Comparing solvers for symmetric positive definite matrices in the Julia programming language

Julius Johannes Taraz, 410064
TU Berlin: Advanced Topics from Scientific Computing
September 2024

Contents

1 Introduction

Symmetric positive definite matrices frequently arise in various scientific and engineering applications, such as finite element methods, optimisation problems, and machine learning algorithms. Efficiently solving linear systems involving symmetric positive definite matrices is crucial for the performance of these applications. The Julia programming language [1], known for its high performance and ease of use, offers a variety of solvers designed to handle such systems effectively.

This report presents a comprehensive comparison of different solvers available in Julia for solving linear systems with symmetric positive definite matrices. We explore both direct methods, such as Cholesky factorization, and iterative methods, i.e. the Conjugate Gradient method with various preconditioners.

Our comparison focuses mainly on the runtime of the different algorithms for different matrix sizes. Furthermore, memory usage and accuracy are investigated.

Through this analysis, we aim to provide insights into the strengths and weaknesses of each solver, helping practitioners choose the most appropriate method for their specific applications. By providing an interactive

Pluto notebook and an easily expandable benchmark database, we want to make looking into a specific problem easier for the user. The Pluto notebook and the benchmark data and code can be found on GitHub [2]. This report highlights the capabilities and limitations of different approaches to solving symmetric positive definite systems.

1.1 Contents

In Section 2, we investigate direct and iterative solvers for linear systems of equations. We also explain different preconditioners for iterative solvers and the Finite Difference method, which is used to generate some benchmark matrices.
In Section 3, we go through all matrix types and algorithms used for benchmarks and we briefly explain some tools and methods for evaluating the results.
We then present the results of our benchmarks divided by matrix types in Section 4. Finally, tables with the scaling behaviour of all matrix types and algorithms used for benchmarks are given in the Appendix.

2 Theory

In this section we will discuss direct and iterative methods to solve $Ax = b$ if A is a large sparse symmetric positive definite (SPD) matrix. We will also discuss the Finite Difference method as a way to generate relevant benchmark matrices.

2.1 Direct solvers

From linear algebra many factorizations of matrices are known, e.g. LU and QR for general matrices and the Cholesky factorization as a special case of the LU factorization for symmetric matrices. Solving a linear system of equations (LSE) using such a factorization consists of two steps, computing the factors and then e.g. forward- or back-substitution to solve a triangular system of equations.

2.1.1 Sparse Cholesky factorization

We will look at the Cholesky factorization $A = LL^T$ as an example, where L is a lower triangular matrix with $A, L \in \mathbb{R}^{n\times n}$ [3]. We can simply compute each entry of the cholesky factor L using

$$\begin{aligned} l_{jj} &= \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk} \right)^{1/2}, \\ l_{ij} &= \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right). \end{aligned} \tag{1}$$

For sparse matrices it is not clear that the original matrix' sparsity transfers to the Cholesky factors. But the sparsity of the Cholesky factor can be changed using a permutation, i.e. $A = LPL^T$, where P is a permutation matrix. It is thus common to permute or reorder the matrix such that the fill-in (non-zero entries of the Cholesky matrix that do not appear in the original matrix) is minimised. One common way to achieve this is nested dissection.
Each matrix $A \in \mathbb{R}^{n\times n}$ can be considered as the adjacency matrix of a graph G with n vertices. Each off-diagonal non-zero entry $A_{ij} \neq 0$ thus corresponds to an edge or connection between vertices i and j. For nested dissection a small vertex separator of G must be found, i.e. a set of vertices S such that $G \setminus S$ has two disconnected subgraphs G_1, G_2. The matrix can then be reordered such that

$$\begin{pmatrix} A_{11} & 0 & A_{S1} \\ 0 & A_{22} & A_{S2} \\ A_{S1}^T & A_{S2}^T & A_{SS} \end{pmatrix}.$$

Using equation 1, it is easy to show that the pattern of the reordered matrix translates to the Cholesky factor. The dissection step can obviously be repeated by separating G_1 and G_2.

2.1.2 Forward substitution

To solve a lower triangular system of equations $Ly = z$ forward substitution is used.

$$\begin{aligned} z_1 &= l_{1,1}y_1, \\ z_2 &= l_{2,1}y_1 + l_{2,2}y_2, \\ &\vdots \\ z_n &= l_{n,1}y_1 + l_{n,2}y_2 + \ldots + l_{n,n}y_n \end{aligned}$$

Solving for the entries of y

$$\begin{aligned} y_1 &= \frac{z_1}{l_{1,1}}, \\ y_2 &= \frac{z_2 - l_{2,1}y_1}{l_{2,2}}, \\ &\vdots \\ y_n &= \frac{z_n - \sum_{i=1}^{n-1} l_{n,i}y_i}{l_{n,n}}. \end{aligned}$$

From these formulas we can see that the time complexity of forward substitution for a dense Cholesky factor is $\mathcal{O}(n^2)$. If the Cholesky factor had a fixed number of non-zero entries per row, the time complexity would be $\mathcal{O}(n)$.

2.2 Krylov Subspace methods

We first introduce general projection methods [4]. The approximate solution x_k to $Ax = b$ in the k-th step is defined as

$$x_k = x_0 + \hat{S}_k t_k,$$

where $\hat{S}_k$ is a basis matrix of the k-dimensional search space S_k, i.e. $\hat{S}_k \in \mathbb{R}^{n\times k}$ and $t_k \in \mathbb{R}^k$. This introduces k degrees of freedom, for a unique solution we thus also need k constraints. Those can be expressed by

$$r_k = b - Ax_k \perp C_k.$$

Here r_k is the residual in the k-th step and C_k is the k-dimensional constraint space, it has a basis matrix $\hat{C}_k$. Taking the definition of x_k and the orthogonality condition for the residual, leads to

$$\hat{C}_k^T r_0 = \hat{C}_k^T A \hat{S}_k t_k.$$

Thus we need to solve a k-dimensional system of linear equations. If A is symmetric positive definite and one chooses $\hat{S}_k = \hat{C}_k$, then $\hat{C}_k^T A \hat{S}_k$ is non-singular. One choice for the search and constraint space is the *Krylov* space $\mathcal{K}_k(A, r_0) = \text{span}(r_0, Ar_0, ..., A^{k-1}r_0)$. This characterises the conjugate gradient (CG) method. We can quickly prove that the orthogonality $r_k \perp \mathcal{K}_k$ implies that $||x - x_k||_A = \min_{z\in x_0+\mathcal{K}_k} ||x - z||_A$.
We first note that $r_k \perp \mathcal{K}_k \Leftrightarrow x - x_k \perp_A \mathcal{K}_k$. Let $z \in x_0 + \mathcal{K}_k$.

$$||x - z||_A^2 = ||\underbrace{x - x_k}_{\in \mathcal{K}_k^{\perp_A}} + \underbrace{x_k - z}_{\in \mathcal{K}_k}||_A^2 = ||x - x_k||_A^2 + ||x_k - z||_A^2 \geq ||x - x_k||_A^2$$

It is important to note that because of numerical stability, in practice an orthogonal basis of $\mathcal{K}_k$ is used. In the end, the only operation that has to be executed often is $v \to Av$.

