AC(K): ROBUST SOLUTION OF LAPLACIAN EQUATIONS BY RANDOMIZED APPROXIMATE CHOLESKY FACTORIZATION *

YUAN GAO†, RASMUS KYNG‡, AND DANIEL A. SPIELMAN §

Abstract. We introduce a new Approximate Cholesky factorization algorithm and software for solving linear equations in symmetric diagonally dominant matrices with nonpositive off-diagonal entries (SDDM matrices), including Laplacian matrices. We use preconditioned conjugate gradient to solve these systems. Our preconditioner is a variant of the Approximate Cholesky factorization of Kyng and Sachdeva (FOCS 2016). Our factorization approach is simple: we eliminate matrix rows/columns one at a time, and use sampling to update the remaining entries of the matrix, approximating complete Cholesky factorization. Unlike Kyng-Sachdeva, we do not prove theoretical guarantees about our algorithm, but we observe experimentally that our algorithm runs much faster. Our algorithm broadly resembles a common approach of using Incomplete Cholesky factorization as a preconditioner in conjugate gradient (ICCG). Unlike earlier approaches such as Kyng-Sachdeva and Incomplete Cholesky variants, our sampled entries always maintain a connected support graph on the neighbors of the eliminated variable. We implement our solver algorithm in Julia and experimentally evaluate its performance when using 1 or 2 samples for each original entry. We compare our single-threaded performance to that of state-of-the-art solvers. We evaluate performance across many problem classes, including all large SDDM matrices from the SuiteSparse collection, and a broad array of programmatically generated instances, with tests ranging up to 200 million nonzeros per system of linear equations. We hope that others will adopt this suite of tests as a benchmark, which we refer to as SDDM2023. Our experiments suggest that our solvers are much more robust than existing solvers for SDDM linear equations, while retaining good performance across all instances. Our code is public and we detail precisely the tests we run and provide a tutorial on how to replicate them.

Key words. Laplacian linear equation solver, SDDM linear equation solver, approximate elimination, approximate Cholesky factorization, preconditioned conjugate gradient.

MSC codes. 65F08

---

*The authors are listed alphabetically. The research leading to these results has received funding from the grant "Algorithms and complexity for high-accuracy flows and convex optimization" (no. 200021 204787) of the Swiss National Science Foundation. It was also supported in part by NSF Grant CCF-1562041, ONR Award N00014-16-2374, and a Simons Investigator Award to Daniel Spielman.

†CISPA, part of the work was conducted as a master's thesis at ETH Zurich; part of this work was also conducted while the author was at MPI for Informatics. (yuan.gao@cispa.de)

‡ETH Zurich (kyng@inf.ethz.ch)

§Yale University (spielman@cs.yale.edu)

1. Introduction.

Linear equations in symmetric diagonally dominant matrices with nonpositive off-diagonals (SDDM matrices) appear in many applications, including solvers for discretized scalar elliptic partial differential equations [63, 40, 22, 10, 61], many problems in computer vision and computer graphics [44], and in machine learning [68]. They are essentially equivalent to Laplacian matrices associated with graphs. To solve the broader class of symmetric diagonally dominant (SDD) linear equations, one can use a reduction of Gremban [32] to convert the problem of solving an $n \times n$ SDD linear equation into one of solving a $2n+1 \times 2n+1$ Laplacian linear equation, but in this work we do not test the performance of our algorithm on instances coming from general SDD matrices.

We introduce a new algorithm, Approximate Cholesky (AC), for solving SDDM linear equations, as well as a single-threaded implementation of the algorithm and several variations in Julia. Our solver is a variant of the Kyng-Sachdeva solver [46], but introduces a new sampling rule for computing an approximate Cholesky factor.

1.1. Related work.

There is a long history of practical methods for solving SDDM linear equations. The most popular approaches broadly fall into two categories: methods based on incomplete factorizations [58], starting with Incomplete Cholesky Factorization [50], and multigrid methods [12, 14]. In both cases, these methods typically combine an iterative solver such as preconditioned conjugate gradient with one of the preceding methods used as a preconditioner. There has been tremendous theoretical success in developing SDDM linear equation solvers with running times asymptotically nearly-linear in the number of nonzeros of the matrix starting with the work of Spielman and Teng [60]. In the following, we review the literature on practical and theoretical algorithms for solving SDDM linear equations.

Incomplete factorizations. Incomplete Cholesky factorizations were developed over a long sequence of works [64, 18, 54, 27, 50] as a method for solving linear equations arising from discretized elliptic partial differential equations. Incomplete factorizations are obtained by omitting most entries that appear during Cholesky factorization or LU-decomposition [40], and the factorizations they produce can be shown to approximate the original matrix. Meijerink and van der Vorst [50] suggested using incomplete Cholesky factorizations as preconditioners for conjugate gradient. This overall approach to solving linear systems is known as ICCG (Incomplete Cholesky conjugate gradient). When they were introduced, they provided the fastest solutions to the discrete approximations of two-dimensional elliptic PDEs. A popular implementation of this method is MATLAB's `ichol`. Incomplete LU decomposition has been a major subject of study since. One significant advance was the introduction of level-of-fill approaches, where ILU(k) allows nonzero entries in the LU-factors when the input matrix to the $k+1$th power has a nonzero (ignoring cancellations) [66, 58]. Another major advance was the development of threshold-based ILU, known as ILUT, an approach to incomplete factorization that drops entries based on comparison with a value threshold [57]. Finally, we highlight that versions of Incomplete Cholesky or ILU factorization approaches exist that modify the diagonal entries of the remaining matrix when dropping off-diagonal entries, usually known as MILU [27, 28, 33, 58]. This diagonal adjustment is somewhat similar to approaches motivated by theoretical computer science, which adjust diagonal and off-diagonal entries together when approximating a Laplacian matrix [60, 59] or its Cholesky factor [46].

Multigrid methods. The Multigrid algorithm [15] provided a significant improvement in the solution of systems of equations arising from discrete approximations of elliptic PDEs, and its generalizations provide the fastest solvers for many families of linear equations [17]. Algebraic Multigrid (AMG) builds upon the principles of Multigrid methods but has a wider range of applications as it does not require the problem to have a simple underlying geometry, making it applicable to general symmetric M-matrices [56], a family that includes SDDM matrices. Brandt designed an AMG solver applicable to Dirac equations [13]. For linear equations in graph Laplacians, Livne and Brandt presented a variant of AMG solver, called the Lean Algebraic Multigrid (LAMG), and provided a MATLAB implementation [49]. Another multigrid variant specialized for graph Laplacians was developed by Napov and Notay [51].

BoomerAMG, a parallel AMG implementation, was introduced in [34], and can be accessed via the popular software library Hypre [29]. The Portable, Extensible Toolkit for Scientific Computation (PETSc) provides another interface for accessing the Hypre-BoomerAMG implementation and its own implementations of conjugate gradients [2]. Combinatorial Multigrid (CMG) [44] is a solver that combines ideas from support theory [62, 9] with ideas from (Algebraic) Multigrid methods. The MATLAB implementation of CMG was applied to problems arising in computer vision and image processing in [44]. Langston et al. [48] presented another implementation of CMG in C using PETSc, however, to the best of our knowledge, this implementation is not publicly available. As part of the Trilinos software package, MueLu [6] provides a multigrid algorithm designed for solving sparse linear systems of equations arising from PDE discretizations using massively parallel computing environments.

Theory. There are some theories of convergence for (Algebraic and Geometric) Multigrid methods [11, 67] and preconditioned iterative methods using Incomplete Cholesky factorization [50, 33, 5, 52, 53, 7], but these do not establish fast rates for general SDDM matrices.

Spielman and Teng [60] built on the work of Vaidya [62] to develop the first provably correct nearly-linear time algorithm for solving SDD linear equations [60]. Koutis, Miller and Peng substantially simplified the algorithm of Spielman and Teng and greatly reduced its running time [42, 43]. Building on this line of work, Cohen et al. [23] reduced the running time below that of comparison sorting the input, and recently Jambulapati and Sidford [36] reduced the running time to linear in the number of matrix nonzeros, up to poly-loglog factors and a $\log(1/\epsilon)$ dependence on the error, ϵ. The solver of Kelner et al. [39] introduced a dual-solution based approach, resulting in a simple algorithm, once a low-stretch spanning tree for the matrix has been computed. Another line of work introduced a parallel algorithm [55] and removed the reliance on combinatorial sparsification, replacing it with more spectral methods [45].

Finally, Kyng and Sachdeva [46] introduced and analyzed a very simple algorithm that is the inspiration for the algorithm in the present paper, developing a sampling-based approach to computing a sparse Cholesky factorization. They then use this factorization as preconditioner in an iterative solver (e.g. preconditioned Richardson or conjugate gradient). This was arguably the culmination of a long line of research on simplifying the Spielman-Teng solver [42, 39, 55, 45], and was the first formally proven nearly-linear time Laplacian solver that avoids complicated graph-theoretic constructs such as expanders and low-stretch trees.

Experimental evaluation of solvers from the theoretical computer science community. Chen and Toledo [21] evaluated an SDD solver based on Vaidya's ideas on tree-based preconditioning combined with PCG. They compared the solver with PCG preconditioned using Incomplete Cholesky factorization and found that Vaidya's approach was often much slower, but sometimes lead to convergence in cases where Incomplete Cholesky with PCG did not. Deweese et al. [25] studied the performance of variants of a cycle toggling SDD solver [39]. This algorithm works with a 'dual space' representation (sometimes known as the 'flow space'). The paper focused on contrasting these implementations without a direct comparison to other solvers. Boman, Deweese, and Gilbert [8] evaluated a cycle toggling algorithm in 'primal space' ('voltage space') known as Primal Randomized Kaczmarz (PRK). They compared this with an implementation of the cycle toggling solver [39] and with a PCG implementation using Jacobi diagonal scaling. The authors find that their results "do not at present support the practical utility of" cycle toggling solvers in primal or dual space. Hoske et al. [35] reached similar conclusions in another experimental evaluation of an implementation of the flow-space cycle toggling solver [39]. A recent paper by Chen, Liang, and Biros has provided another implementation of our sampling procedure [20], which they refer to as RCHOL[1].

1.2. Contributions. We introduce a variant of the approximate Cholesky factorization algorithm developed by Kyng and Sachdeva [46] that greatly improves its practical performance while retaining its simplicity. We conduct experiments to study this factorization as a preconditioner in conjugate gradient, and we compare it with other state-of-the-art preconditioners. Our main algorithm, Approximate Cholesky (AC), performs well on many SDDM matrices, but it fails on specially constructed *Sachdeva star* graphs. To address this, we introduce a slightly slower but more robust variant that splits each matrix entry into k "multi-entries" (analogous to graph multi-edges) and then runs the original algorithm; each sampled elimination produces a union-of-trees sparsity pattern with at most as many multi-entries as the eliminated row/column. We call this method AC(k), and with $k = 2$ it performs well across all our tests.

Our algorithm solves linear equations in SDDM matrices, including Laplacian matrices. We conduct extensive experiments, testing the performance of two variants of our algorithm across a wide range of SDDM matrices and comparing our solver with a suite of state-of-the-art solvers, including the BoomerAMG preconditioner with Hypre and PETSc Krylov Solvers, Combinatorial Multigrid, Lean Algebraic Multigrid, and Matlab's ichol Cholesky Factorization with preconditioned conjugate gradient. Our experiments measure the single-threaded performance of these solvers, and we find that our solvers AC and AC(2) are much more robust than all other solvers, while still achieving competitive running times across all problem classes. While Hypre-BoomerAMG offers the best performance on Poisson grid problems, our solver promises to finally make solving general Laplacian and SDDM linear equations practical. We also make a detailed comparison between our solvers and existing approaches to incomplete Cholesky factorization, which shows that we achieve better preconditioning quality at the same level of fill-in as other methods, including IC(k) and ILUT.

[1] The authors used the standard Cholesky lower-triangular format for the output, whereas our implementation uses a row-operation representation of the output. Chen et al. use a METIS-based elimination ordering while ours is adaptive.

Three key differences between these incomplete factorizations and our new Approximate Cholesky factorizations are that (1) our factorizations increase the magnitudes of the entries they keep (2) that they select entries to keep at random and (3) we always preserve the rank of the factorization, by locally sampling in a way that preserves connectivity of graph associated with the nonzero structure of the input.

1.3. Our algorithms. Our Approximate Cholesky (AC) algorithms solve an SDDM linear equation by reducing it to a Laplacian linear equation, computing an approximate Cholesky factorization of that Laplacian matrix, and using it as a preconditioner for the Laplacian in conjugate gradient. Our algorithms eliminate rows (and the corresponding columns) of a Laplacian matrix one at a time. For each off-diagonal entry in the eliminated row, a new entry is sampled and inserted into the remaining matrix. To readers familiar with the graphical view of Cholesky factorization/Gaussian elimination, we can describe our rule as follows:

> *"In the graphical view of Gaussian elimination, we may interpret a row operation as removing an edge (i, j) incident to the vertex i that we eliminate, and replacing the edge with a star graph connecting j to the remaining neighbors. To approximate Gaussian elimination, sample an edge from each star generated by a sequence of row operations used to eliminate i."*

Note the sampled edges form a *spanning tree* on the neighbors, because each neighbor is connected to a neighbor with a strictly higher index in the ordering, except for the last vertex. Thus, this preserves the connectivity structure of the graph. Furthermore, we choose our sampling probabilities and output weights to ensure the expected value of the output samples agree exactly with the output of Cholesky factorization, similar to Kyng-Sachdeva [46]. Heuristically, a reason our solver seems to work with much fewer samples than Kyng-Sachdeva may be that we always maintain the basic guarantee of connectivity on neighbors when sampling. We introduce a variant that samples more entries per original nonzero, increasing the size of the factorization, but improving its preconditioning quality.

1.4. Relationship to Kyng-Sachdeva [46] and (lack of) theoretical guarantees of our algorithms. We rely on a broadly similar framework to the Kyng-Sachdeva solver [46], but introduce a new sampling rule which experimentally seems to work with a lower sampling rate than their approach. Kyng-Sachdeva [46] uses a multi-edge sampling rule with $k = \Theta(\log^2 m)$ multi-edges per original edge, for an input matrix with m nonzeros. This yields a factorization with $O(m \log^3 m)$ nonzeros in the output, and provably constructs a preconditioner with relative condition number $O(1)$. In contrast, the preconditioning quality achieved by our algorithms has not been rigorously analyzed. However, we do prove that when sampling k multi-edges per original edge, our algorithms produce a factorization with $O(mk \log m)$ nonzeros in the output in expectation. Note that we use $k = 1$ and $k = 2$ in our main software, AC and AC(2) respectively, and thus these both produce factorizations with $O(m \log m)$ expected nonzero count. In this regime, we do not hope for $O(1)$ relative condition number, but rather $O(\text{poly}\log(n))$, which seems to match experimental observations. The time required to compute the factorization is $O(mk \log m)$ in expectation when using the alias method [65] to implement our sampling. In practice, we use a binary search tree for sampling instead of the alias method. In the worst case, this incurs an additional $\log(\text{degree}) \leq \log m$ factor in the asymptotic running time, but works well in practice, because the typical degree of eliminated vertices is small in most practical settings. In Table 4.6, we make an experimental comparison between the Laplacians.jl implementation of the Kyng-Sachdeva solver and our solvers. We find our solvers run 5-10 times faster.

1.5. Our implementations and data. We provide Julia implementations of AC and AC(k):

https://github.com/danspielman/Laplacians.jl/

Our benchmark data and a tutorial on how to replicate our results are available at

https://rjkyng.github.io/SDDM2023/

1.6. Organization of the paper. In Section 2, we introduce some standard facts about SDDM matrices, Laplacian matrices, and linear algebra relating to Cholesky factorization of these. In Section 3 we describe our Approximate Cholesky factorization algorithm, and its implementation AC, along with a more robust variant, AC(k). In Section 4, we describe an extensive and detailed set of experiments measuring the performance of AC and AC(k) and a comparison with other incomplete Cholesky methods. In Section 5, we compare our solvers to other state-of-the-art solvers across a broad range of different SDDM linear equations. Finally, in Section 6, we state our conclusions.

2. Preliminaries.

In this section, we introduce several families of matrices for which we are able to construct fast linear system solvers, along with some basic linear algebra.

Indexing notation. We write $[n] \stackrel{\mathrm{def}}{=} \{1, \ldots, n\}$. At times we use "Matlab notation": Given a matrix $\boldsymbol{M}$, we use $\boldsymbol{M}(i,:)$ and $\boldsymbol{M}(:,i)$ to refer to the ith row and column respectively.

Matrix families. A matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ is said to be symmetric diagonally dominant (SDD), if it is symmetric and for each row i we have $\boldsymbol{M}(i,i) \geq \sum_{j\neq i} |\boldsymbol{M}(i,j)|$. A matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ is said to be a symmetric diagonally dominant M-matrix (SDDM) if it is SDD and has nonpositive off-diagonal entries. A matrix $\boldsymbol{L} \in \mathbb{R}^{n\times n}$ is called a Laplacian, when it is SDDM and all row sums are exactly zero.

Graph Laplacians. Every Laplacian matrix (i.e. SDDM with zero row sums) is in one-to-one correspondence with a weighted undirected graph. We consider an undirected graph $G = (V, E)$, with positive edge weights $w : E \to \mathbb{R}_+$. Let $n = |V|$ be the number of vertices and $m = |E|$ the number of edges, and $V = [n]$. Let $\boldsymbol{e}_i$ denote the ith standard basis vector. Given an ordered pair of vertices (u, v), we define the pair-vector $b_{u,v} \in \mathbb{R}^n$ as $\boldsymbol{b}_{u,v} = \boldsymbol{e}_v - \boldsymbol{e}_u$. For an edge e with endpoints u and v (arbitrarily ordered), we define $\boldsymbol{b}_e = \boldsymbol{b}_{u,v}$. By assigning an arbitrary direction to each edge of G we define the Laplacian of G as $\boldsymbol{L} = \sum_{e\in E} w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top$. Note that the Laplacian does not depend on the choice of direction for each edge. Given a single edge e, we refer to $w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top$ as the Laplacian of e. The Laplacian of a weighted, undirected graph is unique given the graph and vice versa, and we will treat these as interchangeable.

