UPPSALA
UNIVERSITET

Symbol based Multigrid method

Hreinn Juliusson, Johanna Brodin, Tianhao Zhang

Project in Computational Science: Report

February 20, 2018

PROJECT REPORT

Department of Information Technology

Abstract

Discretization of certain groups of partial differential equations (PDEs) on simple domains in 2D or 3D results in large sparse linear systems. When the domains in question are rectangular or brick forms and the discretization is done using regular quadrilateral or hexagonal meshes, these specific systems possess a certain structure that can be utilized in various ways. The corresponding system matrices can be seen as parts of matrix sequences with respect to the discretization parameter and belong to the class of the so-called Generalized Locally Toeplitz (GLT) (see [7]). This is of particular interest as the matrix entries of GLT systems can be computed with an analytical function referred to as the symbol. Multilevel and Multigrid methods are very efficient methods for solving large sparse linear systems. The GLT theory has been used to develop a function-based preconditioner called GLT Multigrid (GLT-MG).

In this work we confirm numerically what has been proven theoretically, namely, that the GLT-MG V-cycle is of optimal order and that it can reduce the overall computational cost.

All previous implementations of GLT-MG have been made in Matlab, which does not allow for a comprehensive performance study. We perform the numerical study in the C++ open-source Finite Element framework deal.II. The isotropic Laplace equation is uniformly discretized on the unit cube domain, and the resulting sparse linear system is used as a benchmark problem when comparing the performance of GLT-MG and Trilinos AMG, a broadly used Multigrid implementation that can handle general linear systems.

A problem that has previously been observed with algebraic Multigrid with respect to its scalability is the bottleneck of matrix-vector multiplications. This is why we also, in addition to the GLT-MG implementation, implement a sparse storage format more suitable for GLT matrices. The tested format is a diagonal-wise sparse storage, which has not shown results matching its potential when implemented in Matlab.

Keywords: Multigrid methods, Function-based methods, Symbol-based methods, Three-dimensional Laplace, Generalized Locally Toeplitz, High performance computing, Efficiency analysis, Preconditioner, Algebraic Multigrid, Geometric Multigrid, Performance study, deal.II

Contents

1 Introduction

Systems of linear algebraic equations are usually written in the form

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where A is a matrix of size $n \times n$, $\boldsymbol{b}$ is a given right-hand-side vector and $\boldsymbol{x}$ is the solution we want to compute. When n is large, the system can not be easily solved using a direct solution method. For this task iterative solution methods are being used and developed.

In this project we consider very large scale linear systems resulting from finite element discretization of partial differential equations (PDEs). More specifically, we focus on systems arising when using regular quadrilateral or hexagonal meshes in 2D or 3D.

The matrices of these linear systems are *large*, *sparse* and *structured*. There is already much experience on how to solve large and sparse discrete PDE problems, and the methods of choice are preconditioned iterative methods. Preconditioning means using a particular technique to accelerate the convergence of the iterative methods and thereby ensuring faster and more reliable large scale computer simulations.

Among the best known preconditioners for linear systems from discretized PDEs are the so-called Multigrid methods. The reason they are widely used is their optimality properties, obtained for a rather broad class of problems. When properly constructed, Multigrid possess

(i) optimal convergence rate and

(ii) optimal computational complexity.

Property (i) means that Multigrid-preconditioned iterative methods converge in a number of iterations that is independent of the total number of degrees of freedom (DoF) - the size n of the system matrix A. Property (ii) reveals that the number of arithmetic operations per one Multigrid-preconditioned iteration is linearly proportional to n. Thus, (i) and (ii) result in an overall computational cost linearly proportional to n.

Although very attractive, the Multigrid methods have some drawbacks. They have high memory demands (as is outlined further in Section 2.3) and high construction costs. Furthermore, as show in e.g. [3], Multigrid may exhibit less favorable scaling when implemented in parallel, the main reason being badly scaling sparse matrix-vector multiplication.

In this project we emphasize *structure*, the last property of the target matrices, namely, that they are structured and fall in the class of the so-called Generalized Locally Toeplitz (GLT) matrices [10]. GLT matrices are characterized by an analytical function that can be used to compute their entries and that can also give information about the spectrum of

the matrices. The latter allows for constructing efficient preconditioners, and in particular, Multigrid preconditioners, based on the knowledge of the above-mentioned symbol of the matrix. We refer to the symbol-based Multigrid as GLT-MG, which stands for Generalized Locally Toeplitz Multigrid.

The classical GLT theory and the construction of the GLT-based Multigrid method assume simple domains (rectangular or brick forms) and regular quadrilateral or hexahedral meshes. We note that, although seemingly simple, such discretizations are used for solving various problems of practical interest. Furthermore, the framework can be applied for more general discretization meshes as long as they are topologically equivalent to the above mentioned regular meshes.

The GLT-MG idea is not new. It was introduced 1991 (see [9]) and has been studied and implemented for various 1D and 2D problems, see [8]. All earlier GLT-MG implementations are in Matlab. The symbol for the 3D discrete Laplace operator (using trilinear finite elements) is derived in [6], and is used as a test problem for a GLT-MG implementation in [7]. The GLT-MG implementation from that paper is also in Matlab and this severely limits the size of the tested problems to only a couple of thousands of DoF. Further, the Matlab implementation does not allow for a thorough performance study and comparison of the computational efficiency of GLT-MG with other optimized, publicly available Multigrid implementations.

