

RSLAB

A Pure-Rust Sparse Direct Solver: Algorithms, Implementation, and Benchmarks

Milan Rother

August 1, 2026

Abstract

RSLAB is a pure-Rust sparse direct solver with three factorization paths: a supernodal complex-symmetric $\mathbf{LDL}^\top$ with Bunch–Kaufman pivoting, an unsymmetric supernodal/multifrontal $\mathbf{LU}$ with threshold partial pivoting, and a KLU-style path (block triangular form + per-block Gilbert–Peierls $\mathbf{LU}$) for circuit-shaped systems. It is generic over the scalar type (`f64`, `f32`, and their complex counterparts) and links no external numeric library: the SIMD dense kernels, the fill-reducing orderings, and the factorizations are all Rust. The analyze phase computes a-priori peak-memory, work, and runtime predictors as exact functions of the matrix structure; the default configuration is a deterministic, model-free heuristic pick (adaptive ordering, an exact nested-dissection bakeoff, and a worker count from a one-time cached hardware calibration), with an optional learned tuner for class-specific specialization under a deterministic memory backstop; and the numeric factorization is bit-identical regardless of the worker count. A mixed-precision mode factors in single precision and refines to double against the original matrix with an explicit backward-error certificate. When the factor is made approximate (static-pivoted or thresholded) it becomes a preconditioner, and a matching Krylov layer (COCG/COCR, flexible GMRES, batched multi-RHS block GMRES with within-cycle deflation, and GCRO-DR subspace recycling) closes the accuracy gap. This report documents the core algorithms, their implementation, and the measured performance against `faer` and MKL PARDISO over a complete-distribution corpus.

1 Scope and architecture

RSLAB factorizes and solves $Ax = b$ for sparse A that is complex-symmetric (routed to $\mathbf{LDL}^\top$), general unsymmetric (routed to $\mathbf{LU}$), or circuit-shaped (routed to the KLU path). The primary target is the complex-symmetric systems of frequency-domain electromagnetic and FEM analysis (curl–curl Maxwell problems with lossy media or PML are complex-symmetric rather than Hermitian), with the KLU path serving the MNA/SPICE-class operators of circuit extraction. The implementation is stable Rust with no C/Fortran dependency and no BLAS/LAPACK/METIS linkage, and is exposed to Python through a thin PyO3 layer that mirrors the Rust surface (`rslab.ldlt/lu/klu/spsolve`).

All paths follow the standard three-phase flow. *Analyze* orders the matrix, builds the elimination structure, and produces the a-priori predictors; *factor* performs the numeric decomposition on the ordered, equilibrated matrix; *solve* runs the triangular substitutions. Analyze is separated from factor at the API level: a symbolic object (`LdltSymbolic`, `LuSymbolic`, `KluSymbolic`) is produced once and factors any number of matrices sharing its sparsity pattern, so a frequency sweep or Newton iteration pays the ordering and symbolic cost once. Across the corpus the numeric factor dominates the wall-clock, so the tuning and the parallelism target the factor. The factored solvers implement the crate’s `Preconditioner` and `Factorization` traits, which is how the direct and iterative layers compose.

2 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, sharing one quotient-graph elimination engine.

Approximate minimum degree. AMD [1] selects the pivot of least approximate external degree from degree-indexed bucket lists over a sliding index workspace with in-place garbage collection. The update performs aggressive element absorption, mass elimination, and supervariable detection by adjacency hashing; rows of degree above $\max(16, \lfloor 10\sqrt{n} \rfloor)$ are deferred to the tail as dense. AMD is the default for very large, very sparse patterns.

Approximate minimum fill. AMF (HAMF4 [2, 3]) replaces the degree score with an approximate fill score $\text{RMF}(i) = (\deg(i)(\deg(i) - 1 + 2\deg_{me}) - \text{WF}(i)) / (nv(i) + 1)$, where the working fill WF is accumulated per eliminated element in 64-bit integers. It is the default for $n \leq 10\,000$, staying within $1.10\times$ of the reference HAMF4 fill on a 183 277-matrix oracle suite.

Nested dissection. A multilevel recursive bisection [4, 5] driven by an explicit work stack (no recursion): coarsen, bisect, refine with Fiduccia–Mattheyses, uncoarsen with projection and refinement; the vertex separator is a minimum vertex cover via König’s theorem on the crossing-edge bipartite graph. Subgraphs below 200 vertices or splits with a $\geq 90\%$ side fall back to AMD leaves. Nested dissection is the default for $n > 10\,000$.

Dispatch. `OrderingMethod::Auto` routes by cheap structural predicates (very large and sparse $\rightarrow$ AMD; arrow/bordered patterns $\rightarrow$ AMF; otherwise the size rule). `AutoRace` instead runs the full symbolic factorization for each candidate and keeps the smallest exact fill, at about $4\times$ the single-pass analysis cost, the deterministic fallback the tuner uses out of distribution.

3 Symbolic analysis

From the ordered pattern RSLAB builds the elimination tree and detects fundamental supernodes, maximal column runs 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 [6] merge with their parent, using a column renumbering that re-postorders the tree so a desired parent–child merge becomes structurally adjacent. *Relaxed* amalgamation [7] further merges an adjacent child within a width and extra-row budget, trading a little explicit zero fill for higher-rank Schur updates. Column counts from a depth-first tree walk drive the factor-size estimate.

4 Numeric factorization

4.1 Equilibration

Before factoring, A is symmetrically equilibrated as $\hat{A} = DAD$ with $s_i = 1/\sqrt{\max_j |A_{ij}|}$ in one pass over the lower triangle; the real, symmetric, positive diagonal preserves complex symmetry and the pivot signs and tolerates zero diagonals (saddle-point systems). The **LU** path applies a two-sided $\hat{A} = D_r A D_c$; iterative Ruiz and MC64 matching-based scaling are selectable alternatives.

4.2 Supernodal left-looking LDL^T

The default symmetric path assembles one dense panel per supernode, updates it by every previously factored descendant (`cm`od: pull the descendant’s columns, apply its block-diagonal D , subtract the rank update, a scalar triple loop below a flop threshold, a SIMD GEMM above), then factors the fully-summed block in place (`cd`iv) by a blocked Bunch–Kaufman partial factorization [8] with panel width 64. For complex-symmetric A the kernel is the real `sytf2` control flow

with magnitudes $|z|$ and no conjugation; the pivot threshold is $\alpha = (1 + \sqrt{17})/8$, pivot selection is bounded to the supernode’s fully-summed rows, and a 2×2 block is rejected as singular when $|d_{11}d_{22} - d_{21}^2| < \varepsilon \max(|d_{11}|, |d_{21}|, |d_{22}|)^2$ with $\varepsilon = 10^{-14}$, bounding the element growth injected into the trailing update.