FACT 2.1. *If G is connected, then the kernel of the corresponding Laplacian $\boldsymbol{L}$ is* $\operatorname{span}\{\mathbf{1}\}$.

Linear algebra. Let $\mathbf{1} \in \mathbb{R}^n$ denote the all ones vector, with dimension n that will always be made clear in the context of its use. Similarly, we let $\mathbf{0}$ denote the all-zero vector or matrix, depending on context. We use $\boldsymbol{B}^\dagger$ to denote the Moore-Penrose pseudo-inverse of a matrix $\boldsymbol{B}$.

In Supplementary Material Section SM1, we give additional background on the standard linear algebra used to solve linear equations in matrices with rank deficiency but a simple kernel.

Measuring fill-in. Fill-in is a crucial concept in incomplete and approximate Cholesky factorization. While the precise definition varies, fill-in measures the space usage of a factorization relative to the space used for the input. We use the following definition:

DEFINITION 2.2 (Fill-in). *When we produce an LU decomposition $\boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}$ of an $n \times n$ matrix $\boldsymbol{M}$, we define the* fill-in *as* $f \stackrel{\mathrm{def}}{=} (\mathrm{nnz}(\boldsymbol{\mathcal{L}}) + \mathrm{nnz}(\boldsymbol{\mathcal{U}}) - n)/\,\mathrm{nnz}(\boldsymbol{M})$.

Note that in the case of $\boldsymbol{M}$ being a Laplacian matrix, we only store the lower-triangular off-diagonals of the input, and we do not store $\boldsymbol{\mathcal{U}}$ as $\boldsymbol{\mathcal{U}} = \boldsymbol{\mathcal{L}}^\top$.

3. A linear equation solver for Laplacian and SDDM matrices.

Our linear equation solvers are based on Preconditioned Conjugate Gradient (PCG), largely following [3]. Our approach to testing for stagnation[2] is mostly based on the MATLAB PCG implementation. We refer the reader to the textbook by Golub and Van Loan [30, Chapter 11] for additional background. We assume the reader is familiar with basics of PCG and preconditioners. Our goal in this section is to describe our algorithm(s) for computing approximate Cholesky factorizations of Laplacian matrices, which we then use as preconditioners inside PCG. We refer to our algorithm and implementation as AC ("Approximate Cholesky") and a second, more robust variant we call AC(k). Our evaluations focus on $k \in \{1, 2\}$ (AC and AC(2)). We use the overall algorithmic framework of Kyng-Sachdeva [46], but we introduce a new approximate elimination rule that performs better in practice. It yields a somewhat worse (superconstant) relative condition number for the preconditioner, but with a much lower sampling rate.

In Subsection 3.3, we describe a standard approach to reducing SDDM linear equations to Laplacian linear equations with one more variable. This approach was developed by Gremban [32, Lemma 4.2]. We note that the solver can be extended to work for all SDD matrices, again using a standard reduction due to Gremban [32], with a later, more numerically robust variant introduced in [39] (see their Appendix A). These reductions from Laplacian to SDD double the variable count and matrix dimensions. We do not test our solver on non-SDDM SDD matrices in this work.

3.1. Background: Approximate Cholesky factorization of Laplacian matrices.

In this section, we briefly review how to compute a Cholesky factorization of a Laplacian matrix, using the additive formulation of

[2]In the literature on practical iterative solvers for linear equations, 'stagnation testing' refers to tests that assess whether the solver is converging at an appropriate rate, or is failing to make sufficient progress.

Kyng-Sachdeva [46]. We then briefly discuss how the *elimination clique* that arises from each row and column elimination can be sampled and how this yields a fast algorithm for approximate Cholesky decomposition [46]. This decomposition may then be used as a preconditioner in PCG to build a linear equation solver.

Producing a Cholesky factorization. Cholesky factorization expresses a matrix in the form $\boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{L}}^T$. A Cholesky factorization can be found for every symmetric positive definite matrix, and for every positive semidefinite matrix after a possible permutation of its rows and columns. One perspective on Cholesky factorization is that it proceeds by writing a symmetric positive semi-definite matrix as a sum of rank 1 matrices. We will outline this view here. Let $\boldsymbol{M}$ be a symmetric positive semi-definite matrix. We define the sequence of Schur complements $\boldsymbol{S}^{(i)}$ by $\boldsymbol{S}^{(0)} \stackrel{\text{def}}{=} \boldsymbol{M}$, and then define

$$\boldsymbol{l}_i = \frac{1}{\sqrt{\boldsymbol{S}^{(i-1)}(i,i)}} \boldsymbol{S}^{(i-1)}(:,i) \text{ and } \boldsymbol{S}^{(i)} = \boldsymbol{S}^{(i-1)} - \boldsymbol{l}_i \boldsymbol{l}_i^\top, \tag{3.1}$$

unless $\boldsymbol{S}^{(i-1)}(:,i) = \boldsymbol{0}$, in which case we define $\boldsymbol{S}^{(i)} = \boldsymbol{S}^{(i-1)}$. However, one can easily show that given the Laplacian of a connected graph, this will occur exactly once, namely $\boldsymbol{S}^{(n-1)} = \boldsymbol{0}$. We do not define the Schur complement when the diagonal is zero but off-diagonals are not — this can be shown not to occur when performing Cholesky factorization on Laplacians. By setting $\boldsymbol{\mathcal{L}} = \begin{pmatrix} \boldsymbol{l}_1 & \boldsymbol{l}_2 & \cdots & \boldsymbol{l}_n \end{pmatrix}$ we get a Cholesky factorization $\boldsymbol{M} = \boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{L}}^\top$. The algorithm described above always produces a Cholesky factorization when applied to a Laplacian, with no pivoting required.

The clique structure of Schur complements. In this section, we recall a standard proof that the Schur complement of a Laplacian is another Laplacian, while also observing that the Schur complement of a Laplacian onto $n-1$ indices has additional structure that will help us develop algorithms for approximating these Schur complements. Given a Laplacian $\boldsymbol{L}$, let $\textsc{Star}[\boldsymbol{L}]_v \in \mathbb{R}^{n\times n}$ denote the Laplacian corresponding to the edges incident on vertex v (the *star* on v), i.e. $\textsc{Star}[\boldsymbol{L}]_v \stackrel{\text{def}}{=} \sum_{e\in E: e\ni v} w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top$. For example, if the first column of $\boldsymbol{L}$ is $\begin{pmatrix} d \\ -\boldsymbol{a} \end{pmatrix}$, then $\textsc{Star}[\boldsymbol{L}]_1 = \begin{bmatrix} d & -\boldsymbol{a}^\top \\ -\boldsymbol{a} & \operatorname{diag}(\boldsymbol{a}) \end{bmatrix}$. We can write the Schur complement $\textsc{Sc}[\boldsymbol{L}]_{[n]\setminus\{v\}}$ as

$$\textsc{Sc}[\boldsymbol{L}]_{[n]\setminus\{v\}} = \boldsymbol{L} - \textsc{Star}[\boldsymbol{L}]_v + \textsc{Star}[\boldsymbol{L}]_v - \frac{1}{\boldsymbol{L}(v,v)}\boldsymbol{L}(:,v)\boldsymbol{L}(v,:).$$

It is immediate that $\boldsymbol{L} - \textsc{Star}[\boldsymbol{L}]_v$ is a Laplacian matrix, since $\boldsymbol{L} - \textsc{Star}[\boldsymbol{L}]_v = \sum_{e\in E: e\not\ni v} w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top$. A more surprising (but well-known) fact is that

$$\textsc{Clique}[\boldsymbol{L}]_v \stackrel{\text{def}}{=} \textsc{Star}[\boldsymbol{L}]_v - \frac{1}{\boldsymbol{L}(v,v)}\boldsymbol{L}(:,v)\boldsymbol{L}(v,:) \tag{3.2}$$

is also a Laplacian, and its edges form a clique on the neighbors of v. It suffices to show this for $v = 1$. We write $i \sim j$ to indicate $(i,j) \in E$. Then

$$\textsc{Clique}[\boldsymbol{L}]_1 = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0}^\top \\ \boldsymbol{0} & \operatorname{diag}(\boldsymbol{a}) - \frac{\boldsymbol{a}\boldsymbol{a}^\top}{d} \end{bmatrix} = \sum_{i\sim 1}\sum_{j\sim 1, i<j} \frac{w(1,i)w(1,j)}{d}\boldsymbol{b}_{(i,j)}\boldsymbol{b}_{(i,j)}^\top.$$

Thus $\textsc{Sc}[\boldsymbol{L}]_{[n]\setminus\{v\}}$ is a Laplacian since it is a sum of two Laplacians. By induction, for all $C \subseteq [n]$, $\textsc{Sc}[\boldsymbol{L}]_C$ is a Laplacian.

Laplacian Cholesky factorization with cliques. The clique structure described above lets us write the Cholesky factorization of a Laplacian in a convenient form. Algorithm 3.1 gives pseudo-code for this.

A framework for Approximate Cholesky factorization via clique sampling. Kyng and Sachdeva [46] introduced the idea of constructing a preconditioner by replacing the elimination clique in Cholesky factorization with an unbiased, sampled approximation of it. In the previous section, we interpreted Gaussian elimination of a variable v (the vth row/column) as constructing the Schur complement on the remaining variables (submatrix) by subtracting the Laplacian corresponding to the edges incident on the vertex v, i.e. $\textsc{Star}[\boldsymbol{S}]_v$, and then adding a Laplacian which corresponds to a complete graph on the neighbors of v, i.e. $\textsc{Clique}[\boldsymbol{S}]_v$. In matrix terms, this is the familiar procedure of deleting a row and column, and, in the resulting submatrix, updating the diagonal and off-diagonal entries corresponding to each pair of off-diagonals in the eliminated row and column. Kyng and Sachdeva proposed

Algorithm 3.1 Algorithm $\textsc{Cholesky}(\boldsymbol{L})$ outputs a lower triangular Cholesky factor $\boldsymbol{\mathcal{L}} \in \mathbb{R}^{n\times n}$ of the input Laplacian $\boldsymbol{L} \in \mathbb{R}^{n\times n}$.

procedure $\textsc{Cholesky}$ $(\boldsymbol{L})$
  $\boldsymbol{S} \leftarrow \boldsymbol{L}$
  for $v = 1$ to $n-1$ **do** ▷ the order can be chosen adaptively
    $\boldsymbol{l}_v \leftarrow \frac{1}{\sqrt{\boldsymbol{S}(v,v)}}\boldsymbol{S}(:,v)$ ▷ $\boldsymbol{S}(:,v)$ is the vth column of $\boldsymbol{S}$
    $\boldsymbol{S} \leftarrow \boldsymbol{S} - \textsc{Star}[\boldsymbol{S}]_v + \textsc{Clique}[\boldsymbol{S}]_v$
  return $\boldsymbol{\mathcal{L}} = \begin{pmatrix}\boldsymbol{l}_1 & \boldsymbol{l}_2 & \cdots & \boldsymbol{l}_{n-1} & \boldsymbol{0}\end{pmatrix}$ ▷ The final column should be the all-zero vector.

replacing this Laplacian of a clique on the neighbors with the Laplacian of a randomly-sampled, sparser subgraph on the neighbors. In matrix terms, instead of updating the diagonals and off-diagonals associated with all pairs of neighbors of v; we instead sample a subset of the off-diagonal entries to update, while also updating all the diagonal entries of neighbors. If we wish to ensure the output agrees in expectation with the update $\textsc{Clique}[\boldsymbol{S}]_v$, which Gaussian elimination dictates, then we can increase the magnitude of the entries we sample to correct for using fewer entries. For now, we state a generic algorithm by using $\mathsf{CliqueSample}$ as a placeholder for the sampler and we allow any ordering. This gives a generic algorithm for approximate Cholesky factorization stated in Algorithm 3.2. The major factors governing the performance of such a preconditioner are the sampling procedure and the order in which vertices are eliminated.

Algorithm 3.2 Algorithm $\textsc{ApproximateCholesky}(\boldsymbol{L})$ outputs a lower triangular matrix $\boldsymbol{\mathcal{L}} \in \mathbb{R}^{n\times n}$ that gives an approximate Cholesky factorization of the input Laplacian $\boldsymbol{L} \in \mathbb{R}^{n\times n}$.

procedure $\textsc{ApproximateCholesky}$ $(\boldsymbol{L})$
  $\boldsymbol{S} \leftarrow \boldsymbol{L}$
  for $v = 1$ to $n-1$ **do** ▷ the order can be chosen adaptively
    $\boldsymbol{l}_v \leftarrow \frac{1}{\sqrt{\boldsymbol{S}(v,v)}}\boldsymbol{S}(:,v)$ ▷ $\boldsymbol{S}(:,v)$ is the vth column of $\boldsymbol{S}$
    $\boldsymbol{S} \leftarrow \boldsymbol{S} - \textsc{Star}[\boldsymbol{S}]_v + \mathsf{CliqueSample}(v, \boldsymbol{S})$
  return $\boldsymbol{\mathcal{L}} = \begin{pmatrix}\boldsymbol{l}_1 & \boldsymbol{l}_2 & \cdots & \boldsymbol{l}_{n-1} & \boldsymbol{0}\end{pmatrix}$ ▷ The final Schur complement is necessarily zero.

Formally, the requirement that the sampler be unbiased means $\mathbb{E}\left[\mathsf{CliqueSample}(v, \boldsymbol{S})\right] = \textsc{Clique}[\boldsymbol{S}]_v$. Kyng and Sachdeva [46] proved that unbiased clique sampling produces a factorization that in expectation equals the original input matrix.[3] We state this as a formal claim below, and for completeness, we include a proof in Supplementary Material Section SM4.

Claim 3.1. *When* $\textsc{ApproximateCholesky}$ *is instantiated with an unbiased clique sampling routine* $\mathsf{CliqueSample}$ *whose output is positive semidefinite, its output* $\boldsymbol{\mathcal{L}} = \textsc{ApproximateCholesky}(\boldsymbol{L})$ *satisfies*

$$\mathbb{E}\left[\boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{L}}^\top\right] = \boldsymbol{L}.$$

We will introduce a sampling procedure $\textsc{CliqueTreeSample}$ that outputs at most as many multi-edges as the multi-degree of the vertex being eliminated. The analysis of Kyng-Sachdeva may be used to bound the running time of approximate elimination with a clique sampler that satisfies such a guarantee. Suppose the clique sampler has the property that it always outputs at most as many (multi-)edge samples as the (multi-edge) degree of the vertex v in $\boldsymbol{S}$. If $\textsc{ApproximateCholesky}$ is then run with an elimination ordering that always picks a vertex whose degree in the remaining graph is upper bounded by a constant multiple of the average degree of vertices in the remaining graph, then the number of nonzeros in the output $\boldsymbol{\mathcal{L}}$ will be at most $O(m \log m)$. This bound always holds, and does not depend on the random choices made by the clique sampler. If the degree bound holds in expectation for each elimination, the nonzero bound will hold in expectation as well. We state this claim formally below.

[3] See [46] Section 4.1. They considered a specific sampling rule, but their proof uses no properties of the sampling rule except that it is unbiased.

Claim 3.2. *Suppose CliqueSample$(v, \boldsymbol{S})$ outputs at most as many multi-edge samples as the multi-edge degree of the vertex v in $\boldsymbol{S}$. Then if* ApproximateCholesky *is run with an elimination ordering that ensures that when a vertex v is eliminated, we have*

$$\text{degree of } v \text{ in current graph} \leq O(\text{average degree in current graph}).$$

then ApproximateCholesky *always outputs a lower triangular matrix $\boldsymbol{\mathcal{L}}$ with $O(m \log m)$ nonzeros. In fact, it suffices for the degree of v in the current graph to be bounded by a constant times the number of multi-edges in the original graph divided by the number of vertices in the current graph.*

If the bound on the degree of v holds at each elimination step in expectation over a random choice of v, then ApproximateCholesky *outputs a lower triangular matrix $\boldsymbol{\mathcal{L}}$ with $O(m \log m)$ nonzeros in expectation.*

Proof. Observe that the total number of edges in $\boldsymbol{S}$ is non-increasing over time, as each elimination removes the edges incident on the vertex being eliminated and the CliqueSample call adds back at most the same number. Hence $\boldsymbol{S}$ contains at most m edges. The number of nonzeros in column $\boldsymbol{l}_i$ of $\boldsymbol{\mathcal{L}}$ is proportional to the degree of the vertex being eliminated in the current graph of $\boldsymbol{S}$. This is hence on average equal to the average degree in $\boldsymbol{S}$, which is at most $\frac{2m}{n-i}$. Thus the total number of nonzeros in $\boldsymbol{\mathcal{L}}$ is bounded by

$$\sum_{i=1}^{n-1} \frac{2m}{n-i} = O(m \log m).$$

If the bound per elimination holds in expectation, the final bound $O(m \log m)$ will hold in expectation. ∎

3.2. Edgewise elimination and a new clique sampling rule. In this section, we describe how the elimination clique created in each step of Cholesky factorization can in fact be further decomposed into a sum of *elimination stars* on the neighbors of the vertex being eliminated. We then show how subsampling each elimination star gives rise to a clique sampler with a desirably property: The output is always a connected graph. We describe two variants of our sampling rule. The simplest approximates the elimination clique with a tree, and hence we call it CliqueTreeSample. This sampling approach directly yields our main approximate Cholesky algorithm, which we refer to as AC.

