) above which a `cmod` GEMM runs rayon-parallel; `par_cdiv` ($\approx 8 \times 10^6$) above which the panel-trailing and front GEMMs run parallel, which engages node-level parallelism on the large near-root fronts; and `use_gemm_schur`, which forces the scalar path for benchmarking. The tree recursion runs in a scoped rayon pool of the requested width rather than the global pool, so concurrent solves do not oversubscribe each other. The worker stack is sized to the assembly-tree depth, computed iteratively,

$$\text{stack} = \text{clamp}(32 \text{ KB} \cdot \text{depth}, 16 \text{ MB}, 8 \text{ GB}),$$

which lets deep banded chain-trees factor without a stack overflow.

6 A-priori predictors

The analyze phase computes four predictors as exact functions of the symbolic structure and the scalar size $v = \text{value_bytes}$. They are the basis of the memory guarantee, the tuner, the thread policy, and the resource planner.

6.1 Per-path peak memory

The factor size is `factor_nnz` $\cdot (v + 8)$, where the $+8$ is the CSC row index and the count is a structural upper bound because numeric cancellation can only lower it. The transient peak is bounded separately for the two paths.

For the left-looking path, a reference-count simulation of the panel-freeing schedule gives the live-panel peak `panel_live_peak_bytes`: it walks the postorder, adds each panel as it is assembled, and frees a panel when its refcount reaches zero, recording the running maximum of live panels plus already-compacted factor. This is the floor the solver actually operates at. A conservative whole-run bound assumes the parallel frontier holds all panels at once,

$$\text{scratch} = \frac{1}{4}(\text{panels_all} + \text{factor}) + 32 \text{ MB}, \quad (2)$$

$$\text{transient} = \text{panels_all} + \text{factor} + \text{input} + \text{scratch}, \quad (3)$$

where the scratch term covers the per-thread `cmod/cdiv` and emit buffers. For $\mathbf{LDL}^\top$ the panel size is `nrow` $\cdot$ `ncol` $\cdot v$, the compacted factor per supernode is

$$\left(\frac{nc(nc+1)}{2} + \text{cnrow} \cdot nc \right) (v + 8),$$

and the input is `nnz` $(v + 8)$; for LU both L and U panels are held and the input is $2 \text{ nnz} (v + 8)$.

For the multifrontal path a level-parallel model takes, at each assembly-tree level, the sum of the front sizes $\sum \text{nrow}^2 v$ plus the contribution blocks of that level's children $\sum \text{cnrow}^2 v$, and multiplies the peak by $\frac{7}{5}$ to cover the work-stealing overlap of deep leaves of one subtree with mid-level fronts of another. Over the corpus both bounds hold with an estimate/measured ratio of about 1.3 in geomean (Fig. 2); the only violations on the regenerated corpus are two dense BEM/MoM matrices at 3–4% under the measured peak, within the margin the planner's safety headroom absorbs. The panel-freeing floor is the tighter quantity used inside the tuner's veto.

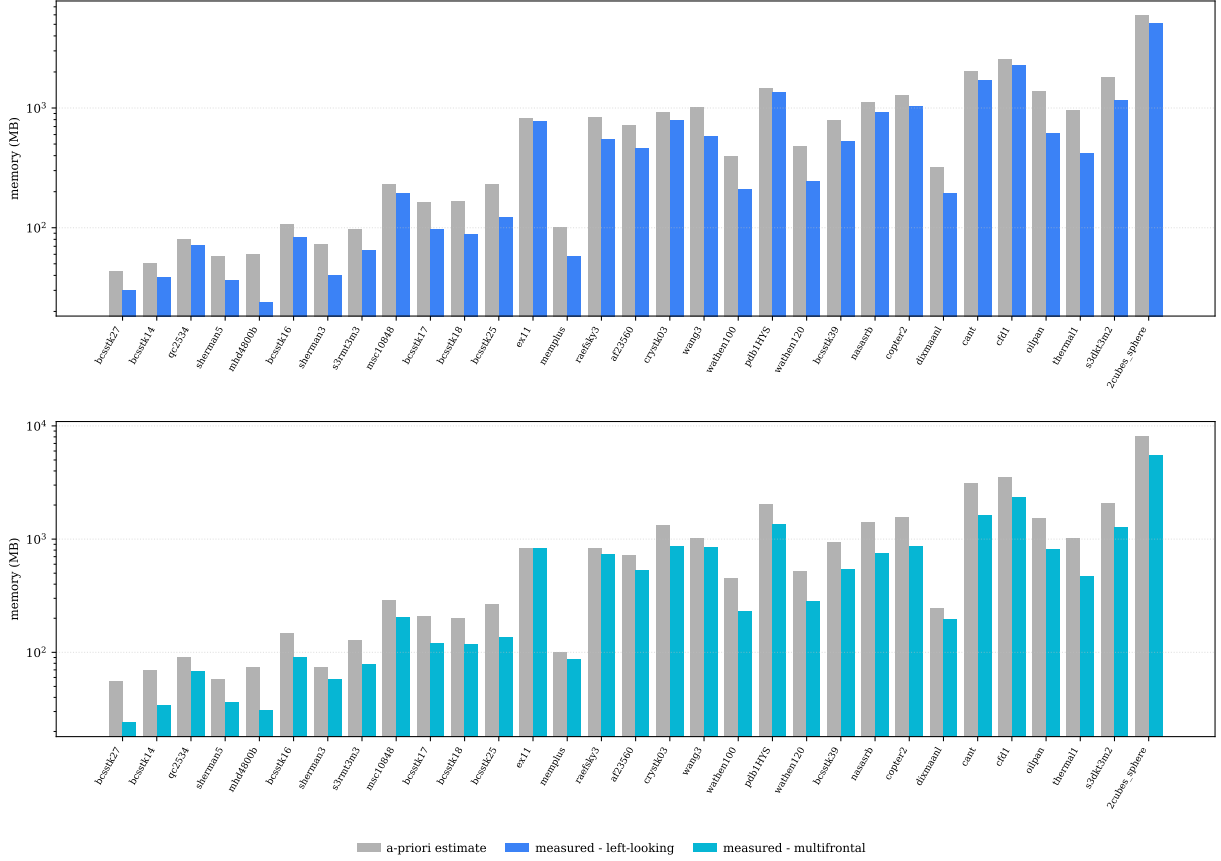


Figure 2: A-priori memory estimate (conservative bound and panel-freeing floor) vs. the measured peak, per RSLAB path. The estimate does not under-predict.

6.2 Work and runtime

The geometric work

$$\text{factor_flops} = \sum_s \text{nrow}_s^2 \text{ncol}_s$$

is a type-independent proxy, so a single `f64` calibration serves all scalar types. The thread-aware runtime estimate is

$$\max\left(\frac{\text{critical_path_flops}}{r}, \frac{\text{factor_flops}}{r \cdot \text{speedup}}\right), \quad r = \text{gflops} \cdot 10^9,$$

where the second term is the parallel work and the first is an Amdahl floor from the assembly tree’s longest leaf-to-root chain (computed in one postorder pass over the supernodes), so a critical-path-bound matrix does not benefit from workers past the point where the parallel term drops to the floor. The `Calibration` measures the single-thread geometric-flop rate (separately for real and complex, since the per-flop cost differs) and the parallel speedup once, by factoring a 24^3 seven-point Laplacian at one thread and at all physical cores, and caches the result keyed by a hardware fingerprint (core counts and a RAM-size bucket) in the manner of FFTW wisdom [14]. The speedup is interpolated linearly from one thread to the calibrated thread count and held flat beyond it, matching the saturating scaling of sparse-direct work; a fallback of 2 Gflop/s and $\sqrt{\text{cores}}$ speedup is used if the measurement fails.

Learned residual. The bare analytical speedup

$$s_{\text{pred}}(t) = \frac{\text{factor_flops}}{\max(\text{critical_path_flops}, \text{factor_flops}/\text{speedup}(t))}$$

mispredicts the achieved scaling systematically: the critical-path flops overstate the true serial fraction (sibling subtrees overlap in time), so the model is too pessimistic on serial-heavy trees and too optimistic on wide ones. A ridge regression of $\log(s_{\text{meas}}/s_{\text{pred}})$ on dimensionless structural features $[1, \text{amdahl_frac}, \ln t, \ln \text{tree_width}]$, where $\text{amdahl_frac} = \text{critical_path_flops}/\text{factor_flops}$, is fit offline over a generated thread-scaling corpus of 54 matrices and reduces the held-out speedup error by $\sim 26\%$ under leave-one-matrix-out cross-validation. Almost all of the signal is in `amdahl_frac` (dropping it leaves a 2% gain), which validates the critical-path feature. The residual is kept additive on the calibrated base rather than a standalone model, so the absolute rate still comes from the hardware calibration and only the shape of the speedup curve is relearned — a machine-transferable correction. It is clamped, and the corrected time is floored at the critical path, so the linear fit can never extrapolate a true chain ($\text{amdahl_frac} \rightarrow 1$) into a physically impossible speedup.

6.3 Thread count

The thread-count predictor maps three structural quantities, the factor flops, the largest front height, and the peak tree width, to a worker count. A thin and narrow problem (`front_nrow_max` < 512 and `tree_width_max` < 128) is capped at two workers, where more only regress; a small problem (`factor_flops` $< 3 \times 10^8$) is capped at four, where scheduling overhead dominates; otherwise all cores are used. On the corpus this lands within about 10% of the per-matrix optimal in geomean, against about 50% for a fixed budget of two workers. The need for a per-matrix policy is visible in the measured thread scaling (Fig. 3): on the complex corpus RSLAB reaches a geomean $\sim 3.2\times$ (left-looking) / $\sim 3.0\times$ (multifrontal) at 12 to 24 workers, but sparse-direct factorization is largely memory-bandwidth bound, so it saturates well below linear, and the min-max band is wide (some matrices regress past a few threads, others reach $\sim 4.8\times$). This spread is why the worker count is picked per matrix — and, in the v2 cost model, a decision variable that may choose fewer threads when scaling saturates.

6.4 Structural features

The tuner consumes a structural fingerprint computed in the analyze phase. The pattern features, computed in $\mathcal{O}(\text{nnz})$, are n , `nnz`, the mean and maximum degree, the degree coefficient of variation (an arrow-pattern signal), the maximum and relative-mean bandwidth, and the diagonal-dominance and diagonal-presence fractions. The symbolic features, computed after analysis, are the supernode count, the fill `nnz` and fill ratio, the mean supernode width, the maximum front height, the tree depth, the maximum and mean tree width, the factor flops, the arithmetic intensity (flops per stored factor entry), and the fraction of flops in the largest and in the top 1% of fronts. These are the quantities that determine which configuration is fastest and smallest, and are the model’s input.