The aim of this project is divided into three parts:

(1) Implement GLT-MG for the 3D discrete Laplace operator in C++ using the open-source finite element library deal.II ([1]).

(2) Implement sparse matrix storage and matrix-vector multiplication using a diagonal storage scheme, to take full advantage of the structure of the target matrices.

(3) Compare the performance of the so-implemented GLT-MG with that of a state-of-the-art Multigrid solver from the broadly used scientific computing library Trilinos (see [2]).

The structure of the remaining report is as follows. In Section 2, we briefly describe the idea of iterative methods, preconditioning and Multigrid. In Section 3, we introduce matrices with Toeplitz structure, how the matrices relate to GLT and their symbol, and finally the GLT-MG algorithm itself. Section 4 is devoted to sparse matrices in diagonal storage and the corresponding algorithm for matrix-vector multiplication. In Section 5, we show various numerical experiments and we finalize the report with Section 6, where the conclusions and future extensions of this work are presented.

2 Iterative methods, Preconditioning and Multigrid

In the following section, we give some background to preconditioning and Multigrid by first briefly describing the idea of the iterative methods in Subsection 2.1. After introducing the need of preconditioning the iterative methods, we give some details on various types of preconditioning in Subsection 2.2. This leads to Subsection 2.3, where we introduce the concept of Multigrid methods, the difference between Geometric and Algebraic Multigrid, and the most often used V-cycle recursion scheme.

2.1 Iterative methods

The most basic iterative scheme is the following simple iteration

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - (A\boldsymbol{x}^k - \boldsymbol{b}), \quad k = 0, 1, \ldots \tag{2}$$

The cost of one iteration is linear with respect to n, however, the problem is that convergence can be very slow. Therefore, there are various techniques to speed up the convergence.

One illustrative example is

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - M^{-1}(A\boldsymbol{x}^k - \boldsymbol{b}), \tag{3}$$

where M is some suitably chosen matrix referred to as a preconditioner. As the name suggests, the role of M is to improve the properties of A so that the convergence gets faster. Indeed, Equation (3) can be rewritten as

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - (M^{-1}A\boldsymbol{x}^k - M^{-1}\boldsymbol{b}), \tag{4}$$

thus, Equation (3) is Algorithm (2) applied to the system

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}. \tag{5}$$

Ideally, the fastest convergence will be achieved if $M \equiv A$, however this is impractical due to the high cost of computing A^{-1}. Instead M is chosen as an approximation of A.

Even preconditioned, the simple iteration methods of the type (2) and (3), as well as other methods like Jacobi, Gauss-Seidel iterations, SOR, SSOR etc, turn out to be slow and inefficient. Therefore, in practice we use Krylov subspace iteration methods, such as the Conjugate Gradient (CG) method for symmetric positive definite matrices, the Generalized Minimal Residual (GMRES) method for general non-symmetric matrices and many more, described for instance in [11].

2.2 Preconditioning

Even though Krylov subspace iteration methods are much more efficient than the simple iteration methods, they still need to be preconditioned in order to exhibit fast convergence. There are various ways to precondition the original system $Ax = b$, as presented below.

(i) Left preconditioning: we consider the system

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b} \tag{6}$$

which has the same solution as $A\boldsymbol{x} = \boldsymbol{b}$. The reason for the slow convergence of the original system is the high condition number of A. When preconditioning in this way we aim $M^{-1}A$ to have a much smaller condition number than A.

(ii) Right preconditioning:

$$AM^{-1}\boldsymbol{y} = \boldsymbol{b} \tag{7}$$

$$M\boldsymbol{x} = \boldsymbol{y}, \tag{8}$$

where we recover x by one more solution with M. We note that the condition number for $M^{-1}A$ is the same as that of AM^{-1}

(iii) Implicitly defined preconditioning:

$$[M^{-1}]A\boldsymbol{x} = \boldsymbol{b} \tag{9}$$

or

$$A[M^{-1}]\boldsymbol{y} = \boldsymbol{b} \tag{10}$$

are used.

where $[\bullet]$ denotes the action of a matrix on a vector. In both cases (i) and (ii) M is assumed to be an explicitly given matrix, constructed in advance. This is not necessary in general. To precondition A it suffices to have a procedure that describes the action M^{-1} on a vector without constructing M explicitly. The most efficient preconditioners nowadays belong to this class: the Multilevel and Multigrid preconditioners. Here we deal with Multigrid preconditioners, which are described in more detail in the next subsection.

The most important requirements for a good preconditioner are:

- the condition number of $M^{-1}A$ should be much less than that of A;
- the solution of systems with M should be much easier than with A;
- the construction of M should be relatively cheap;
- to construct and solve systems with M should be well-parallelizable.

2.3 Multigrid Methods

Here we briefly describe the serial framework of the Multigrid method. However, we start with defining the error we want to reduce by using the method. The error vector to the iterative solution of the linear system (1) is the difference between the exact (unknown) solution $\boldsymbol{x}$ and the iterate solution $\boldsymbol{e}_\ell = \boldsymbol{x} - \boldsymbol{x}_\ell$.