Our second variant, called AC(2), approximates the elimination clique with a tree and potentially a few extra edges, roughly as many as in the tree. This variant arises naturally from combining our CliqueTreeSample approach with the multi-edge splitting that is used in [46] to improve the approximation quality by doing more fine-grained sampling. We call this CliqueTreeSampleMultiEdge, and a slight but important tweak of it is called CliqueTreeSampleMultiEdgeMerge. This tweak limits the maximum number of multi-edges between two vertices, which gives large performance gains in practice. These modifications are trivial to implement by introducing multi-edges. Nonetheless, we describe them in detail in in Supplementary Material Section SM2. In general, we can use k multi-edges per original edge, and this leads to a version of our algorithm which we refer to as AC(k). In our evaluations we find AC (i.e. $k = 1$) and AC(2) usually perform best, as the improved preconditioner quality obtained by further increasing k is not worth the additional time and space spent on constructing the higher-sample approximate Cholesky factor.

Decomposing the elimination clique into elimination stars. When we eliminate a vertex, we can further decompose the resulting clique $\textsc{Clique}[\boldsymbol{L}]_v$ as a sum of smaller graphs. This decomposition may appear a little mysterious, but it arises naturally if we consider the process of eliminating a column as a sequence of eliminations that each remove one entry of the column. To arrive at the decomposition, we pick an ordering on the neighbors of the vertex being eliminated. To simplify the discussion, we consider eliminating the first vertex $v = 1$, and the order we pick for eliminating the neighbors is by increasing vertex number. However, any vertex can be eliminated this way and any ordering on the neighbors can be used. As before, we let the first column of the Laplacian $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ be $\begin{pmatrix} d \\ -\boldsymbol{a}(2:n) \end{pmatrix}$ where $\boldsymbol{a} \in \mathbb{R}^n$ has first entry 0. We also define $\boldsymbol{a}_i \in \mathbb{R}^n$ given by

$$\boldsymbol{a}_i(j) = \begin{cases} \boldsymbol{a}(l) & \text{for } j > i \\ 0 & \text{for } j \leq i \end{cases}.$$

Thus $\boldsymbol{a}_i$ is the off-diagonal entries of the first column, after removing the first i entries. We can then break $\textsc{Clique}[\boldsymbol{L}]_1$ into a sum of smaller terms, where each term is a Laplacian consisting of edges from neighbor i of vertex

1 to neighbors $l \geq i$. This term is given by $\textsc{ElimStar}[\boldsymbol{L}]_{1,i} \stackrel{\text{def}}{=} \left(\operatorname{diag}\left(\frac{\boldsymbol{a}(i)\boldsymbol{1}^\top \boldsymbol{a}_i}{d}\boldsymbol{e}_i + \frac{\boldsymbol{a}(i)}{d}\boldsymbol{a}_i\right) - \frac{\boldsymbol{a}(i)}{d}\left(\boldsymbol{e}_i\boldsymbol{a}_i^\top + \boldsymbol{a}_i\boldsymbol{e}_i^\top\right)\right)$. We can also write this as

$$\textsc{ElimStar}[\boldsymbol{L}]_{1,i} = \left(\sum_{j>i} \frac{\boldsymbol{a}(j)\boldsymbol{a}(i)}{d}(\boldsymbol{e}_i - \boldsymbol{e}_j)(\boldsymbol{e}_i - \boldsymbol{e}_j)^\top\right). \tag{3.3}$$

We call $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ the *elimination star* of neighbor i. It is straightforward to verify that $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ is a graph Laplacian and it is nonzero only when $\boldsymbol{a}(i) \neq 0$ and consists of edges from the ith neighbors $l \geq i$ where $\boldsymbol{a}(l) \neq 0$. Note that $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ depends on the ordering chosen on the neighbors, and again, any ordering can be used.

Furthermore, these Laplacians together add up to give the clique created by eliminating vertex 1. This is immediate from the fact that each $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ contains exactly the off-diagonal entries of $\textsc{Clique}[\boldsymbol{L}]_1$ for i and for each $j > i$, and thus summing up across all i, we get $\textsc{Clique}[\boldsymbol{L}]_1$. Because $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ is nonzero only when i is a neighbor of vertex 1 in the current graph, we can write

$$\textsc{Clique}[\boldsymbol{L}]_1 = \sum_{i\sim 1} \textsc{ElimStar}[\boldsymbol{L}]_{1,i}. \tag{3.4}$$

Approximating the elimination clique by sampling a tree. Like [46], we will approximate the elimination clique by sampling. However, we engineer our sampler so that each elimination star $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$ is approximated by a single randomly chosen reweighted edge, chosen so that in expectation, this edge Laplacian yields the elimination star $\textsc{ElimStar}[\boldsymbol{L}]_{1,i}$.

We choose a random index γ_i according to a probability distribution given by $\Pr[\gamma_i = j] = p_i(j)$ where $p_i(j) = \frac{\boldsymbol{a}_i(j)}{\boldsymbol{1}^\top \boldsymbol{a}_i}$. We can also write this as

$$p_i(j) = \begin{cases} \frac{\boldsymbol{a}(j)}{\sum_{l>i}\boldsymbol{a}(l)} & \text{for } j > i \\ 0 & \text{for } j \leq i \end{cases}.$$

To make the expectations work out, if the outcome is $\gamma_i = j$, we then choose a weight of

$$\tilde{w}(i,j) = \frac{1}{p_i(j)}\frac{1}{d}\boldsymbol{a}(j)\boldsymbol{a}(i) = \frac{\boldsymbol{a}(i)\boldsymbol{1}^\top\boldsymbol{a}_i}{d}, \tag{3.5}$$

and we output the single edge Laplacian $\tilde{\boldsymbol{S}}_i = \tilde{w}(i,j)(\boldsymbol{e}_i - \boldsymbol{e}_j)(\boldsymbol{e}_i - \boldsymbol{e}_j)^\top$. We can show that $\mathbb{E}_{\gamma_i}\left[\tilde{\boldsymbol{S}}_i\right] = \textsc{ElimStar}[\boldsymbol{L}]_{1,i}$, i.e. the sampling of the elimination star is correct in expectation.

Claim 3.3. $\mathbb{E}_{\gamma_i}\left[\tilde{\boldsymbol{S}}_i\right] = \textsc{ElimStar}[\boldsymbol{L}]_{1,i}$.

Proof. By Equation (3.3) and the observations above, we have that

$$\mathbb{E}_{\gamma_i}\left[\tilde{\boldsymbol{S}}_i\right] = \sum_{j>i} p_i(j)\tilde{w}(i,j)(\boldsymbol{e}_i - \boldsymbol{e}_j)(\boldsymbol{e}_i - \boldsymbol{e}_j)^\top = \sum_{j>i}\frac{1}{d}\boldsymbol{a}(j)\boldsymbol{a}(i)(\boldsymbol{e}_i - \boldsymbol{e}_j)(\boldsymbol{e}_i - \boldsymbol{e}_j)^\top = \textsc{ElimStar}[\boldsymbol{L}]_{1,i}.$$

∎

We now approximate the whole elimination clique $\textsc{Clique}[\boldsymbol{L}]_1$ by the union of one sample for each of the elimination stars, leading to an approximation of the clique by $\sum_{i\sim 1}\tilde{\boldsymbol{S}}_i \approx \textsc{Clique}[\boldsymbol{L}]_1$. We summarize the pseudo-code for this sampling procedure in Algorithm 3.3, which we call $\textsc{CliqueTreeSample}$ for reasons that will be apparent in a moment.

Algorithm 3.3 Algorithm $\textsc{CliqueTreeSample}(v, \boldsymbol{S})$ returns $\tilde{\boldsymbol{C}}$ which approximates the elimination clique $\textsc{Clique}[\boldsymbol{S}]_v$

procedure $\textsc{CliqueTreeSample}(v, \boldsymbol{S})$
  $\tilde{\boldsymbol{C}} \leftarrow \boldsymbol{0} \in \mathbb{R}^{n\times n}$
  $d \leftarrow \boldsymbol{S}(v,v)$; $\boldsymbol{a} \leftarrow -\boldsymbol{S}(v,:)$; $\boldsymbol{a}(v) \leftarrow 0$
  for all i s.t. $\boldsymbol{a}(i) \neq 0$ **do** ▷ pick any ordering on the neighbors
    $c \leftarrow \boldsymbol{a}(i)$; $\boldsymbol{a}(i) \leftarrow 0$
    Sample index j with probability $p(j) = \frac{\boldsymbol{a}(j)}{\boldsymbol{1}^\top\boldsymbol{a}_i}$
    $\tilde{\boldsymbol{C}} \leftarrow \tilde{\boldsymbol{C}} + c \cdot \frac{\boldsymbol{1}^\top\boldsymbol{a}}{d}(\boldsymbol{e}_i - \boldsymbol{e}_j)(\boldsymbol{e}_i - \boldsymbol{e}_j)^\top$
  return $\tilde{\boldsymbol{C}}$

We can directly conclude from Equation (3.3) and Claim 3.3 that, in expectation, the output of CliqueTreeSample(v,$\mathbf{S}$) equals $\text{Clique}[\boldsymbol{S}]_v$.

Corollary 3.4. $\mathbb{E}\left[\text{CliqueTreeSample}(v, \boldsymbol{S})\right] = \text{Clique}[\boldsymbol{S}]_v$.

It turns out that CliqueTreeSample always outputs the Laplacian of a tree on the neighbors of the vertex being eliminated.

Claim 3.5. *The random matrix returned by* CliqueTreeSample$(v, \boldsymbol{S})$ *is always the Laplacian of a tree on the neighbors of v in the graph associated with $\boldsymbol{S}$.*

Proof. We can trivially ignore all the vertices that are not neighbors of vertex v, and w.l.o.g. assume the neighbors of v are numbered $1, \ldots, k$. The result follows by a simple induction, starting from $i = k-1$ and down to $i = 1$. Note that the ith sample connects vertex i to a vertex $j > i$. Our induction hypothesis is that samples $i, \ldots, k$ form a tree on vertices $i, \ldots, k$. The base case $i = k-1$ is trivially true as the last sample connects vertices $k-1$ and k by a single edge (deterministically). For the inductive step, observe that the ith sample connects to one of the vertices $i+1, \ldots, k$, where by induction, we already have a tree. Thus i is now connected by an edge to the tree on vertices $i+1, \ldots, k$. And the new graph must again form a tree as it connects $k-i+1$ vertices using $k-i$ edges. ∎

From this claim, it follows by a simple induction that ApproximateCholesky (Algorithm 3.2) returns a lower triangular factor $\boldsymbol{\mathcal{L}}$ with rank equal to the rank of the input Laplacian matrix, when using CliqueTreeSample, because replacing a clique with a connected graph never changes the matrix rank.

CliqueTreeSample can be instantiated with different orderings on the neighbors of v. We use an ordering by increasing weight $w(v, i)$. This is motivated by numerical stability of the sampling and heuristic arguments about global variance. It is likely other orderings may work well too. Using an ordering by increasing weight also means the algorithm can be implemented in expected linear time in the degree of v by using the alias method [65] for sampling, and combining with rejection sampling to avoid updating the sampling weights until a constant fraction of entries have been processed. However, in practice we opt for maintaining probabilities using a binary search tree, with $O(k \log k)$ overall complexity processing a vertex of degree k. In the worst case, this may add an additional factor $\log m$ to the running time. However, when the typical degree of an eliminated vertex is low, this is highly performant. For each edge being eliminated, we always average the weights of its multi-edges. This step can be omitted, but in practice we observe that this step makes the code numerically more stable. We also expect it improves spectral approximation slightly as the averaged samples have lower maximum leverage score.

Approximate Cholesky factorization: unbiased, fast, and sparse. We plug in CliqueTreeSample for CliqueSample as our choice of clique sampling routing in Algorithm 3.2 to obtain a new algorithm for computing an approximate Cholesky factorization. By Corollary 3.4, the sampling rule is unbiased, and hence by Claim 3.1, the overall output satisfies $\mathbb{E}\left[\boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{L}}^\top\right] = \boldsymbol{L}$.

We eliminate the vertices in an order that guarantees that the degree of the vertex being eliminated is always at most twice the number of multi-edges in the original graph divided by the number of vertices in the current graph. Thus, Claim 3.2 applies and the number of nonzeros in the output decomposition is bounded by $O(m \log m)$ when the input Laplacian has $O(m)$ nonzeros. We can summarize our observations in the following theorem:

Theorem 3.6. *Calling* ApproximateCholesky*($\boldsymbol{L}$) with the sampling rule* CliqueTreeSample *outputs a lower triangular matrix $\boldsymbol{\mathcal{L}} \in \mathbb{R}^{n\times n}$ that gives an approximate Cholesky factorization of the input Laplacian $\boldsymbol{L} \in \mathbb{R}^{n\times n}$ with* $\text{rank}(\boldsymbol{\mathcal{L}}) = \text{rank}(\boldsymbol{L})$ *and* $\mathbb{E}\left[\boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{L}}^\top\right] = \boldsymbol{L}$. *The nonzero count of $\boldsymbol{\mathcal{L}}$ is bounded by $O(m \log m)$ and the running time is $O(m \log m)$ in expectation when sampling with the alias method [65]. Sampling with a binary tree data structure increases the running time upper bound by a factor $O(\log m)$.*

When using a multi-edge version of algorithm with k multi-edges (splits) per original edge, this increases the sparsity and running time to $O(km \log m)$ - we refer to this multi-edge version as AC(k). In practice, we focus on $k = 1$ and $k = 2$, i.e. AC and AC(2).

3.3. Our full solver algorithms. In this section we describe how, using our clique sampling approach, we obtain our full solver algorithms. In particular, we describe our vertex elimination ordering, our slightly non-standard output format, and the reduction from SDDM matrices to Laplacian matrices.

Elimination order. Our ordering is greedy on the (approximate) unweighted degree of the vertices. More precisely, the vertex being eliminated is a vertex with unweighted degree within a factor 2 of the lowest unweighted degree. This is non-standard in the context of Incomplete Cholesky factorization, where the "natural" ordering is often preferred for low fill-in or Reverse Cuthill McKee for higher fill-in [58]. Our ordering is related to conventional minimum degree heuristics [1, 26], but it is adaptive to the *actual* vertex degrees created by approximate elimination, not the degrees induced by exact elimination.

While we expect that one could obtain sparser preconditioners by always eliminating a vertex of lowest degree in the current graph, the priority queue required to implement this would slow down the algorithm. We use a lazy approximate version of this rule because it is faster, but we also note that there is some theoretical evidence that choosing randomly among low-degree vertices is beneficial [46]. For very large instances, where memory consumption is a key concern, it may be worthwhile to use a more precise minimum-degree priority queue. We leave this question for future exploration.

Factorization output format. Our approximate Cholesky factorization approach can be used directly with a standard Cholesky factorization output format as shown in Algorithm 3.2. However, we found in practice that additional (small constant factor) speed-ups could be achieved by using a slightly non-standard row operation format. We describe this row-operation format in detail in Supplementary Material Section SM3. We implemented a routine for applying the inverse of the output factorization without converting to the standard Cholesky lower-triangular format. With appropriate optimizations, a standard Cholesky factorization format can possibly be made equally efficient and is potentially easier to parallelize.

Extending our solver to SDDM matrices. We reduce the problem of solving linear equations in SDDM matrices to that of solving linear equations in Laplacian matrices by using a technique introduced by Gremban [32, Lemma 4.2]. An SDDM Matrix $\boldsymbol{M}$ can be written in the form $\boldsymbol{D} + \boldsymbol{L}$ where $\boldsymbol{L}$ is a Laplacian and $\boldsymbol{D}$ is a non-negative diagonal matrix. Let $\boldsymbol{d}$ be the vector of entries on the diagonal of $\boldsymbol{D}$. Gremban constructs a Laplacian matrix $\widehat{\boldsymbol{L}}$ that is identical to $\boldsymbol{L}$ except that it has one extra vertex that is connected to vertex i by an edge of weight $\boldsymbol{d}[i]$. To solve the system $\boldsymbol{M}\boldsymbol{x} = b$, we first solve $\widehat{\boldsymbol{L}}\boldsymbol{y} = \widehat{\boldsymbol{b}}$ where $\widehat{\boldsymbol{b}}$ is identical to $\boldsymbol{b}$ except at the extra vertex, where its value is set to make the sum of the entries of $\widehat{\boldsymbol{b}}$ zero. The vector $\boldsymbol{x}$ is then obtained by subtracting the value of $\boldsymbol{y}$ at the extra vertex from the values of $\boldsymbol{y}$ at the original vertices.

Our full solver algorithms. We provide two solvers for Laplacian linear equations, AC and AC(k). Both compute an approximate Cholesky factorization using the approaches described in previous sections and then solve the linear equations using PCG with a preconditioner given by an approximate Cholesky factorization. These solvers are then extended to SDDM matrices using Gremban's approach to add one more variable and obtain a Laplacian linear system. Our first implementation, AC, uses the CliqueTreeSample sampling rule (Algorithm 3.3) and formats the output as row operations (Algorithm SM3.2). Our second implementation, AC(k), uses the CliqueTreeSampleMultiEdgeMerge sampling rule (Algorithm SM2.3) with split parameter k and merge parameter k and formats the output as row operations (Algorithm SM3.2). Because we work with Laplacian matrices (which have kernel of dimension one or higher), we need to account for the kernel when using an (approximate) Cholesky decomposition as a preconditioner. This is standard, but for completeness we describe this step in Appendix SM1.