Transient memory is controlled by *panel freeing*: every supernode carries a reference count of the ancestors that will still update from it, and when a `cmod` decrements it to zero the panel is compacted into its final CSC fragment and the dense buffer released (acquire/release atomics make this safe under tree parallelism). The resident data is the compact factor plus only the live panels. Sibling subtrees factor concurrently in a scoped rayon pool of caller-set width; per-node GEMMs parallelize above their flop thresholds.

4.3 Multifrontal path

The alternative multifrontal schedule [9, 10] assembles a dense frontal matrix per supernode from its eliminated columns and its children’s contribution blocks (CBs), factors the fully-summed part, and passes the Schur complement up as its CB. A work-stealing tree recursion factors children in parallel; each CB is freed the moment it is extend-added into the parent, and where the CB stack is large children are reordered by Liu’s (peak – *cb*) rule [11], a scheduling hint that changes neither numbering nor fill. The front is denser and more BLAS-3-rich than a left-looking panel but carries a higher transient peak, which is why both schedules ship and the choice is per matrix.

4.4 Unsymmetric LU

General matrices reuse the symmetric analysis machinery on the symmetrized pattern $A \cup A^T$ and ship both numeric schedules. The default supernodal left-looking **LU** uses UMFPACK-style [12] threshold partial pivoting: the diagonal candidate is kept when $|a_{jj}| \geq u |a_{\text{colmax}}|$ with tunable $u = \text{pivot_u}$ (default 0.1); $u = 1$ recovers full partial pivoting, $u = 0$ keeps the natural order and skips the pivot search (the static fast path for fixed-pattern sequences). The multifrontal **LU** factors each front by a blocked right-looking `getrf` with unrestricted partial pivoting over the fully-summed rows. L is stored CSC (unit lower), U CSR; the column permutation is the fill-reducing ordering and a separate row permutation records the interchanges.

4.5 KLU path for circuit-shaped matrices

Circuit-class operators (MNA/SPICE: extremely sparse, unsymmetric, near-triangularizable) defeat supernodal methods: their irreducible blocks are far too small for BLAS-3 fronts to pay off. RSLAB therefore ships a third path following SuiteSparse KLU [14], reimplemented in pure Rust.

Analyze permutes to block *upper* triangular form (BTF): an MC21-style maximum transversal (augmenting-path matching with per-column lookahead, iterative) pairs every column with a row holding a structural nonzero (an incomplete matching proves the matrix *structurally* singular before any numeric work) and Tarjan’s SCC algorithm on the matched digraph, emitted in reverse topological order, yields the symmetric permutation under which every entry outside the irreducible diagonal blocks lies above them. AMD then orders each block on its symmetrized pattern. Fill is confined to the diagonal blocks; the off-block entries are never factored and enter only the block back-substitution.

Factor runs a left-looking Gilbert–Peierls **LU** per block: for each column a depth-first reach over the pattern of the L columns factored so far yields the fill pattern in topological order, so the numeric work is proportional to the flop count. Threshold partial pivoting prefers the (structurally nonzero) diagonal unless it falls below 10^{-3} of the column maximum; row-max scaling equilibrates the rows first. The path is strictly sequential and allocation-deterministic, so its results are **bit-identical across runs and thread counts**; it doubles as the determinism arbiter for the parallel paths.

Refactor is the path’s distinctive fast lane: for a new value set on the *same* pattern (frequency sweep, time stepping, Newton), the stored pattern and pivot sequence are replayed with no symbolic work and no pivot search; a changed pattern is detected and rejected exactly, and a frozen pivot that becomes zero fails cleanly so the caller can re-factor with pivoting. The symbolic object also carries an exact a-priori fill/flop estimator (a pattern-only Gilbert–Peierls pass under the diagonal-pivot assumption, computed lazily and cached), mirroring the estimators of the other two paths.

4.6 Static pivoting and preconditioner mode

The `ZeroPivotAction` policy governs near-singular pivots on the $\mathbf{LDL}^\top/\mathbf{LU}$ paths: `Fail` (default) returns a rank-deficiency error, `PerturbToEps` lifts any sub-floor pivot to the floor while preserving its phase, so the factorization satisfies $\mathbf{LDL}^\top = A + \Delta$ with bounded Δ and never fails. Together with threshold dropping (`drop_tol`: incomplete factors) and `f32` mixed precision, this never-fail factor is the preconditioner the Krylov layer consumes. The KLU path never perturbs: a vanishing pivot is a hard, attributed error.

5 A-priori predictors

The analyze phase computes a set of predictors as *exact functions of the symbolic structure* and the scalar size v , before any numeric work, the basis of the memory guarantee, the heuristic configuration pick, the thread policy, and the resource planner of Sec. 7.

Per-path peak memory. The factor size is `factor_nnz` · ($v + 8$), a structural upper bound (cancellation can only lower it). The transient peak is bounded separately per schedule. For the left-looking path, a reference-count simulation of the panel-freeing schedule walks the postorder, adds each panel as it is assembled, frees it when its `refcount` reaches zero, and records the running maximum of live panels plus already-compacted factor, the floor the solver actually operates at; a conservative whole-run bound adds all panels, the input, and a scratch term $\frac{1}{4}(\text{panels} + \text{factor}) + 32$ MB for the per-thread `cmod/cdiv` buffers. For the multifrontal path a level-parallel model takes, per assembly-tree level, the front sizes $\sum \text{nrow}^2 v$ plus that level’s child contribution blocks, times $\frac{7}{5}$ for work-stealing overlap. Over the corpus both bounds hold at an estimate/measured ratio of ≈ 1.3 in geomean (Fig. 1); the panel-freeing floor is the tighter quantity the tuner’s memory veto uses. The KLU path carries the same contract: a pattern-only Gilbert–Peierls pass (Sec. 4.5) gives its fill and flops exactly under diagonal pivoting.