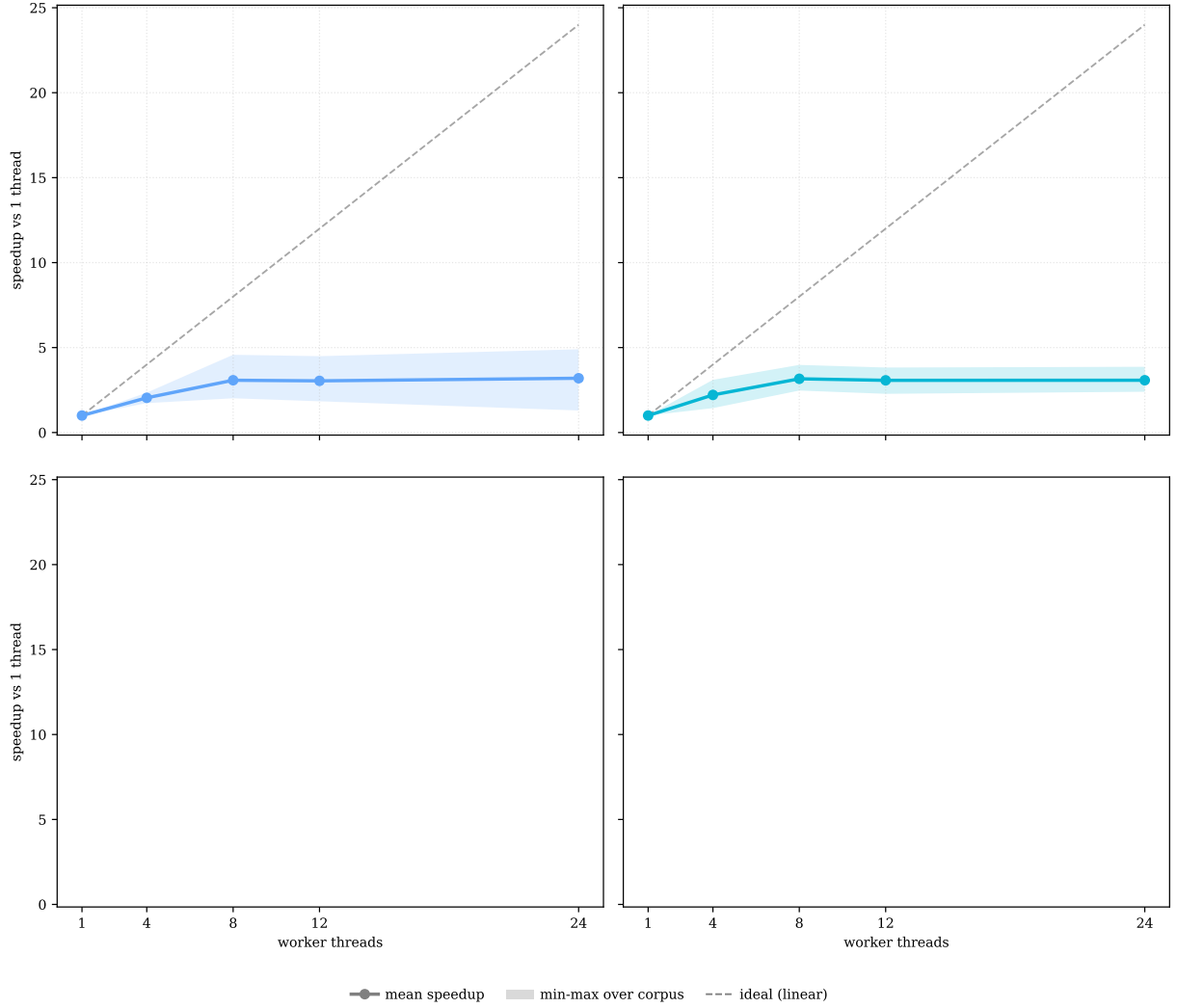


Figure 3: Thread scaling of the RSLAB left-looking and multifrontal kernels, 1–24 workers, geomean with min–max band over the complex corpus.

7 Tunable stages

Every analyze, factor, and kernel parameter is exposed through one flat `SolverSettings` struct, so a caller and the tuner address the same interface. Table 2 lists the parameters and the stage each affects. The analyze-stage parameters change fill and front shape; the factor-stage parameters change the transient schedule and the pivoting; the kernel-stage parameters change the scalar/SIMD and serial/parallel boundaries without changing the result. The tuner (Sec. 9) searches the ordering, `nemin`, relaxation width, `panel_nb`, the kernel thresholds, the method, the LU threshold-pivot tolerance `pivot_u`, the equilibration strategy `scaling`, and the memory mode; static pivoting and the incomplete-factor drop tolerance are set directly by the caller. Two further options are caller- or race-selected rather than searched: the RCM band/profile ordering (a candidate in `AutoRace`) and the right-looking method (an opt-in schedule). Every added axis is exact-path: it changes speed or memory but leaves the factorization exact, and each keeps fill predictable so the a-priori memory backstop (Sec. 9) still holds.

Parameter	Stage	Effect
ordering	analyze	Fill-reducing ordering: AMD, AMF, ND (METIS/Scotch/KaHIP), RCM, or a race (Sec. 3)
nemin	analyze	Supernode amalgamation threshold
relax	analyze	Relaxed amalgamation width / extra-row budget
reorder	analyze	Child-reorder strategy (CB-stack peak vs. leaf parallelism)
scaling	analyze	Equilibration strategy (one-pass ∞ -norm, iterative Ruiz, MC64, off)
method	factor	Left-looking, multifrontal, or right-looking
threads	factor	Worker count (fixed, or the auto predictor)
pivot.u	factor	LU threshold partial-pivot tolerance $u \in [0, 1]$ ($u=0$: static)
on_zero_pivot	factor	Exact fail or static-pivot floor
drop_tol	factor	Incomplete-factor threshold (preconditioner)
memory	factor	Factor emit / peak-RSS strategy
panel_nb	kernel	Bunch–Kaufman / LU panel width
scalar_gate	kernel	Scalar-loop vs. SIMD-GEMM flop gate
par_gemm	kernel	cmod GEMM parallel gate
par_cdiv	kernel	Front / trailing GEMM parallel gate
use_gemm_schur	kernel	SIMD vs. scalar Schur update

Table 2: The flat settings interface. Kernel-stage parameters change the schedule of the dense work but not the numeric result.

8 Determinism

Three determinism properties hold. First, the numeric factorization is bit-identical regardless of the worker count: the parallel schedule changes the order in which independent subtrees and GEMMs execute, but not the accumulation within a pivot block, so a factor computed at one thread and at all cores agree to the last bit. This extends to the front-level parallelism: the trailing Schur GEMM and the trailing-block subtraction of a large top-of-tree front are split over disjoint output blocks, so the write set is a partition and the result does not depend on the worker count. The multi-RHS triangular solve is likewise bit-identical whether the right-hand sides are solved serially or split into per-thread column chunks (each column is independent), so the parallel wide-RHS solve is a pure speedup. The thread count is therefore a performance parameter only. Second, the analyze-phase choices (the auto ordering, the auto thread count, the amalgamation strategy) are deterministic functions of the matrix pattern. Third, the resource planner (Sec. 10) is a pure function of the memory estimate, the budget, and the cached calibration, so a plan is reproducible and can be computed before the matrix is factored. The pivot sequence is value-dependent (Bunch–Kaufman and threshold partial pivoting both branch on magnitudes), so re-factoring the same numeric matrix reproduces the same pivots; a sequence of matrices that share a pattern but differ in values may pivot differently unless the pivot order is fixed. Setting `pivot.u=0` does exactly that: it keeps the natural pivot order and skips the pivot search across refactorizations (static pivoting for frequency sweeps and time stepping), with iterative refinement recovering accuracy when the values drift. The calibration is a timing measurement and is not bit-reproducible, but once cached it makes subsequent planning on the same hardware reproducible.

9 Learned configuration tuner

The tuner selects a `SolverSettings` from the structural feature vector, with **one model per solver path** (symmetric $\mathbf{LDL}^\top$ and unsymmetric LU): the paths have different relevant axes (`pivot.u` only affects LU) and different speed/memory profiles, so each is fit and selected independently and the LU grid searches the threshold-pivot tolerance the $\mathbf{LDL}^\top$ grid pins. Each model is a multilayer perceptron predicting (log factor_ms, log peak_mb) from the standardized features concatenated with an encoding of a candidate configuration: an ordering one-hot, `nemin`, relaxation width,

`panel_nb`, `scalar_gate`, `par_cdiv`, the method, the equilibration-strategy one-hot, the memory mode, and (LU) `pivot_u` — a $37/38 \rightarrow 64 \rightarrow 32 \rightarrow 2$ network with ReLU hidden layers, trained offline with scikit-learn [15] on a complete-distribution corpus (structured-grid generators across real/complex $\times$ symmetric/unsymmetric $\times$ conditioning, including curl-curl Maxwell, Stokes/KKT saddle-point, and convection–diffusion over the grid-Péclet range, plus the SuiteSparse matrices) and exported as JSON weights. Each path is fit on its own class: the LU model on 103 unsymmetric matrices (held-out R^2 0.98 in both time and memory), the $\mathbf{LDL}^\top$ model on 205 symmetric ones. The input encoding is data-driven: the Rust inference builds the same vector the training exported, and a candidate axis is searched only when the shipped model references it, so adding an axis is a retrain, not a code change. Inference is pure Rust and runs at analyze time over the candidate grid; the outputs are destandardized to milliseconds and megabytes. The selected configuration minimizes $w \log(\hat{t}) + (1 - w) \log(\hat{m})$, with the default weight $w = 0.7$; $w = 1$ and $w = 0$ give the speed and memory extremes.

The prediction is constrained by a deterministic guard stack, applied in order:

1. If `factor_flops` exceeds the training range (`flops_ood_cap` $\approx 1 \times 10^{10}$), the model is not trusted; the fallback is not the plain default but a deterministic *ordering race* (`AutoRace`) that picks the minimum-exact-fill ordering (with the default among the candidates, so never worse). The ordering is decidable from the exact a-priori fill, not extrapolated, so this recovers the large-3D / complex-indefinite wins the model cannot: on curl-curl it cut the worst case from $19\times$ to $1.5\times$ of PARDISO.
2. A candidate whose predicted `log_peak_mb` exceeds the default’s by more than $\ln(1.02)$ is rejected.
3. A multifrontal candidate whose estimated transient exceeds the left-looking floor is rejected, using the deterministic estimate rather than the model.
4. The selected configuration is re-analyzed, and accepted only if its exact estimates satisfy $\text{fill} \leq 1.02\times$ default, $\text{flops} \leq 1.05\times$ default, and its memory floor stays under the default’s; otherwise the default is used.
5. The survivor must improve the score by at least a deviate threshold (0.08 by default, or 0.20 if it changes the factorization method). Under a hardware-calibrated profile (Sec. 9.1) this threshold is set to $z \cdot \text{CV}$, the machine’s own single-shot timing noise ($z = 2$), so the tuner never chases a predicted gain smaller than the measurement variance.

The asymmetry between the memory and time guarantees is a consequence of what is predictable. Peak memory is a deterministic function of structure, so a candidate can be guaranteed to never exceed the default’s memory by the estimate-based checks (2)–(4). Factor time depends on the achieved BLAS-3 efficiency, which the model predicts only approximately, so time is optimized in aggregate and backed by the exact-flop proxy in check (4) but is not guaranteed per matrix. The same learned-prediction-with-a-deterministic-fallback structure appears in hardware branch prediction [16], where a misprediction costs only re-execution and not correctness.

Measured against the untuned default, each path is evaluated on *its own* class over 100+ matrices (generated + SuiteSparse), since the two are separate models a caller dispatches to explicitly. The balanced $\mathbf{LDL}^\top$ tuner ($w = 0.7$) is $1.33\times$ faster at $0.83\times$ the memory over 167 matrices; the balanced LU tuner is $1.92\times$ faster at $0.72\times$ the memory over 97 matrices, and the guarded picks never regress memory (Figs. 4–5). The LU result is a direct consequence of corpus coverage: with only a handful of generated unsymmetric matrices the LU tuner was statistically indistinguishable from the default ($1.02\times$); broadening the unsymmetric training set to 90 generated convection–diffusion problems (grid-Péclet $\times$ flow field $\times$ discretization) lifted the per-path held-out R^2 from 0.75 to 0.98 and the tuner from neutral to $1.92\times$ (and to $2.2\times$ on the pure convection–diffusion class). The gains are largest on the big, hard matrices, where a mispicked ordering costs most: on the symmetric head-to-head classes (Sec. 14) the tuner is $1.94\times$ over the default, its full effect on curl-curl and saddle-point.