3.4. Implementation. We provide two main implementations of our SDDM and Laplacian linear equation solvers, which we refer to as AC and AC(k), where the latter uses k multi-edges per original edge, and AC is an optimized code for the case $k = 1$. Both AC and AC(k) use preconditioned conjugate gradient (PCG) to compute a solution to the input system of linear equations, up to the specified residual error tolerance. Our implementation of PCG is based on the book *Templates for the solution of linear systems: building blocks for iterative methods* [3]. Both solvers are initially designed for Laplacian matrices, and use a standard reduction to convert linear equations in SDDM matrices into linear equations in Laplacians. In Section 3.2, we describe the overall pseudocode for AC, with the row-operation output format further described in Supplementary Material Section SM3. In Supplementary Material Section SM2, we describe the multi-edge splitting and merging used to obtain AC(k).

Tuning the sampling quality: AC vs. AC(2) and more. We define an even broader family of AC variants, called AC-skmj, which first *splits* every original edge into k multi-edges, and then during execution *merges* multi-edges whenever more than j of them accumulate between a pair of vertices. We use AC(k) to refer to AC-skmk, i.e. with split and merge count both equal to k. In Supplementary Material Section SM2, we run experiments with a range of parameters for AC-skmj, namely $k = 1, 2, 3$ and $j = 1, 2, 3, \infty$, and conclude that generally setting $j = k$ is preferable, and $k = 1$ or $k = 2$ provides the best trade-offs between preconditioner quality and size. This leads us to focus on AC and AC(2) for the rest of our experiments. Increasing the split-count or merge count always improves the preconditioner quality, but at the expense of using more space.

For easy problems, the factorizations given by AC are reliable enough and hence have the best runtime. However, for harder problems, such as the Sachdeva star (see Figure 5.1 and Table 5.4), AC cannot produce reliable factorizations and the PCG algorithm might stagnate. In contrast, AC(2) is still able to produce reliable factorizations for Sachdeva stars, while maintaining a relatively good runtime. Thus, AC should be used as the main version for easier problems and AC(2) for harder problems.

4. Experimental evaluation and comparison. In this section, we provide experimental evaluation of our solvers AC and AC(2). We start by describing our general experimental setup. In Subsection 4.1, we first show plots that demonstrate the nearly-linear running time scaling of AC and AC(2) across the entire set of instances we use for our experiments. We then investigate the timing and memory consumption of AC and AC(2), scaling up to large instances; and we study the performance variability of AC and AC(2). In Subsection 4.2, we investigate the impact of sampling rates on fill-in and preconditioner quality, and we study different ways to tune fill-in by splitting and merging multi-edges. Next, we study the impact of our elimination ordering compared with the suggested random ordering in theoretical work [46]. Finally, in Subsection 4.3, we conduct a more in-depth study of the trade-off between fill-in and preconditioner quality obtained by AC(k) compared with other approaches to incomplete Cholesky factorization.

Our largest set of experiments are deferred to Section 5, where we measure the single-threaded performance of our solvers and compare them to a suite of state-of-the-art SDDM solvers on broad classes of SDDM and Laplacian matrices.

Experimental set-up. We run all our experiments on a 24-core Intel Xeon Gold 6240R 2.4GHz processor with 512 GB memory, on Ubuntu. Experiments were run with single-threaded versions of each solver. Our main experiments focus on (a) generating a preconditioner for a given matrix and (b) solving linear equations in the given matrix using said preconditioner combined with preconditioned conjugate gradient. We measure various performance statistics in this context. When solving a linear equation in a matrix $\boldsymbol{M}$, we apply $\boldsymbol{M}$ to a random Gaussian $\boldsymbol{g}$ and normalize, yielding $\boldsymbol{M}\boldsymbol{g}/\left\|\boldsymbol{M}\boldsymbol{g}\right\|_2$ as the right-hand side. This ensures the right-hand side is in the image of $\boldsymbol{M}$. This is important for some solvers in the context of Laplacian linear equations, which typically have a rank deficit of 1. On the other hand, our experiments on general SDDM matrices always use full-rank matrices.

Solver error tolerances. For each linear equation, we report the time and iterations required to obtain a relative residual of 10^{-8}. *If the solver has higher residual error, we mark the time entry with $*$ if the residual error exceeds the target by a factor $(1, 10^4]$ and by $**$ if it exceeds the target by a factor $(10^4, 10^8)$, and finally we report the time as ∞ (INF) if the residual error is too large by a factor 10^8 or more.* The zero vector obtains a residual error of 1, matching our threshold for reporting the time as ∞.

Measurements per solve. We report wall-clock times, but exclude the time required to load the input matrix into memory before starting the solver (because our file loading system for Hypre is slow). For each experiment, we report a subset of the following statistics. When we summarize across many instances, we report the median, 75th percentile, and worst-case running times seen across these instances.

- n: The number of variables in the linear equation.
- nnz: The number of nonzeros in the original matrix.
- t_{total}: The total time in seconds (wall-clock time) to build our approximate Cholesky factorization and solve a linear equation in the input matrix. $t_{\text{total}} = t_{\text{build}} + t_{\text{solve}}$.
- t_{build}: The total time in seconds (wall-clock time) to build our approximate Cholesky factorization of the input matrix.
- t_{solve}: The total time in seconds (wall-clock time) to solve a linear equation in the input matrix.
- N_{iter}: The number of iterations of PCG required to reach the desired relative residual.
- f: Fill-in (see Definition 2.2).

In every figure and table, we report the units of measurement, using standard SI units and standard metric prefixes. Our experiments mostly use nonzero counts ranging from thousands ($1K\,\text{nnz} = 10^3\,\text{nnz}$) to millions ($1M\,\text{nnz} = 10^6\,\text{nnz}$). Running times are generally reported in microseconds per nonzero, i.e. $\mu s/\text{nnz} = 10^{-6} s/\text{nnz}$, as this leads to an interpretable scale.

4.1. Timing and memory experiments for AC and AC(2).

How running time scales with problem size. Our overall benchmarking data set includes hundreds of different instances. We describe these instances in detail in Section 5. In Figure 4.1, we show plots that display the running time of our AC and AC(2) solvers plotted against nonzero count for all our instances. We report total running time including factorization and solve in seconds.

Analysis. In Figure 4.1, we also plot the function $10^{-8}\,\mathrm{nnz}\log_{10}(\mathrm{nnz})^3$. This corresponds to our conjectured asymptotic running time (see Section 6 for an in-depth discussion) when using an $O(1)$ time algorithm per edge sample (such as the alias method). We find that overall, observed performance is consistent with this conjectured scaling behavior. However the largest solves take longer for AC(2). This is possibly due to garbage collection issues, c.f. the discussion of memory usage in the next subsection. We caution that (a) we do not have a theoretical analysis that establishes such a running time for AC or AC(2) and (b) our implementations use an edge sampler with $\log m$ worst-case time per edge sample.

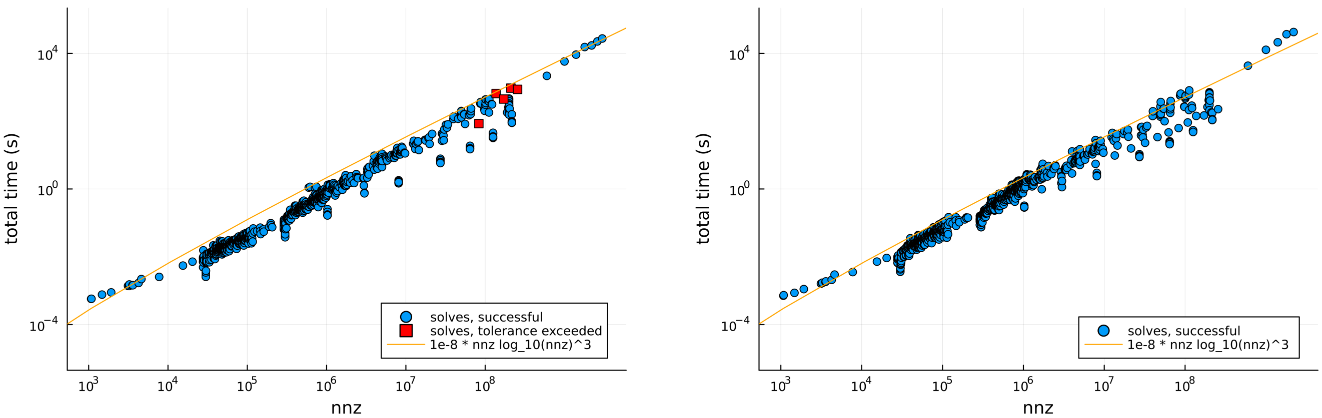


Fig. 4.1: The plot shows data from all our experiments with AC (left) and AC(2) (right) with total running time in seconds on the y-axis plotted against the number of nonzeros of the instance on the x-axis on a log-log scale. Solves that reached the target tolerance are shown with a blue circle, while solves that failed to reach tolerance are shown with a red square. The only failed solves occurred with AC on Sachdeva stars. An orange line shows the function $\mathrm{nnz}\log_{10}(\mathrm{nnz})^3$ (seconds) where nnz is the nonzero count.

What is the memory and time consumption of AC and AC(2) on very large instances. As we show later in Table 4.3, on difficult instances, the fill-in of AC is usually below 2.6 and of AC(2) usually below 3.6. Using these numbers we estimate the memory consumption of AC and AC(2). Given an input graph with n vertices and m edges, we get $N = 2m$ nonzeros in the adjacency matrix, we can estimate memory requirements as follows.

Input matrix: The input is stored in CSC (compressed sparse column format), which means we store one adjacency list per vertex, and for neighbor of the vertex, we store two 64-bit numbers = 16 bytes, (an integer index and a double float for the value). Altogether, this uses $4n + 16N$ bytes.

Intermediate graph: We use a format similar to CSC to store the intermediate matrix. When using k multi-edges, this format requires $32n + 24N \cdot k$ bytes. This intermediate graph is updated in-place.

Cholesky factor: We use a slightly non-standard format for the Cholesky factor output. However, the storage required equals that of a conventional Cholesky factor: One float and one integer index per off-diagonal entry in the row/column being eliminated. We again use 64-bit numbers for each field. We also store one additional field per column. Altogether, when fill-in is f, it requires $4n + 8N \cdot f$ bytes.

Approximate priority queue: We also use an approximate priority queue to enable adaptive greedy ordering of the vertices. This data structure requires $(16k + 32)n$ bytes.

Total space: These allocations are the main consumers of space beyond smaller edge sampling data structures and local variables, and combined they require $(16k + 72)n + (16 + 24k + 8f) \cdot N$ bytes.

To test the above estimates, we used AC and AC(2) to solve linear equations in 3D anisotropic Poisson SDDM matrices with nonzeros N in a range from $0.6 \cdot 10^9$ to $3 \cdot 10^9$ and equal side lengths, and $n \approx N/6$. Results are shown in Table 4.1.

For $N \approx 2.2 \cdot 10^9$, we get a fill-in of 2.62 for AC and of 3.79 for AC(2). This yields memory consumption estimates for our core data structures of 155 GB for AC and 229 GB for AC(2). We observe that during our build phase the total memory allocations including various smaller arrays (which are not reused) is 507 GB for AC and 631 GB for AC(2), which is 3.3 and 2.8 times our estimate for the core data structures respectively. We get a total time of 7.81 μs/nnz for AC and 19.4 μs/nnz for AC(2). We find that further increasing the grid size to $N \approx 2.6 \cdot 10^9$ causes AC(2) to slow down catastrophically on account of memory usage. In contrast, AC continues to work well throughout our largest experiment size of $N \approx 3 \cdot 10^9$, on account of lower memory consumption.

Analysis. The discrepancy between predicted and actual memory consumption appears to arise from repeated memory allocations for temporary variables that are allocated to the heap instead of stack by Julia. We expect a more careful implementation could essentially achieve our target of $(16 + 24k + 8f) \cdot N$ bytes for AC(k). Further savings could be made (a) by only keeping two of the three key data structures in memory at a time (albeit requiring the input matrix to be loaded twice from storage instead of once) and (b) switching to 32-bit integers for indexing vertices, as long as at most $n = 2 \cdot 10^9$ variables are used, or 48-bit integers otherwise. Combining these savings, with $k = 2$ and $f = 4$, we then estimate roughly $(14 + 22k + 7f)N = 86N$ bytes should suffice. This should allow solving instances with nonzero count $N \approx 6 \cdot 10^9$ using 512 GB RAM.

We see in Table 4.1 that fill-in and solve time per nonzero are growing slowly, while total time per nonzero grows faster. This tells us that the build time per nonzero is growing, which may be due to garbage collection or other issues, such as decreasing memory locality. The slowly growing solve time per nonzero suggests that our algorithm continues to produce a good preconditioner at large sizes. It may also be that our log(degree) time-per-sample approach is starting to have an impact at these scales, and switching to the alias method with constant expected time per sample could help. We believe that more careful memory management and removing the reliance on the Julia garbage collection could alleviate the slowing build-time issue.

nonzeros	AC					AC(2)				
nnz	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	build memory	solve memory	fill-in	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	build memory	solve memory	fill-in
(M)	(μs)	(μs)	(GB)	(GB)		(μs)	(μs)	(GB)	(GB)	
599	1.84	3.63	126	96.9	2.58	2.35	7.23	154	79.3	3.68
1000	1.92	5.78	231	164	2.6	2.41	12.8	289	130	3.73
1400	1.95	6.57	326	226	2.61	2.48	15.2	403	179	3.76
1800	2.4	8.56	419	295	2.62	2.91	20.4	525	230	3.77
2200	2.04	7.81	507	360	2.62	2.54	19.4	631	281	3.79
2600	2.09	8.66	610	427	2.63	N/A	N/A	N/A	N/A	N/A
3000	2.18	9.17	691	518	2.63	N/A	N/A	N/A	N/A	N/A

Table 4.1: Large scale memory usage experiment for AC and AC(2), on anisotropic coef. Poisson grid with variable weights.

How performance varies across repeated runs of AC and AC(2). We run tests to assess the variability in the performance of AC(2) and AC, with results shown in Table 4.2. We run these repeated experiments on a few matrices, chosen because they were particularly challenging for our solvers. We evaluate the variation in the performance of these solvers with a weighted Chimera with 10 million variables, a weighted SDDM Chimera with 10 million variables, the Sachdeva star with parameter $k = 700$, high contrast coefficient Poisson grid with 200 million nonzeros and axis interval 32, and Anisotropic coefficient Poisson grid with fixed discretization and weight 0.001. For each system of linear equations, we run each solver 10 times.

Analysis. These experiments show that our solvers' solve time and total time is tightly concentrated, with the only exception occurring with AC on Sachdeva stars. This is because AC failed to produce reliable factorizations for Sachdeva stars. In comparison, AC(2) still manages to exhibit a very small variance of runtime on Sachdeva star.

Instance	nonzeros	AC $t_{\text{solve}}/$ nnz			AC(2) $t_{\text{solve}}/$ nnz			AC $t_{\text{total}}/$ nnz			AC(2) $t_{\text{total}}/$ nnz		
	nnz	median	0.75	max	median	0.75	max	median	0.75	max	median	0.75	max
	(M)	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs
Weighted chimera	54.6	1.43	1.51	1.6	1.37	1.44	1.56	2.7	2.78	2.91	3.86	3.97	4.28
Weighted SDDM chimera	54.2	1.38	1.43	1.65	1.26	1.31	1.37	3.02	3.16	3.36	4.02	4.21	4.32
Sachdeva star	172	4.04	4.48*	4.55**	0.198	0.201	0.209	4.46	4.9 *	4.97**	0.832	0.836	0.846
High contrast coefficient Poisson grid	200	1.71	1.73	1.77	2.14	2.15	2.25	2.6	2.62	2.66	3.34	3.35	3.45
Anisotropic coef. Poisson grid, variable weight	200	1.12	1.14	1.18	1.15	1.16	1.2	1.72	1.75	1.82	2.33	2.34	2.38

Table 4.2: Variation in total time and solve time of AC and AC(2) computed across 10 runs per system of linear equations.

4.2. Design choices in AC(k): The impact of splitting/merging and elimination order. In this section, we run experiments to analyze the impact of two important design choices in our algorithm: the splitting and merging of multi-edges, and the elimination ordering. We run these experiments on the same five matrices we used for the performance variability experiments in Table 4.2, which we selected as they were particularly challenging for our solvers.

How AC(k) splitting and merging impacts fill-in and preconditioner quality. The two main algorithms analyzed in this paper are AC and AC(2), i.e. AC(k) with split/merge parameter set to 1 or 2. To understand the impact of this parameter, we report detailed measurements of several quantities across five difficult instances: For each instance, we report fill-in of AC and AC(2), i.e. the ratio of the size of our approximate Cholesky factor to the size of the original input matrix. We also show the total solve time measured per nonzero entry, the number of iterations used to solve a linear equation in the matrix to the target tolerance, and the relative condition number of the input matrix and the preconditioner computed by AC and AC(2). The results are shown Table 4.3.

Instance	nonzeros	AC				AC(2)			
	nnz	t_{total}/ nnz	N_{iter}	fill-in factor	Condition num.	t_{total}/ nnz	N_{iter}	fill-in factor	Condition num.
	(M)	μs				μs			
Weighted chimera	54.6	2.88	26	2.5	14.1	3.9	19	3.45	7.81
Weighted SDDM chimera	54.2	3.05	24	2.42	11.3	4.26	19	3.27	8.18
Sachdeva Star	172	2.2 *	408	1.0	9660.0	0.84	44	1.0	64.6
High contrast coefficient Poisson grid	200	2.24	60	2.54	80.2	3.44	45	3.57	40.6
Anisotropic coef. Poisson grid, variable weight	200	1.78	39	2.43	27.2	2.33	26	3.33	11.5

Table 4.3: Comparison between AC and AC(2): We measure total solve time per nonzero, iteration count, fill-in factor (i.e. nonzero count in the output relative to the input), and condition number. We used five of our most difficult instances for this comparison table.