Furthermore, for any $z \in x_0 + \mathcal{K}_k$ there exist coefficients β_i such that

$$x - z = x - x_0 + \sum_{i=0}^{k-1} \beta_i A^i r_0 = \left(I - \sum_{i=0}^{k-1} \beta_i A^{i+1} \right) (x - x_0) = p(A)(x - x_0).$$

Where $p(y) = 1 - \sum_{i=0}^{k-1} \beta_i y^{i+1}$ is a polynomial. The space of degree k polynomials with $p(0) = 1$ is denoted by P_k. Since A is SPD, there exists an orthogonal diagonalisation $A = UDU^T$, where D is a diagonal matrix with $D_{ii} = \lambda_i$ and $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n > 0$. The minimality of $||x - x_k||_A$ implies

$$\begin{aligned} ||x - x_k||_A &= \min_{p \in P_k} ||p(A)(x - x_0)||_A = \min_{p \in P_k} ||A^{1/2} p(A)(x - x_0)||_2 \\ &= \min_{p \in P_k} ||p(A) A^{1/2} (x - x_0)||_2 \leq \min_{p \in P_k} ||p(A)||_2 ||x - x_0||_A \\ &= \min_{p \in P_k} ||p(D)||_2 ||x - x_0||_A = \min_{p \in P_k} \max_{1 \leq i \leq n} ||p(\lambda_i)||_2 ||x - x_0||_A. \end{aligned} \tag{2}$$

Using Chebyshev polynomials, it can be shown that

$$\min_{p \in P_k} \max_{1 \leq i \leq n} ||p(\lambda_i)||_2 \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k, \tag{3}$$

where $\kappa = \lambda_1 / \lambda_n$ is the 2-norm condition number, since λ_i is the i-th eigenvalue of A. Thus, the expression above is an upper bound on the convergence rate $\frac{||x - x_k||_A}{||x - x_0||_A}$ of the CG method. The bound is obviously strictly increasing for $\kappa \geq 1$ with $\kappa = 1$ meaning immediate convergence. Since the solution of $Ax = b$ is equal to the solution of $M^{-1}Ax = M^{-1}b$, solving the second system of linear equations is equivalent but can be much faster than solving the original system of linear equations, if $\kappa(M^{-1}A) < \kappa(A)$.
M is then called the preconditioner. As mentioned above, the CG algorithm often has to compute matrix-vector products $v \to Av$. In the case of a preconditioner M, the operation is $v \to M^{-1}Av$. In practice the system of linear equations $My = Av$ is solved. The preconditioner is thus chosen in a way such that this can be done efficiently, i.e. M must be easily invertible.
There are many different types of preconditioners, we will present two important categories.

2.2.1 Incomplete LU Factorisation (ILU)

A common preconditioner is the so-called incomplete LU factorization (ILU) [5]. This stems from the classic LU factorization, but since the L and U factors of a sparse matrix do not have to be sparse too, this may result in dense matrices. The simplest version of the ILU is called ILU0, or ILUZero. It neglects all entries of L and U which are on positions where A had a zero-entry. I.e. the incomplete factors have zero fill-in. Since $\hat{L}\hat{U} \neq A$, where $\hat{L}$ and $\hat{U}$ are the incomplete factors, ILU is not a true factorization.
Alternatively, a threshold τ can be defined such that fill-in entries under the threshold are neglected. This is called ILUT or ILU(τ). ILUZero is thus equivalent to ILU(∞).
Similar to the direct solvers, the ILU factorization has to be computed once for each matrix and for each right hand side forward- and backward-substitution has to be done.

2.2.2 Algebraic Multigrid (AMG)

Another common preconditioner is the algebraic multigrid method (AMG) [6]. We will now introduce the (geometrical) multigrid method in detail and then briefly touch the practically used algebraic multigrid method. First, we introduce notation for classical iterative schemes. The notation used here will then be used for the smoother and the coarse grid correction steps.
Consider the linear system of equations $Az = b$, with the solution $z \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ being SPD. A classical iterative method to find an approximate solution $x \in \mathbb{R}^n$ for this system of equations is the following. Let $R \approx A$ be an easily invertible matrix that is SPD. If A has only positive diagonal entries, diag(A) would be a good candidate for R, which describes the Jacobi method. In each iteration the approximate solution x is updated to x' by

$$x' = x + R^{-1}(b - Ax) =: \text{Smooth}(x, A, R, b).$$

For notational convenience we define a function Smooth(x, A, R, b). The error is defined as

$$e := z - x.$$

After one iteration, the error is

$$\begin{aligned} e' &= z - x' = z - x - R^{-1}(b - Ax) \\ &= z - x - R^{-1}A(z - x) = (I - R^{-1}A)(z - x) \\ &= (I - R^{-1}A)e. \end{aligned}$$

From this we can see that this iterative method can converge to the right solution $x \to z$ for each initial vector x_0 if and only if the spectral radius $\rho(I - R^{-1}A) < 1$.
We consider the V-cycle as an example of a multigrid method. It is defined on the levels $0 \leq k \leq J$. Here, J is the finest level and 0 is the coarsest. On each level, a grid with n_k nodes is defined together with a prolongation operation $P_k \in \mathbb{R}^{n_{k+1} \times n_k}$ and a smoother $R_k \in \mathbb{R}^{n_k \times n_k}$. From the prolongations the coarse grid matrices follow $A_k = P_k^T A_{k+1} P_k$. Thus, A_J is the matrix of the original LSE. Pre- and post-smoothing are both done ν times.

Algorithm $MG(b, x, k)$

$$\begin{aligned} \text{pre-smoothing: } \bar{x} &\leftarrow \text{Smooth}(x, A_k, R_k, b)^\nu \\ \text{fine residual: } r_k &\leftarrow b - A_k \bar{x} \\ \text{coarse residual: } r_{k-1} &\leftarrow P_{k-1}^T r_k \\ \text{coarse grid correction: } d_{k-1} &\leftarrow \begin{cases} A_0^{-1} r_0, & \text{if } k = 1 \\ MG(r_{k-1}, 0, k-1), & \text{otherwise} \end{cases} \\ \text{prolongate: } \tilde{x} &\leftarrow \bar{x} + P_{k-1} d_{k-1} \\ \text{post-smoothing: } x' &\leftarrow \text{Smooth}(\tilde{x}, A_k, R_k, b)^\nu \end{aligned}$$

An important detail is the choice of the prolongation operator. In a geometric multigrid method, where A_J is the discretisation of a differential operator, a prolongation as shown in Figure 1 can be used. If there is no grid forming the basis of the matrix, or the grid is unknown, only the matrix is used. This is called an algebraic multigrid method. There are two very prominent examples of algebraic multigrid methods. In Ruge-Stüben AMG, each point of the coarse grid corresponds to a point of the fine grid. The points are chosen such that each fine grid point has a connection (i.e. an off-diagonal entry with large value) to a coarse grid point. The prolongation operator is constructed such that strong connections are preserved and weak connections are disregarded. In Smoothed-Aggregation AMG, each point in the coarse grid corresponds to a set (or aggregate) of fine grid points which are interconnected. A preliminary prolongation is built similar to the one in Ruge-Stüben AMG, but then each column of it is smoothed using e.g. a Jacobi or Gauß-Seidel smoother.