The error vector can be described as a linear combination of the eigenvectors of A. The components of $\boldsymbol{e}_\ell$ corresponding to larger eigenvalues are referred to as high frequency errors while error components corresponding to small eigenvalues are referred to as low frequency errors. High frequency errors can usually be removed using local averaging, sometimes referred to as smoothing. Many iterative methods handle the high frequencies well but need more time to eliminate the error components corresponding to the lower frequencies. Multigrid deals with this problem by creating a hierarchy of coarse levels (meshes). The idea is to change from fine grid to coarse grid and from coarse to fine, in order to use different grid resolutions to remove different error frequencies. The aim is to eliminate the high frequencies on the fine grid and then eliminate the remaining error frequencies on the coarser grids. Iterative methods such as Jacobi or Gauss-Seidel are relatively cheap for error corrections and work as smoothers for Multigrid. These methods can be applied on each level of the hierarchy to target the different frequencies of the error.

Multigrid is built on a hierarchy of coarse levels with coarse level matrices. When A is based on a discretization mesh the fine level matrix becomes $A = (A_h)$, with a mesh size parameter h, and the coarse level matrices become $A_{2h}, A_{4h} \ldots$. Multigrid methods generally contain three principal parts:

1. the prolongation operator P,
2. the restriction operator R,
3. and a smoother operator S.

The prolongation and restriction operators work in reverse order, quite often $R = P^T$. The restriction operator downsamples the residual error from a finer grid to a coarser while the prolongation operator does the opposite and interpolates a correction of the residual error from a coarser grid to a finer. The smoother reduces high frequency errors relative to the current grid level. The smoothers for a Multigrid implementation are chosen depending on the problem type. It is worth noting that P, R, and A_i are usually represented as matrices, however they can e.g. be implicitly defined via procedures.

2.3.1 Geometric and Algebraic Multigrid

The are two types of Multigrid, geometric Multigrid and algebraic Multigrid (AMG). Whether a Multigrid method is classified as geometric or algebraic depends on how the prolongation and restriction operators are created. The geometric Multigrid is build on

a (nested) sequence of discretization meshes and the level matrices are discrete operators on each grid. The restriction operator works on a vector by averaging neighboring grid points. Similarly, the prolongation operator works on a vector by interpolating between each neighboring grid point.

AMG doesn't consider any underlying hierarchy of grids, discretizing (1). The prolongation and restriction operators are matrices created consequently, based on some graph algorithms and aggregation techniques. The operators act on the level matrix $A_{\ell-1}$ by matrix multiplication to create a coarser level matrix A_ℓ, as shown below in Equation (11),

$$A_\ell = P_\ell^{\ell-1} A_{\ell-1} R_{\ell-1}^\ell. \tag{11}$$

2.3.2 The V-cycle

Performing one iteration with the Multigrid method is usually done using the recursion form referred to as a V-cycle. The name is related to how the Multigrid is visually represented (see Figure 1). Figure 1 illustrates how the restriction operator takes data from the finest level to the coarsest and then the prolongation operator brings it back up again. In every intermediate step the smoother operator works on the current level to reduce error frequencies.

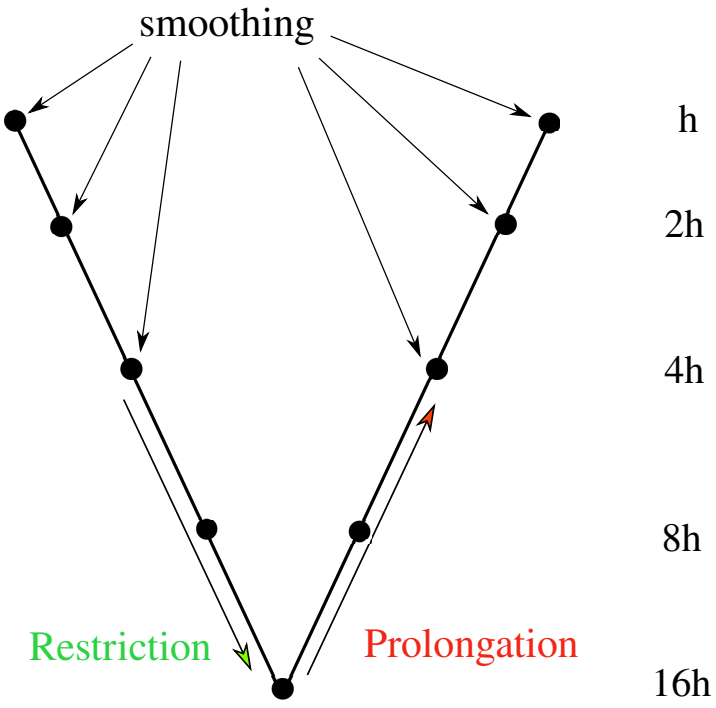


Figure 1: V-cycle Schedule.

Figure 1 depicts in this case a geometric Multigrid V-cycle with the decreasing mesh size represented by the length between mesh points h, $2h$, $4h$ etc. Multigrid may have different

recursion forms depending on what types of error frequencies are prevalent. It might be necessary to do extra smoothing iterations on the two coarsest grids in one cycle to target low frequency errors. In that case the recursion would visually look more like a 'W' , see [4].

The recursive form can be changed and developed to suit the problem at hand, if it's beneficial the Multigrid method could restrict and prolongate as much as needed in one recursion cycle. However, it is worth to note that the V-cycle is by far the most commonly used recursion form, since other forms tend to be more expensive and not worth their cost in practice.

The Multigrid V-cycle iteration scheme (depicted in Figure 1) is described below, first with an algorithm for a V-cycle with a hierarchy of only two grids, followed by a V-cycle with m number of grids.