Analysis. The measurements shed light on why AC usually outperforms the more reliable AC(2), while the latter wins out on a few difficult instances. We see that AC(2) builds a preconditioner with much smaller relative condition number on difficult instances, but that AC has a significantly lower fill-in factor. The trade-off usually works out in favor of AC, but on one matrix class AC fails to reach the target tolerance, namely the *Sachdeva stars*. We see that for this class, AC(2) works well. On other classes, the higher sampling rate of AC(2) increases fill-in by a factor roughly 1.5, while also yielding an improved condition number, which tends to reduce iteration count, also by a factor slightly less than 1.5. Overall, this yields similar performance of AC and AC(2) on these difficult instances. On easier instances, AC is generally faster than AC(2) by a factor 1.5-2, c.f. Table 5.1. In Supplementary Material Section SM5, we conduct evaluations across a broader set of parameters, i.e. other choices of 'split' and 'merge' parameters, but we find that AC and AC(2) outperform other choices.

Elimination ordering: approximate-degree-greedy vs. random. We compare our elimination ordering, which is an approximate greedy ordering on actual degrees during elimination, to the random elimination ordering suggested by Kyng-Sachdeva [46] when they first introduced approximate Gaussian elimination.[4] We compare AC with a variant that uses a uniformly random elimination order, which we call AC-random. We note that our approximate minimum-degree ordering is *not* equivalent to conventional greedy or approximate minimum degree orderings for Gaussian elimination, as we use a greedy approach based on the actual (unweighted) degrees realized by our algorithms.

In our first experiment, we reuse the setup from our performance variability experiments (see Table 4.2), i.e. we run the AC-random solver 10 times on each of 5 different systems of linear equations and we report the median, 75th percentile, and maximum solve time, normalized by nonzero count, to compare it with our results for AC. The results of this experiment are shown in Table 4.4.

Instance	nonzeros	AC t_{solve}/ nnz			AC-random t_{solve}/ nnz			AC t_{total}/ nnz			AC-random t_{total}/ nnz		
	nnz	median	0.75	max	median	0.75	max	median	0.75	max	median	0.75	max
	(M)	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs	μs
Weighted chimera	54.6	1.43	1.51	1.6	10.8	10.9	12	2.7	2.78	2.91	11.3	11.5	12.6
Weighted SDDM chimera	54.2	1.38	1.43	1.65	10.5	10.9	12.4	3.02	3.16	3.36	11.5	11.9	13.5
Sachdeva Star	172	4.04	4.48*	4.55**	0.878	1.27	1.49*	4.46	4.9 *	4.97**	1.07	1.45	1.68*
High contrast coefficient Poisson grid	200	1.71	1.73	1.77	8.75	9.08	9.74	2.6	2.62	2.66	9.28	9.62	10.3
Anisotropic coef. Poisson grid, variable weight	200	1.12	1.14	1.18	4.36	4.45	4.6	1.72	1.75	1.82	4.92	5.02	5.14

Table 4.4: Variation in total time of AC and AC-random computed across 10 runs per system of linear equations.

In our second experiment, we reuse the setup from our fill-in and preconditioning quality experiments (see

[4] As an alternative, Kyng-Sachdeva also suggested a random choice among low-degree vertices, which is closer to our approach.

Table 4.3), i.e. we run the AC-random solver once on each of 5 different systems of linear equations and compute the total solve time per nonzero, the ratio of sizes between the input and the output factorization, and the condition number. The results of this experiment are shown in Table 4.5.

Instance	nonzeros	AC				AC-random			
	nnz	$t_{\text{total}}/$ nnz	N_{iter}	fill-in	Condition num.	$t_{\text{total}}/$ nnz	N_{iter}	fill-in	Condition num.
	(M)	μs				(μs)			
Weighted chimera	54.6	2.88	26	2.5	14.1	11	134	3.25	491
Weighted SDDM chimera	54.2	3.05	24	2.42	11.3	12.9	145	3.62	891
Sachdeva Star	172	2.2 *	408	1.0	9660.0	1.21	215	1	4530
High contrast coefficient Poisson grid	200	2.24	60	2.54	80.2	9.88	140	3.15	684
Anisotropic coef. Poisson grid, variable weight	200	1.78	39	2.43	27.2	4.63	66	3.3	108

Table 4.5: Comparison between approximate Cholesky factorization with different elimination orders.

Analysis. In [46], the random ordering plays a role in the proof that the algorithm produces a good factorization, and thus it is interesting to measure how much randomness is needed, if any. We find that our adaptive approximately minimum degree vertex outperforms random ordering. In both experiments, we see that AC outperforms AC-random except on Sachdeva stars, where AC does worse on all parameters. This is not so surprising, as the Sachdeva star is designed to challenge the greedy ordering approach in particular. Concretely, eliminating the central vertex of the Sachdeva star early tends to give worse performance, and the parameters we use for the star are such that the greedy ordering eliminates the central vertex first. The results suggest that our greedy ordering approach generally works well, but perhaps could be improved slightly by adding a little more randomness to the elimination ordering. In Subsection 4.3, we show further experiments comparing our elimination ordering with the given ordering.

4.3. Contrasting incomplete factorizations. Our algorithm can be viewed as a new approach to incomplete Cholesky factorization. There is a long line of research on incomplete factorizations and in this section, we compare with two important families of incomplete factorizations, namely IC(k) (i.e. symmetric matrix ILU(k) c.f. [58]) and ILUT [57]. These methods both give ways to trade off between fill-in and preconditioner quality, and we compare their trade-offs with the trade-offs obtained by AC(k). We also compare with Kyng-Sachdeva [46], which is a provably nearly-linear time solver for SDD matrices based on an incomplete factorization that motivated our approach. Finally, we compare with RCHOL, an alternative implementation of our algorithm.

AC(k) gives us a tunable way to improve preconditioner quality by increasing fill-in (using more samples). Similarly, it is possible to use IC(k) with increasing level-of-fill parameter k to improve preconditioner quality by increasing fill-in. ILUT comes with two parameters (maximum fill-in and relative entry thresholds, see [57]) that we can set to obtain different trade-offs between fill-in and preconditioner quality. We use the ITSOL implementation of ILUT for these experiments.

How we compare to IC(k) and ILUT: measuring fill-in and preconditioner quality. We run experiments to compare the trade-off achieved by AC(k), in particular AC and AC(2), with the trade-off achieved by IC(k) and ILUT. We study the performance of ILUT with parameters similar to those suggested by Saad ([57] Section 6.1). We defer to Saad for a description of the parameters. The results on five test problems is shown in Figure 4.2. Again, we use the same five test problems as for our performance variability experiment Table 4.2. In addition, we add one more test problem, McRae/ecology2 from SuiteSparse. For the high-contrast Poisson problem, Figure 4.2(c)), there is no data for AC(2) with the given ordering, as this required more memory than our machine has.

Analysis. Across the test problems, the Sachdeva star (Figure 4.2(e)) stands out because AC performs poorly, while AC(2) and IC(k) both achieve few PCG iterations at relatively low fill-in. For the other problems, AC(2) consistently lowers the iteration count compared to AC, albeit with a moderate increase in fill-in. IC(k) follows a similar trend to AC(k), but generally requires more PCG iterations at every fill-in level, and even at large k (with high fill-in), it fails to match AC's PCG iteration count. Notably, for the weighted Chimera (Figure 4.2(a)) problem, IC(k) does not improve monotonically with increasing k, indicating that more fill-in does not always help IC(k) on that system. We find that for some parameter choices ILUT performs well on the anisotropic coefficient Poisson grid problem and on the SuiteSparse McRae/ecology2 problem, but it fails to converge on several other problems.

How does our ordering impact fill-in vs preconditioner quality.. For IC(k), we use the given elimination ordering, as no other ordering is consistently preferred, to the best of our knowledge [58]. To understand the role of our elimination ordering in the performance of AC and AC(2), we contrast these with the same algorithm but using the given ordering. The results are also shown in Figure 4.2.

Analysis. We find that AC(k) with the given ordering is more robust than IC(k), outperforming the latter on the weighted Chimera (Figure 4.2 (a)) and McRae/ecology2 (Figure 4.2 (f)). But, for the problems (b)-(d) in the fill-in vs. iteration count trade-off for AC(k) (given ordering) and IC(k) are quite similar. This shows our elimination ordering plays an important role in further improving the performance of AC(k). Further, we see that for problems (a)-(c) in Figure 4.2, not only does our ordering reduce the fill-in, it also reduces iteration counts. This does not fit neatly with the theory developed by Kyng-Sachdeva [46]. This phenomenon warrants further investigation, and perhaps a theoretical explanation.

How we compare to Kyng-Sachdeva. We compare the performance of our solvers AC and AC(2) to an implementation of the simple solver by Kyng-Sachdeva [46] provided in Laplacians.jl, henceforth referred to as KS16. Results are summarized in Table 4.6.

Analysis. We see that with the recommended parameter choice, the solver takes roughly 5-10 times longer than AC and AC(2) on most problems, with the exception of the Sachdeva star, where it converges successfully, unlike AC, but needs twice the time of AC(2). The Kyng-Sachdeva algorithm has a substantially higher sampling rate than AC and AC(2), but achieves a better relative condition number.

How we compare to RCHOL. We compare the performances of our solvers AC and AC(2) to RCHOL [20]. RCHOL is another implementation of our approximate Cholesky factorization procedure.[5] We reuse the setup from our performance variability test (see Table 4.2), i.e. we run the RCHOL solver 10 times on each of 5 different systems of linear equations and we report the median, 75th percentile, and maximum solve time, normalized by nonzero count, to compare it with our results for AC and AC(2). Results are summarized in Table 4.7. We note that the performance of RCHOL is worse than the AC and AC(2) implementations across all instances.

Instance	nonzeros	AC $t_{\text{total}}/$ nnz			AC(2) $t_{\text{total}}/$ nnz			KS16 $t_{\text{total}}/$ nnz		
	nnz	median	0.75	max	median	0.75	max	median	0.75	max
	(M)	μs	μs	μs	μs	μs	μs	μs	μs	μs
Weighted chimera	54.6	2.7	2.78	2.91	3.86	3.97	4.28	27.1	27.8	29.7
Weighted SDDM chimera	54.2	3.02	3.16	3.36	4.02	4.21	4.32	21.9	23	24.2
Sachdeva Star	172	4.46	4.9 *	4.97**	0.832	0.836	0.846	1.76	1.77	1.8
High contrast coefficient Poisson grid	200	2.6	2.62	2.66	3.34	3.35	3.45	17.3	19.3	20.5
Anisotropic coef. Poisson grid, variable weight	200	1.72	1.75	1.82	2.33	2.34	2.38	17.8	17.9	18.4

Table 4.6: Total solve times for AC, AC(2), and KS16 solver implemented in Laplacians.jl.

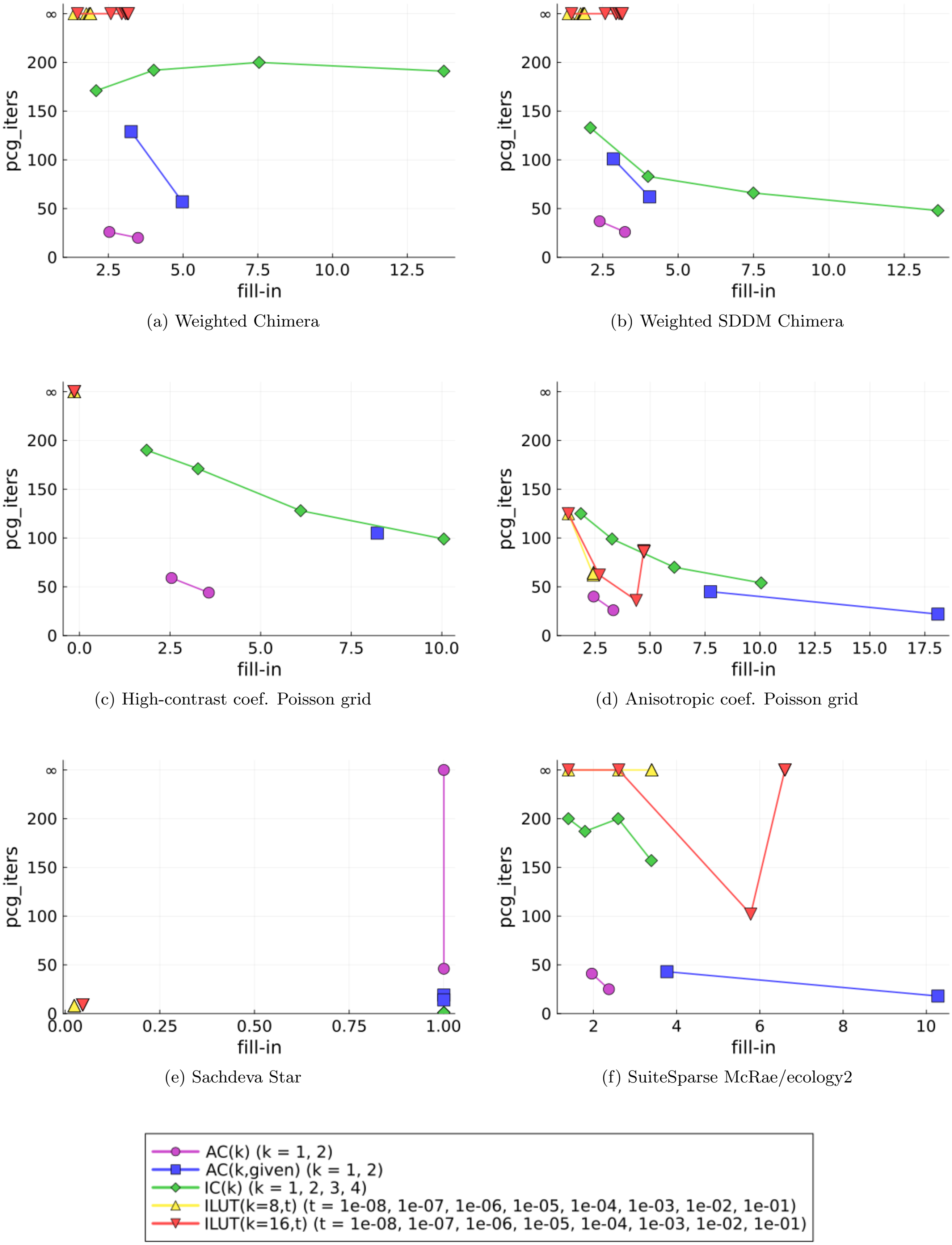
Instance	nonzeros	AC $t_{\text{total}}/$ nnz			AC(2) $t_{\text{total}}/$ nnz			RCHOL $t_{\text{total}}/$ nnz		
	nnz	median	0.75	max	median	0.75	max	median	0.75	max
	(M)	μs	μs	μs	μs	μs	μs	μs	μs	μs
Weighted chimera	54.6	2.7	2.78	2.91	3.86	3.97	4.28	Inf	Inf	Inf
Weighted SDDM chimera	54.2	3.02	3.16	3.36	4.02	4.21	4.32	9.9	10.3	10.6
Sachdeva Star	172	4.46	4.9 *	4.97**	0.832	0.836	0.846	Inf	Inf	Inf
High contrast coefficient Poisson grid	200	2.6	2.62	2.66	3.34	3.35	3.45	5.85	5.93	6.04
Anisotropic coef. Poisson grid, variable weight	200	1.72	1.75	1.82	2.33	2.34	2.38	2.29	2.32	2.4

Table 4.7: Total solve times for AC, AC(2), and RCHOL.

5. Solver Comparisons on a Broad Set of Benchmark Data. In this section, we present our final, largest set of experiments. These focus on measuring the single-threaded performance of our solvers and comparing them with the popular SDDM linear equation solvers across a wide range of problems. We use the same experimental setup as in Section 4.

Which solvers do we compare to. 1. Combinatorial Multigrid (CMG), 2. Hypre's Krylov Solver with BoomerAMG preconditioning (Hypre), 3. PETSc's Krylov Solver with BoomerAMG preconditioning (PETSc), 4. MATLAB preconditioned conjugate gradient using the ichol implementation of Incomplete Cholesky factorization (ICCG), 5. preconditioned conjugate gradient using the Hypre implementation of Incomplete Cholesky factorization with

[5] While we use an edgewise representation of the Cholesky factor, RCHOL uses a more standard column format.



(a) Weighted Chimera

(b) Weighted SDDM Chimera

(c) High-contrast coef. Poisson grid

(d) Anisotropic coef. Poisson grid

(e) Sachdeva Star

(f) SuiteSparse McRae/ecology2

Fig. 4.2: Comparing fill-in and PCG iteration count for several solvers: (i) AC(k) with our default ordering, (ii) AC(k) with the given ordering, (iii) IC(k) with the given ordering, (iv) ILUT with the given ordering. For each solver, we try a range of parameters, as specified in the legend.

level-of-fill = 2 (IC(2)CG), 6. Lean Algebraic Multigrid (LAMG)[6]. Of the third-party solvers we test, i.e. CMG, Hypre, PETSc, ICCG/IC(2)CG, and LAMG, the latter three seem to be dominated in performance by CMG or Hypre, with few exceptions, but ICCG exhibits greater reliability. Consequently, our discussion will focus on comparing AC and AC(2) with CMG, Hypre, and ICCG/IC(2)CG. Among the possible level-of-fill choices for Incomplete Cholesky, we choose ICCG (level-of-fill = 0) because it is the most common, and IC(2)CG because it typically has fill-in comparable to AC(k), as we show in Figure 4.2.

Which instances do we compare on. We test four main categories of SDDM matrices. 1. All large SDDM matrices in SuiteSparse. 2. Programmatically generated matrices, which we break into three subgroups: (a) "Chimeras": a broad class of matrices that we generate programmatically to test solvers on a wide array of graph geometries. (b) "Sachdeva stars": A special family of matrices designed to challenge our algorithm[7]. 3. "Flow IPM" matrices: We run an interior point method for solving maximum flow problems whose Newton step updates lead to Laplacian linear equations, and we solve these equations. We use Chimera graphs and *Spielman graphs*, a family of graphs that are thought to be particularly challenging for interior point methods, for our maximum flow instances. 4. Discretizations of partial differential equations on 3D grids: (a) Matrices based on fluid simulations from a Society of Petroleum Engineering (SPE) benchmark. (b) Poisson problems on 3D grids with uniform, high-contrast coefficients, or anisotropic coefficients. The matrices we use have up to 256 million nonzeros.