2.2.3 Time complexity

Since the most time consuming step of each CG-iteration is the matrix-vector product, each step needs $\mathcal{O}(m)$ operations where m is the number of non-zero entries. As shown in equations 2 and 3

$$\frac{||x - x_k||_A}{||x - x_0||_A} \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k.$$

We consider the solution x_k of the k-th CG step to be good enough with a specified tolerance ε, if

$$\frac{||x - x_k||_A}{||x - x_0||_A} \leq \varepsilon.$$

From this we can show that for the maximum number of iterations k_{max}

$$k_{max} \lesssim \frac{\sqrt{\kappa}}{2} \ln \left(\frac{2}{\varepsilon} \right).$$

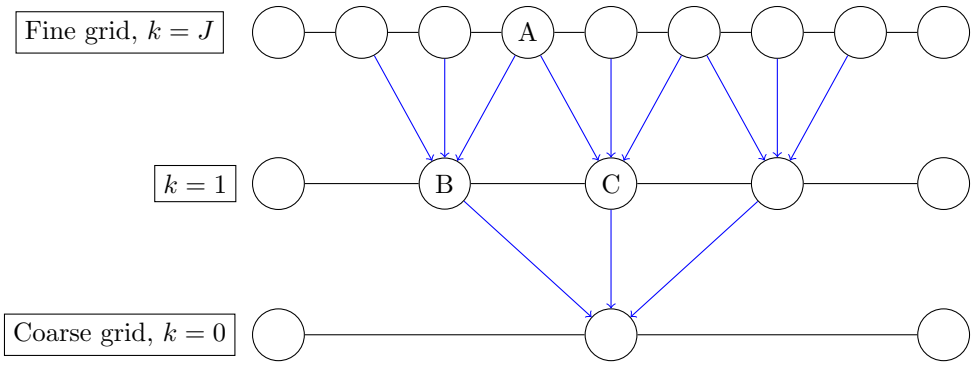

Figure 1: Sequence of refining grids $k = 0, 1, 2$. Blue arrows indicate a non-zero entry in the prolongation matrix, i.e. the prolongation operator for $\text{grid}_{k=1}$ to $\text{grid}_{k=2}$ is P_1 with $(P_1)_{AB}, (P_1)_{AC} \neq 0$.

We furthermore know that for finite difference (see Section 2.3) and finite element discretisations of second-order elliptic PDEs in d dimensions $\kappa \in \mathcal{O}(n^{2/d})$ [7]. Then we get the total time complexity $\mathcal{O}(n^{1+1/d})$ under the assumption that the number of non-zero elements is $\in \mathcal{O}(n)$.

2.3 Finite difference method

The Poisson equation is an elliptic partial differential equation which can be used to model e.g. electrostatics and gravity fields. It states that

$$-\Delta u = f,$$

where $\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$ is the Laplace operator. Let us impose homogeneous Neumann boundary conditions $\frac{\partial}{\partial x} u|_{\partial\Omega} = 0$. If we consider one dimension, the equation reduces to $-\frac{\partial^2 u}{\partial x^2} = f$. The first derivative can be approximated using the Taylor series for $h \ll 1$

$$\begin{aligned}
\frac{\partial}{\partial x} u(x) &\approx \frac{u(x + h/2) - u(x - h/2)}{h}. \\
\text{Thus } \frac{\partial^2}{\partial x^2} u(x) &= \frac{\partial}{\partial x} \left(\frac{\partial}{\partial x} u(x) \right) \approx \frac{1}{h} \left(\frac{\partial}{\partial x} u(x + h/2) - \frac{\partial}{\partial x} u(x - h/2) \right) \\
&\approx \frac{1}{h^2} \left[u(x + h) - u(x) - (u(x) - u(x - h)) \right] \\
&\approx \frac{u(x + h) - 2u(x) + u(x - h)}{h^2}.
\end{aligned}$$

This is the finite difference discretisation of the 1-dimensional Laplace operator. Let us now consider a vector $U \in \mathbb{R}^n$ such that the entries $U_i = u(ih)$ for $1 \leq i \leq n$. Let $A \in \mathbb{R}^{n \times n}$ with A_j being the j-th row. For

$$A_j = (0 \ldots 0 \quad \text{-1} \quad \underbrace{2}_{A_{jj}} \quad \text{-1} \quad 0 \ldots 0)$$

for $2 \leq j \leq n - 1$ and

$$\begin{aligned}
A_1 &= (1 \quad \text{-1} \quad 0 \ldots 0), \\
A_n &= (0 \ldots 0 \quad \text{-1} \quad 1),
\end{aligned}$$

it holds that $\frac{1}{h^2} AU \approx -\Delta u$. Thus, $U = h^2 A^{-1} F$ is the approximate solution of the Poisson equation.

3 Methodology

3.1 Matrices

3.1.1 Finite Difference Matrices

We conducted benchmarks for the Finite-Difference discretisation for equidistant meshes of the Laplace operator with homogeneous Neumann boundary conditions in 1, 2 and 3 dimensions.
To ensure that the resulting matrices are positive definite, we added 0.01 to the diagonal values.

3.1.2 Finite Volume Matrices for inhomogeneous meshes

We also investigated inhomogeneous meshes. Using GMSH [8] we generated a mesh for a square (and a cube). All corners but one had the same meshsize h, the remaining corner had meshsize $h/10$. Such a mesh can be seen in Figure 2. The mesh was then exported to ExtendableGrids.jl [9] and VoronoiFVM.jl [10] was then used to generate the system matrix for the Laplace operator with homogeneous Neumann boundary conditions. The used meshsizes and properties of the resulting matrices can be seen in Section 6.2.
As with the Finite Difference matrices, we added 0.01 to the diagonal values to ensure that the resulting Finite Volume matrices are positive definite.

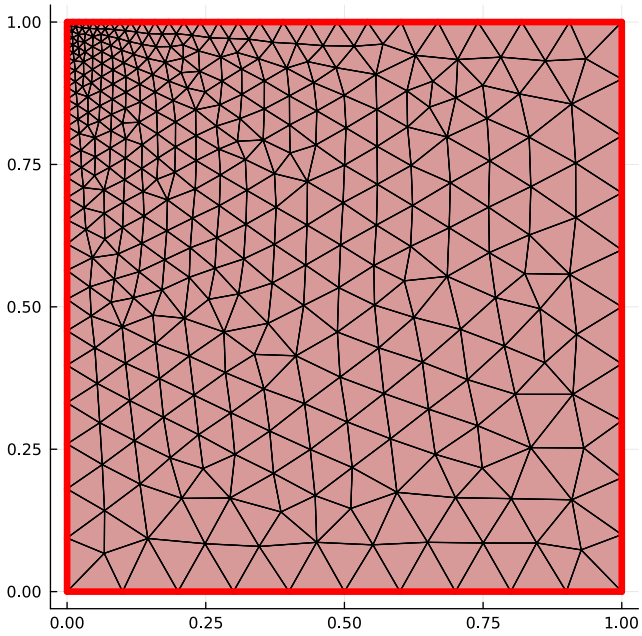

Figure 2: Inhomogeneous mesh for a square.

3.1.3 Random structure

We used SparseArrays.jl's *sprand_sdd!* function which generates a strictly diagonally dominant sparse matrix B with *nnzrow* non-zero entries per row. The bandwidth of $B \in \mathbb{R}^{n \times n}$ is bounded by $\sqrt{n}$ [11]. Since this matrix is neither symmetric nor positive definite, we used $A = I + B + B^T$ as a matrix. It is important to note that such a matrix is not guaranteed to be positive definite, but it is very likely. Whether the matrix is truly SPD can be seen from the accuracy of our solvers.