1. Apply smoothing $A_h\boldsymbol{x_h} = \boldsymbol{b_h}$ using Jacobi or Gauss-Seidel iterations.
2. Compute the residual $\boldsymbol{r_h} = \boldsymbol{b_h} - A_h\boldsymbol{x_h}$
3. Restrict the residual to the coarser grid $R(\boldsymbol{r_h}) \rightarrow \boldsymbol{r_{2h}}$
4. Solve $A_{2h}\boldsymbol{e_{2h}} = \boldsymbol{r_{2h}}$, where $\boldsymbol{e_{2h}}$ is the coarse error correction.
5. Prolong $\boldsymbol{e_{2h}}$ back to the finer grid $P(\boldsymbol{e_2}) \rightarrow \boldsymbol{e_h}$
6. Add the correction to the solution vector $\boldsymbol{x_h} = \boldsymbol{x_h} + \boldsymbol{e_h}$.
7. Apply smoothing again on $A_h\boldsymbol{x_h} = \boldsymbol{b_h}$

The V-cycle is next expanded to handle m levels $\ell = 1, \ldots, m$ with $\ell = 1$ being the finest level and $\ell = m$ the coarsest. The algorithm is recursive and the base case is for the coarsest grid where usually a direct solver is used to compute the coarsest level solution (see [5]). Algorithm 1 shows one iteration of a Multigrid V-cycle.

Algorithm 1 $\mathcal{MG}(\ell, \mathbf{x}_\ell, \mathbf{b}_\ell)$, a Multigrid V-cycle

if $\ell = m$ **then**
  $\mathbf{x}_\ell \leftarrow A_\ell^{-1}\mathbf{b}_\ell$ ▷ Coarsest level solver
else
  $\mathbf{x}_\ell \leftarrow S_\ell^{\nu_\ell}(\mathbf{x}_\ell, \mathbf{b}_\ell)$ ▷ do ν_ℓ pre-smoothing steps
  $\mathbf{r}_\ell \leftarrow \mathbf{b}_\ell - A_\ell\mathbf{x}_\ell$ ▷ compute residual r
  $\mathbf{r}_{\ell+1} \leftarrow R_\ell(\mathbf{r}_\ell, \mathbf{x}_\ell)$ ▷ restrict to coarser grid
  $\boldsymbol{\delta}_{\ell+1} \leftarrow \mathcal{MG}(\ell+1, \mathbf{0}_{N_{\ell+1}}, \mathbf{r}_{\ell+1})$ ▷ apply MG recursively
  $\boldsymbol{\delta}_\ell \leftarrow P_\ell(\boldsymbol{\delta}_{\ell+1}, \mathbf{x}_\ell)$ ▷ prolong the error δ to finer grid
  $\mathbf{x}_\ell \leftarrow \mathbf{x}_\ell + \boldsymbol{\delta}_\ell$ ▷ update the iterative solution on the current level
  $\mathbf{x}_\ell \leftarrow S_\ell^{\tilde{\nu}_\ell}(\mathbf{x}_\ell, \mathbf{b}_\ell)$ ▷ do $\tilde{\nu}_\ell$ post-smoothing steps

The main drawbacks of the Multigrid method are its high memory demand, setup time and problem-dependent smoothers. With every level we associate a coarse matrix, a prolongation operator and a restriction operator, all of them usually stored as matrices. For large systems the storage demand can make the method unfeasible depending on how much resources are available. The setup time is maybe not as big a problem as the storage but is often large, and if it can not be amortized by the solution time, then other methods might be more appropriate. Further, even though simple smoothers can be used (e.g. when using simple relaxations in AMG), the smoothers might need to be sufficiently robust and problem-specific (a common case when using Geometric Multigrid in 3D). The construction of these smoothers can become costly.

3 Toeplitz matrices and GLT-MG

In this section we introduce the concept of GLT and GLT-MG by first defining Toeplitz matrices and GLT sequences (of matrices) in Subsection 3.1. We continue by showing how these sequences can be represented by analytical functions (the so-called matrix symbols) in Subsection 3.2. Subsection 3.3 ties together the symbol with Multigrid and presents the GLT-MG method and algorithm.

3.1 Toeplitz and GLT

In this project we focus on large and sparse matrices that also possess a particular band or block-band structure. These matrices belong to the Toeplitz matrix class or to the generalization of Toeplitz-like matrices, a class referred to as Generalized Locally Toeplitz (GLT). The GLT theory uses a rather abstract and difficult mathematical framework that falls beyond the scope of this project, but below we briefly describe some GLT elements that have been used in the work presented in this report.

A Toeplitz matrix $A^{\mathcal{T}}$ is a matrix with constant diagonals from the upper left to the lower right corner. It can be written on the form

$$A^{\mathcal{T}} = \begin{bmatrix} a_0 & a_1 & a_2 & \dots & \dots & \dots & a_n \\ a_{-1} & a_0 & a_1 & a_2 & & & \vdots \\ a_{-2} & a_{-1} & \ddots & \ddots & & & \vdots \\ \vdots & a_{-2} & \ddots & \ddots & \ddots & a_2 & \vdots \\ \vdots & & \ddots & a_{-1} & a_0 & a_1 & a_2 \\ \vdots & & & a_{-2} & a_{-1} & a_0 & a_1 \\ a_{-n} & \dots & \dots & \dots & a_{-2} & a_{-1} & a_0 \end{bmatrix}, \tag{12}$$

where each element a_i is constant along the related offset diagonal $i \in [-n, n] \subset \mathbb{Z}$. (Note that the elements a_i does not have to be equal to the elements a_{-i}.)