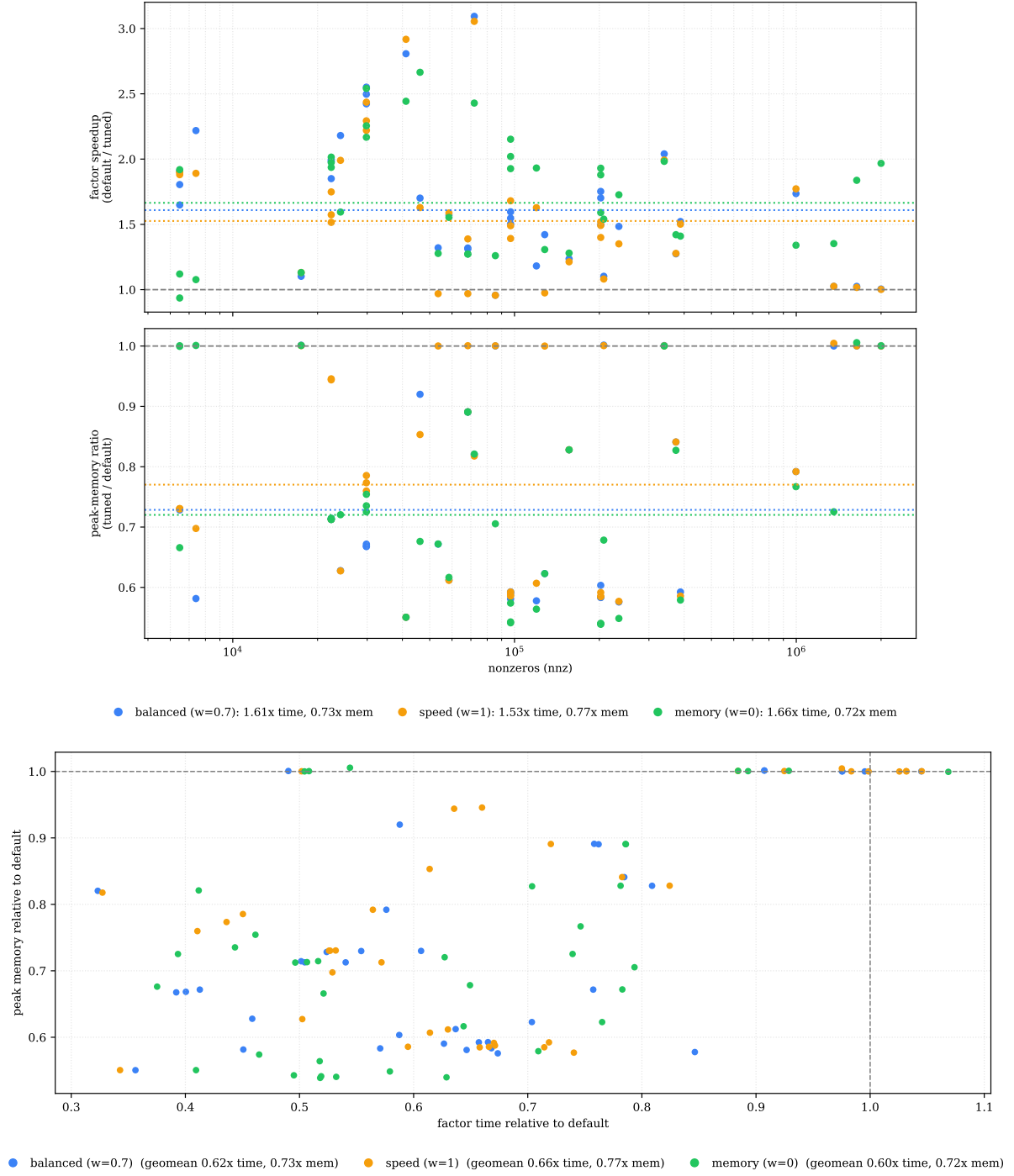


Figure 4: $\text{LDL}^\top$ tuner vs. untuned default over 167 matrices: (top) factor speedup and peak-memory ratio by problem size for the three tuner modes (dotted = geomean); (bottom) the three Pareto modes as a time-memory cloud, at full text width.

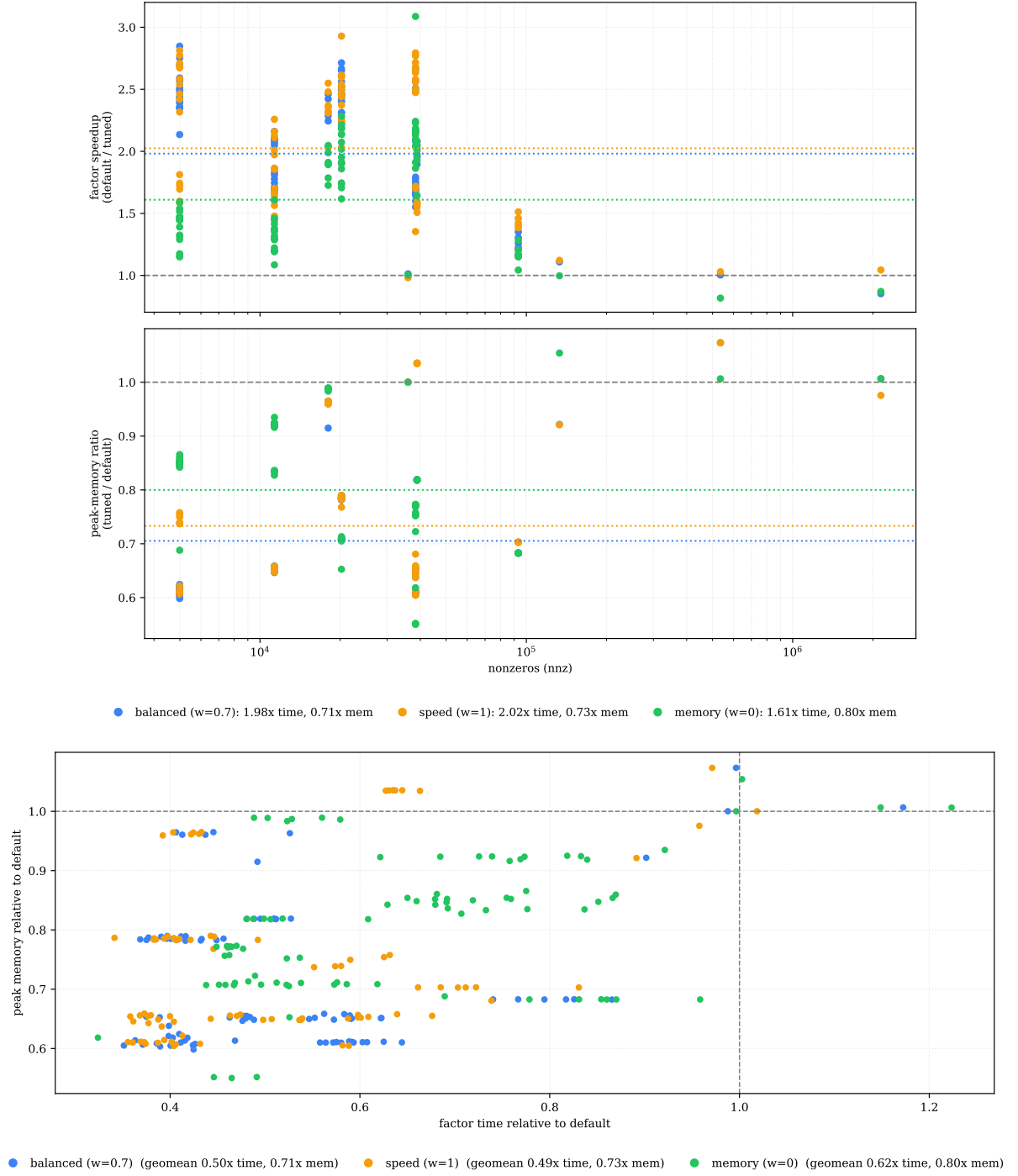


Figure 5: LU tuner vs. untuned default over 97 matrices: (top) factor speedup and peak-memory ratio by problem size; (bottom) the three Pareto modes as a time–memory cloud, at full text width. Nearly every matrix lands below-and-left of the default (faster *and* lighter).

9.1 Runtime profile and the meta-tuner

The two per-path models and the guard thresholds ship compiled-in, but they are also exposed as a runtime *tuner profile*: a JSON artifact bundling both models with the calibrated guards. A profile is applied programmatically (`apply_profile`) or by naming it in the `RSLAB.TUNER.PROFILE` environment variable, which the tuner loads on first use — specializing the solver to a machine or problem class with no recompile. The default profile is a semantic no-op (the embedded models, default guards), so the pathway is exercised on every build.

Profiles are produced by an offline meta-tuner, invoked as `cargo xtask tune`, that chains the whole pipeline deterministically: sweep the corpus, train the two models, measure this machine’s calibration (proxy-flop rate per scalar type, parallel speedup, and the single-shot timing CV that sets the deviate guard), assemble a candidate profile, and *validate* it on a held-out generator corpus (curl-curl and saddle-point at sizes disjoint from training) by factoring each matrix with the shipped default’s recommendation and the candidate’s, and taking the geometric-mean speedup. A *ship-gate* then writes the profile only if it does not regress the default beyond the timing noise floor; a candidate that merely ties still ships (it is the freshly calibrated one), but a regression is rejected and the proven default is kept. This makes retuning safe to run unattended: the worst case a bad sweep or an unlucky machine can produce is the status-quo default, never a slower solver.

10 Solver-in-the-loop and resource governance

Two features support running many solves under a fixed resource budget. First, the analyze and factor phases are separate: a symbolic object is computed once and factors any matrix that shares its pattern, so a sequence of same-pattern systems (a frequency sweep, a time-stepping or Newton iteration) pays the ordering and symbolic cost once and only re-runs the numeric factorization. Second, each factorization runs in a scoped thread pool of a caller-set width, so many concurrent solves coexist without oversubscribing the machine; the fixed-budget thread setting is the intended mode for concurrent solving, where the per-solve auto thread count would otherwise contend.

The resource planner `tuning::plan` turns a memory estimate, a budget (a memory ceiling and a thread budget), and the cached calibration into a plan: a predicted peak, a predicted runtime, and a decision on whether the factorization fits the ceiling. It is a pure function of its inputs. Because the memory estimate is an upper bound rather than a sample, a scheduler can pack concurrent solves against a memory ceiling without the safety margin an unbounded estimate would require. Each solve consumes a bounded memory and a thread width; the thread predictor supplies a scaling-aware width per solve, so non-scaling solves can run many-abreast at low thread counts while a BLAS-3-rich solve claims more cores. Because the per-solve peaks and runtimes are predicted, the aggregate memory footprint and the makespan of a batch are computable before the batch runs. The limits on this are the memory-bandwidth contention between concurrent factorizations, which the current calibration does not model (it measures per-solve speedup, not contention), the conservatism of the memory estimate, which reduces packing density, and NUMA placement on multi-socket machines.

11 Preconditioned and recycled iterative solution

The default factorization is exact and the solve is a direct back-substitution. Two modes relax exactness: static pivoting (Sec. 5) lifts sub-floor pivots so an indefinite or near-singular system never fails, and threshold dropping / block-low-rank make the factor approximate to save memory. In both the factor is a preconditioner $M \approx A$ and accuracy is recovered by iteration. RSLAB ships a Krylov layer for exactly this: the short-recurrence COCG and COCR for the complex-symmetric case, restarted flexible GMRES for the general case, a batched multi-RHS block GMRES, and GCRO-DR Krylov-subspace recycling for sequences of related solves. The iteration and the preconditioner

are decoupled by two traits. `LinearOperator` supplies $y \leftarrow Ax$ and a block $Y \leftarrow AX$; an explicit `CscMatrix/GeneralCsc` implements it, but it may equally be *matrix-free* — an FMM/MLFMA MoM operator the caller writes — with RSLAB factoring only the sparse near-field as the preconditioner. `Preconditioner` supplies $z \leftarrow M^{-1}r$ and a block $Z \leftarrow M^{-1}R$; a factored `LdltSolver` or `LuSolver` implements it through the parallel `solve_many`, so a block Krylov solve inherits the 8 to 19× wide-RHS speedup of the triangular solve. The equilibration and pivot axes (Sec. 7) set the conditioning of that preconditioner factor. Everything below is `src/numeric/iterative.rs` (with the small dense harmonic-Ritz eigensolver in `src/numeric/dense_eig.rs`).