Work, runtime, and threads. The geometric work `factor_flops` = $\sum_s \text{nrow}_s^2 \text{ncol}_s$ is a type-independent proxy; 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 first term is an Amdahl floor from the assembly tree’s longest leaf-to-root chain, so a critical-path-bound matrix does not benefit from more workers. The single-thread rate and parallel speedup come from a one-time hardware calibration cached by machine fingerprint; a learned additive residual on the speedup curve (ridge regression on dimensionless structure features, dominated by `critical_path/factor_flops`) cuts the held-out speedup error by $\sim 26\%$. A thread-count predictor caps thin/narrow problems at two workers and small ones at four, landing within $\sim 10\%$ of the per-matrix optimum in geomean (a fixed two-worker budget is off by $\sim 50\%$).

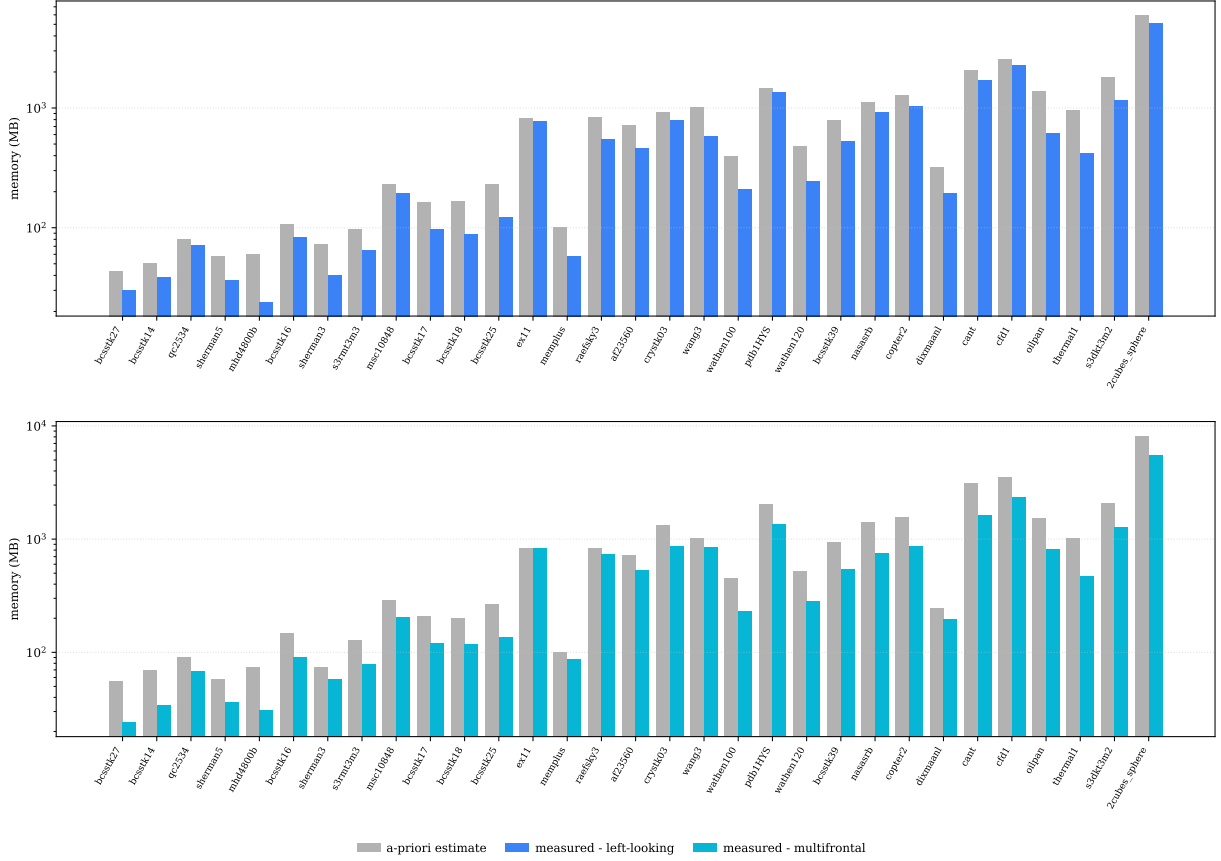


Figure 1: A-priori memory estimate (conservative bound and panel-freeing floor) vs. the measured peak, per RSLAB path. The estimate does not under-predict.

6 Determinism and the learned tuner

Three determinism properties hold. The numeric factor is bit-identical regardless of worker count: parallel subtrees and GEMMs write disjoint blocks and every reduction sums in a fixed order, so the thread count is a performance parameter only (the parallel multi-RHS solve is likewise bit-identical to serial). The analyze-phase choices are deterministic functions of the pattern. And the resource planner is a pure function of its inputs. The pivot sequence is value-dependent; `pivot_u = 0` (or the KLU refactor) freezes it across same-pattern refactorizations.

The shipped default configuration is a deterministic, model-free heuristic: the adaptive ordering choice, the measured default kernel knobs, an exact nested-dissection bakeoff on large systems (re-analyze with `MetisND`, adopt only on a clear predicted-flops win with no regression in exact fill or transient peak), and, after a one-time cached hardware calibration, a critical-path-aware worker count from the calibrated cost model. Every input to the pick is an exact a-priori quantity; nothing is modeled.

The mixed-precision solvers (`MixedLdltsolver/MixedLuSolver`) factor $\text{cast}_{32}(A)$ at half the memory and solve through an explicit refinement ladder (plain iterative refinement escalating to GMRES-IR with the single-precision factor as the preconditioner), reporting the achieved normwise backward error and whether the double-precision certificate holds (Carson–Higham multi-precision IR); on the reference class the single-precision factor runs *1.64 times* and certifies at the roundoff level after two refinement steps.

For specializing to a *specific problem class on specific hardware* an optional learned tuner remains: one MLP per path predicting $(\log t, \log m)$ per candidate configuration, trained offline [15] on a complete-distribution corpus (curl–curl Maxwell, Stokes/KKT saddle-point, convection–diffusion across the grid–Péclet range, plus SuiteSparse), retrainable into a runtime `tuner_profile.json`

artifact. Its picks are constrained by a deterministic guard stack (a re-analysis check that the pick’s exact fill, flops, and memory floor stay within $1.02\times/1.05\times/1.0\times$ of the default’s, and a minimum-improvement threshold), so peak memory is *guaranteed* never to exceed the default’s.