A Block Toeplitz matrix $A^{\mathcal{BT}}$, denoted as

$$A^{\mathcal{BT}} = \begin{bmatrix} a_{11}^{\mathcal{T}} & a_{12}^{\mathcal{T}} & \dots & a_{1n}^{\mathcal{T}} \\ a_{21}^{\mathcal{T}} & a_{22}^{\mathcal{T}} & & \vdots \\ \vdots & & \ddots & \\ a_{n1}^{\mathcal{T}} & \dots & & a_{nn}^{\mathcal{T}} \end{bmatrix}, \tag{13}$$

is a matrix where each element $a_{ij}^{\mathcal{T}}$ contains a Toeplitz matrix like the one shown in Equation (12). Such matrices often occur when discretizing PDEs. For instance, for the case of the discretized Laplace problem where the resulting linear system has the GLT attribute. The corresponding system matrix A can be rearranged through matrix permutations such that A becomes a block-Toeplitz matrix $A^{\mathcal{BT}}$ (as shown in Figure 5, see Subsection 5.1).

3.2 The Symbol

When we solve a PDE (or an ODE) numerically, we choose a suitable mesh for the discretization. To this mesh, we associate a discretization parameter h which describes how fine the mesh is. Thus, for each h we obtain a matrix A_n where n is related to $\frac{1}{h}$. In this way, when $h \to 0$ we obtain a sequence of matrices $\{A_n\}_n$ of increasing size. Such a sequence can be seen as a GLT sequence, and can under certain conditions be related to an analytical function f, referred to as the *generating function* or the matrix *symbol* (see [10]). The symbol of A_n can be used to compute its entries. Sequences of discretized matrices such as $\{A_n\}_n$ are common for many classes of differential equations discretized using local methods such as the finite difference method, the finite element method with locally supported basis functions, finite volumes, isogeometric analysis, etc. (see [10]).

The target problem in this work, the 3D anisotropic Laplace equation is given in Equation (14),

$$\mathcal{L}(u) \equiv \sum_{i=1}^{3} \varepsilon_i \frac{\partial^2 u}{\partial x_i^2} = g, \tag{14}$$

where each partial derivative is weighted with a factor ε_i, $i = 1, 2, 3$. To each partial derivative $\frac{\partial^2}{\partial x_i^2}$ corresponds a symbol function f_i which is a Fourier coefficient of the matrix symbol f, defined on the domain $Q^d = [-\pi, \pi]^d, d \geq 1$. The Fourier coefficient functions f_i are given in equation (15) for the equidistantly discretized unit cube domain with step size h (see [7]).

$$\begin{aligned} f_1(\theta_1, \theta_2, \theta_3) &= (4 + 2cos(\theta_1))(1 + 0.5cos(\theta_2))(2 - 2cos(\theta_3)), \\ f_2(\theta_1, \theta_2, \theta_3) &= (4 + 2cos(\theta_1))(2 - 2cos(\theta_2))(1 + 0.5cos(\theta_3)), \\ f_3(\theta_1, \theta_2, \theta_3) &= (2 - 2cos(\theta_1))(4 + 2cos(\theta_2))(1 + 0.5cos(\theta_3)). \end{aligned} \tag{15}$$

The resulting system matrix symbol f given in equation (16) is a linear combination of its Fourier components f_i.

$$f(\theta_1, \theta_2, \theta_3) = \frac{h}{9}(\varepsilon_1 f_1(\theta_1, \theta_2, \theta_3) + \varepsilon_2 f_2(\theta_1, \theta_2, \theta_3) + \varepsilon_3 f_3(\theta_1, \theta_2, \theta_3)) \tag{16}$$

For a more complete description and theoretical results on the GLT theory and matrix symbols, see [10].

3.3 GLT-MG Method

As already mentioned, the symbol of a matrix sequence can be used to compute the entries of the matrices A_n for any n. Another important property of the symbol is that it can give information about the spectrum of A_n. Most important for this project is that we can construct a Multigrid method that makes use of the symbol. This Multigrid method is referred to as GLT-MG, and according to theory it shows better performance than the corresponding geometric or algebraic Multigrid.

In GLT-MG the symbol is used to create the hierarchy of level matrices as well as the prolongation, and restriction operators. In all other aspects the GLT-MG follows the conventional Multigrid steps given in Algorithm 1. The goal of this project is to show that the symbol-based Multigrid GLT-MG is more efficient than a general AMG when applied to GLT systems. A very important aspect of the GLT-MG theory is the option to use the symbol to make the method matrix-free, trading a decrease in memory complexity for an increase computation complexity. This becomes especially interesting when implementing GLT-MG for modern HPC clusters (see [7]).

As already mentioned, what sets GLT-MG apart from a general AMG is the use of the symbol to create the prolongation and the restriction operators. The algorithm for one V-cycle of GLT-MG is given in Algorithm 2.