11.1 Complex-symmetric short recurrences

For $A = A^\top$ (complex symmetric, PARDISO `mtype 6`, the frequency-domain EM/FEM target) the iterative method of choice is COCG [38], structurally conjugate gradient but with every inner product taken in the *unconjugated* bilinear form $x^\top y = \sum_i x_i y_i$ — the geometry a complex-symmetric (not Hermitian) operator induces; for $T = \text{f64}$ it reduces to preconditioned CG. COCR [39] is the conjugate-residual analogue, minimising a residual-like quantity and smoother on the strongly indefinite operators (high-frequency 3D Helmholtz) where COCG’s residual can oscillate. Each costs one matvec and one preconditioner apply per step.

Implementation. `cocg` and `cocr` take the unconjugated inner product through the `dotu` helper (no conjugation), and use the genuine Euclidean norm only in the stopping test. A zero bilinear denominator ($p^\top Ap$ or $r^\top z$, possible for an indefinite operator) is a breakdown: the iteration stops and returns the best iterate with `converged = false` rather than dividing by zero.

11.2 GMRES and flexible GMRES

For a general (unsymmetric) operator the method is restarted GMRES [31]. Over the Krylov space

$$\mathcal{K}_m = \text{span}\{r_0, Ar_0, \dots, A^{m-1}r_0\}$$

the Arnoldi process builds an orthonormal basis V_m and an upper-Hessenberg $\bar{H}_m \in \mathbb{C}^{(m+1) \times m}$ with $AV_m = V_{m+1}\bar{H}_m$; the iterate $x = x_0 + V_m y$ that minimises $\|b - Ax\|_2$ is the solution of the small least-squares problem $\min_y \|\beta e_1 - \bar{H}_m y\|_2$, kept in triangular form by applying a Givens rotation to each new Hessenberg column so the update is a back-substitution. After m steps the basis is discarded and the cycle restarts from the current residual: GMRES(m).

Right preconditioning builds the space on $\tilde{A} = AM^{-1}$. Plain right-preconditioned GMRES forms $w = AM^{-1}v_j$ each step and, at the restart, spends one *extra* M^{-1} solve rebuilding the update $x += M^{-1}(V_m y)$. Flexible GMRES [32] instead *keeps* the preconditioned Arnoldi vectors $z_j = M^{-1}v_j$ as a second basis $Z_m = [z_0 \dots z_{m-1}]$ and updates $x += Z_m y$ directly. This (a) removes that per-cycle solve and (b) makes the method flexible: M may *vary* between steps (an inner Krylov solve, or a preconditioner strengthened across iterations), at the cost of one extra $n \cdot (m+1)$ basis.

Implementation. `gmres` is FGMRES: it stores the Z basis `zb` and updates $x += Zy$, so it spends exactly one M^{-1} apply per Arnoldi step and none at the restart (Sec. 14.4 quantifies the saving). The Arnoldi basis is one flat $n \times (m+1)$ buffer and the Hessenberg a flat $(m+1) \times m$ buffer; the per-restart Krylov scalars (Hessenberg, Givens c/s , LS right-hand side, back-substitution) are hoisted out of the outer loop and cleared each cycle, so a many-restart (ill-conditioned) solve does no per-cycle heap allocation. Each cycle first measures the *true* residual $\|b - Ax\|$ — the only reliable stop test on ill-conditioned MoM near-field operators, where the Hessenberg least-squares estimate can dip below `tol` while the true residual is orders larger — and reports it as `final_res` with no separate post-loop matvec. The inner sweep early-exits on the least-squares estimate but does *not* treat it as convergence; the top-of-cycle true-residual check is authoritative.

11.3 Orthogonalization

The Arnoldi orthogonalization sets the backward stability of the whole solve, and RSLAB uses two schemes for the two paths, each backward stable but with a different arithmetic profile. Modified Gram–Schmidt (MGS) loses orthogonality in proportion to the condition number of the basis; a second pass restores it, and the classical DGKS criterion [33] takes that second pass only when one sweep cancels more than a fixed fraction $\eta = 1/\sqrt{2}$ of the vector norm. Classical Gram–Schmidt (CGS) is a single BLAS-2/3 sweep — higher arithmetic intensity, but numerically weaker; CGS with one reorthogonalization (CGS2) recovers a backward-stable basis to working precision [34], and the block generalization and its naming follow [35].

Implementation. The single-RHS `gmres` uses MGS with the conditional DGKS second pass ($\eta = 1/\sqrt{2}$), the latency-bound but strongest scheme for one right-hand side. The block path (below) uses *block* CGS2: the whole s -column panel is projected against the basis in two panel-wide sweeps (`block_project` / `block_subtract`) for BLAS-3 arithmetic intensity, with the conditional second pass decided *per column* from that column’s own norm collapse, so a single ill-conditioned right-hand side no longer imposes the second orthogonalization on the well-conditioned ones. Because MGS and CGS accumulate the projections in a different order, a block solve with $s = 1$ is *not* bit-identical to the single-RHS `gmres` and may differ by up to ± 1 iteration at a residual straddling `tol`; both converge to the requested tolerance. This is a deliberate design point (the panel-wide reductions are what make the multi-RHS path fast and thread-count deterministic), covered by the `gmres_block_single_rhs_matches_scalar_gmres` test.

11.4 Block GMRES: batched-independent systems with within-cycle deflation

The multi-RHS solver drives s right-hand sides in lockstep so the two expensive operations — the operator matvec and the preconditioner solve — are issued once per step as *block* applies (`apply_block`), each matrix or factor value touched once for all s columns (the BLAS-3 arithmetic intensity that makes a wide solve pay over s separate ones). To be precise, this is *not* a shared-subspace block Krylov method (a single Krylov space over all right-hand sides); that is future work (issue #3). Each right-hand side keeps its *own* Arnoldi basis, Hessenberg, and Givens state and converges identically to the single-RHS `gmres`; the systems share only the batched operator and preconditioner calls. This batched-independent design is what lets a column drop out cleanly the moment it converges: a right-hand side whose true residual reaches `tol` is *deflated* from the block, and — the v0.13 refinement — a column whose Hessenberg estimate reaches `tol` *mid-cycle* is finalized and compacted out immediately, so the batched applies shrink to the still-active width *within* a cycle, not only at the restart. The fast-converging right-hand sides are never dragged along by the slowest one.

The block matvec and block preconditioner apply are BLAS-3 and parallel; the remaining serial, latency-bound piece was the orthogonalization, done per right-hand side by MGS+DGKS — $O(m \cdot s)$ sequential BLAS-1 reductions per Arnoldi step. RSLAB replaces it with the block CGS2 of Sec. 11.3: the whole s -column panel is projected against the basis in two panel-wide sweeps parallelized over the vector dimension in fixed row chunks. The result is memory-neutral (the same block Krylov basis $n \cdot s \cdot (m+1)$ dominates; the projection scratch is $\sim 0.05\%$ of it) and *bit-identical across thread counts* — the chunked reduction sums in a fixed order independent of the worker count.

Implementation. `gmres_block` sizes the Arnoldi basis for the full width s but uses only the first `sa` columns each cycle, where `sa` is the count of active right-hand sides; the basis stride is therefore `sa`, recomputed per cycle. When a column converges mid-cycle, `finalize_block_column` back-substitutes its frozen Hessenberg state, forms Vy from the basis at the *current* stride, applies the preconditioner once, and adds the contribution to x ; the surviving columns are then compacted

to the narrower stride (a monotone copy that never clobbers unread data) and their per-column Arnoldi state is remapped by $O(1)$ vector swaps. The parallel orthogonalization reductions run in a scoped pool derived from the preconditioner’s thread policy (Sec. 8), so factor and solve share one concurrency budget.

Figure 6 measures a right-preconditioned convection–diffusion solve ($n = 40\,000$, 63 to 77 GMRES iterations, complex `f64`). BCGS2 scales to $\sim 2.2\times$ at 12 cores, where the per-RHS MGS path is flat-to-negative — its serial orthogonalization does not scale at any thread count, so more threads (fighting the parallel matvec/solve for memory bandwidth) can even regress it. The efficiency knee is at 4 to 6 threads: the orthogonalization is no longer the bottleneck, and the residual serial fraction is the sparse triangular preconditioner solve. Per-RHS cost stays near-flat in the block width s (adding right-hand sides is cheap, BLAS-3 reuse) and 1.5 to $1.6\times$ below the MGS reference at 12 threads.

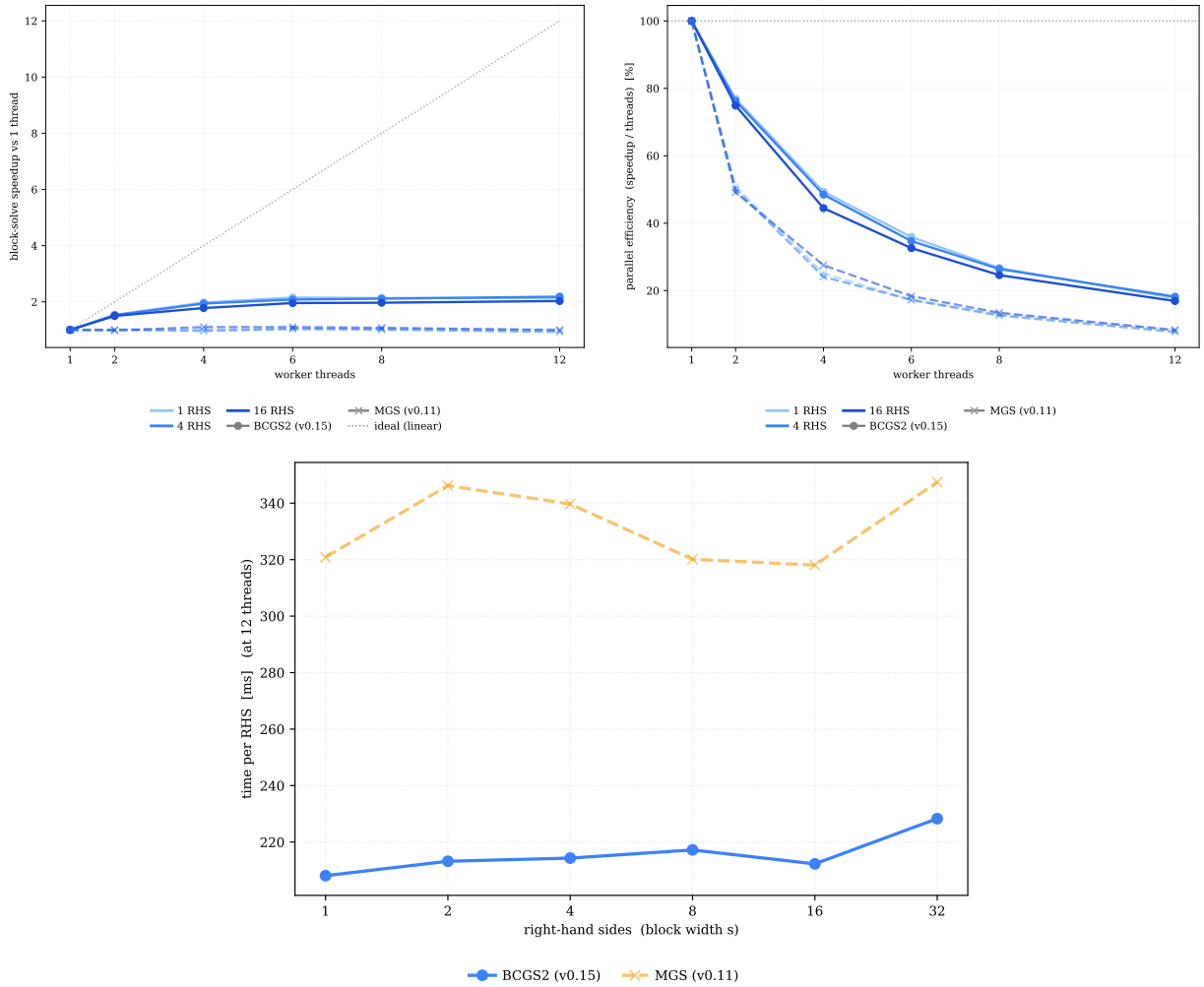


Figure 6: Multi-RHS block GMRES, BCGS2 (v0.15, solid) vs. the pre-BCGS2 per-RHS MGS (v0.11, dashed). Left: strong-scaling speedup vs. threads per block width. Right: parallel efficiency. Bottom: per-RHS cost vs. block width at 12 threads. Preconditioned convection–diffusion, $n = 40\,000$, complex `f64`.