3.2 Algorithms

All iterative methods we used are based on the Conjugate Gradient (CG) method. We either used CG based solvers from *AMGCLWrap* or LinearSolve.jl [12] as a framework and *KrylovJL_CG* from Krylov.jl [13] as a solver.

3.2.1 ILUZero

We used the function *ilu0* from ILUZero.jl [14].

3.2.2 ILU(τ)

We used *IncomleteLU.ilu(A, τ)* from IncompleteLU.jl [15] with $\tau \in \{0.1, 0.2, 0.3, 0.4\}$.

3.2.3 AMG

We used *AlgebraicMultigrid.smoothed_aggregation* and *AlgebraicMultigrid.ruge_stuben* as preconditioners for *KrylovJL_CG*. These are single-threaded implementations [16].
Secondly, we used *AMGCLWrap.AMGPrecon* as a preconditioner. Thirdly, we used *AMGCLWrap*'s algebraic multigrid solver *AMGSolver* with CG as a solver using Sparse Approximate Inverse (SPAI) and ILU as relaxation steps with Smoothed-Aggregation coarsening. Finally, we used *AMGCLWrap*'s one-step Relaxation solver *RLXSolver* with CG as a solver using ILU as preconditioner. *AMGCLWrap* runs on multiple threads [17, 18, 19].

3.2.4 Direct solvers

We used *LinearSolve.jl*'s *CholeskyFactorization*, *LUFactorization*, *UMFPACKFactorization*, *SparspakFactorization*, *KLUFactorization* and *QRFactorization*.

3.3 Power law fitting

In Sections 2.1.2 and 2.2.3 we derived upper bounds given by power laws for the time complexity of forward substitution and the CG algorithm. This is a motivation to use power law fits for the runtime t as a function of the matrix size n. We use a reference matrix size n_0.

$$t(n) = t_0 \left(\frac{n}{n_0} \right)^{\alpha} \tag{4}$$

For a dataset $\{(n_i, t_i) | 1 \leq i \leq N\}$ and a fitfunction $f(n)$, the error is defined as

$$\max_i |\ln(f(n_i)) - \ln(t_i)|.$$

The errors of the power law fits of our benchmarking data are given in the tables in Section 6.1.

3.4 Timings

As mentioned above, for direct solvers and for preconditioned Krylov methods there is an overhead, i.e. computations which have to be done only once for each matrix. For direct methods, the overhead is the factorization and for preconditioned methods it is the computation of the preconditioner. On the other hand there are also computations which have to be done for each new right hand side. These take the time t_{rhs}. The sum of the overhead and t_{rhs} will be denoted by t_{all}.
As a computation time (for both t_{rhs} and t_{all} respectively), we take the minimum elapsed time of multiple executions of the same algorithm for the same matrix size and matrix class. For each execution we used a random vector $x \in \mathbb{R}^n, x_i \in (0, 1)$ as exact solution and then compute $b = Ax$.
The benchmarks were computed on the Weierstrass Institute for Applied Analysis and Stochastics's computers, leonhard-18 (inhomogeneous square), leonhard-22 (random structure, *nnzrow*= 4), leonhard-23 (Finite

Difference $d = 2$ and $d = 3$ and random structure, *nnzrow*$= 10$) and leonhard-24 (Finite Difference $d = 1$ and inhomogeneous cube).
leonhard-18 has 2×18 cores (3 GHz), leonhard-22 has 2×16 cores (4 GHz) and leonhard-23 and leonhard-24 have 2×32 cores (4 GHz). All computers have 768 Gigabytes of RAM. The difference in computers makes a direct comparison of computation time for different matrices difficult. The number of used BLAS threads was always 1.

4 Results

4.1 Runtime

Using Equation 4, we can get t_0, the time required to solve an $n_0 \times n_0$ system. Here we used $n_0 = 10^7$. The times to solve $Ax = b$ for $n = 10^7$, the scaling exponents and the errors are all given as tables in Section 6.1. For three dimensional inhomogeneous meshes the mesh- and matrix-generation are very time intensive steps, such that only matrices with $n \leq 3 \times 10^6$ were investigated. The power law fit obviously still allows a projection for $n = 10^7$.

4.1.1 Finite Difference $d = 1$

The time required to solve a new right hand side, i.e. t_{rhs}, scales linear ($\alpha \approx 1$) for all investigated algorithms (single- and multi-threaded) with the matrix size n. For t_{all} *AlgebraicMultigrid.jl*'s Ruge-Stüben is the only algorithm with a different scaling exponent, namely $\alpha \approx 3$, the others have $\alpha \approx 1$ for t_{all} as well.
We do observe that t_0 hugely varies between different algorithms. t_{rhs} of CG with an ILU based preconditioner is around 30 times smaller than just CG and around 20 times smaller than single-threaded AMG. But some direct solvers such as KLU can be twice as fast as ILU preconditioned CG. The multi-threaded AMG solvers are slower than direct solvers as well.
Considering t_{all}, ILUZero and AMGCLWrap with SPAI as relaxation perform equally well and better than all other algorithms. They are around twice as fast as KLU.
It is interesting to note that our numerical results show a jump in t_{rhs} for Cholesky around $n = 10^7$, while t_{all} is relatively smooth. A visualisation corresponding to this subsection's discussion is contained in Figure 3.

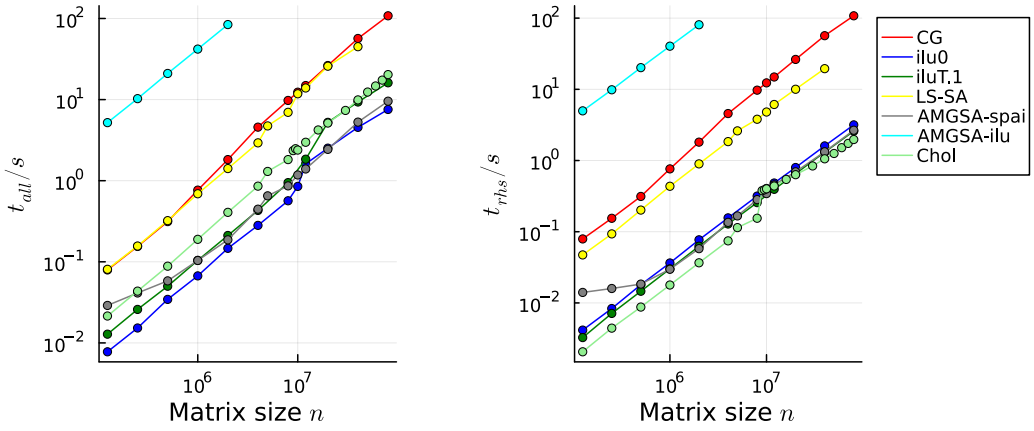

Figure 3: Benchmarking results for Finite Difference matrices with $d = 1$ for some selected algorithms: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

4.1.2 Finite Difference $d = 2$

The first observation is that only ILU based multi-threaded AMG algorithms are able to achieve sublinear scaling, for all other algorithms $\alpha \approx 1$.
t_{rhs} for ILU($\tau = 0.1$) is at least twice as small as for CG, ILUZero and ILU($\tau = 0.4$). Single-threaded AMG performs slightly better than ILU($\tau = 0.1$). There is a significant performance increase for multi-threaded AMG and direct solvers. Cholesky is the fastest single-threaded algorithm for t_{rhs}. CG with Smoothed-Aggregation is the fastest single-threaded algorithm for t_{all}. For both categories SPAI relaxed AMGCLWrap is the fastest solver in general. A visualisation corresponding to this subsection's discussion is contained in Figure 4.