Algorithm 2 $\mathcal{GLTMG}(\ell, \mathbf{x}_\ell, \mathbf{b}_\ell)$

if $\ell = m$ **then**
  $\mathbf{x}_\ell \leftarrow A_\ell^{-1}\mathbf{b}_\ell$
else
  $\mathbf{x}_\ell \leftarrow S_\ell^1(\mathbf{x}_\ell, \mathbf{b}_\ell)$ ▷ one Jacobi step
  $\mathbf{r}_\ell \leftarrow \mathbf{b}_\ell - A_\ell \mathbf{x}_\ell$ ▷ compute residual
  $\mathbf{r}_{\ell+1} = R_\ell \mathbf{r}_\ell$, ▷ **R constructed using the symbol**
  $\boldsymbol{\delta}_{\ell+1} \leftarrow \mathcal{GLTMG}(\ell + 1, \mathbf{0}_{N_{\ell+1}}, \mathbf{r}_{\ell+1})$ ▷ GLT-MG recursion, initial guess 0
  $\boldsymbol{\delta}_\ell = P_\ell \boldsymbol{\delta}_{\ell+1}$ ▷ $\mathbf{P} = \mathbf{R}^\mathbf{T}$
  $\mathbf{x}_\ell \leftarrow \mathbf{x}_\ell + \boldsymbol{\delta}_\ell$ ▷ update iterative solution
  $\mathbf{x}_\ell \leftarrow \tilde{S}^1(\mathbf{x}_\ell, \mathbf{b}_\ell)$ ▷ one weighted Jacobi step with factor $\frac{2}{3}$

Let $\ell = 1, 2, \ldots, m$ be the level numbers in a given hierarchy of *nested* meshes $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$, obtained by regular refinements of a given coarsest mesh $\mathcal{T}_m$. Thus, $\mathcal{T}_1$ is the finest mesh, where we want to compute the solution. Let ℓ be the current level number. The most

important ingredient of GLT-MG is the prolongation operator P_ℓ. Then $R_\ell = P_\ell^T$ and $A_{\ell+1} = R_\ell A_\ell P_\ell$. The prolongation P_ℓ is defined as

$$P_\ell = T_{n_\ell}(p_\ell) H_\ell, \tag{17}$$

where T_{n_ℓ} is a Toeplitz matrix, generated by a special trigonometric polynomial, related to the symbol of the fine level matrix $A = (A_1)$. H_ℓ is a rectangular matrix of size $(n_\ell, n_{\ell+1})$ and n_ℓ is the size of the mesh at level ℓ.

Due to the complexity of the mathematics involved, we omit the details regarding the trigonometric polynomial p_ℓ. We only mention that the construction is recursive, it is explicitly available and uses information of the zeros of the symbol on the finest level $\ell = 1$. The matrix H_ℓ, referred to as the "cutting" or "shrinking" matrix, is also explicitly defined on the regular meshes in 1D, 2D and 3D.

For instance, in 3D for the twice refined unit cube the matrices T_2, H_2 and P_2 are as follows:

$$T_2 = \begin{pmatrix}
8 & 4 & & 4 & 2 & & & & 4 & 2 & & 2 & 1 & & & & & & & & & & & \\
4 & 8 & 4 & 2 & 4 & 2 & & & 2 & 4 & 2 & 1 & 2 & 1 & & & & & & & & & & \\
 & 4 & 8 & & 2 & 4 & & & & 2 & 4 & & 1 & 2 & & & & & & & & & & \\
4 & 2 & & 8 & 4 & & 4 & 2 & & 2 & 1 & & 4 & 2 & & 2 & 1 & & & & & & & \\
2 & 4 & 2 & 4 & 8 & 4 & 2 & 4 & 2 & 1 & 2 & 1 & 2 & 4 & 2 & 1 & 2 & 1 & & & & & & \\
 & 2 & 4 & & 4 & 8 & & 2 & 4 & & 1 & 2 & & 2 & 4 & & 1 & 2 & & & & & & \\
 & & & 4 & 2 & & 8 & 4 & & & & & 2 & 1 & & 4 & 2 & & & & & & & \\
 & & & 2 & 4 & 2 & 4 & 8 & 4 & & & & 1 & 2 & 1 & 2 & 4 & 2 & & & & & & & \\
 & & & & 2 & 4 & & 4 & 8 & & & & & 1 & 2 & & 2 & 4 & & & & & & & \\
4 & 2 & & 2 & 1 & & & & 8 & 4 & & 4 & 2 & & & & & 4 & 2 & & 2 & 1 & & & \\
2 & 4 & 2 & 1 & 2 & 1 & & & 4 & 8 & 4 & 2 & 4 & 2 & & & & 2 & 4 & 2 & 1 & 2 & 1 & & \\
 & 2 & 4 & & 1 & 2 & & & & 4 & 8 & & 2 & 4 & & & & & 2 & 4 & & 1 & 2 & & \\
2 & 1 & & 4 & 2 & & 2 & 1 & & 4 & 2 & & 8 & 4 & & 4 & 2 & & 2 & 1 & & 4 & 2 & & 2 & 1 & \\
1 & 2 & 1 & 2 & 4 & 2 & 1 & 2 & 1 & 2 & 4 & 2 & 4 & 8 & 4 & 2 & 4 & 2 & 1 & 2 & 1 & 2 & 4 & 2 & 1 & 2 & 1 \\
 & 1 & 2 & & 2 & 4 & & 1 & 2 & & 2 & 4 & & 4 & 8 & & 2 & 4 & & 1 & 2 & & 2 & 4 & & 1 & 2 \\
 & & & 2 & 1 & & 4 & 2 & & & & & 4 & 2 & & 8 & 4 & & & & & 2 & 1 & & 4 & 2 & \\
 & & & 1 & 2 & 1 & 2 & 4 & 2 & & & & 2 & 4 & 2 & 4 & 8 & 4 & & & & 1 & 2 & 1 & 2 & 4 & 2 \\
 & & & & 1 & 2 & & 2 & 4 & & & & & 2 & 4 & & 4 & 8 & & & & & 1 & 2 & & 2 & 4 \\
 & & & & & & & & & 4 & 2 & & 2 & 1 & & & & & 8 & 4 & & 4 & 2 & & & & \\
 & & & & & & & & & 2 & 4 & 2 & 1 & 2 & 1 & & & & 4 & 8 & 4 & 2 & 4 & 2 & & & \\
 & & & & & & & & & & 2 & 4 & & 1 & 2 & & & & & 4 & 8 & & 2 & 4 & & & \\
 & & & & & & & & & 2 & 1 & & 4 & 2 & & 2 & 1 & & 4 & 2 & & 8 & 4 & & 4 & 2 & \\
 & & & & & & & & & 1 & 2 & 1 & 2 & 4 & 2 & 1 & 2 & 1 & 2 & 4 & 2 & 4 & 8 & 4 & 2 & 4 & 2 \\
 & & & & & & & & & & 1 & 2 & & 2 & 4 & & 1 & 2 & & 2 & 4 & & 4 & 8 & & 2 & 4 \\
 & & & & & & & & & & & & 2 & 1 & & 4 & 2 & & & & & 4 & 2 & & 8 & 4 & \\
 & & & & & & & & & & & & 1 & 2 & 1 & 2 & 4 & 2 & & & & 2 & 4 & 2 & 4 & 8 & 4 \\
 & & & & & & & & & & & & & 1 & 2 & & 2 & 4 & & & & & 2 & 4 & & 4 & 8
\end{pmatrix},$$