The within-cycle deflation itself is a work reduction, measured in Fig. 7 on a controlled multi-rate testbed (a diagonal operator whose right-hand sides have staggered minimal-polynomial degrees, so the columns converge at spread rates — the benchmark-scale mirror of the width-counting deflation test in `src/numeric/iterative.rs`). A counting operator records the width of every batched apply. As the block widens, the compaction removes a growing fraction of the batched column-applies: at $s = 16$ the operator does 908 column-applies against the 1376 a schedule with no mid-cycle compaction would do ($0.66\times$), and the active panel drains monotonically from

16 to a single column within the first cycle.

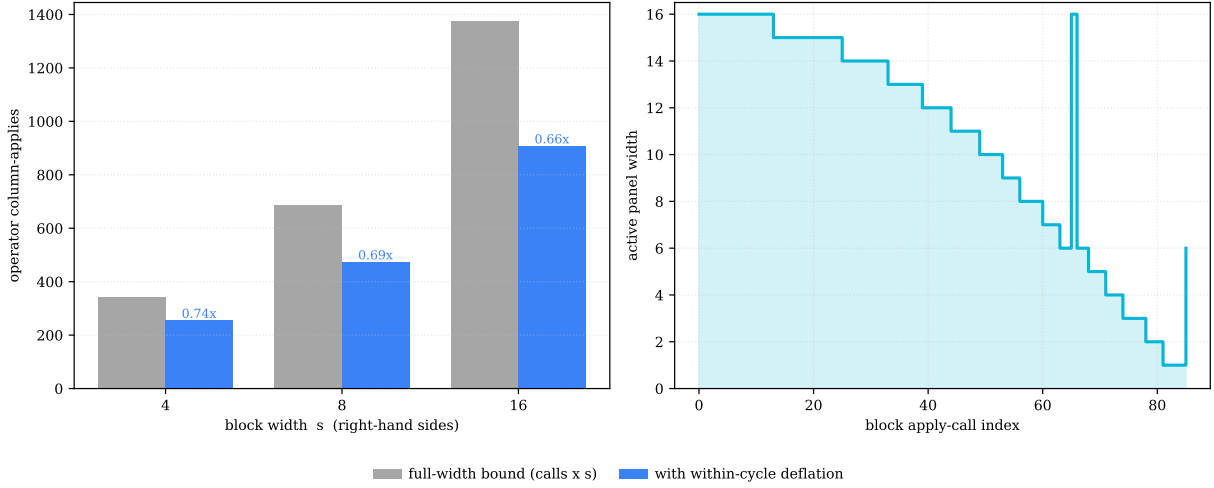


Figure 7: Block-GMRES within-cycle deflation. Left: total operator column-applies with mid-cycle compaction vs. the full-width bound (`calls` $\times$ s), per block width. Right: the active-panel width per batched apply for $s = 16$ — the panel drains as columns converge, then refills briefly at the once-per-cycle full-width residual check. Multi-rate diagonal testbed, $n = 20\,000$.

11.5 Breakdown guard

The least-squares back-substitution divides by the Hessenberg diagonal, which after the Givens rotation is $\sqrt{|f|^2 + |g|^2}$ and normally nonzero; but under exact stagnation or a rank-deficient Hessenberg (hard, indefinite, or singular operators) some diagonal can be zero, and an unguarded reciprocal would inject ∞/NaN into the iterate and the residual. `well_conditioned_dim` returns the largest leading dimension whose Hessenberg diagonals all clear a relative threshold $\varepsilon \cdot \max_i |h_{ii}|$; the solve is truncated to that well-conditioned prefix, the degenerate tail dropped, and the outer true-residual check reports non-convergence — a *deterministic breakdown* rather than a NaN. In the well-conditioned case every diagonal clears the threshold and the normal path is unaffected. Both the single-RHS and the block paths carry the guard (the block on each column’s Hessenberg before its restart update).

11.6 Restart length and the memory budget

The restart length m trades convergence against memory. A longer basis captures more of the Krylov space, so fewer restart cycles are needed and the iteration count falls (with diminishing return); but the Arnoldi basis is allocated *up front*, so m fixes the memory floor regardless of how few iterations actually run — $2n(m+1)$ scalars for the single-RHS V/Z pair, $ns(m+1)$ for the block basis, which at $n = 100\,000$, $s = 10$, $m = 80$, complex `f64` is ≈ 13 GB. The Python default therefore does not pin m : it caps an unspecified restart so the basis stays under a 1 GiB budget, clamped to $[20, 80]$, with an explicit value honoured exactly. Figure 8 shows the trade-off the policy navigates: a longer restart cuts iterations, and the adaptive rule rides at the maximum restart until the basis would exceed the budget, then declines to hold memory flat — the largest restart that fits the ceiling.

11.7 Warm start

On a sequence of related systems — a frequency sweep, a time-stepping or Newton iteration — the previous solution is a good initial guess. All three GMRES paths take an optional x_0 (single-RHS, or column-major $n \times s$ for the block): the cycle’s true residual $b - Ax$ then measures progress

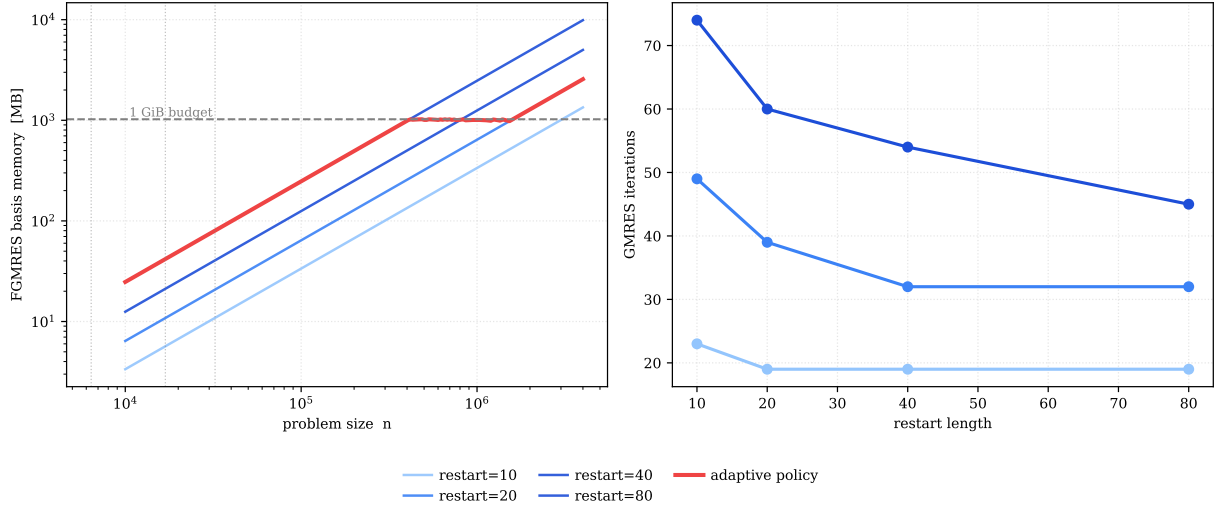


Figure 8: Adaptive GMRES restart under a 1 GiB basis budget. Left: FGMRES basis memory vs. problem size for fixed restarts and the adaptive policy (analytic); the policy tracks the maximum restart until the budget binds, then flattens along it. Right: measured GMRES iterations vs. restart per size — a longer restart cuts iterations with diminishing return. Preconditioned convection–diffusion.

from that guess, while convergence stays relative to $\|b\|$. When the right-hand side barely moves the per-solve iteration count collapses.

11.8 Incomplete-factor trade-off

Threshold dropping trades factor fill for GMRES iterations: entries of the factor below a relative `drop_tol` are discarded, shrinking the preconditioner, and GMRES makes up the lost accuracy. Figure 9 sweeps `drop_tol` on one convection–diffusion system ($n = 32\,400$). Fill falls monotonically while the iteration count rises — slowly at first, then steeply once the factor is too sparse to precondition well. Because the factor is the memory driver, the useful regime is the flat-iteration knee: at `drop_tol`= 1×10^{-2} the fill halves ($26 \rightarrow 11$ MB) for ~ 15 iterations and a *total* wall time within a few percent of the exact direct solve — a factor-memory halving essentially for free. Past $\sim 3 \times 10^{-2}$ the iteration cost dominates and total time climbs, bounding the useful range.

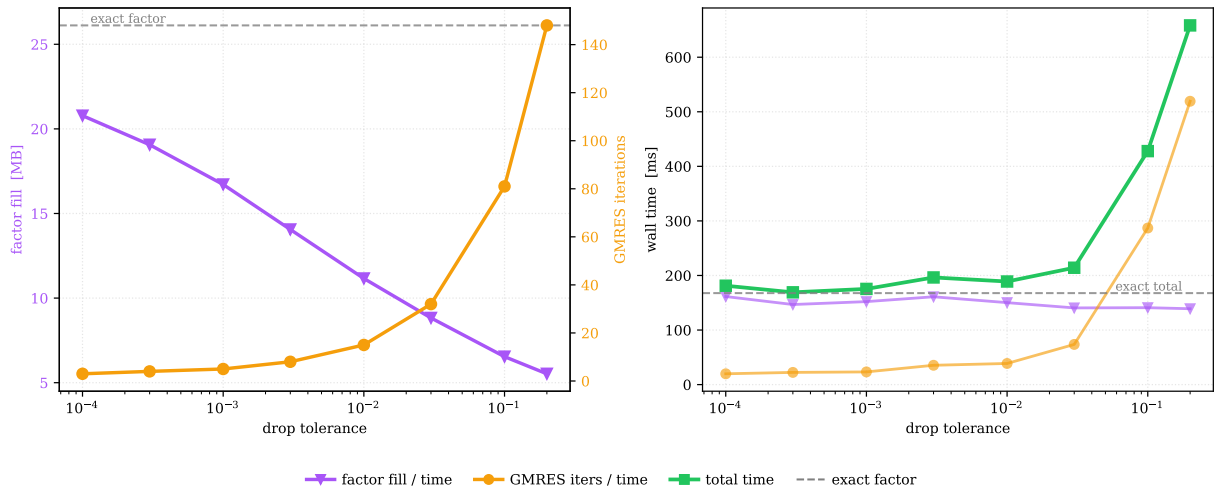


Figure 9: Incomplete-factor preconditioner + GMRES. Left: factor fill (memory) and GMRES iterations vs. `drop_tol`, with the exact factor as reference. Right: factor / GMRES / total wall time, exposing the sweet spot. Convection–diffusion, $n = 32\,400$, complex `f64`.

11.9 GCRO-DR: Krylov subspace recycling

RSLAB targets solver-in-the-loop workloads: many solves of a slowly varying $A/M/b$. On a *stagnating* system — a cluster of small eigenvalues of $\tilde{A} = AM^{-1}$ that restarted GMRES keeps re-discovering and discarding at every restart — plain FGMRES(m) throws away exactly the near-invariant subspace that governs its convergence. GCRO-DR (Parks, de Sturler, Mackey, Johnson & Maiti [36], combining de Sturler’s GCRO with Morgan’s GMRES-DR deflated restarting [37]) retains k approximate smallest eigenvectors of $\tilde{A}$ and (a) *deflates* them across restarts *within* a solve, and (b) *recycles* them *across* solves through an opaque handle, so the next related system starts already knowing where the trouble is.

Harmonic Ritz vectors. The retained subspace is spanned by the k harmonic-Ritz vectors of smallest value. Over the augmented search space $[U \mid V_p]$ of the current cycle (U the previous recycle vectors, V_p the $p = m - k$ Arnoldi vectors), the harmonic-Ritz pairs (θ, g) solve the generalized eigenproblem $\tilde{G}^H \tilde{G} g = \theta \tilde{G}^H (\hat{W}^H [U, V]) g$, where $\tilde{G} = \begin{bmatrix} I_k & B \\ 0 & \tilde{H} \end{bmatrix}$ is the $(d+1) \times d$ matrix ($d = k + p$) relating the space to its image $\hat{W} = [C \mid V_{p+1}]$ under $\tilde{A}$, $B = C^H \tilde{A} Z$ records the C -components of the Arnoldi directions, and $\tilde{H}$ is the Arnoldi Hessenberg. The k smallest $|\theta|$ give the new $U \leftarrow [U, V] G_k$: the directions $\tilde{A}$ has barely resolved, which is what deflating accelerates.