Summary of results. AC(2) is able to converge in a reasonable time on every single test, while AC is typically faster but fails to converge on large Sachdeva stars. On general graphs, our solvers achieve a worst-case running time of about 7.4 μs per nonzero entry, while on grid graphs we get a worst-case running time of about 3.6 μs per nonzero. This should be contrasted with all other solvers we test, which entirely fail on some SuiteSparse matrices or Chimeras, but achieve a worst-case running time on grid graphs of about 1.27 μs per nonzero (CMG) and 0.77 μs per nonzero (Hypre). Our experiments show that AC and AC(2) obtain good practical performance across many different types of SDDM-matrices. Importantly, AC and AC(2) show significantly greater reliability than existing solvers. Our test results are summarized in Table 5.1, and they are included in Figure 4.1.

Table 5.1: Summary of worst case total solve time per nonzero matrix entry across all our comparison experiments.

Instance family	AC	AC(2)	CMG	Hypre	PETSc	ICCG	IC(2)CG
	$t_{\text{total}}/$ nnz		$t_{\text{total}}/$ nnz				
	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
SuiteSparse Matrix Collection	0.994	1.4	371	Inf	27.1	1450 *	269
Chimeras	4.33	7.35	Inf	Inf	N/A	Inf	Inf
Sachdeva stars	4.76 **	1.04	1.18	3.52 **	7.22**	1.02	1.33
Maximum flow IPMs	1.2	2.39	Inf	5.04 *	N/A	22.1 *	28
SPE benchmark	1.62	1.91	1.1	0.648	2.58	6.13	3.01
Poisson grid problems	2.33	3.57	1.27	0.772	4.92	6.56	3.8

* relative residual error exceeded tolerance 10^{-8} by a factor $(1, 10^4]$.
** relative residual error exceeded tolerance 10^{-8} by a factor $(10^4, 10^8)$.
Inf: Solver crashed or returned a solution with relative residual error 1 (error of the trivial all-zero solution).
N/A: Experiment omitted as the solver crashed too often with (only occurred for PETSc).

5.1. Detailed comparisons across matrix classes. We now describe the experiments summarized in Table 5.1 in detail.

Linear equation systems used for experimental evaluation. Our instances are divided into four categories:

1. Matrices from the SuiteSparse Matrix Collection which are SDDM or approximately SDDM.
2. Programmatically generated SDDM matrices. (a) Chimera graph Laplacians and variants that are strictly SDDM. (b) Sachdeva star graph Laplacians.
3. Laplacian matrices arising from solving maximum flow problems using an interior point method.

[6]We omit LAMG from our result tables, as we found it unable to convergence to our target tolerance across most matrix families we tested.

[7]We thank Sushant Sachdeva for suggesting this graph construction.

4. Discretizations of partial differential equations on 3D grids: (a) Matrices based on fluid simulations in a Society of Petroleum Engineering (SPE) benchmark. (b) Uniform coefficient Poisson problems on 3D grids. (c) Variable coefficient Poisson problems on 3D grids with a checkerboard pattern using variable resolution and fixed weight. (d) Anisotropic coefficient Poisson problems on 3D grids with variable discretization and fixed weight. (e) Anisotropic coefficient Poisson problems on 3D grids with fixed resolution and variable weight.

The SuiteSparse matrix collection. In this section, we report the performances of the solvers on matrices from the SuiteSparse Matrix Collection[8], previously known as the University of Florida Sparse Matrix Collection [24]. In addition to the SDDM matrices in SuiteSparse Matrix Collection, we also included matrices that are "approximately" SDD, in the sense that they have positive diagonals, nonpositive off-diagonals, and are symmetric, while not diagonally dominated but close. In particular, for such an $n \times n$ matrix M, if $\forall i \in [n]$, such that $(M\mathbf{1})_i < 0, |(M\mathbf{1})_i / M_{ii}| \leq \epsilon$ for some small value ϵ, then we say M is "approximately" SDDM. In other words, for these non-SDDM matrices, we also included those satisfying $\left|\min_{i\in[n]}(M\mathbf{1})_i/M_{ii}\right| \leq \epsilon$ and we call this quantity the *SDDM-nearness*. In our experiments, we set this ϵ to be ten times the machine epsilon. In particular, we included the following non-SDDM matrices with SDDM-nearness roughly 10^{-16}: McRae/ecology1, McRae/ecology2, HB/nos7. Similarly, in our experiments, we treat some of the SDDM matrices as approximately Laplacian. If $\max_{i\in[n}(M\mathbf{1})_i/M_{ii} \leq \epsilon$, then we say M is approximately Laplacian. If the tolerance ϵ is small enough, then solving the system while discarding the excess on the diagonals is sufficient for solving the system in M. We excluded matrices with less than 1000 nonzeros so that the overheads are negligible. We also excluded the matrix Cunningham/m3plates, which is a diagonal matrix with some zeros on the diagonal. This matrix significantly slowed down some of the solvers, but not AC and AC(2). In total, we tested the solvers on 28 matrices from the SuiteSparse Matrix Collection. We observe that our solvers also work well on the "approximately" SDDM matrices.

Results and analysis. Results are shown in Table 5.2. Only AC and AC(2) performed well across all the tested matrices from the SuiteSparse Matrix Collection. The nonzero count of tested matrices ranges from 1,000 to 5 million, while the number of variables ranges from 675 to 1 million.

Table 5.2: SuiteSparse Matrix Collection Time / nnz

AC t_{total}/ nnz			AC(2) t_{total}/ nnz			CMG t_{total}/ nnz		
median	0.75	max	median	0.75	max	median	0.75	max
(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
0.434	0.503	0.994	0.553	0.698	1.4	20.1	115	371

Hypre t_{total}/ nnz			PETSc t_{total}/ nnz			ICCG t_{total}/ nnz			IC(2)CG t_{total}/ nnz		
median	0.75	max	median	0.75	max	median	0.75	max	median	0.75	max
(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
0.0586	0.345	Inf	1	1.83	27.1	80.8	275	1450*	0.972	16.9	269

[8]https://sparse.tamu.edu

Table 5.3: Chimeras Time / nnz

variables	# instances	AC $t_{\text{total}}/\text{nnz}$			AC(2) $t_{\text{total}}/\text{nnz}$			CMG $t_{\text{total}}/\text{nnz}$		
n		median	0.75	max	median	0.75	max	median	0.75	max
(K)		(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
10	412	0.38	0.412	0.656	0.541	0.621	1.24	8.03*	10.9 *	30.9*
100	431	0.607	0.738	1.87	0.936	1.28	3.18	1.67*	2.36*	Inf
1000	98	1.44	1.63	2.39	1.72	2.28	3.58	1.38	2.17*	Inf
10 000	36	3.02	3.69	4.33	3.78	5.46	7.35	2.16*	3.83*	Inf

variables	# instances	Hypre $t_{\text{total}}/\text{nnz}$			PETSc $t_{\text{total}}/\text{nnz}$			ICCG $t_{\text{total}}/\text{nnz}$			IC(2)CG $t_{\text{total}}/\text{nnz}$		
n		median	0.75	max	median	0.75	max	median	0.75	max	median	0.75	max
(K)		(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
10	412	0.895	1.25	4.75**	N/A	N/A	N/A	8.62	11.4	24.8*	1.21	1.8	36.7*
100	431	1.2	1.81	6.24**	N/A	N/A	N/A	2.12	4.01	Inf	2.16	4.82	Inf
1000	98	2.76	4.05	15.4	N/A	N/A	N/A	3.37	6.83	Inf	6.04	12.5	160
10 000	36	5.92	9.93	Inf	N/A	N/A	N/A	3.46	14.9	48.4*	12.2	18.5	Inf

"Chimera" graph Laplacian and SDDM problems. In this section, we study Laplacian and SDDM systems of linear equations with a wide range of nonzero structures. We first focus on graph Laplacians arising from a class of recursively generated graphs which we call *Chimeras.*

Chimeras are formed by randomly combining standard base graphs. We generate Chimera graphs with a given vertex count by a recursive process which uses a seeded pseudo-random generator to pseudo-randomly choose among the following options.

(a) Form a graph from one of several classes: paths, trees, 2D grids, rings and generalized rings, the largest connected component of an Erdős-Renyi graph, random regular graphs, and random preferential-attachment-like graphs.
(b) Recursively form two smaller Chimeras and combine them in one of several ways: by adding random edges between them, or forming their Cartesian graph product, or forming a 'generalized necklace' from the two graphs. We define a generalized necklace of two graphs G and H as the result of expanding each vertex in G to an instance of H, and for each edge in G adding a number of random edges between the corresponding copies of H in the new graph.
(c) Select one of two operations that create a larger graph from a single smaller graph. The first operation is a pseudo-random two-lift, which doubles the vertex set and replace each edge with either a pair of internal edges internal to the original set and the copy or a pair crossing edges between the original and the copy. The second operation is 'thickening,' which adds a subset of the current 2-hop paths as new edges in the graph.

We consider both unweighted Chimeras and weighted variants. Weights are obtained by a pseudo-random process that first either chooses uniform edge weights in $[0, 1]$ or chooses random potentials to assign vertices and then sets the weights of edges to the differences between the potentials at the endpoints. These potentials can either be uniform in $[0, 1]$, or the result of multiplying such vertex potentials by a small power of the normalized Laplacian matrix or the corresponding walk matrix. Finally, with probability one half, all the edge weights are replaced by their reciprocals.

For each vertex count n, when we report statistics for C different instances, these are always the Chimeras with n vertices and seed indices $i = 1, \ldots, C$. We always choose the Chimeras generated by the first seed indices, and we do not exclude any Chimeras. Our Chimera generator is deterministic given the seed. When using Chimeras for benchmarking, we strongly encourage this approach, as arbitrary exclusions could give misleading statistics.

We also consider a variant of the graph Laplacian problem, where we modify the linear system by setting a boundary condition of $u(x) = 0$ for $|V|^{2/3}$ of the vertices $x \in V$. The boundary size of $|V|^{2/3}$ was chosen to get a similar boundary size as in the 3D cube examples. We enforce the boundary condition at vertices with index divisible by $|V|^{1/3}$, which spreads the boundary vertices across the graph. Expanding these tests to include 'adversarially' chosen boundary conditions remains an interesting open question. This modification results in a strictly SDDM linear equation.

Table 5.4: Sachdeva star: A star graph with $(k/2)$ leaves replaced by the complete graph on k vertices

k	nonzeros	variables	AC				AC(2)				CMG			
	nnz	n	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.
	(K)	(K)		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
200	4000	20	167	0.682	0.931	0.955	37	0.155	0.716	0.877	7	0.149	0.487	0.001 02
400	32 000	80	459	2.12	2.44	0.967	40	0.186	0.718	0.985	7	0.12	0.41	0.001 97
600	108 000	180	773	3.55	3.93	0.997	45	0.208	0.849	0.741	7	0.117	0.395	0.002 51
800	256 000	320	669	3.07	3.39	32.9	45	0.207	0.883	0.956	7	0.116	0.386	0.002 74

k	nonzeros	variables	Hypre				PETSc				ICCG				IC(2)CG			
	nnz	n	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.	N_{iter}	$t_{\text{solve}}/$ nnz	$t_{\text{total}}/$ nnz	res. err.
	(K)	(K)		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
200	4000	20	100	0.692	1.4	2.87×10^3	2	0.0376	1.93	2.09×10^4	1	0.218	0.384	1.27×10^{-6}	1	0.01	0.36	1.06×10^{-7}
400	32 000	80	100	0.842	2.22	1.54×10^3	1	0.0394	3.69	1.2×10^4	1	0.195	0.366	1.54×10^{-5}	1	0.009 37	0.68	1.44×10^{-7}
600	108 000	180	47	0.396	2.42	0.562	2	0.0572	5.47	9.17×10^3	1	0.236	0.451	2.36×10^{-5}	1	0.0131	1.07	1.77×10^{-7}
800	256 000	320	100	0.821	3.52	2.44×10^3	2	0.0388	7.22	8.43×10^3	1	0.281	0.542	5.54×10^{-6}	1	0.008 95	1.33	1.99×10^{-7}

Results and analysis. Results across all Chimera instances are shown in Table 5.3. Our experiments show that Chimeras are a particularly challenging problem because only AC and AC(2) successfully reached the desired tolerance 10^{-8} for all problem instances. Beyond this worst case behavior, our solver AC also achieved the best median runtimes, and 75th percentile runtime, beating Hypre and CMG and ICCG/IC(2)CG. The nonzero count in our tested Chimeras range from 27,000 to 122 million, with variable count ranging from 10,000 to 10 million.

Sachdeva star graph Laplacians.. In this section, we report the performance of solvers on a graph Laplacian specifically designed to make our ApproximateCholesky/ApproximateEdgewiseCholesky algorithm fail. The construction was suggested by Sushant Sachdeva. We construct a star graph with l leaves, and then replace each leaf with a complete graph on k vertices. In the experiments, we set l to be $\frac{k}{2}$. We then consider a linear equation in the associated graph Laplacian. An example is shown in Figure 5.1. The intuition is that because the center vertex of the star has relatively low degree, it gets eliminated early, and this creates high variance as every incident edge is important as it crosses between two otherwise disconnected components.

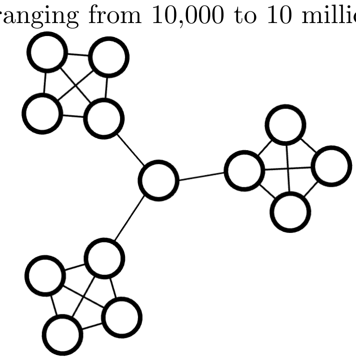


Fig. 5.1: Sachdeva star with $l = 3$ and $k = 4$.

Results and analysis. Results are shown in Table 5.4. We test on graphs with k up to 800, which leads to 256 million matrix nonzeros and 320,000 vertices. This is the most challenging problem for AC. The iteration counts of AC spiked when compared to other problems. This suggests that AC failed to produce reliable factorizations for the Sachdeva star Laplacians. In contrast, AC(2) produced much more reliable factorizations and the iteration counts are still comparable to other problems of similar sizes. We observe that CMG performs the best for this problem, and ICCG/IC(2)CG performed well too, while Hypre and PETSc failed to achieve the desired tolerance on most instances.

Interior point method matrices. In this section we report the performances of solvers on matrices generated from running an interior point method (IPM) on undirected maximum flow problems on two classes of graphs. Our maximum flow interior point solver is based on the [37] framework, and uses too many iterations to be practical. Hence, our main interest is to examine if the resulting linear equations are tractable using various solvers.

First, we tested the solvers using Laplacians generated from running a maximum flow IPM on unweighted Chimeras (see section 5.1 for details on Chimera graphs). We ran the IPM on five Chimeras with 10^5 vertices and with the generating seed index i ranging from 1 to 5. We compute and store a number of the Laplacian linear equations encountered during a run of the IPM. Concretely, the IPM has a homotopy parameter, which measures roughly how close we are to having an optimal maximum flow solution. As the algorithm progresses, the homotopy parameter changes and the currently maintained approximate maximum flow solution improves in quality. We collect matrices at five different stages, corresponding to different values of the homotopy parameter, and hence different current solution qualites. First, we collect matrices when the maximum flow solution is guaranteed error at most 10^{-1} by the homotopy parameter; then we collect when the guaranteed error is $10^{-2}, 10^{-3}, 10^{-4}$ and 10^{-5}. At each of these five collection stages, we store roughly 5 Laplacians corresponding to consecutive Newton steps in each precision range, for a total of 25 Laplacians per maximum flow problem.

In general, the weights in the Laplacian matrices will get increasingly extreme as the precision parameter decreases (meaning the error is smaller). Thus we might expect the problems to become increasingly difficult as the precision parameter decreases. Table 5.5 reports detailed performances of the solvers in each precision range,

Table 5.5: Chimera IPM

IPM param.	# instances	AC $t_{\text{total}}/$ nnz			AC(2) $t_{\text{total}}/$ nnz			CMG $t_{\text{total}}/$ nnz		
		median	0.75	max	median	0.75	max	median	0.75	max
(10^{-3})		(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
100	28	0.68	0.774	1.2	1.27	1.79	2.39	1.6	1.83	Inf
10	27	0.678	0.777	1.15	1.27	1.8	2.36	1.62	1.97	2.55
1	27	0.68	0.763	1.14	1.25	1.79	2.38	1.56	1.91	2.51
0.1	27	0.683	0.784	1.15	1.27	1.79	2.36	1.55	1.89	2.71
0.01	19	0.681	0.777	1.15	1.28	1.78	2.37	1.57	1.87	2.33

IPM param.	# instances	Hypre $t_{\text{total}}/$ nnz			PETSc $t_{\text{total}}/$ nnz			ICCG $t_{\text{total}}/$ nnz			IC(2)CG $t_{\text{total}}/$ nnz		
		median	0.75	max	median	0.75	max	median	0.75	max	median	0.75	max
(10^{-3})		(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)	(μs)
100	28	1.54	2.27	5.04	N/A	N/A	N/A	1.71	2.16	6.25	2.93	3.31	6.14
10	27	1.52	2.29	4.82	N/A	N/A	N/A	1.72	2.21	6.79	2.86	3.17	6.05
1	27	1.52	2.32	4.94	N/A	N/A	N/A	1.51	2.21	4.14	2.87	3.29	6.11
0.1	27	1.53	2.29	4.86	N/A	N/A	N/A	1.7	2.29	3.76	2.97	3.3	6.08
0.01	19	1.52	2.3	4.58	N/A	N/A	N/A	1.46	2.33	4	2.93	3.29	6.04

showing the median, 75th percentile, and worst case running times across the five Chimera instances we tested. AC, AC(2), CMG and ICCG/IC(2)CG reached the desired tolerance on all instances. AC is clearly the fastest solver across precision ranges, while the other solvers still managed a comparable run time. We omitted PETSc because we found it to be very unstable on these IPM-weighted Chimera Laplacians and frequently crashed.