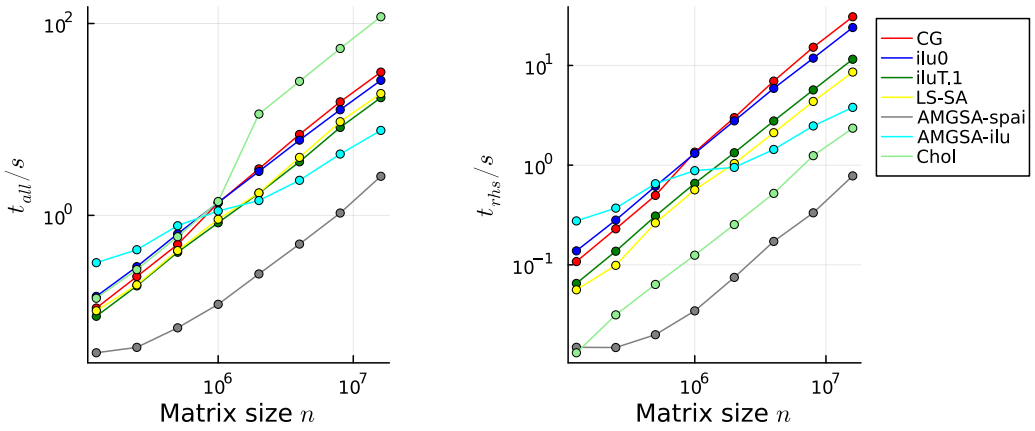

Figure 4: Benchmarking results for Finite Difference matrices with $d = 2$ for some selected algorithms: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

4.1.3 Finite Volume : Inhomogeneous square

One relevant finding is that most algorithms scale with $\alpha \approx 1.2$ for the inhomogeneous mesh in a square. For CG and single-threaded AMG algorithms α increases more than for ILU preconditioned CG. One interesting exception is SparspakFactorization whose scaling exponent only increases from $\alpha = 1.06$ to $\alpha = 1.08$. The scaling exponents of ILU based multi-threaded AMG algorithms increase from 0.68 and 0.47 to 0.77 and 0.71 respectively.
The results are very similar to the Finite Difference matrices for $d = 2$. Cholesky remains the fastest single-threaded algorithm to solve a right hand side. Interestingly the best direct solver is at least 10 times faster than the iterative solvers for the inhomogeneous square, while direct solvers are only three times faster than iterative solvers for Finite Difference matrices. CG with Smoothed-Aggregation is again the fastest algorithm to solve a new LSE, i.e. t_{all} is smallest for this algorithm. A visualisation corresponding to this subsection's discussion is contained in Figure 5.

4.1.4 Finite Difference $d = 3$

All iterative methods, whether single- or multi-threaded, scale linearly. The direct solvers scale significantly worse, for t_{rhs} we find $\alpha \in (1.38, 1.83)$ and for t_{all} we find $\alpha \in (1.79, 2.53)$.
Multigrid-solvers are the fastest single-threaded (and multi-threaded) solvers. The difference to ILU preconditioned CG is significantly more prominent for t_{rhs} and negligible for t_{all}. The use of 128 logical processors leads to a speedup of 16 for t_{rhs}. A visualisation corresponding to this subsection's discussion is contained in Figure 6.

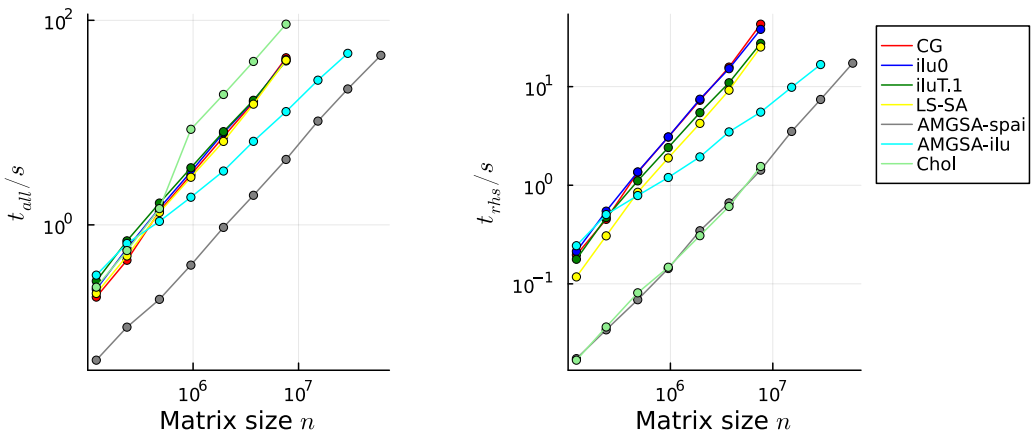

Figure 5: Benchmarking results for Finite Volume on an inhomogeneous square: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

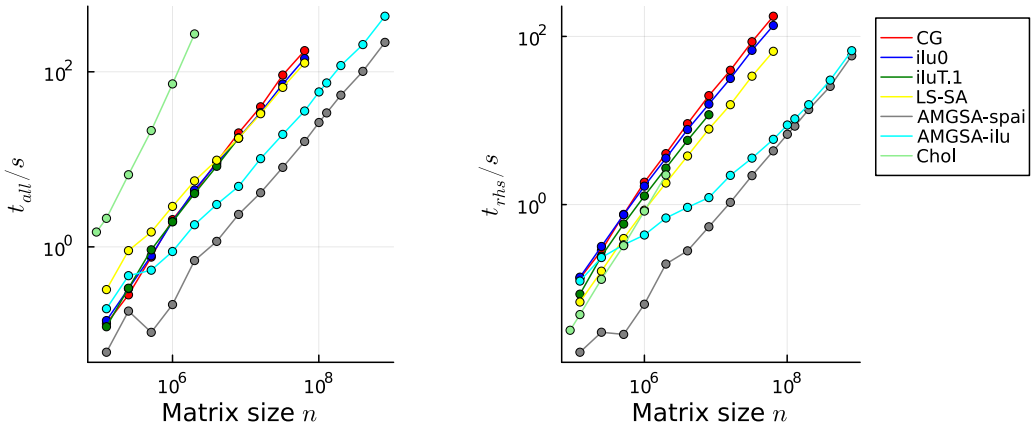

Figure 6: Benchmarking results for Finite Difference matrices with $d = 3$ for some selected algorithms: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

4.1.5 Finite Volume : Inhomogeneous cube

Similar as the transition from Finite Difference $d = 2$ to the inhomogeneous square, the scaling exponents for the inhomogeneous cube are generally larger than for Finite Difference $d = 3$, with single-threaded AMG being especially affected for t_{rhs}. It is interesting to mention that even though we find $\alpha \approx 0.4$ for ILU based multi-threaded AMG solvers, the prefactor t_0 is very large such that the ILU relaxed AMGSolver is projected to be only insignificantly faster than SPAI relaxed AMGSolver ($\alpha = 1.02$) for $n = 10^7$. For t_{all} ILU based multi-threaded AMG solvers also have $\alpha \approx 1$.
CG without a preconditioner is the fastest single-threaded algorithm for both t_{rhs} and t_{all}. For single-threaded algorithms, ILU is a better preconditioner than AMG for our inhomogeneous mesh, contrary to Finite Difference matrices for $d = 3$. A visualisation corresponding to this subsection's discussion is contained in Figure 7.