$$H_2 = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}, \quad \text{and} \quad P_2 = \begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 \\ 2 & 0 & 2 & 0 & 2 & 0 & 2 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 2 & 0 & 2 & 0 & 2 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}.$$

Keeping in mind that $P_2 = T_2 H_2$, we see the effect of the cutting matrix — it 'cuts' block-rows and columns from T_2 to match its size with the fine–coarse mesh dimensions.

4 Efficiency aspects of matrix-vector multiplication with banded GLT matrices

As noted earlier in the report, executions analysis has shown that sparse matrix-vector multiplication becomes a performance bottleneck both in serial and parallel AMG implementations (see [3]). Below, we discuss possibilities to enhance the performance for GLT matrices, using their structure and the availability of the symbol.

4.1 Sparse matrices

As the name suggests, these are matrices in which most of the elements are zero. To quantify 'sparse', the usual assumption is that the number of nonzero elements are $\mathcal{O}(n)$,

where n is the size of the matrix. For general sparse matrices, the most commonly used storage formats are compressed sparse row format (CSR) and compressed sparse column format (CSC).

The CSR format (used in Trilinos) stores the a sparse $n \times n$ matrix A in row form using three (one-dimensional) arrays VA, IA and JA. Let NNZ denote the number of nonzero entries in A. (Note that zero-base indexes shall be used here.)

- The array VA is the length NNZ and holds all the nonzero entries of A in left-to-right top-to-bottom("row-major") order;
- The array IA is of length $n + 1$. It is defined by this recursive definition:
  - IA[0] = 0;
  - IA[i] = IA[i-1] + (number of nonzero elements on the $(i - 1)$-th row in the original matrix);
  - Thus, the first n elements of IA store the index into VA of the first nonzero element in each row of A, and the last element of IA stores NNZ, the number of elements in VA, which are the index in VA of first element of a phantom row just beyond the end of the matrix A. The values of the i-th row of the original matrix is read from the elements VA[IA[i]] to A[IA[i+1]-1] (inclusive on both ends), i.e. from the start of one row to the last index just before the start of the next;
- The third array, JA, contains the column index in A of each element of VA and hence is of length NNZ as well.

A simple example:

$$\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix} \tag{18}$$

is a 4×6 matrix with 8 nonzero elements, so
VA = [10 20 30 40 50 60 70 80],
IA = [0 2 4 7 8], and
JA = [0 1 1 3 2 2 3 4 5].

4.2 Banded matrices

Banded matrices are sparse matrices, where the nonzero elements are located on a few diagonals, forming a band (see Figure 2).

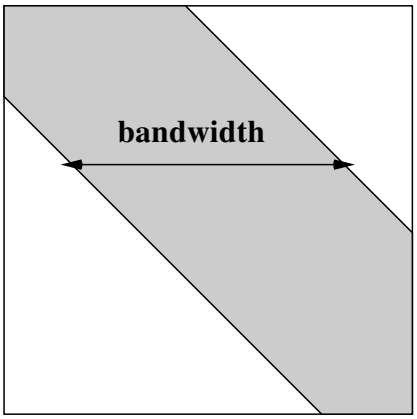


Figure 2: Banded matrix, where the nonzero elements form a band.

The GLT matrices we are dealing with in this project are baned and the so called compressed sparse diagonal storage (CSD) format turns out to be a particularly suitable format. Consider the following sparse matrix:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & \cdots & \cdots & a_{1,n-1} & 0 \\ a_{21} & a_{22} & a_{23} & 0 & & & & a_{2n} \\ 0 & a_{32} & a_{33} & a_{34} & \ddots & & & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & & & \ddots & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & 0 \\ \vdots & & & & 0 & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & a_{n,n-1} & a_{nn} \end{pmatrix} \tag{19}$$

In the CSD format, the position of an element in a matrix is not described using its row and column indexes but using diagonal indexing. Each element in a matrix is positioned on a specific diagonal with a specific diagonal index, commonly referred to as an offset. The main diagonal starts with the top left element, and ends with the bottom right. This is the zero offset diagonal. All elements on the same row to the right are positioned on increasing positive offset diagonals. While all elements in the same column, below the top left element are on negative offset diagonals.