In Supplementary Material Section SM6, we show additional experiments on using linear equation solvers on matrices from IPM Newton steps. These experiments investigate solver performance on matrices that arise from solving maximum flow problems on *Spielman graphs*, a family of graphs that is thought to be challenging IPM maximum flow solvers. These experiments find that performance of AC and AC(2) on *Spielman graph* IPM Newton steps is similar to the in the Chimera IPM Newton step case that we report here.

Results and analysis. Our experiments on matrices arising from maximum flow IPMs do not show that extreme weights coming from interior point methods yield much harder Laplacian linear equation instances than our unweighted Chimeras, at least for AC and AC(2). This suggests that practical Laplacian solvers may finally be ready for use in interior point methods. However, obtaining good performance for solving flow problems in practice using interior point methods remains a major open problem.

The SPE benchmark. In this section, we report the performance of solvers on SDDM matrices from the Society of Petroleum Engineering benchmark [22], a collection of discretized Poisson problems. These matrices are generated from a 3D waterflood of a geostatistical model. We sourced the data set from authors of [19]. The problem size ranges from 0.5 million to 16 million nonzeros.

Results and analysis. Results are shown in Table 5.6. On this problem, CMG and Hypre have particularly good performances and beat other solvers. On the other hand, ICCG performed poorly with very large iteration counts for the two largest problem instances. IC(2)CG does better, and is only slower than AC by a factor 2. The runtimes of AC(2) are very close to those of AC, but again, the SPE benchmark is not hard enough for AC(2) to out-perform AC.

5.2. Further Poisson problems on 3D domains. We consider a suite of experiments on programmatically generated 3D Poisson problems similar to Sundar et al. [61], which evaluated the performance of the Hypre-BoomerAMG algebraic multigrid on discretizations of several Poisson problems, also known as scalar elliptic diffusion problems. The nonzero counts in our Poisson grid experiments range from 2 million to 200 million, while variable counts range from 290,000 to 30 million.

We consider systems of linear equations arising from Poisson problems with Dirichlet boundary conditions on a domain Ω with a spatially varying scalar diffusion coefficient μ, and consider both the case of μ isotropic and anisotropic,

$$\begin{cases} \operatorname{div}(\mu(x)\boldsymbol{\nabla} u(x)) = f(x) & \text{for } x \in \Omega \\ u(x) = 0 & \text{for } x \in \partial\Omega. \end{cases} \tag{5.1}$$

Table 5.6: SPE Benchmark

nonzeros	variables	AC				AC(2)				CMG			
nnz (K)	n (K)	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})
2910	422	17	0.24	0.718	0.433	13	0.23	1.07	0.288	15	0.306	1.07	0.709
14 600	2100	22	0.359	0.967	0.337	14	0.291	1.31	0.315	15	0.294	0.882	0.644
28 500	4100	22	0.404	0.928	0.474	14	0.337	1.34	0.416	15	0.312	0.873	0.957
55 800	8000	39	0.893	1.49	0.495	27	0.796	1.79	0.256	23	0.52	1.1	0.981
112 000	16 000	41	1.03	1.62	0.331	26	0.868	1.91	0.28	23	0.531	1.1	0.663

nonzeros	variables	Hypre				PETSc				ICCG				IC(2)CG			
nnz (K)	n (K)	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})
2910	422	6	0.199	0.508	0.0937	9	1.63	2.38	1×10^{-6}	38	0.618	0.811	0.935	18	0.268	0.618	0.811
14 600	2100	6	0.213	0.518	0.378	9	1.71	2.46	4.33×10^{-6}	59	0.892	1.1	0.973	26	0.427	0.789	0.979
28 500	4100	6	0.215	0.519	0.384	9	1.73	2.49	5.78×10^{-6}	57	0.844	1.04	0.913	25	0.403	0.759	0.985
55 800	8000	10	0.34	0.646	0.287	9	1.77	2.57	0.009 28	372	5.78	5.99	0.988	168	2.65	3.01	0.993
112 000	16 000	10	0.342	0.648	0.301	9	1.78	2.58	0.009 39	382	5.9	6.13	0.976	167	2.65	3.01	0.944

We adopt a finite volume method discretization approach similar to [16]. We consider a unit cube domain $\Omega = [0,1]^3$ using a standard 3D 7-point stencil. We discretize along three dimensions with n_1, n_2, and n_3 variables spaced evenly along each dimension, leading to a 3D grid with $n = n_1 \cdot n_2 \cdot n_3$ variables, indexed by $u_{i,j,k}$ with $(i,j,k) \in [n_1] \times [n_2] \times [n_3]$, at positions $\left(\frac{i-1}{n_1-1}, \frac{j-1}{n_2-1}, \frac{k-1}{n_3-1}\right) \in \Omega$.

When a point is on the boundary $\partial\Omega$, we introduce the equation $u_{i,j,k} = 0$ (equivalently, we remove the corresponding matrix row and column). When a point is not on the boundary of Ω, we set

$$
\begin{aligned}
&-(a_{\text{north}} + a_{\text{south}} + a_{\text{east}} + a_{\text{west}} + a_{\text{up}} + a_{\text{down}})u_{i,j,k} \\
&+ a_{\text{north}}u_{i+1,j,k} + a_{\text{south}}u_{i-1,j,k} + a_{\text{east}}u_{i,j+1,k} + a_{\text{west}}u_{i,j-1,k} + a_{\text{up}}u_{i,j,k+1} + a_{\text{down}}u_{i,j,k-1} = f_{i,j,k}
\end{aligned}
$$

where we used a_{north} to refer to the coefficient $\mu(x)$ at the domain location $x \in \Omega$ halfway between the domain locations of $u_{i,j,k}$ and $u_{i+1,j,k}$ and so on. This discretization scheme directly leads to an SDDM matrix. Unlike [16], for simplicity, when our domain Ω has regions with varying diffusion coefficient μ, we ensure these regions are aligned so that there is always a variable on the boundary between regions and the $x \in \Omega$ giving rise to each coefficient $a_{\text{direction}}$ falls entirely in one region. This lets us avoid averaging over different x to obtain our coefficients $a_{\text{direction}}$.

Uniform coefficient 3D cube. In this experiment, we consider a unit cube domain $\Omega = [0,1]^3$ with uniform coefficient $\mu(x) = 1$ everywhere and equally fine-grained discretization along each dimension, i.e. $n_1 = n_2 = n_3$. Results are shown in Table 6.1.

High-contrast variable coefficient 3D cube. In this experiment, we consider a unit cube domain $\Omega = [0,1]^3$, with equally fine-grained discretization along each dimension, i.e. $n_1 = n_2 = n_3$. We now adopt a checkerboard pattern for the coefficient μ, dividing this into subregions with varying coefficients, similar to experiments by Sundar et al. [61, 16]. Formally, we divide $[0,1]^3$ into k intervals along each axis, leading to k^3 smaller cubes, which we refer to as subregions. Each subregion has a fixed μ value, and we pick these to form a checkerboard pattern. Formally, the checkerboard pattern is defined for $(x,y,z) \in \Omega$ by

$$
\mu(x,y,z) = \begin{cases} 1 & \text{if } \lfloor k \cdot x \rfloor + \lfloor k \cdot z \rfloor + \\ & \quad \lfloor k \cdot y \rfloor \equiv 0 \pmod 2 \\ w & \text{otherwise} \end{cases}. \tag{5.2}
$$

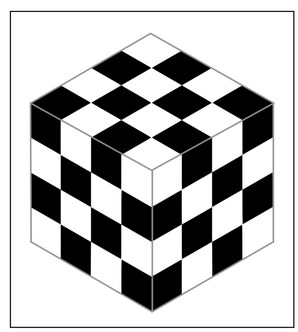


Fig. 5.2: Checkered cube domain Ω following Equation (5.2) with $k = 4$. Black subregions have $\mu(x) = w$, white subregions have $\mu(x) = 1$.

Figure 5.2 shows an example with $k = 4$. Results are shown in Table 6.2.

Anisotropic coefficient 3D cube with variable resolution and fixed weight. In this experiment, we follow Sundar et al. [61], and study anisotropic coefficient diffusion problems on a unit cube domain $\Omega = [0,1]^3$. We include this experiment as Sundar et al. [61] highlighted it as a problematic case for Algebraic Multigrid methods. We consider systems of linear equations arising from Poisson problems with Dirichlet boundary conditions on a domain Ω with a fixed *matrix* coefficient μ. We choose $\mu \in R^{3\times3}$ as a diagonal matrix with $\mu(1,1) = w$ and $\mu(2,2) = \mu(3,3) = 1$.

$$(5.3) \qquad \begin{cases} \operatorname{div}(\mu \boldsymbol{\nabla} u(x)) = f(x) & \text{for all } x \in \Omega \\ u(x) = 0 & \text{for } x \in \partial\Omega \end{cases}.$$

This is equivalent to the continuous uniform scalar coefficient problem, studied on a rectangular domain $\Omega = [0, 1/w] \times [0, 1]^2$ with equal resolution used in the discretization along each dimension. Our discrete matrix is a grid nonzero structure with uniform weights, symmetric diagonally dominant with negative off-diagonals (SDDM), but we vary the resolution used for discretization along each dimension. Concretely, we stretch or shrink the grid along the first dimension by a factor $\eta = 1/w$ so that the number of variables along each, denoted n_1, n_2, n_3 satisfy $n_1 = \eta n_2 = \eta n_3$. Results are shown in Table 6.3.

Anisotropic coefficient 3D cube with fixed resolution and variable weight. We again study the continuous problem from Equation (5.3), as in the previous experiment. But in this experiment, we use the same resolution when discretizing along each axis. This results in a discrete 3D grid, with weights along the first axis of w and weights along the second two axes of 1, and all three axes having the same number of variables, denoted n_1, n_2, n_3 and satisfying $n_1 = n_2 = n_3$. The results are shown in Table 6.4.

Overall Poisson analysis. We see that broadly speaking, Hypre and CMG are the winners here, and AC beats AC(2). Interestingly, our methods beat ICCG/IC(2)CG, showing that even on grid problems, our approach is favorable to other incomplete factorizations.

6. Conclusion. Our experiments suggest that AC(2) and AC attain a level of robustness and reliability not seen before in solvers for SDDM linear equations, while retaining good performance across all instances. For most problems, AC appears to be a good choice, while some very challenging instances require using AC(2). Our solvers are implemented in Julia and are available at:

https://github.com/danspielman/Laplacians.jl/

We hope that our benchmarking data will be used in future works to provide a rigorous basis for comparison of new SDD solvers. This benchmarking data and an accompanying tutorial are available at:

https://rjkyng.github.io/SDDM2023/

Open problems. We hope that our results will motivate others to study ways to incorporate insights from our solvers into production-level software for solving SDDM systems. Developing a highly-optimized version of our algorithm, even for single-threaded systems, will be an important step in this direction. Creating an efficient parallel implementation is even more important, and presents interesting questions about which parallelization approach is likely to be most successful. In follow-up work to this paper, Baumann and Kyng [4] developed a parallel version of our algorithm, based on eliminating large sampled independent sets. This theoretical result may be a promising route for developing a practical parallel implementation. We believe a more in-depth study of the impact of elimination orderings is also warranted. Our experiments (Figure 4.2) suggest our greedy ordering gives AC a significant performance boost. For the particular case of 3D grid problems, [20] studied the performance of many different elimination orders, however, it is not clear in their observations how it translates beyond this case. They also developed a parallel implementation for 3D grid problems.

Another significant open problem is whether one can develop a version of AC that beats Hypre and CMG on grid problems. A potentially interesting avenue here is to develop variants of AC with lower fill-in. Spectral graph theory [59, 46] suggests that a promising direction is to identify low leverage score edges and sample them with lower probability. This may be particularly tractable for graphs such as 3D grids that have a relatively simple structure compared to e.g. Chimera graphs.

We believe our solvers may finally be robust enough to use in interior point methods, and we hope this will lead to further applied research on IPMs for single-commodity flow problems. A major challenge here is to develop practical interior point methods with low iteration counts for these problems.

To further improve SDD solvers and to bring them into widespread practical use for applications such as IPMs, we believe more testing is required. Adding more benchmark data sets could serve as an important guide to which algorithms perform well in practice in these settings.

Conjectured preconditioning quality of our algorithms. We believe that the multi-edge version of our algorithm (AC-skmk, see Section 3.4) with $k = \Theta(\log^2 m)$ multi-edges can be shown to provide the same preconditioning guarantees as Kyng-Sachdeva [46] with this sampling rate. If true, this yields an algorithm with $O(m \log^3 m)$

running time and space to compute the preconditioner and $O(m \log^3 m \log(1/\epsilon))$ time to find an ϵ-accurate solution to a linear system $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$, measured in the $\boldsymbol{L}$-norm,[9] exactly as in [46].

Crucially, the parameter choice of $k = \Theta(\log^2 m)$ samples is *not* the focus of our experiments. Rather, in practice, we have found that our algorithm works well with a much lower sampling rate than Kyng-Sachdeva [46]. In particular, we focus on versions of our algorithm with $k = 1$ (the version AC with no multi-edges) and $k = 2$ (the version AC(2)). In this regime, we cannot hope for constant factor relative condition number, but we conjecture that our AC(2) factorization achieves a relative condition number of $\kappa = O(\log^2 m)$, possibly even $\kappa = O(\log m)$, with high probability. Proving this formally is an exciting open problem. Table 4.3 provides experimental support for these conjectures. If true, this yields an algorithm with $O(m \log m)$ running time and space to compute the preconditioner and $O(m \log m \sqrt{\kappa} \log(1/\epsilon))$ (with κ as above) time to find an ϵ-accurate solution to a linear system in $\boldsymbol{L}$-norm. This means that if we target constant or inverse polynomial residual error in ℓ_2-norm,[10] we expect an overall running time bounded by $O(m \log^2 m \sqrt{\kappa})$, which yields a bound of $O(m \log^3 m)$ if $\kappa = O(\log^2 m)$, and $O(m \log^{2.5} m)$ if $\kappa = O(\log m)$. In contrast, [46] would use $O(m \log^4 m)$ time and $O(m \log^3 m)$ space.

Heuristically, a reason that our preconditioner performs well with a low sampling rate may be that our algorithms maintain basic connectivity properties with probability 1, by always sampling nonzero patterns that contain a connected subgraph (a tree on the 'neighbor' variables of the eliminated variable). Some theoretical results suggests that a clique may be well-approximated by a tree plus a few additional edges: tree-based ultra-sparse sampling can yield convergent iterative methods for solving Laplacians [60], even when the sampling is too sparse to yield concentration of measure w.r.t. spectral norms [23]. By combining spectral sparsification [60, 59] with ultrasparsification [60, 41], one can obtain a poly $\log n$ spectral sparsifier of graph Laplacian using a tree and $O(n)$ edges. Combining this type of guarantee with an analysis along the lines of Kyng-Sachdeva [46] could potentially yield an analysis of our algorithm. Other evidence for the suitability of tree-based sampling in the very sparse regime comes from the observation that in bounded-degree graphs, a union of two random spanning trees forms a good cut sparsifier [31], while $O(\log n)$ random spanning trees form a good spectral sparsifier [47, 38].

We view the guaranteed connectivity-maintaining property of our sampling rule as its key feature, and that other sampling rules that enforce this could potentially achieve even better performance and a systematic study of this is warranted.

Acknowledgments. The authors want to thank anonymous reviewers whose feedback on an earlier draft led to many improvements in the manuscript. We also thank Sushant Sachdeva, Edmond Chow, Richard Peng, Joel Tropp, George Biros, Houman Owhadi, and John Gilbert for many helpful conversations about practical solvers for SDDM linear equations. We thank Sushant Sachdeva for suggesting the Sachdeva Star to break the AC solver. We also thank Kaan Oktay for assistance with setting up PETSc and running experiments.

[9] Note that our experiments target the residual error $\boldsymbol{L}\boldsymbol{x} - \boldsymbol{b}$ in ℓ_2-norm, which may incur an additional $\log m$-factor in the running time for both algorithms when targeting a fixed residual error.

[10] Again, assuming polynomially bounded weights in our input matrix, we can achieve $\epsilon = 1/\operatorname{poly}(m)$ error in ℓ_2-norm by reaching $\epsilon' = \epsilon/\operatorname{poly}(m)$ error in $\boldsymbol{L}$-norm.