7 Solver-in-the-loop and resource governance

RSLAB is built to run *inside* an outer loop (a frequency sweep, adaptive mesh refinement, a Newton or time-stepping iteration, many concurrent extractions) rather than as a one-shot black box, and three mechanisms carry that model. First, analyze and factor are separate phases: a symbolic object factors any number of matrices sharing its pattern, so a same-pattern sequence pays the ordering and symbolic cost once (and on the KLU path the numeric refactor also skips the pivot search). Second, every factorization runs in a *scoped* thread pool of caller-set width, so many concurrent solves coexist without oversubscribing the machine, and a factored preconditioner hands its thread policy to the Krylov layer so factor and solve share one concurrency budget. Third, 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 fits/does-not-fit decision) as a pure function of its inputs. Because the memory estimate of Sec. 5 is an upper bound rather than a sample, a scheduler can pack concurrent solves against a memory ceiling without extra safety margin, fail fast before allocating, or switch to an approximate factor when the exact one cannot fit; and because per-solve peaks and runtimes are predicted, the aggregate footprint and makespan of a batch are computable before the batch runs. Per-call diagnostics (stage timings, fill, the attached estimate) close the loop after the fact, estimate against measured.

8 Preconditioned and recycled Krylov layer

When the factor is approximate, accuracy is recovered by iteration. The iteration and the preconditioner are decoupled by two traits: `LinearOperator` supplies $y \leftarrow Ax$ and a block $Y \leftarrow AX$ (an explicit matrix, or matrix-free; an FMM/MoM operator with RSLAB factoring only the sparse near-field), and `Preconditioner` supplies $z \leftarrow M^{-1}r$ and a block apply that a factored solver implements through the parallel `solve_many` (a 8 to 19 $\times$ wide-RHS speedup the block Krylov methods inherit).

Complex-symmetric short recurrences. For $A = A^\top$ the methods of choice are COCG [26] and COCR [27]: structurally CG/CR but with every inner product in the unconjugated bilinear form $x^\top y$. A zero bilinear denominator is reported as a deterministic breakdown with the best iterate, never a division by zero.

Flexible GMRES. For general operators, restarted GMRES [19] in the flexible variant [20]: the preconditioned Arnoldi vectors $z_j = M^{-1}v_j$ are kept as a second basis and $x \leftarrow x + Zy$, which saves one M^{-1} solve per restart cycle and admits a varying preconditioner. Orthogonalization is MGS with a conditional DGKS [21] second pass. Each cycle measures the *true* residual $\|b - Ax\|$, the only reliable stop test on ill-conditioned near-field operators, and a Hessenberg-diagonal guard truncates the least-squares solve to its well-conditioned prefix, so a rank-deficient cycle reports non-convergence instead of injecting NaN.

Block GMRES. The multi-RHS solver drives s right-hand sides in lockstep so matvec and preconditioner are issued once per step as block applies. Each right-hand side keeps its own Arnoldi basis (batched-independent; a shared-subspace block Krylov method is future work), which lets a column *deflate* out the moment it converges including mid-cycle, compacting the batched applies to the still-active width. The panel orthogonalization is block CGS2 [22, 23]: two panel-wide sweeps, BLAS-3 intensity, parallelized in fixed row chunks and therefore bit-identical across thread counts. Measured on preconditioned convection–diffusion ($n = 40\,000$, complex),

BCGS2 scales to $\sim 2.2\times$ at 12 cores where the per-RHS MGS path is flat-to-negative, and per-RHS cost stays near-flat in the block width.

GCRO-DR subspace recycling. For sequences of related solves, GCRO-DR [24, 25] retains the k harmonic-Ritz vectors of smallest value, the near-invariant subspace that governs restarted-GMRES stagnation, and deflates them across restarts within a solve, and recycles them across solves through an opaque handle refreshed in place. The subspace only *shapes* the Krylov space: the outer true-residual test stays authoritative, every recycle step degrades gracefully to plain FGMRES, and $C = \tilde{A}U$ is recomputed each solve so the split stays exact when A or M drift. On a stagnation-spectrum sequence of 8 solves ($n = 20\,000$), recycling cuts the cumulative iteration count $6.4\times$ versus cold starts ($2.9\times$ on the first solve alone) for a $1.5\times$ wall-clock reduction. Warm starts (x_0), an adaptive restart length that keeps the up-front Arnoldi basis under a memory budget, and f32 mixed-precision preconditioning (`LowPrecisionPreconditioner` / `LowPrecisionLu`) complete the layer.

9 Benchmarks

All cross-solver figures come from the `bench_suite` engine over a complete-distribution corpus: structured-grid generators (curl-curl Maxwell, shifted Helmholtz, Stokes/KKT saddle-point, convection-diffusion over the grid-Péclet range, BEM/MoM near-field kernels) plus the complex SuiteSparse [18] matrices, 8k–125k DOFs, `Complex<f64>`, measured in one run on a quiet 12-core machine, so the cross-solver ratios carry no run-to-run drift. RSLAB runs its shipped default, the deterministic heuristic pick (adaptive ordering, exact ND bakeoff, calibrated worker count); each path is compared on its own class against its own PARDISO mtype [16] and faer [17].

9.1 Scaling and accuracy

Figures 2 and 3 plot factor time and peak memory against nnz with per-solver power-law fits; Table 1 gives the head-to-head geomean ratios. RSLAB sits between the two on both paths: faster than the pure-Rust faer, moderately behind the hand-optimized MKL PARDISO ($5.6\times$ sym / $5.1\times$ unsym). The heuristic pick’s win over the fixed default is visible as a gap that widens with problem size (largest on the big matrices where a mispicked ordering costs most) and on the symmetric path never comes at a memory cost. On accuracy (Fig. 4), 24 of 31 corpus matrices factor to a relative residual below 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. The determinism and equivalence properties were validated over 180 SuiteSparse matrices: right-looking vs. multifrontal, both emit modes, parallel vs. serial front subtraction, and the 32-bit compressed factor are all bit-identical.