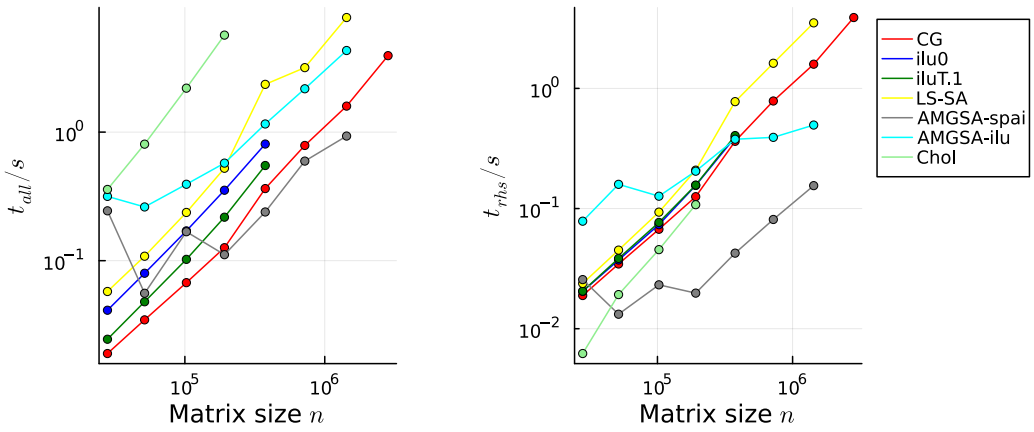

Figure 7: Benchmarking results for Finite Volume on an inhomogeneous cube for some selected algorithms: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

4.1.6 Random structure

The results for $nnzrow = 4$ and for $nnzrow = 10$ are very similar. For t_{all} and for t_{rhs} all single-threaded iterative methods scale approximately linear with the size of the matrix, with AMG preconditioned methods scaling worse than ILU preconditioned methods. In general all tested preconditioners fail to deliver a significant speedup. Interestingly, the AMGSA-ILU performs much worse than AMGSA-SPAI. RLX-ILU has a large t_0 as well, but from our data, the approximated scaling behaviour is sublinear.
Direct solvers have $\alpha \approx 1.5$.
ILU($\tau = 0.1$) is the single-threaded solver with the smallest t_{rhs} and for t_{all} it is CG without any preconditioner. A visualisation corresponding to this subsection's discussion is contained in Figure 8.

4.2 Memory

4.2.1 alloc$_{all}$

For a new LSE, the exact scaling relation of the memory requirements to the problem size depends on the type of matrix. The memory requirements for a new LSE scale linear with the problem size for all single-threaded iterative solvers. CG without a preconditioner requires the least memory from all algorithms. CG with ILU preconditioners requires slightly less memory than AMG preconditioners. For the Finite Difference matrix with $d = 1$, the memory requirements scale linear for direct solvers too. Cholesky requires the least

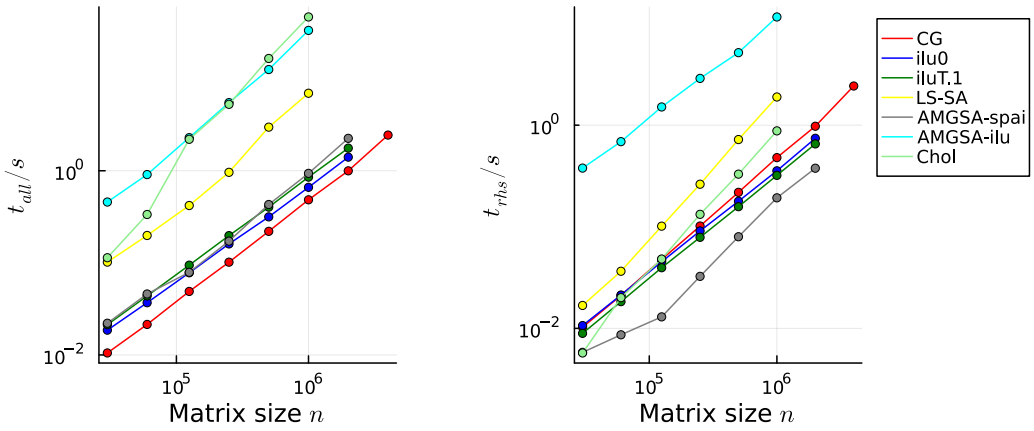

Figure 8: Benchmarking results for SPD matrices with a random structure with *nnzrow*= 4 for some selected algorithms: CG, CG with ILUZero (labeled *ilu0*), with ILU($\tau = 0.1$) (labeled *iluT.1*), with single-threaded Smoothed-Aggregation (*LS-SA*) and two variants of multi-threaded AMG as preconditioner (*AMGSA-spai* and *AMGSA-ilu*) and Cholesky (*Chol*). Both t_{all} (left), the time to solve $Ax = b$ and t_{rhs} (right) the time to solve for a new right hand side are shown as functions of the matrix size n.

and QR, LU and UMFPACK require the highest amount of memory. For all other matrices, QR requires the most amount of memory. Interestingly, the memory requirements of Cholesky for the Finite Difference and Finite Volume matrices in 2D increase a lot around $n = 10^6$. These increases can also be seen in t_{all}. In these cases, Sparspak is the most memory efficient direct solvers.

4.2.2 alloc$_{rhs}$

To solve a new right hand side of any size n, the investigated single-threaded iterative solvers require 80 bytes of memory. The usage of LinearSolve (i.e. LinearSolve with AMGCLWrap.AMGPrecon as preconditoner) is the only investigated multi-threaded solver for which the memory requirements are independent from n as well. For the solvers from AMGCLWrap (i.e. AMGSolver and RLXSolver), the memory requirements scale linear with n. The only direct solvers with constant memory requirements to solve right hand sides are KLU (64 bytes) and UMFPACK (48 bytes). The memory required to solve a new right hand side for Cholesky, LU, QR and Sparspak increases with the size of the problem. Here, LU needs the least and QR the highest amount of memory.

4.3 Accuracy

We use the maximum-norm and the absolute backward error as an error, i.e.

$$d := ||\hat{x} - x||_\infty$$

with $\hat{x}$ being the approximate solution and x being the true solution. For the sake of simplicity, we used each solver's default settings for tolerance.
Even though the specifics depend on the type of matrix used and the matrix size, in general we see that direct solvers achieve errors of around 10^{-13}. The solvers from AMGCLWrap achieve errors of order 10^{-9}. Iterative solvers from LinearSolve.jl achieve errors between 10^{-8} and 10^{-6}. Between the different preconditioners, AMG preconditioners perform best for most matrix types.