Table 6.1: Uniform coefficient 3D cube

nonzeros	variables	AC				AC(2)				CMG			
nnz (K)	n (K)	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})
1990	287	24	0.319	0.769	0.524	18	0.32	1.17	0.739	27	0.464	1.25	0.749
19 900	2860	25	0.453	1.07	0.913	20	0.549	1.74	0.472	27	0.485	0.915	0.872
200 000	28 700	27	0.833	1.46	0.612	20	0.99	2.49	0.679	27	0.619	1.03	0.956

nonzeros	variables	Hypre				PETSc				ICC				IC(2)CG			
nnz (K)	n (K)	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}}/\text{nnz}$ (μs)	res. err. (10^{-8})
1990	287	7	0.201	0.488	0.197	5	0.62	1.62	0.004 02	61	0.911	1.13	0.926	33	0.619	0.992	0.7
19 900	2860	8	0.259	0.572	0.113	5	0.771	2.02	0.005 18	109	1.6	1.78	0.958	59	1.28	1.65	0.817
200 000	28 700	7	0.235	0.587	0.206	6	1.01	2.62	0.000 333	185	2.86	3.1	0.968	83	1.79	2.17	0.971

Table 6.2: High-contrast variable coefficient 3D cube. Unit cube domain with $\Omega = [0,1]^3$, with equally fine-grained discretization along each dimension. 28.7 M variables and 200 M nonzeros. Weight $w = 10^7$. Times are reported in seconds.

axis intervals	AC				AC(2)				CMG			
k	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})
2	41	1.28	1.88	0.742	30	1.48	3.01	0.987	30	0.668	1.07	0.643
4	47	1.47	2.1	0.733	35	1.74	3.14	0.786	28	0.569	0.981	0.796
8	52	1.61	2.24	0.836	38	1.87	3.26	0.985	26	0.531	0.937	0.912
16	52	1.57	2.19	0.915	46	2.2	3.57	0.707	28	0.571	1	0.969
32	59	1.73	2.33	0.802	45	2.06	3.38	0.967	28	0.552	0.971	0.718
64	57	1.59	2.16	0.925	48	2.02	3.23	0.762	25	0.506	0.93	0.671
128	53	1.39	1.94	0.98	35	1.31	2.46	0.828	28	0.519	0.956	0.989

axis intervals	Hypre				PETSc				ICC				IC(2)CG			
k	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})	N_{iter}	$t_{\text{solve}}/\text{nnz}$ (μs)	$t_{\text{total}/\text{nnz}}$ (μs)	res. err. (10^{-8})
2	8	0.265	0.616	0.308	7	1.35	3.12	0.000 145	175	2.71	2.95	0.991	90	1.43	1.81	0.931
4	8	0.264	0.656	0.738	7	1.65	3.79	0.000 555	236	3.65	3.89	0.994	127	1.99	2.37	0.975
8	9	0.295	0.734	0.351	7	1.77	3.85	0.001 42	290	4.46	4.71	0.979	167	2.6	2.98	1
16	9	0.289	0.752	0.184	8	1.95	3.84	0.000 297	337	5.33	5.61	0.91	164	2.57	2.95	0.984
32	8	0.257	0.732	0.744	8	1.83	3.43	0.001 17	343	5.27	5.51	0.993	171	3.42	3.8	0.921
64	8	0.258	0.64	0.513	9	1.43	2.35	0.000 498	375	5.77	6.01	0.986	169	2.65	3.02	0.99
128	7	0.244	0.55	0.98	7	0.664	1.1	0.001 06	411	6.32	6.56	0.966	143	3.1	3.47	0.998

Table 6.3: Anisotropic coefficient 3D cube with variable resolution and fixed weight.

aniso. stretch	nonzeros	variables	AC				AC(2)				CMG			
$\eta = 1/w$	nnz	n	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.
	(M)	(M)		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
0.001	203	29.6	26	0.637	1.16	0.661	19	0.729	1.99	0.871	25	0.539	0.968	0.922
0.01	200	28.8	26	0.724	1.32	0.904	20	0.864	2.26	0.592	26	0.564	0.979	0.682
0.1	200	28.7	27	0.809	1.42	0.618	20	0.956	2.42	0.681	26	0.586	1	0.759
1	200	28.7	26	0.811	1.44	0.906	20	0.994	2.5	0.753	28	0.64	1.05	0.591
10	200	28.7	27	0.858	1.48	0.635	20	0.991	2.47	0.61	28	0.761	1.19	0.677
100	196	28.6	25	0.741	1.31	0.845	19	0.837	2.12	0.842	28	0.633	1.07	0.637
1000	189	29.9	23	0.494	0.956	0.733	18	0.502	1.39	0.454	17	0.379	0.845	0.749

aniso. stretch	nonzeros	variables	Hypre				PETSc				ICC				IC(2)CG			
$\eta = 1/w$	nnz	n	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.
	(M)	(M)		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
0.001	203	29.6	7	0.222	0.589	0.192	5	0.677	1.62	0.003 66	47	0.741	0.887	0.967	25	0.531	0.896	0.721
0.01	200	28.8	7	0.228	0.583	0.2	5	0.732	1.84	0.004 56	80	1.27	1.44	0.987	43	0.832	1.21	0.884
0.1	200	28.7	7	0.234	0.581	0.204	5	0.808	2.15	0.006 18	135	2.09	2.27	0.999	63	1.09	1.46	1
1	200	28.7	7	0.235	0.587	0.205	6	1.01	2.6	0.000 371	187	2.91	3.15	0.978	82	1.34	1.72	0.956
10	200	28.7	8	0.261	0.614	0.103	5	0.638	1.77	0.004 14	100	1.57	1.79	0.921	54	0.892	1.29	0.86
100	196	28.6	7	0.223	0.574	0.187	4	0.494	1.52	0.116	35	0.563	0.78	0.766	19	0.319	0.731	0.501
1000	189	29.9	7	0.194	0.476	0.0744	4	0.545	1.41	0.002 11	12	0.212	0.437	0.651	7	0.118	0.537	0.269

Table 6.4: Anisotropic coefficient 3D cube with fixed resolution and variable weight. 28.7 M variables and 200 M nonzeros

weight	AC				AC(2)				CMG			
$\eta = 1/w$	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.
		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
0.001	39	1.09	1.67	0.902	25	1.09	2.48	0.657	28	0.568	0.979	0.645
0.01	33	0.951	1.53	0.944	23	0.997	2.41	0.963	27	0.55	0.974	0.83
0.1	26	0.778	1.37	0.992	20	0.925	2.39	0.645	28	0.573	0.983	0.619
1	26	0.814	1.52	0.952	20	0.991	2.5	0.846	28	0.636	1.05	0.653
10	20	0.571	1.21	0.818	15	0.63	1.95	0.413	31	0.857	1.27	0.785
100	17	0.399	0.916	0.464	12	0.378	1.51	0.612	30	0.722	1.15	0.83
1000	14	0.302	0.78	0.814	10	0.257	1.05	0.91	30	0.684	1.09	0.865

weight	Hypre				PETSc				ICC				IC(2)CG			
$\eta = 1/w$	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.	N_{iter}	$t_{\text{solve}}/\,\text{nnz}$	$t_{\text{total}}/\,\text{nnz}$	res. err.
		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})		(μs)	(μs)	(10^{-8})
0.001	7	0.297	0.662	0.165	6	0.621	1.16	0.000 816	205	3.18	3.42	0.998	100	1.82	2.2	0.951
0.01	7	0.299	0.683	0.159	6	0.733	1.38	0.000 871	193	3.01	3.25	0.989	91	1.43	1.81	0.949
0.1	8	0.321	0.772	0.134	7	2.76	4.92	0.008 65	164	2.54	2.78	0.989	80	1.26	1.64	0.944
1	7	0.235	0.588	0.207	5	0.861	2.46	0.005 81	154	2.39	2.63	0.971	96	1.47	1.85	0.992
10	7	0.323	0.625	0.618	5	1.09	2.56	0.0704	155	2.4	2.65	0.969	84	1.64	2	0.991
100	8	0.282	0.449	0.34	5	0.629	1.05	0.0317	75	1.17	1.43	0.96	56	0.865	1.22	0.996
1000	7	0.223	0.332	0.128	6	0.644	0.904	0.000 392	30	0.483	0.733	0.866	26	0.575	0.927	0.874

REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, and W. Gropp, *PETSc users manual*, (2019).

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.

[4] Y. Baumann and R. Kyng, *A Framework for Parallelizing Approximate Gaussian Elimination*, in Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '24, New York, NY, USA, June 2024, Association for Computing Machinery, pp. 195–206, .

[5] R. Beauwens, *Approximate factorizations with modified s/p consistently ordered m-factors*, Numerical linear algebra with applications, 1 (1994), pp. 3–17.

[6] L. Berger-Vergiat, C. A. Glusa, J. J. Hu, C. Siefert, R. S. Tuminaro, M. Matthias, P. Andrey, and W. Tobias, *MueLu User's Guide.*, tech. report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia . . . , 2019.

[7] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo, *Support-graph preconditioners*, SIAM Journal on Matrix Analysis and Applications, 27 (2006), pp. 930–951.

[8] E. G. Boman, K. Deweese, and J. R. Gilbert, *Evaluating the Dual Randomized Kaczmarz Laplacian Linear Solver*, Informatica, 40 (2016), pp. 95–107.

[9] E. G. Boman and B. Hendrickson, *Support Theory for Preconditioning*, SIAM Journal on Matrix Analysis and Applications, 25 (2003), pp. 694–717, .

[10] E. G. Boman, B. Hendrickson, and S. Vavasis, *Solving Elliptic Finite Element Systems in Near-Linear Time with Support Preconditioners*, SIAM Journal on Numerical Analysis, 46 (2008), pp. 3264–3284, .

[11] D. Braess and W. Hackbusch, *A new convergence proof for the multigrid method including the V-cycle*, SIAM journal on numerical analysis, 20 (1983), pp. 967–975.

[12] A. Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of computation, 31 (1977), pp. 333–390.

[13] A. Brandt, *General highly accurate algebraic coarsening*, Electronic Transactions on Numerical Analysis, 10 (2000), p. 21.

[14] A. Brandt, *Multiscale scientific computation: Review 2001*, Multiscale and multiresolution methods, (2002), pp. 3–95.

[15] A. Brandt and N. Dinar, *Multigrid Solutions to Elliptic Flow Problems*, in Numerical Methods for Partial Differential Equations, S. V. Parter, ed., Academic Press, Jan. 1979, pp. 53–147, .

[16] J. Brannick, F. Cao, K. Kahl, R. D. Falgout, and X. Hu, *Optimal interpolation and compatible relaxation in classical algebraic multigrid*, SIAM Journal on Scientific Computing, 40 (2018), pp. A1473–A1493.

[17] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*, Society for Industrial and Applied Mathematics, second ed., Jan. 2000, .

[18] N. I. Buleev, *A numerical method for solving two-dimensional diffusion equations*, The Soviet Journal of Atomic Energy, 6 (1960), pp. 222–224, .

[19] L. Cambier, C. Chen, E. G. Boman, S. Rajamanickam, R. S. Tuminaro, and E. Darve, *An Algebraic Sparsified Nested Dissection Algorithm Using Low-Rank Approximations*, arXiv:1901.02971 [cs, math], (2020), .

[20] C. Chen, T. Liang, and G. Biros, *RCHOL: Randomized Cholesky factorization for solving SDD linear systems*, SIAM Journal on Scientific Computing, 43 (2021), pp. C411–C438.

[21] D. Chen and S. Toledo, *Vaidya's preconditioners: Implementation and experimental study*, Electronic Transactions on Numerical Analysis, 16 (2003), p. 03.

[22] M. A. Christie and M. J. Blunt, *Tenth SPE Comparative Solution Project: A Comparison of Upscaling Techniques*, in SPE Reservoir Simulation Symposium, OnePetro, Feb. 2001, .

[23] M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu, *Solving SDD linear systems in nearly m log1/2 n time*, in Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, 2014, pp. 343–352.

[24] T. A. Davis and Y. Hu, *The university of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1–1:25, .

[25] K. Deweese, J. R. Gilbert, G. Miller, R. Peng, H. R. Xu, and S. C. Xu, *An empirical study of cycle toggling based Laplacian solvers*, in 2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, SIAM, 2016, pp. 33–41.

[26] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 2017.

[27] T. Dupont, R. P. Kendall, and H. H. Rachford, Jr., *An Approximate Factorization Procedure for Solving Self-Adjoint Elliptic Difference Equations*, SIAM Journal on Numerical Analysis, 5 (1968), pp. 559–573, .

[28] H. C. Elman, *A Stability Analysis of Incomplete LU Factorizations*, Mathematics of Computation, 47 (1986), pp. 191–217, , .

[29] R. D. Falgout and U. M. Yang, *Hypre: A library of high performance preconditioners*, in International Conference on Computational Science, Springer, 2002, pp. 632–641.

[30] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, fourth edition ed., 2013.

[31] N. Goyal, L. Rademacher, and S. Vempala, *Expanders via random spanning trees*, in Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2009, pp. 576–585.

[32] K. D. Gremban, *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*, PhD Thesis, Carnegie Mellon University, 1996.

[33] I. Gustafsson, *A class of first order factorization methods*, BIT Numerical Mathematics, 18 (1978), pp. 142–156.

[34] V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177, .

[35] D. Hoske, D. Lukarski, H. Meyerhenke, and M. Wegner, *Engineering a combinatorial Laplacian solver: Lessons learned*, Algorithms, 9 (2016), p. 72.

[36] A. Jambulapati and A. Sidford, *Ultrasparse Ultrasparsifiers and Faster Laplacian System Solvers*, in Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2021, pp. 540–559.

[37] T. Kathuria, Y. P. Liu, and A. Sidford, *Unit Capacity Maxflow in Almost O(m4/3) Time*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2020, pp. 119–130.

[38] T. Kaufman, R. Kyng, and F. Soldá, *Scalar and Matrix Chernoff Bounds from l-infinity-Independence*, SODA '22: Proceedings of the Thirty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, (2022), .

[39] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, *A simple, combinatorial algorithm for solving SDD systems in nearly-linear time*, in Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, 2013, pp. 911–920.

[40] D. S. Kershaw, *The incomplete Cholesky—conjugate gradient method for the iterative solution of systems of linear equations*, Journal of Computational Physics, 26 (1978), pp. 43–65, .

[41] A. Kolla, Y. Makarychev, A. Saberi, and S.-H. Teng, *Subgraph sparsification and nearly optimal ultrasparsifiers*, in Proceedings of the Forty-Second ACM Symposium on Theory of Computing, Cambridge Massachusetts USA, June 2010, ACM, pp. 57–66, .

[42] I. Koutis, G. L. Miller, and R. Peng, *Approaching Optimality for Solving SDD Linear Systems*, in Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science, FOCS '10, USA, Oct. 2010, IEEE Computer Society, pp. 235–244, .

[43] I. Koutis, G. L. Miller, and R. Peng, *A nearly-m log n time solver for sdd linear systems*, in 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science, IEEE, 2011, pp. 590–598.

[44] I. Koutis, G. L. Miller, and D. Tolliver, *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*, Computer Vision and Image Understanding, 115 (2011), pp. 1638–1646, .

[45] R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. A. Spielman, *Sparsified cholesky and multigrid solvers for connection laplacians*, in Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, 2016, pp. 842–850.

[46] R. Kyng and S. Sachdeva, *Approximate gaussian elimination for laplacians-fast, sparse, and simple*, in 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2016, pp. 573–582.

[47] R. Kyng and Z. Song, *A matrix chernoff bound for strongly rayleigh distributions and spectral sparsifiers from a few random spanning trees*, in 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2018, pp. 373–384.

[48] M. H. Langston, M. T. Harris, P.-D. Letourneau, R. Lethin, and J. Ezick, *Combinatorial Multigrid: Advanced Preconditioners For Ill-Conditioned Linear Systems*, in 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, Sept. 2019, IEEE, pp. 1–7, .

[49] O. E. Livne and A. Brandt, *Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver*, SIAM Journal on Scientific Computing, 34 (2012), pp. B499–B522.

[50] J. A. Meijerink and H. A. van der Vorst, *An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix*, Mathematics of Computation, 31 (1977), pp. 148–162, .

[51] A. Napov and Y. Notay, *An Efficient Multigrid Method for Graph Laplacian Systems II: Robust Aggregation*, SIAM Journal on Scientific Computing, 39 (2017), pp. S379–S403, .

[52] Y. Notay, *Solving positive (semi) definite linear systems by preconditioned iterative methods*, in Preconditioned Conjugate Gradient Methods, Springer, 1990, pp. 105–125.

[53] Y. Notay, *Conditioning of Stieltjes matrices by S/P consistently ordered approximate factorizations*, Applied numerical mathematics, 10 (1992), pp. 381–396.

[54] T. A. Oliphant, *An implicit, numerical method for solving two-dimensional time-dependent diffusion problems*, Quarterly of Applied mathematics, 19 (1961), pp. 221–229.

[55] R. Peng and D. A. Spielman, *An efficient parallel solver for SDD linear systems*, in Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, 2014, pp. 333–342.

[56] J. W. Ruge and K. Stüben, *Algebraic multigrid*, in Multigrid Methods, SIAM, 1987, pp. 73–130.

[57] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numerical Linear Algebra with Applications, 1 (1994), pp. 387–402, .

[58] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, second ed., Jan. 2003, .

[59] D. A. Spielman and N. Srivastava, *Graph sparsification by effective resistances*, in Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, Victoria British Columbia Canada, May 2008, ACM, pp. 563–568, .

[60] D. A. Spielman and S.-H. Teng, *Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems*, in Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, STOC '04, New York, NY, USA, June 2004, Association for Computing Machinery, pp. 81–90, .

[61] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, *Parallel geometric-algebraic multigrid on unstructured forests of octrees*, in SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Nov. 2012, pp. 1–11, .

[62] P. M. Vaidya, *Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners.* 1990.

[63] H. van der Vorst, and J. Meijerink, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comput, 31 (1977), p. 148.

[64] R. S. Varga, *Factorization and normalized iterative methods*, tech. report, Westinghouse Electric Corp. Bettis Plant, Pittsburgh, 1959.

[65] A. Walker, *New fast method for generating discrete random numbers with arbitrary frequency distributions*, Electronics Letters, 10 (1974), pp. 127–128, .

[66] J. W. Watts, *A conjugate gradient-truncated direct method for the iterative solution of the reservoir simulation pressure equation*, Society of Petroleum Engineers Journal, 21 (1981), pp. 345–353.

[67] H. Yserentant, *Old and new convergence proofs for multigrid methods*, Acta numerica, 2 (1993), pp. 285–326.

[68] X. Zhu, Z. Ghahramani, and J. D. Lafferty, *Semi-supervised learning using gaussian fields and harmonic functions*, in Proceedings of the 20th International Conference on Machine Learning (ICML-03), 2003, pp. 912–919.