RSLAB vs	$\mathbf{LDL}^\top$ (sym)	$\mathbf{LU}$ (unsym)
MKL PARDISO: factor time	$5.6\times$ slower	$5.1\times$ slower
MKL PARDISO: peak memory	$2.8\times$ more	$3.6\times$ more
faer $\mathbf{LU}^\dagger$: factor time	$6.7\times$ faster	$2.7\times$ faster
faer $\mathbf{LU}^\dagger$: peak memory	$1.8\times$ less	$1.4\times$ more
fixed default cfg: factor time	$1.84\times$ faster	$1.49\times$ faster
fixed default cfg: peak memory	$0.90\times$ (less)	$1.11\times$ more

Table 1: Head-to-head geomean ratios (63 sizes per path, 1k–110k DOFs), each path on its own class, over the matrices both solvers factor to below 0.1 residual. The last rows are the shipped heuristic pick against the fixed RSLAB default configuration. Unsym memory ratios above 1 are the worker-count trade: the calibrated pick runs more workers than the capped fixed config, raising the transient panel peak. † faer has no symmetric path (it factors symmetric matrices as $\mathbf{LU}$), so its $\mathbf{LDL}^\top$ gap is structurally largest; it also OOMs on the largest matrices, so its head-to-head is a conservative floor.

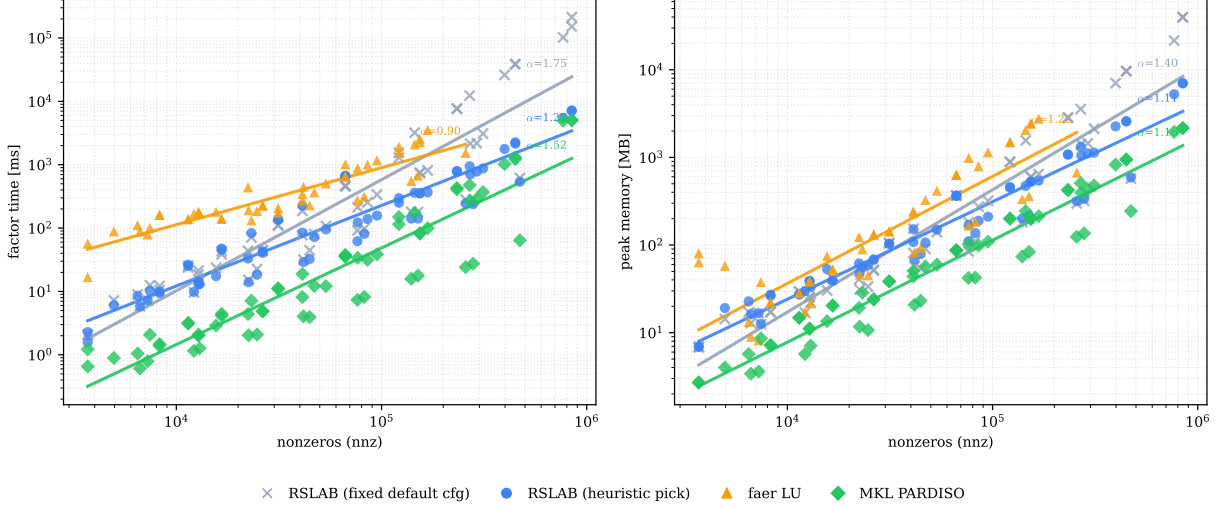


Figure 2: $\text{LDL}^\top$ path (symmetric, PARDISO mtype 6): factor wall-clock (left) and peak memory (right) vs. problem size: RSLAB vs. faer vs. MKL PARDISO, with power-law fits.

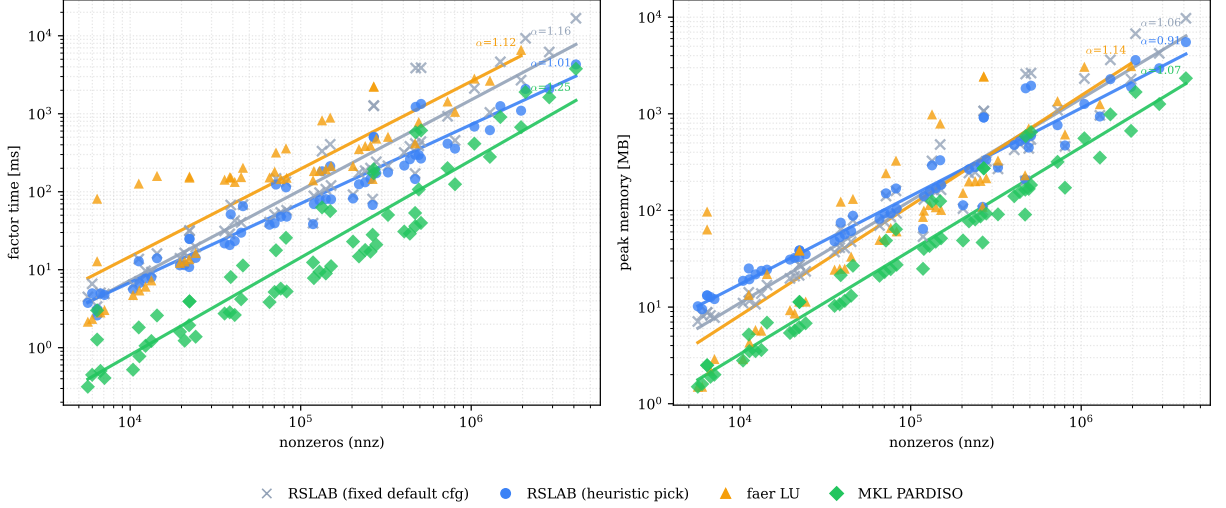


Figure 3: LU path (unsymmetric, PARDISO mtype 13): factor wall-clock (left) and peak memory (right) vs. problem size. The fixed RSLAB default configuration (gray) accompanies the heuristic pick (blue); on time the pick scales flatter than PARDISO ($\alpha \approx 1.01$ vs. 1.25).

9.2 Apple Silicon: RSLAB vs. Accelerate sparse solvers

A second, independent same-run measurement of all three solver paths on an Apple M3 (8 cores, 16 GB, macOS 15.6) against Apple Accelerate’s sparse direct solvers, which gained complex support and a sparse LU in macOS 15.5. RSLAB runs its shipped default (heuristic pick, hardware-calibrated on this machine); Accelerate runs its vendor-best configuration per class: Cholesky-first with an $\text{LDL}^\top$ fallback for real-symmetric matrices, its new sparse LU (TPP, `pivotTolerance` 0.01) for unsymmetric and circuit-shaped ones, and, because its complex $\text{LDL}^\top$ is Hermitian-only (verified empirically: the lower triangle is mirrored conjugated), LU on the *full* matrix for the complex-symmetric families, the same structural handicap faer carries. Each matrix gets an AMD-vs-Metis ordering bakeoff (`SparseOrderDefault` is plain AMD, up to $4.4\times$ the Metis fill on the large EM/FEM systems); the bakeoff cost is charged to Accelerate’s analyze time, mirroring RSLAB’s exact ND bakeoff. Peak memory uses the same live-bytes semantics as the Rust solvers, via instrumented allocation callbacks.