Preconditioned-space formulation. Because the Krylov space is built on $\tilde{A} = AM^{-1}$ (FGMRES stores $Z = M^{-1}V$ and updates $x \leftarrow x + Zy$), U lives in the *preconditioned* space. Each cycle recomputes $P = M^{-1}U$ (k preconditioner solves) and $C = AP = \tilde{A}U$ (k matvecs), orthonormalizes $C = QR$ while applying the same R^{-1} to P and U so the invariants $C = \tilde{A}U$ and $P = M^{-1}U$ still hold, then runs the GCRO split: the residual is first minimised over $\text{range}(C)$ ($x \leftarrow x + PC^H r$, $r \leftarrow C C^H r$), and the rest of the Krylov space is built *orthogonal to* C (each Arnoldi direction has its C -component removed, recorded in B). The restart least-squares then decouples: y is the ordinary GMRES solution of the Hessenberg system and the recycle coordinate is $z_1 = -By$, giving $x \leftarrow x + Pz_1 + Zy$. Recomputing C every solve (the default) is k matvecs + k M -solves and keeps the split exact when A or M change between solves — there is never a stale C — which is precisely what a slowly-varying sequence needs.

Correctness and cost. U only *shapes* the Krylov space to accelerate convergence; the outer true-residual stop test is authoritative, so an inaccurate or rank-deficient recycle subspace can only make a solve slower, never wrong. Every recycle step is accordingly guarded (rank-dropping in the orthonormalization, singular-projection fallbacks) and degrades gracefully to plain FGMRES; k is capped at $m/2$ so the Arnoldi space is never starved. U , C , P are each $n \cdot k$ scalars, of which only U persists between solves.

Implementation. `gmres_recycled` carries the subspace in an opaque `Recycle` handle, refreshed in place at the end of every solve; it composes with the x_0 warm start (seed from the previous solution *and* recycle its stagnation subspace). The harmonic-Ritz eigenproblem is assembled in `recompute_recycle` and solved in `src/numeric/dense_eig.rs` entirely in `Complex<f64>` regardless of the working field: the standard-form matrix $M_2^{-1}M_1$ is formed by a partial-pivot dense LU, its eigenvalues by Householder Hessenberg reduction and shifted Givens QR, and the coefficient vectors by inverse iteration — all capped and best-effort, since an approximate spectrum only slows the outer solve. Recycling is implemented for the single-RHS FGMRES path only; the block path’s within-cycle basis compaction does not compose cleanly with a shared recycle subspace and is left as future work. For the *real* field a real U cannot store a complex eigenvector, so `combine_ritz` splits each complex conjugate Ritz pair into its real and imaginary parts (two real recycle vectors). Section 14.4 and Fig. 13 measure the effect on a sequence of related solves.

11.10 Mixed-precision preconditioning

A factor stored in `Complex<f32>` occupies half the bytes and its triangular solves run in single precision, but an *f64* outer iteration keeps the *solution* at full *f64* accuracy: the standard mixed-precision setup for large 3D EM. `LowPrecisionPreconditioner` (symmetric $\mathbf{LDL}^\top$) and `LowPrecisionLu` (unsymmetric) down-cast A , factor in single precision, and up-/down-cast inside apply, while COCG/COCR/GMRES iterate in the working precision; on a strongly diagonally dominant operator a couple of iterations recover the factor’s $\sim 10^{-6}$ apply floor at half the factor memory.

12 Python API surface

The PyO3 layer (`python/src/lib.rs`) exposes the `analyze/factor/solve` flow and the Krylov layer with the diagnostics a solver-in-the-loop caller needs. A factored `Ldl` or `Lu` object carries `drop_tol=` and static-pivot options at construction, and its `gmres` / `gmres_block` methods return the full diagnostics tuple (`X`, `converged`, `iters`, `final_res`) rather than a bare solution — `converged` is `True` only when *every* right-hand side reached the tolerance, so a caller never mistakes a stalled iterate for a solution. Both accept an optional `x0=` warm start (same $n \times \text{nrhs}$ layout as B). The single-RHS `gmres` additionally accepts a `recycle=` handle: `rslib.Recycle(k)` is an opaque object (`dim`, `active`, `dtype`, `clear`) carried across a sequence of related solves and refreshed in place, exposing the GCRO-DR path of Sec. 11.9 to Python. When `restart` is left unspecified the binding applies the adaptive memory cap of Sec. 11.6 (`adaptive_restart`: basis under 1 GiB, clamped [20, 80]), so a Python caller cannot silently allocate a multi-gigabyte basis; an explicit `restart=` is honoured exactly. The memory formulas ($2n(m+1)$ single-RHS, $n s (m+1)$ block) are documented on the methods so the cost is predictable before the call.

13 Test matrix corpus

The tuner is fit and the solver evaluated on a corpus deliberately spanning the full problem distribution — both paths ($\mathbf{LDL}^\top$ and LU), real and complex values, and the conditioning classes (SPD, symmetric indefinite, unsymmetric, ill-scaled, saddle-point) — rather than a single family. The SuiteSparse collection holds few genuinely complex, non-Hermitian matrices, so the backbone is a set of structured-grid generators assembled directly with finite differences (no external FEM library), complemented by the complex SuiteSparse matrices that do exist (the Bai *qc/dwg*, QCD lattice, waveguide, and dielectric-filter EM matrices).

Curl-curl Maxwell (complex symmetric indefinite). The time-harmonic operator $\nabla \times \nabla \times E - (\omega^2 - i\omega\sigma) E$, discretized on a structured grid through the identity $\nabla \times \nabla \times = L - \text{grad div}$ (the component-wise vector Laplacian L minus $D^\top D$ from the discrete divergence D), with lumped mass $M = I$. The $-\omega^2 M$ shift makes the operator indefinite; the conductivity term $i\omega\sigma M$ makes it complex symmetric (not Hermitian); and $\nabla \times \nabla \times$ is singular on discrete gradients, giving the gradient near-null-space that makes edge-element electromagnetics hard — the primary target class [28, 27].

Stokes / KKT saddle-point (symmetric indefinite). The block system

$$\begin{bmatrix} A & B^\top \\ B & -\beta C \end{bmatrix}$$

with A the velocity vector Laplacian, B the discrete divergence coupling velocity to pressure, and $-\beta C$ a Brezzi–Pitkäranta pressure-Laplacian stabilization ($\beta > 0$) regularizing the singular zero (2, 2) block on a collocated grid — the MAC / mixed-FEM form [29, 30].

Convection–diffusion (unsymmetric). The operator $-\varepsilon \nabla^2 u + \mathbf{b} \cdot \nabla u = f$ finite-differenced on a structured grid with Dirichlet boundaries — the canonical unsymmetric class (the LU path’s workload). The first-derivative advection term $\mathbf{b} \cdot \nabla u$ breaks symmetry; central differencing gives the sharp, oscillation-prone form that stresses pivoting at small ε (high grid-Péclet $|\mathbf{b}|h/\varepsilon$), first-order upwinding a diagonally dominant M-matrix. Sweeping ε moves the operator from diffusion-dominated (nearly symmetric) to advection-dominated (strongly unsymmetric), and the flow field $\mathbf{b}$ (a uniform wind or a recirculating vortex) varies the coupling structure. The parameter grid (four Péclet levels $\times$ two flows $\times$ two schemes $\times$ several grids, 2-D and 3-D) yields 90 generated unsymmetric matrices spanning the distribution real CFD/transport solves live on — the LU tuner’s training and held-out corpus.

Remaining classes. Shifted Helmholtz (complex symmetric), banded / arrow / jumping-coefficient stencils, BEM/MoM near-field kernels (complex unsymmetric), and random SPD / unsymmetric / exactly-conditioned spectral matrices span the rest. Every generator is type-agnostic, so each class is available in `f64` and `Complex<f64>`. This complete-distribution corpus was load-bearing: it exposed the tuner mispicks on complex curl-curl that a real-SPD corpus had hidden, and the too-thin unsymmetric coverage that had left the LU tuner neutral until the convection–diffusion set was added (Sec. 14).

14 Evaluation

All measured figures come from the `bench_suite` engine over the complete-distribution corpus of Sec. 13 (the structured-grid generators and the complex SuiteSparse [24] matrices, 8k–125k DOFs, all `Complex<f64>`) on a 12-core / 24-thread machine; the per-stage, estimator, thread-scaling, and tuner figures appear with the concepts they support in the preceding sections. This section reports the cross-solver comparison. RSLAB, faer [22], and MKL PARDISO [21] were measured in one run, so the cross-solver numbers are not affected by run-to-run drift.

14.1 Scaling

The two paths are separate solvers a caller dispatches to explicitly, so each is compared on *its own* class against its own PARDISO mtype (6 for complex-symmetric, 13 for unsymmetric) and faer: Figs. 10 and 11 plot factor time and peak memory against nnz with a least-squares power-law fit $\sim \text{nnz}^\alpha$ per solver (solves with residual above 0.1 excluded). The RSLAB curve is the auto-tuned default, which picks left-looking or multifrontal per matrix under the guards. Table 3 gives the head-to-head geomean ratios.

RSLAB vs	$\mathbf{LDL}^\top$ (sym)	LU (unsym)
MKL PARDISO — factor time	7.0 $\times$ slower	4.0 $\times$ slower
MKL PARDISO — peak memory	2.2 $\times$ more	2.4 $\times$ more
faer LU [†] — factor time	14.5 $\times$ faster	6.7 $\times$ faster
faer LU [†] — peak memory	2.3 $\times$ less	1.1 $\times$ less
untuned default — factor time	1.94 $\times$ faster	1.78 $\times$ faster
untuned default — peak memory	0.68 $\times$ (less)	0.72 $\times$ (less)

Table 3: Head-to-head geomean ratios (~ 100 matrices per path, 5k–1M nonzeros), each path on its own class ($\mathbf{LDL}^\top$: curl-curl, Helmholtz, saddle-point; LU: convection–diffusion, BEM/MoM), over the matrices both solvers factor to below 0.1 residual. The last row is the auto-tuned solver’s speedup over the untuned RSLAB default. All passes of this regeneration were measured on a verified-quiet machine (no concurrent load); the peak-memory ratios are allocation counts and load-independent, and reproduce the previous run to within 5%. [†]faer has no symmetric path (it factors symmetric matrices as LU), so its $\mathbf{LDL}^\top$ gap is structurally largest; it also OOMs on the largest matrices (factoring only the smaller subset), so its head-to-head is a conservative floor.

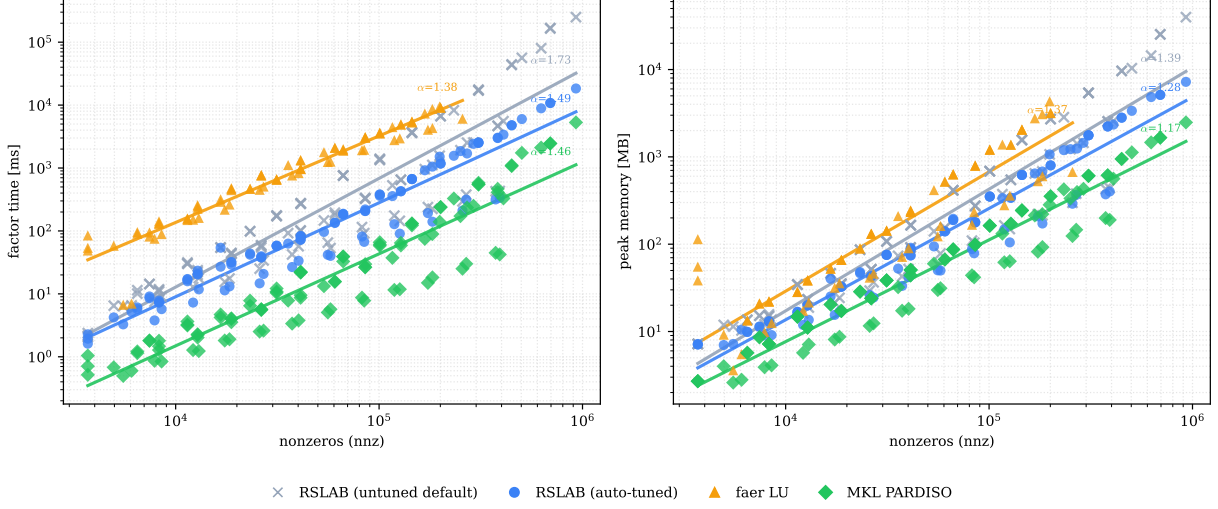


Figure 10: $\text{LDL}^\top$ path (symmetric, PARDISO mtype 6): (left) factor wall-clock time and (right) peak memory vs. problem size — RSLAB vs. faer vs. MKL PARDISO, with power-law fits (the fitted exponent α is annotated on each curve).