5 Conclusion

In this report we have investigated the performance of direct and iterative solvers with various preconditioners for linear systems of equations. We have put an emphasis in analysing the runtime's scaling behaviour with varying problem size.
Which solver performs best, depends not only on the problem class and size, but obviously on the used notion of 'best'. We offer advice on choosing the right solver for your specific application, by distinguishing between t_{all}, the time needed to solve $Ax = b$ and t_{rhs}, the time required to solve $Ax' = b'$, if $Ax = b$ is already solved (i.e. the factorization or the preconditioner is already assembled). E.g. when solving a linear system of equations originating from the discretisation of a 2D elliptic differential operator, direct solvers have a very low t_{rhs}, but a very high t_{all}. Thus, a direct solver should be used if the same differential operator is used for many different right hand sides. For 3D differential operators, an iterative method minimises both t_{all} and t_{rhs}.
We also analysed memory requirements and accuracy of different solvers. Firstly, it is important to note that some solvers such as Cholesky or AMGCLWrap.jl's AMGSolver require ≈ 10 Gigabytes of memory to solve $Ax' = b'$, if $Ax = b$ is already solved, (for $A \in \mathbb{R}^{10^6 \times 10^6}$) while e.g. Conjugate Gradient (CG) with an ILU preconditioner or LinearSolve.jl's CG with AMGCL as a preconditioner require only 80 bytes of memory for the same problem.
Secondly, direct methods are substantially more accurate using the default settings than iterative solvers.
Finally, it is important to note that we also investigated multi-threaded solvers. For some matrix types they were able to achieve sublinear scaling, but this depends on the specific solver and the exact matrix type.
Using an interactive Pluto notebook and an easily expandable benchmark database, you can look closely at the results for a specific problem class you are interested in.

References

[1] The Julia Programming Language. https://julialang.org/.

[2] Johannes Taraz. https://github.com/jotaraz/SPDsolversBenchmarksJL.

[3] Tarjan R.E Gilbert, J.R. The analysis of a nested dissection algorithm. *Numer. Math.*, 1986.

[4] Erin Carson, Jörg Liesen, and Zdeněk Strakoš. Towards understanding CG and GMRES through examples, 2024.

[5] Yousef Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[6] W. Hackbusch. Multi-grid methods and applications. *Springer*, 1985.

[7] Jonathan Richard Shewchuk: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[8] Gmsh. https://gmsh.info/.

[9] Jürgen Fuhrmann, Christian Merdon, Johannes Taraz: ExtendableGrids.jl. https://github.com/j-fu/ExtendableGrids.jl.

[10] Jürgen Fuhrmann, Dilara Abdel, Jan Weidner, Alexander Seiler, Patricio Farell, Matthias Liero: VoronoiFVM.jl. https://github.com/j-fu/VoronoiFVM.jl.

[11] Daniel Karrasch et al.: SparseArrays.jl. https://github.com/JuliaSparse/SparseArrays.jl.

[12] Christopher Rackauckas et al.: LinearSolve.jl. https://github.com/SciML/LinearSolve.jl.

[13] Alexis Montoison et al.: Krylov.jl. https://github.com/JuliaSmoothOptimizers/Krylov.jl.

[14] Matt Covalt et al.: ILUZero.jl. https://github.com/mcovalt/ILUZero.jl.

[15] Harmen Stoppels et al.: IncompleteZero.jl. https://github.com/haampie/IncompleteLU.jl.

[16] Ranjan Anantharaman et al.: AlgebraicMultigrid.jl. https://github.com/JuliaLinearAlgebra/AlgebraicMultigrid.jl.

[17] Jürgen Fuhrmann: AMGCLWrap.jl. https://github.com/j-fu/AMGCLWrap.jl.

[18] Denis Demidov. AMGCL: An Efficient, Flexible, and Extensible Algebraic Multigrid Implementation. *Lobachevskii Journal of Mathematics*, 40(5):535–546, May 2019.

[19] Denis Demidov. AMGCL – A C++ library for efficient solution of large sparse linear systems. *Software Impacts*, 6:100037, 2020.

6 Appendix

6.1 Benchmarking results: scaling behaviour

Table 1: Benchmarking results for t_{all} and t_{rhs} for Finite Difference, d=1

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	12.608	1.049	0.049	12.612	1.048	0.05
ilu0	1.416	0.817	0.034	0.403	0.993	0.006
iluT.1	1.888	1.092	0.264	0.329	0.993	0.006
iluT.2	1.903	1.083	0.259	0.332	0.989	0.002
iluT.3	1.908	1.082	0.255	0.331	0.99	0.004
iluT.4	1.914	1.08	0.255	0.331	0.991	0.005
LS-SA	11.991	0.981	0.087	4.959	0.996	0.033
LS-RS	3053.522	3.045	0.133	7.293	1.097	0.016
LS-AMGCL	2.065	1.096	0.061	1.239	1.014	0.009
AMGSA-spai	1.192	1.023	0.07	0.373	0.934	0.029
AMGSA-ilu	429.219	1.01	0.009	412.235	1.01	0.009
RLX-ilu	3.74	1.03	0.054	2.492	1.005	0.01
Chol	2.257	1.053	0.011	0.271	0.956	0.008
KLU	2.484	0.965	0.075	0.146	1.022	0.027
UMF	6.614	1.003	0.037	0.194	1.013	0.063
LU	6.557	1.0	0.021	0.215	1.046	0.153
QR	7.302	0.966	0.008	0.717	0.82	0.023
SP	11.522	0.947	0.061	3.074	0.965	0.006

Table 2: Benchmarking results for t_{all} and t_{rhs} for Finite Difference, d=2

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	18.957	1.118	0.032	18.83	1.121	0.037
ilu0	15.765	1.052	0.012	14.88	1.034	0.021
iluT.1	10.189	1.112	0.036	7.125	1.038	0.008
iluT.2	11.302	1.14	0.024	8.428	1.051	0.014
iluT.3	11.406	1.145	0.04	8.389	1.05	0.008
iluT.4	15.473	1.113	0.019	13.109	1.054	0.014
LS-SA	11.399	1.156	0.072	5.375	1.018	0.016
LS-RS	20.884	1.119	0.079	6.636	1.02	0.008
LS-AMGCL	2.731	1.087	0.023	1.632	1.004	0.017
AMGSA-spai	1.439	1.123	0.057	0.452	1.11	0.06
AMGSA-ilu	5.174	0.82	0.051	2.784	0.679	0.04
RLX-ilu	9.765	0.567	0.014	7.649	0.474	0.079
Chol	69.779	1.125	0.01	1.465	1.089	0.079
KLU	744.821	1.631	0.269	1.997	1.082	0.032
UMF	93.828	1.411	0.12	1.398	1.074	0.017
LU	93.739	1.411	0.121	1.555	1.116	0.075
QR	176.631	1.278	0.095	11.605	1.171	0.023
SP	53.14	1.383	0.065	3.288	1.055	0.039

Table 3: Benchmarking results for t_{all} and t_{rhs} for Finite Volume on an inhomogeneous square

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	58.094	1.261	0.061	58.105	1.261	0.061
ilu0	55.83	1.204	0.052	51.783	1.198	0.045
iluT.1	53.471	1.151	0.042	36.392	1.161	0.057
iluT.2	52.977	1.169	0.047	41.882	1.177	0.058
iluT.3	55.892	1.181	0.044	44.574	1.187	0.051
iluT.4	59.586	1.173	0.039	48.473	1.176	0.044
LS-SA	55.075	1.269	0.049	33.47	1.242	0.069
LS-RS	71.755	1.253	0.037	32.651	1.22	0.05
LS-AMGCL	12.256	1.179	0.051	8.203	1.215	0.034
AMGSA-spai	6.157	1.13	0.04	2.031	1.201	0.04
AMGSA-ilu	16.93	0.968	0.019	7.16	0.77	0.051
RLX-ilu	29.257	0.808	0.118	22.57	0.709	0.133
Chol	123.368	1.139	0.015	1.989	1.125	0.076
KLU	3289.605	1.766	0.034	3.826	1.149	0.002
UMF	202.21	1.454	0.032	2.811	1.169	0.011
LU	201.521	1.45	0.033	3.068	1.205	0.053
QR	527.021	1.348	0.024	27.919	1.196	0.014
SP	248.1	1.45	0.055	4.534	1.081	0.038