Figures 5–7 show the per-path scaling; Table 2 the geomean ratios. On its home silicon the

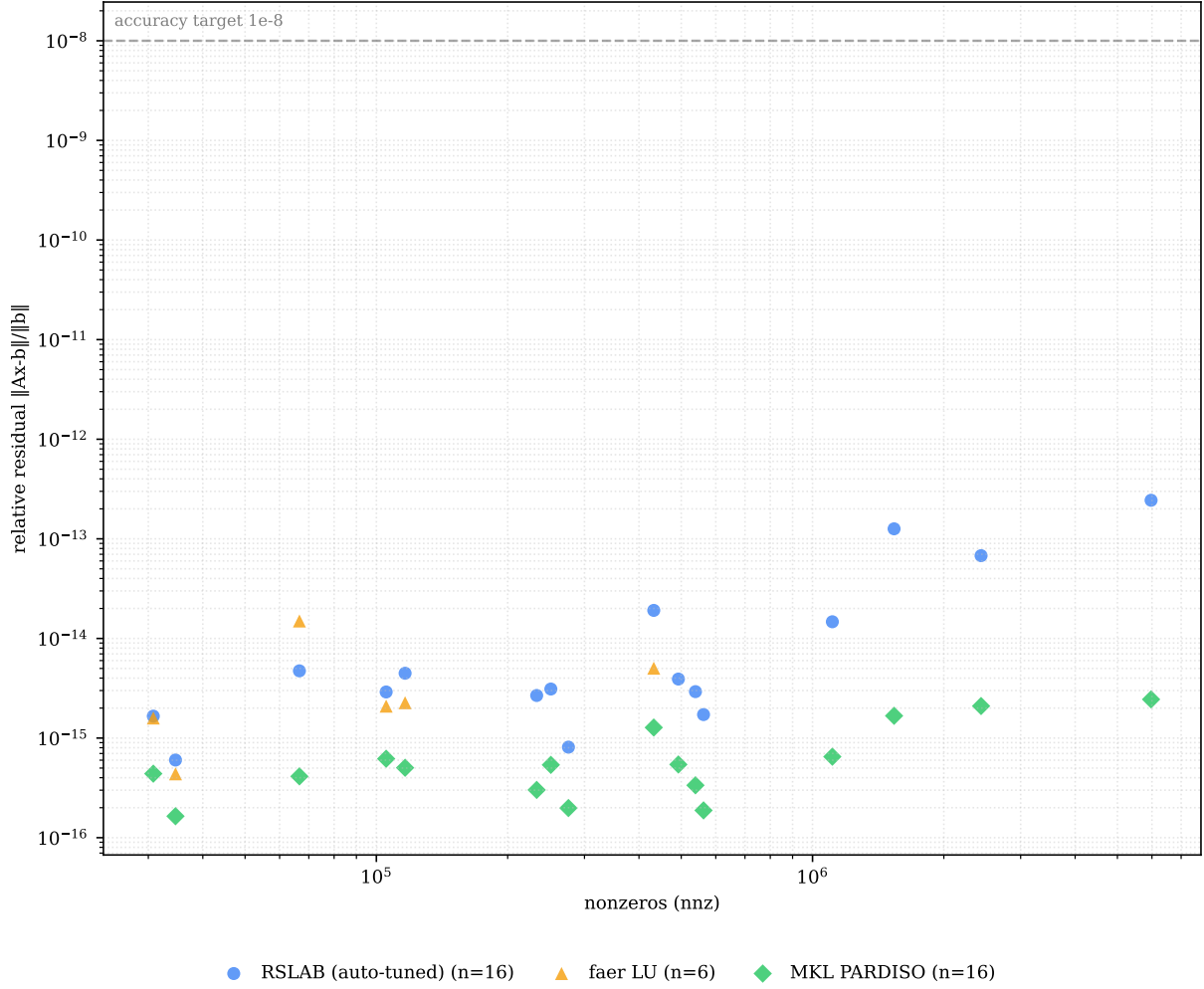


Figure 4: Relative residual $\|Ax - b\|/\|b\|$ across the corpus, per solver.

vendor library wins most of the distribution: its AMX-backed kernels take the small and mid sizes outright ($0.23\times$ – $0.77\times$ RSLAB’s factor time in geomean). The structural story survives at scale: above $\sim 3 \cdot 10^5$ nonzeros on the complex-symmetric systems, RSLAB’s $A = A^T$ exploitation turns the tables (Accelerate 1.3 – $1.35\times$ slower there; its time scales at $\alpha \approx 1.67$ vs. 1.38). The circuit comparison is one-shot analyze+factor+solve against the strictly sequential KLU default; KLU’s numeric-only refactor sweeps and the parallel per-block factor are not exercised. On accuracy, Accelerate reaches 10^{-8} on 25 of 36 corpus matrices, but on five hard indefinite/ill-scaled ones (stokes64, bratu3d, cont-201, rim, lhr34) it returns NaN or garbage *with an OK status*, the same silent-degradation mode as faer, where RSLAB declines cleanly.

Accelerate vs. RSLAB shipped path	factor time	peak memory	matrices
sym ($\mathbf{LDL}^T$ path)	$0.64\times$	$0.58\times$	49
unsym ($\mathbf{LU}$ path)	$0.27\times$	$0.29\times$	60
circuit (vs. KLU)	$0.77\times$	$0.75\times$	15
SuiteSparse corpus	$0.23\times$	$0.45\times$	30

Table 2: Apple M3 head-to-head geomean ratios (Accelerate / RSLAB), over the matrices both solve to below 0.1 residual; ratios below 1 mean Accelerate is faster/leaner. One run, one machine, live-bytes memory on both sides.