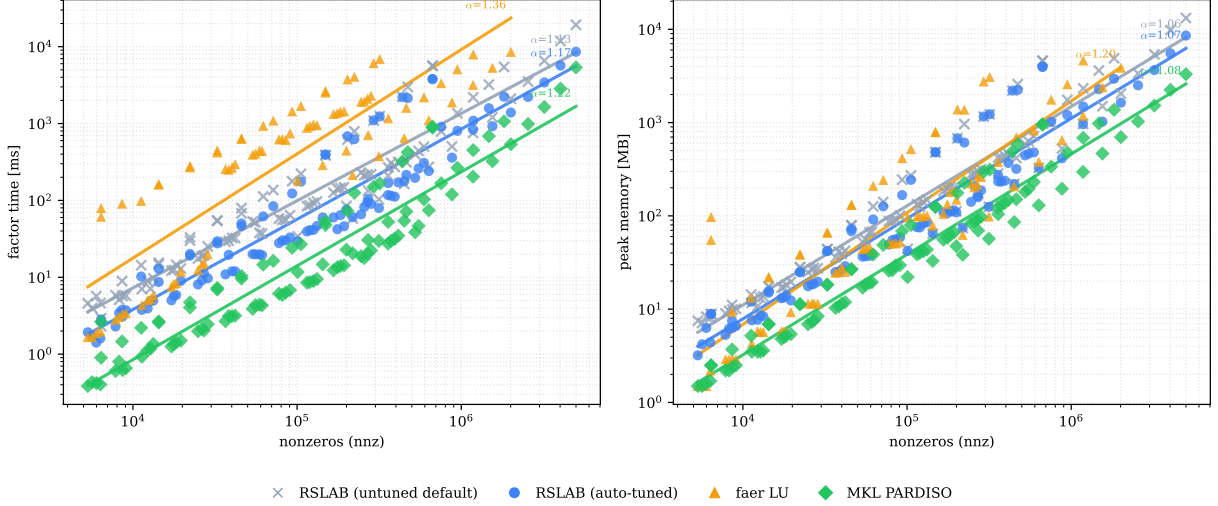


Figure 11: LU path (unsymmetric, PARDISO mtype 13): (left) factor wall-clock time and (right) peak memory vs. problem size. Each panel carries the untuned-default RSLAB curve (gray) beside the auto-tuned one (blue); the tuner closes part of the default→PARDISO distance. On time RSLAB scales slightly flatter than PARDISO ($\alpha \approx 1.17$ vs. 1.22).

RSLAB sits between the two on both paths: faster and lighter than the pure-Rust faer, moderately behind the hand-optimized MKL PARDISO. The unsymmetric path is the closer of the two ($4.0\times$ vs. $7.0\times$ behind PARDISO). Each head-to-head plot carries the untuned default curve beside the tuned one, so the learned tuner’s win (last two table rows, $1.94\times/1.78\times$ faster at $0.68\times/0.72\times$ the memory) is visible as a gap that widens with problem size — the tuner helps most on the big matrices where a mispicked ordering costs most, and never at a memory cost: the backstop compares the *exact* symbolic fill (not a dense-panel estimate, which overshoots non-uniformly across orderings and once let a $2\times$ -fill pick slip through), so a pick can never grow the factor beyond the default’s. The corpus choice was load-bearing throughout: benchmarking on real-SPD matrices flattered RSLAB ($7\times$) and hid two tuner mispicks on complex-indefinite curl-curl, where the adaptive Auto ordering produces $\sim 3\times$ the fill of nested dissection. Adding MetisND to the tuner grid and racing orderings out-of-distribution (the AutoRace fallback, deciding

on the exact a-priori fill rather than the extrapolating model) cut the worst curl-curl case from $19\times$ to $1.5\times$ of PARDISO.

14.2 Accuracy

Figure 12 shows the relative residual $\|Ax - b\|/\|b\|$ per solver. Where RSLAB factors, 24 of 31 matrices are below 1×10^{-8} , matching PARDISO and ahead of faer. The exact $\mathbf{LDL}^\top$ declines a handful of indefinite saddle-point matrices rather than returning a degraded solution.

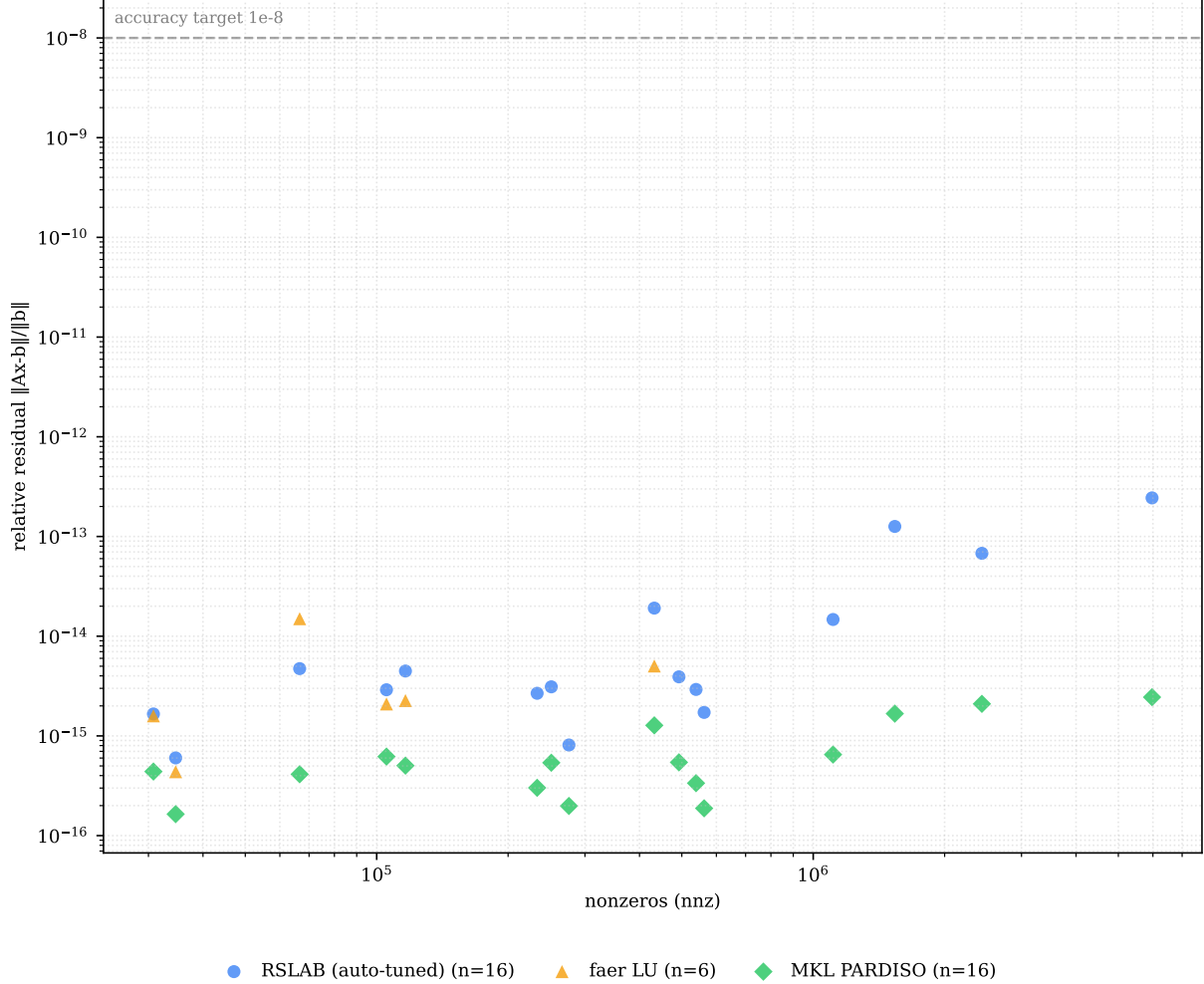


Figure 12: Relative residual across the corpus, per solver.

14.3 Exact-path axes: correctness and speedups

The tunable axes were validated over 180 SuiteSparse matrices spanning structural, CFD, thermal, and saddle-point classes. Every determinism and equivalence property held on all 180: the right-looking factor is bit-identical to the multifrontal one; the `Eager` and `LowMemory` emit modes produce bit-identical factors; the parallel front subtraction is bit-identical to serial; and the 32-bit compressed factor solves bit-identically to the full-width one. Where the axes carry a measurable gain, the parallel multi-RHS solve is 8 to $19\times$ faster than solving the 256 right-hand sides column by column, and 32-bit index compression halves the stored index footprint (for example 459 MB $\rightarrow$ 230 MB on `pwt`) at no accuracy cost. The block/multi-RHS Krylov solvers (block GMRES, COCG, COCR) apply the factored preconditioner through exactly this parallel `solve_many`, so they inherit the wide-RHS speedup directly, and the equilibration/pivot axes set

the quality of that preconditioner factor; the exact-path work is otherwise orthogonal to the Krylov layer. Retraining the tuner with the new axes is neutral on the well-scaled structural corpus (geomean $1.03\times$ versus the pre-axis model, within the $\pm 20\%$ single-shot variance): the threshold-pivot and equilibration axes bind on unsymmetric and ill-scaled classes, and the memory backstop keeps the guarded picks from regressing where they do not help.

14.4 Preconditioned and recycled iterative layer

The Krylov layer of Sec. 11 is evaluated on four studies, each the benchmark-scale mirror of a unit test (`benches/recycle`, `deflation`, `fgmres_accounting`, `adaptive_restart`); the block-scaling and incomplete-factor trade-off figures (Figs. 6, 9) and the deflation and restart figures (Figs. 7, 8) appear with their concepts in Sec. 11.

Subspace recycling. Figure 13 runs a sequence of 8 related solves — a diagonal operator with a stagnation spectrum (a tight cluster of small eigenvalues below a spread bulk), a per-step diagonal perturbation, and a rotating right-hand side, mirroring the `gmres_recycled_cross_solve_beats_warm_and_cold` test at $n = 20\,000$ — comparing cold ($x_0 = 0$), warm ($x_0 = \text{previous solution}$), and GCRO-DR(k) recycling (warm start plus a handle carried across the sequence). The iteration counts order recycling < warm < cold by a wide margin. On the first solve alone (a fresh handle, so purely the *within-solve* deflated restarting) GCRO-DR($k=10$) cuts 360 iterations to 124 ($2.9\times$); over the whole sequence the cumulative *cross-solve* total falls from 2837 (cold) to 444 ($6.4\times$ fewer iterations), for a $1.5\times$ wall-clock reduction after the per-iteration recycle overhead (the dense harmonic-Ritz work and the C -orthogonalization). The $k=20$ curve coincides with $k=10$ because k is capped at $m/2$ (here $m = 20$). The wall-clock win is smaller than the iteration win because the recycle bookkeeping is $O(nk)$ per step against an $O(n)$ diagonal matvec; on a denser operator, where the matvec and preconditioner solve dominate, the wall-clock tracks the iteration count more closely.