Table 4: Benchmarking results for t_{all} and t_{rhs} for Finite Difference, d=3

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	25.18	1.057	0.059	24.736	1.053	0.025
ilu0	21.777	1.012	0.023	19.697	1.043	0.026
iluT.1	21.989	1.055	0.01	15.115	1.072	0.02
iluT.2	22.429	1.058	0.022	15.697	1.08	0.011
iluT.3	23.432	1.048	0.012	18.554	1.086	0.017
iluT.4	23.881	1.064	0.004	18.819	1.095	0.026
LS-SA	21.467	0.955	0.013	9.82	1.033	0.035
LS-RS	35.395	0.96	0.02	10.332	1.054	0.03
LS-AMGCL	3.888	1.017	0.014	1.842	1.033	0.007
AMGSA-spai	2.65	0.999	0.045	0.599	1.036	0.074
AMGSA-ilu	6.837	0.938	0.063	0.751	1.017	0.057
RLX-ilu	3.35	1.057	0.014	1.64	1.074	0.02
Chol	4583.764	1.786	0.065	20.292	1.379	0.037
KLU	1.4502366354e7	2.534	0.154	581.038	1.827	0.13
UMF	71550.545	2.087	0.075	93.667	1.518	0.077
LU	109702.071	2.201	0.115	110.751	1.559	0.076
QR	320669.888	2.032	0.186	2172.982	1.646	0.104
SP	239642.753	2.277	0.031	50.584	1.435	0.019

Table 5: Benchmarking results for t_{all} and t_{rhs} for Finite Volume on an inhomogeneous cube

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	16.353	1.162	0.071	16.135	1.158	0.067
ilu0	36.134	1.165	0.023	17.936	1.185	0.071
iluT.1	29.289	1.226	0.053	18.127	1.181	0.082
iluT.2	28.794	1.225	0.07	17.69	1.176	0.104
iluT.3	28.153	1.22	0.082	17.257	1.171	0.115
iluT.4	28.228	1.22	0.057	17.391	1.172	0.092
LS-SA	96.208	1.258	0.416	57.066	1.377	0.221
LS-RS	131.406	1.272	0.025	55.213	1.349	0.05
LS-AMGCL	12.194	1.068	0.148	5.305	1.057	0.061
AMGSA-spai	8.676	1.089	0.185	1.162	1.021	0.043
AMGSA-ilu	30.222	1.0	0.018	1.148	0.403	0.209
RLX-ilu	14.882	0.946	0.006	0.836	0.382	0.153
Chol	1629.169	1.438	0.036	35.509	1.455	0.142
KLU	2.5965955119e7	2.52	0.054	321.817	1.605	0.033
UMF	194732.93	2.16	0.1	368.288	1.679	0.053
LU	192234.23	2.157	0.099	357.826	1.67	0.006
QR	434046.118	2.053	0.068	2452.512	1.641	0.008
SP	578255.014	2.296	0.156	100.419	1.504	0.021

Table 6: Benchmarking results for t_{all} and t_{rhs} for Random structure, nnzrow=4

	All			RHS		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	6.715	1.146	0.061	6.635	1.146	0.073
ilu0	7.427	1.045	0.027	3.701	1.009	0.02
iluT.1	9.619	1.055	0.013	3.356	1.019	0.006
iluT.2	8.26	1.043	0.009	3.453	1.013	0.009
iluT.3	10.801	1.116	0.106	3.683	1.013	0.019
iluT.4	9.71	1.098	0.032	3.948	1.017	0.016
LS-SA	169.843	1.378	0.092	49.142	1.413	0.02
LS-RS	155.916	1.27	0.057	35.019	1.183	0.068
LS-AMGCL	14.79	1.148	0.041	3.398	1.031	0.065
AMGSA-spai	16.099	1.224	0.045	2.726	1.188	0.086
AMGSA-ilu	614.234	1.28	0.056	101.951	0.967	0.084
RLX-ilu	9.226	0.781	0.055	4.332	0.639	0.009
Chol	1411.246	1.489	0.098	21.488	1.389	0.034
KLU	400068.489	2.278	0.135	97.206	1.55	0.041
UMF	9274.973	1.83	0.233	127.326	1.69	0.052
LU	10881.791	1.87	0.259	125.223	1.685	0.028
QR	32994.384	1.845	0.028	818.887	1.604	0.012
SP	35022.879	1.988	0.047	40.902	1.446	0.023

Table 7: Benchmarking results for t_{all} and t_{rhs} for Random structure, nnzrow=10

	All			**RHS**		
Algorithm	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**	**Time**($n = 10^7$) in s	**Scaling Exponent**	**Error**
CG	8.514	1.022	0.032	8.616	1.032	0.026
ilu0	32.087	1.122	0.132	7.96	1.052	0.026
iluT.1	22.12	1.062	0.029	6.93	1.024	0.012
iluT.2	37.184	1.131	0.643	7.206	1.029	0.059
iluT.3	19.615	1.058	0.009	8.29	1.046	0.021
iluT.4	18.918	1.056	0.022	8.229	1.036	0.044
LS-SA	192.381	1.326	0.219	13.363	1.107	0.106
LS-RS	323.511	1.289	0.133	68.017	1.263	0.102
LS-AMGCL	9.18	0.93	0.071	5.424	0.964	0.016
AMGSA-spai	6.723	1.051	0.033	2.088	1.115	0.141
AMGSA-ilu	807.075	1.066	0.139	771.581	1.078	0.128
RLX-ilu	24.73	0.615	0.005	20.202	0.596	0.013
Chol	11488.342	1.825	0.318	65.001	1.517	0.057
KLU	829747.559	2.215	0.031	71.927	1.353	0.168
UMF	5209.304	1.582	0.073	147.797	1.563	0.169
LU	4704.613	1.562	0.018	130.409	1.538	0.169
QR	9796.6	1.575	0.091	1442.183	1.683	0.327
SP	27035.052	1.742	0.043	52.761	1.411	0.018

6.2 Input for inhomogeneous meshes

Table 8: Inputs for inhomogeneous squares

Meshsize h	Matrix size	Number of nonzero entries
0.01	30633	213001
0.0071428571428571435	59762	416332
0.005	120833	842977
0.0035714285714285718	235986	1647908
0.0025	479579	3351351
0.0017857142857142859	953078	6663568
0.00125	1941892	13581850
0.0008928571428571429	3735804	26134678
0.000625	7609196	53241590
0.0004464285714285714 7	15266903	106836431
0.0003188775510204082	29281187	204923663
0.0002232142857142857 3	60306527	422081915

Table 9: Inputs for inhomogeneous cubes

Meshsize h	Matrix size	Number of nonzero entries
0.1	2312	29252
0.07936507936507937	4321	56553
0.06298815822625346	7661	103191
0.04999060176686783	14270	197336
0.03967508076735542	27968	395952
0.03148815933917097	51536	742154
0.02499060265013569	102531	1499999
0.019833811627091816	191870	2839958
0.01574112033896176	375449	5611805
0.01249295264996965	720176	10846422
0.009915041785690198	1432635	21735001