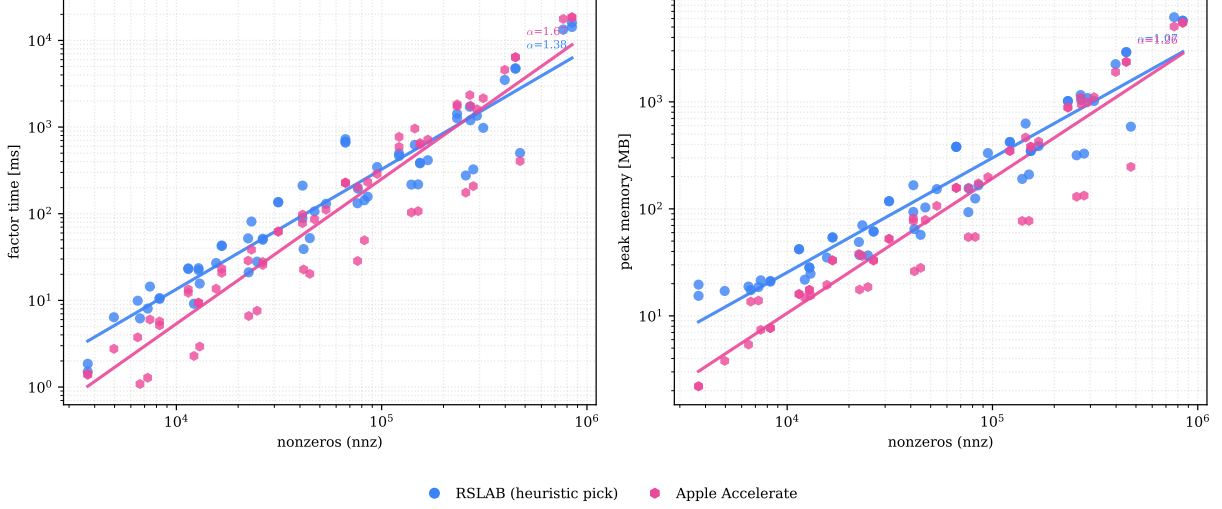


Figure 5: Apple M3: LDL^T path (complex-symmetric + real-symmetric families): factor wall-clock (left) and live-bytes peak (right) vs. problem size, RSLAB shipped pick vs. Apple Accelerate (vendor-best configuration), with power-law fits.

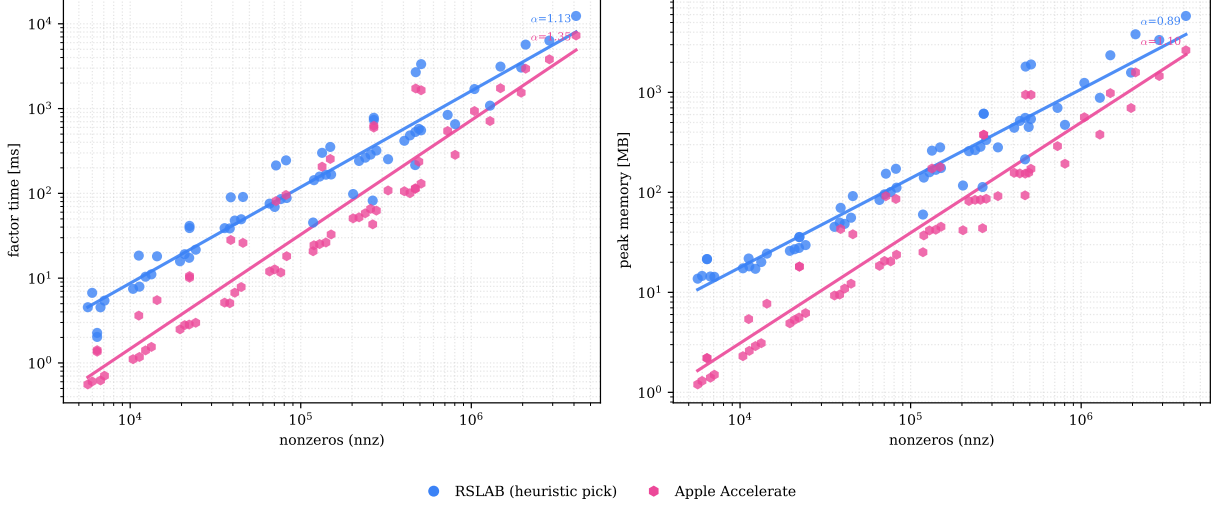


Figure 6: Apple M3: LU path (unsymmetric families). Accelerate’s macOS-15.5 sparse LU scales at $\alpha \approx 1.36$ vs. RSLAB’s 1.15 on time.

9.3 KLU path on circuit-shaped matrices

Table 3 benchmarks the KLU path against the multifrontal LU (defaults) on MNA-like matrices: $\sim 4\text{--}5$ nnz/column, unsymmetric, column-diagonally dominant, cascaded stages giving a genuinely reducible BTF structure (`cargo bench --bench klu_circuit`, Apple M3). The KLU factor is 5 to $12\times$ faster with 1.7 to $5.7\times$ less fill, the gap widening with size as the multifrontal fronts grow; the numeric-only refactor runs $\sim 2\times$ faster still, so a 20-point sweep (refactor+solve vs. factor+solve) is 10 to $40\times$ faster end to end. Both solvers reach machine-precision residuals ($\sim 10^{-15}$) on every size. An opt-in parallel per-block factor (independent BTF blocks on a scoped pool, bit-identical to the sequential result) adds another 2.2 to $3.1\times$.

Against the reference C implementation, SuiteSparse KLU [14] (loaded at runtime when installed), the analyze pipeline is at parity: identical BTF block counts and fill within 1.5% (RSLAB slightly less), confirming the maximum transversal, Tarjan SCC, and per-block AMD match the reference. On the numeric kernel (32-bit index streams, packed DFS marks, Eisenstat-Liu symmetric pruning [13], and a recorded refactor scatter program), the sequential factor runs