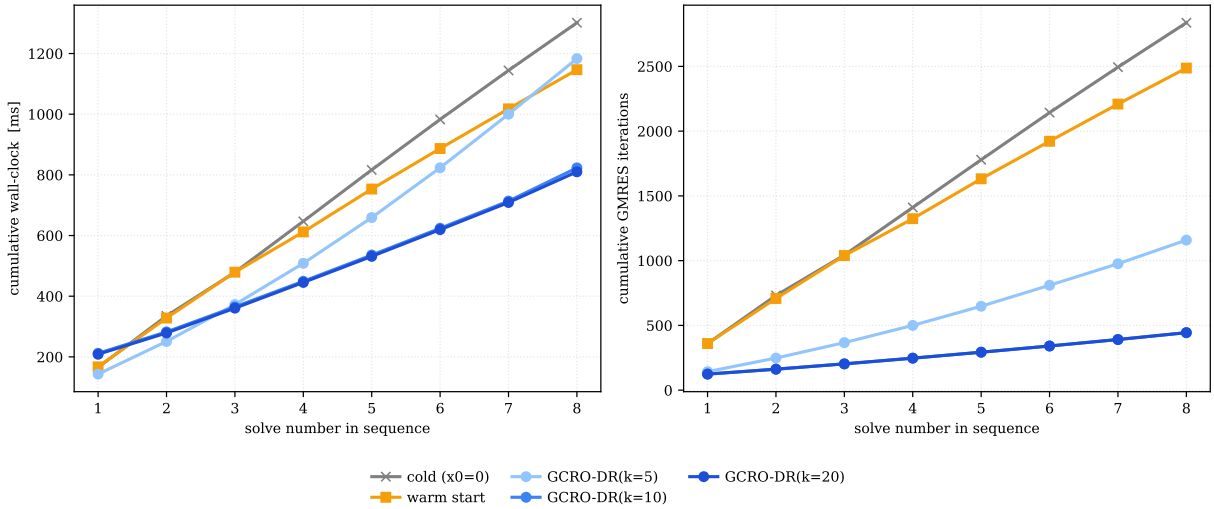


Figure 13: GCRO-DR subspace recycling over a sequence of 8 related solves (stagnation spectrum, $n = 20\,000$). Left: cumulative wall-clock; right: cumulative GMRES iterations, for cold, warm, and GCRO-DR(k). Recycling cuts the cross-solve iteration total $6.4\times$ versus cold ($k=20$ overlaps $k=10$, the $m/2$ cap).

FGMRES accounting. The flexible-basis update saves one preconditioner solve per restart cycle (Sec. 11). Counting the M^{-1} applies of a weak-factor preconditioned convection–diffusion solve (short restart, several cycles) confirms it exactly: FGMRES spends one apply per Arnoldi

step and none at the restart, where plain right-preconditioned GMRES would spend one extra per cycle (Table 4). The saving grows with the cycle count and is free of any extra memory beyond the Z basis already stored for flexibility.

n	iters	cycles	FGMRES M^{-1}	plain M^{-1}
3600	24	3	24	27
10 000	53	4	53	57
19 600	89	6	89	95

Table 4: Preconditioner-apply accounting for FGMRES vs. plain right-preconditioned GMRES (emulated by counting): one M^{-1} solve per Arnoldi step for FGMRES, plus one per restart cycle for the plain variant. Convection-diffusion, `drop_tol`=0.1, restart 20.

The remaining two studies appear in Sec. 11: within-cycle deflation reduces the batched operator column-applies to $0.66\times$ the full-width bound at $s = 16$ (Fig. 7), and the adaptive restart holds the FGMRES basis under its 1 GiB budget while picking the longest restart that fits (Fig. 8). The total wall-clock of the four new benchmarks is under a minute; they are trend measurements, sized for the story rather than records.

15 Related work

Automatic and learned tuning of sparse solvers has been studied along several axes. GPTune [17] tunes HPC libraries including SuperLU_DIST by Bayesian optimization with multitask learning across problem sizes, using online trial runs per problem; the RSLAB model is a feed-forward regressor evaluated once at analyze time. OSKI [18] autotunes at the sparse-kernel level (register and cache blocking for sparse matrix-vector products), not at the solver-configuration level. Lighthouse [19] and the work of Bhowmick, Raghavan et al. [20] apply machine learning to select a solver or preconditioner from a portfolio, a classification problem, rather than to predict and tune a direct factorization’s configuration. Production direct solvers such as MKL PARDISO [21] and MUMPS ship hand-written heuristic automatic modes rather than learned models. The structure used here, a learned performance regressor constrained by a deterministic guarantee and an out-of-range fallback, is closest in form to hardware branch prediction [16], where a learned predictor is paired with a mechanism that makes a misprediction cost re-execution and not correctness.

16 Limitations and future work

The tuner’s gains are aggregate: the geomean speedup is $1.62\times$, and part of it is within the per-matrix single-shot timing variance of about $\pm 20\%$, so the geomean rather than any single matrix is the signal. The model is trained on one machine and one corpus and does not transfer across hardware without retraining, in the same way a branch predictor is tuned per microarchitecture; a meta-tuner that runs the sweep, the training, and the guard-threshold calibration on a user’s own matrix set to emit a problem-class profile is planned. Time is optimized but not guaranteed; only memory is. The exact-path tuning stages that keep the factorization exact are now implemented (Sec. 7): an RCM/band ordering [25], an exposed threshold-pivot tolerance, a tunable equilibration strategy [26], static pivot-order reuse for fixed-pattern sequences via `pivot_u=0`, a parallel blocked multi-RHS solve, a right-looking schedule, and front-level parallelism of the trailing Schur update and subtraction. Each provides an a-priori memory estimate so the deterministic memory guarantee continues to hold, and each either enters the tuner grid or is caller-selected. What remains is a fuller two-dimensional (block-cyclic) factorization of the very largest top-of-tree fronts to push thread scaling past the current $\sim 3\times$, and the cross-hardware meta-tuner above. On the iterative side, the block/multi-RHS path is batched-independent rather than a shared-subspace block

Krylov method, and its within-cycle basis compaction does not yet compose with GCRO-DR recycling; a shared-subspace block solver and block recycling are the planned next step (issue #3).

17 Conclusion

RSLAB is a pure-Rust, scalar-generic sparse direct solver that reaches within a single-digit factor of MKL PARDISO on the SuiteSparse corpus and is faster and smaller than the open-source alternatives. Its analyze phase computes a per-path memory bound, a work and runtime estimate, and a thread recommendation, all as exact functions of structure; a learned tuner selects the solver configuration from that structure under a deterministic guard stack that keeps its memory below the untuned default. The numeric result is independent of the worker count and the resource planner is a pure function of its inputs, which makes batched, solver-in-the-loop execution reproducible and its cost computable in advance. When the factor is made approximate it becomes a preconditioner, and a matching Krylov layer — complex-symmetric short recurrences, flexible GMRES with a backward-stable orthogonalization split, batched multi-RHS block GMRES with within-cycle deflation, and GCRO-DR subspace recycling — closes the accuracy gap and turns the direct factor into the engine of a fast preconditioned and recycled iterative solver.

References

- [1] P. R. Amestoy, T. A. Davis, I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.* 17(4):886–905, 1996; and Algorithm 837: AMD. *ACM Trans. Math. Softw.* 30(3):381–388, 2004.
- [2] P. R. Amestoy. Recent progress in parallel multifrontal solvers and the approximate minimum fill (HAMF) ordering. Technical report / CERFACS, 1999.
- [3] E. Rothberg, S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal. Appl.* 19(3):682–695, 1998.
- [4] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* 10(2):345–363, 1973.
- [5] G. Karypis, V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20(1):359–392, 1998.
- [6] C. M. Fiduccia, R. M. Mattheyses. A linear-time heuristic for improving network partitions. *19th Design Automation Conference*, 175–181, 1982.
- [7] J. D. Hogg, E. Ovtchinnikov, J. A. Scott. A sparse symmetric indefinite direct solver for GPU architectures (SSIDS). *ACM Trans. Math. Softw.* 42(1):1–25, 2016.
- [8] C. Ashcraft, R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw.* 15(4):291–309, 1989.
- [9] J. R. Bunch, L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.* 31(137):163–179, 1977.
- [10] I. S. Duff, J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.* 9(3):302–325, 1983.
- [11] J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review* 34(1):82–109, 1992.
- [12] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw.* 12(3):249–264, 1986.
- [13] T. A. Davis. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.* 30(2):196–199, 2004.

- [14] M. Frigo, S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE* 93(2):216–231, 2005.
- [15] F. Pedregosa et al. Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.* 12:2825–2830, 2011.
- [16] D. A. Jiménez, C. Lin. Dynamic branch prediction with perceptrons. *7th Int. Symp. High-Performance Computer Architecture (HPCA)*, 197–206, 2001.
- [17] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, X. S. Li. GPTune: multitask learning for autotuning exascale applications. *PPoPP*, 234–246, 2021.
- [18] R. Vuduc, J. W. Demmel, K. A. Yelick. OSKI: a library of automatically tuned sparse matrix kernels. *J. Phys.: Conf. Ser.* 16:521–530, 2005.
- [19] E. R. Jessup, P. Motter, B. Norris, K. Sood. Performance-based numerical solver selection in the Lighthouse framework. *SIAM J. Sci. Comput.* 38(5):S750–S771, 2016.
- [20] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, D. Keyes. Application of machine learning in selecting sparse linear solvers. *Int. J. High Perf. Comput. Appl.*, 2006.
- [21] O. Schenk, K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.* 20(3):475–487, 2004.
- [22] S. Quinones. faer-rs: a linear algebra library for the Rust programming language. <https://github.com/sarah-quinones/faer-rs>, 2024.
- [23] X. S. Li. An overview of SuperLU: algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* 31(3):302–325, 2005.
- [24] T. A. Davis, Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38(1):1–25, 2011.
- [25] E. Cuthill, J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proc. 24th ACM National Conference*, 157–172, 1969.
- [26] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, 2001.
- [27] J.-C. Nédélec. Mixed finite elements in $\mathbb{R}^3$. *Numer. Math.* 35(3):315–341, 1980.
- [28] J.-M. Jin. *The Finite Element Method in Electromagnetics*, 3rd ed. Wiley–IEEE Press, 2014.
- [29] H. C. Elman, A. Ramage, D. J. Silvester. Algorithm 866: IFISS, a Matlab toolbox for modelling incompressible flow. *ACM Trans. Math. Softw.* 33(2):14, 2007.
- [30] M. Benzi, G. H. Golub, J. Liesen. Numerical solution of saddle point problems. *Acta Numerica* 14:1–137, 2005.
- [31] Y. Saad, M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* 7(3):856–869, 1986.
- [32] Y. Saad. A flexible inner–outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.* 14(2):461–469, 1993.
- [33] J. W. Daniel, W. B. Gragg, L. Kaufman, G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Math. Comp.* 30(136):772–795, 1976.
- [34] L. Giraud, J. Langou, M. Rozložník, J. van den Eshof. Rounding error analysis of the classical Gram–Schmidt orthogonalization process. *Numer. Math.* 101(1):87–100, 2005.
- [35] J. L. Barlow, A. Smoktunowicz. Reorthogonalized block classical Gram–Schmidt. *Numer. Math.* 123(3):395–423, 2013.
- [36] M. L. Parks, E. de Sturler, G. Mackey, D. D. Johnson, S. Maiti. Recycling Krylov subspaces for sequences of linear systems. *SIAM J. Sci. Comput.* 28(5):1651–1674, 2006.

- [37] R. B. Morgan. GMRES with deflated restarting. *SIAM J. Sci. Comput.* 24(1):20–37, 2002.
- [38] H. A. van der Vorst, J. B. M. Melissen. A Petrov–Galerkin type method for solving $Ax = b$, where A is symmetric complex. *IEEE Trans. Magn.* 26(2):706–708, 1990.
- [39] T. Sogabe, S.-L. Zhang. A COCR method for solving complex symmetric linear systems. *J. Comput. Appl. Math.* 199(2):297–303, 2007.