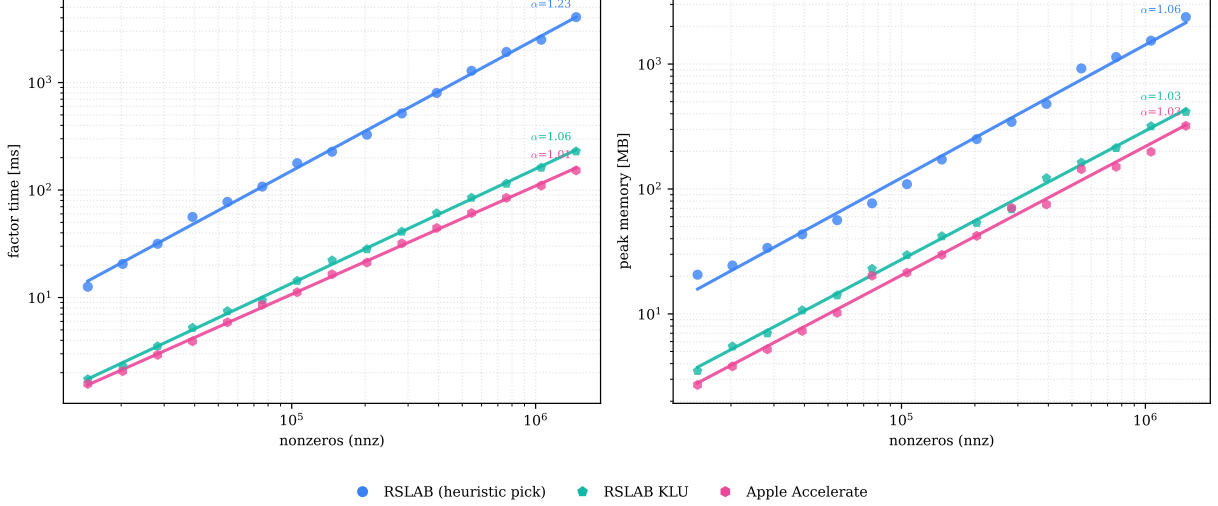


Figure 7: Apple M3: circuit-shaped matrices (complex MNA, $G + j\omega C$). RSLAB KLU vs. the multifrontal LU default vs. Accelerate LU; one-shot factor, so KLU’s refactor-driven sweep niche is not exercised here.

n	nnz	KLU			blocks	multifrontal LU		
		factor	refactor	fill		factor	fill	sweep ratio
2k	15k	1.7 ms	0.8 ms	79k	8	8.1 ms	132k	9.8×
10k	73k	8.0 ms	4.1 ms	439k	16	54.7 ms	1.03M	12.5×
50k	366k	45.4 ms	24.3 ms	2.32M	32	304 ms	9.21M	13.9×
200k	1.47M	179 ms	96 ms	9.17M	64	2.16 s	52.2M	40×

Table 3: KLU vs. multifrontal LU on MNA-like circuit matrices (Apple M3). “Sweep ratio” is the wall-clock ratio of 20 same-pattern value sets solved by KLU refactor+solve vs. multifrontal factor+solve. Fill counts stored factor entries.

within 1.2 to 1.6 $\times$ of the C code (e.g. 170 ms vs. 135 ms at $n = 200k$) and the refactor within 1.15 to 1.4 $\times$. The opt-in bit-identical parallel per-block execution (factor and refactor; two-phase spliced output) is 1.6 to 2.7 $\times$ faster than SuiteSparse KLU on factor and 2.1 to 3.4 $\times$ on refactor (51 ms / 24 ms at $n = 200k$), putting the 20-point sweep $\sim 2.4\times$ ahead end to end. Both reach $\sim 10^{-15}$ residuals.

9.4 Iterative layer

The Krylov results appear with their concepts in Sec. 8: block-CGS2 lifts multi-RHS strong scaling to $\sim 2.2\times$ at 12 cores where per-RHS MGS is flat; within-cycle deflation cuts the batched operator column-applies to 0.66 $\times$ the full-width bound at $s = 16$; GCRO-DR recycling cuts the cross-solve iteration total 6.4 $\times$ on a stagnating sequence; and the incomplete-factor sweet spot at $\text{drop_tol} = 10^{-2}$ halves the factor memory at a total wall time within a few percent of the exact direct solve. FGMRES’s flexible-basis update saves exactly one preconditioner solve per restart cycle, confirmed by counting.

10 Conclusion

RSLAB is a pure-Rust, scalar-generic sparse direct solver with three paths matched to their operator classes: a supernodal complex-symmetric $\mathbf{LDL}^T$, a supernodal/multifrontal threshold-pivoted LU, and a sequential, bit-deterministic KLU path with a numeric-only refactor for fixed-pattern sweeps. It reaches within a single-digit factor of MKL PARDISO while remaining dependency-free,

is faster and smaller than the open-source alternatives, and its analyze phase predicts memory, work, and runtime exactly enough that a learned tuner can optimize configuration under a hard memory guarantee. When the factor is made approximate it becomes a preconditioner, and the matching Krylov layer (COCG/COCR, flexible GMRES, deflating block GMRES, GCRO-DR recycling) turns the direct factor into the engine of a fast preconditioned 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.
- [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] 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.
- [7] C. Ashcraft, R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw.* 15(4):291–309, 1989.
- [8] J. R. Bunch, L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.* 31(137):163–179, 1977.
- [9] I. S. Duff, J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.* 9(3):302–325, 1983.
- [10] J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review* 34(1):82–109, 1992.
- [11] 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.
- [12] T. A. Davis. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.* 30(2):196–199, 2004.
- [13] S. C. Eisenstat, J. W. H. Liu. Exploiting structural symmetry in unsymmetric sparse symbolic factorization. *SIAM J. Matrix Anal. Appl.*, 13(1), 1992.
- [14] T. A. Davis, E. Palamadai Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.* 37(3):36, 2010.
- [15] F. Pedregosa et al. Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.* 12:2825–2830, 2011.
- [16] O. Schenk, K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.* 20(3):475–487, 2004.
- [17] S. Quinones. faer-rs: a linear algebra library for the Rust programming language. <https://github.com/sarah-quinones/faer-rs>, 2024.
- [18] T. A. Davis, Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38(1):1–25, 2011.
- [19] 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.
- [20] Y. Saad. A flexible inner–outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.* 14(2):461–469, 1993.
- [21] 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.
- [22] 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.
- [23] J. L. Barlow, A. Smoktunowicz. Reorthogonalized block classical Gram–Schmidt. *Numer. Math.* 123(3):395–423, 2013.
- [24] 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.
- [25] R. B. Morgan. GMRES with deflated restarting. *SIAM J. Sci. Comput.* 24(1):20–37, 2002.

- [26] 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.
- [27] T. Sogabe, S.-L. Zhang. A COCR method for solving complex symmetric linear systems. *J. Comput. Appl. Math.* 199(2):297–303, 2007.