In CSD, the matrix in (19) can be represented in the diagonal storage format in the

following way:

$$\text{Values} = \begin{pmatrix} 0 & a_{11} & a_{12} & a_{1,n-1} \\ a_{21} & a_{22} & a_{23} & a_{2n} \\ a_{32} & a_{33} & a_{34} & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & 0 \\ a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} & 0 \\ a_{n,n-1} & a_{n,n-1} & 0 & 0 \end{pmatrix} \tag{20}$$

The offset vector keeps track off the distance from the main diagonal.

$$\text{offset} = \begin{pmatrix} -1 & 0 & 1 & n-1 \end{pmatrix} \tag{21}$$

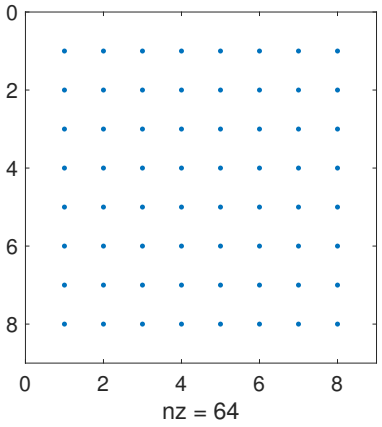
Figure 3 and 4 shows what the pattern of the sparse matrix looks like when its size grows. The left pictures are in the original format, while the right ones are CSD.

4.3 Matrix-vector multiplication algorithm for a diagonal format

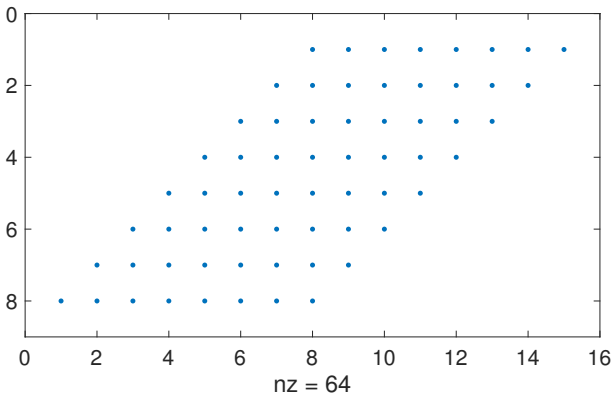
A matrix–vector multiplication algorithm for a matrix stored in CSD is expressed in pseudo-code in Algorithm 3.

Algorithm 3 Diagonal MatVec$(n, k, \mathbf{values}, \mathbf{offset}, \mathbf{x}, \mathbf{y})$

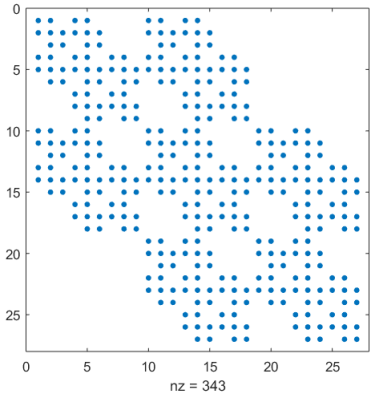
for $i = 1, 2, \ldots, k-1, k$ **do**
  $d = \mathbf{offset}[i]$
  $r0 = max(1, 1-d)$
  $r1 = min(n, n-d)$
  $c0 = max(1, 1+d)$
  for $r = r0, r0+1, \ldots, r1-1, r1$ **do**
    $c = r - r0 + c0$
    $\mathbf{y}[r] = \mathbf{y}[r] + \mathbf{values}[r, i] \cdot \mathbf{x}[c]$



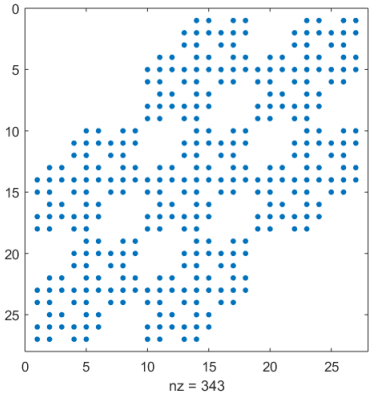
(a) Size 8×8, original form



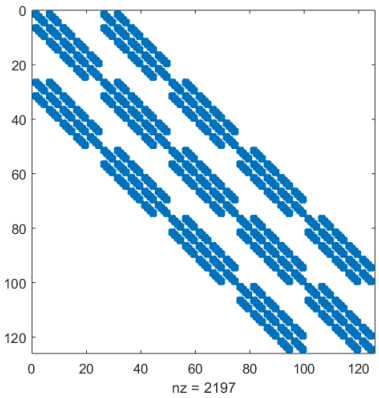
(b) Size 8×8, diagonal storage



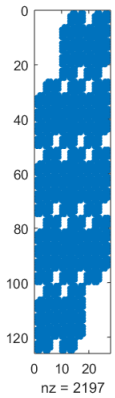
(c) Size 27×27, original form



(d) Size 27×27, diagonal storage

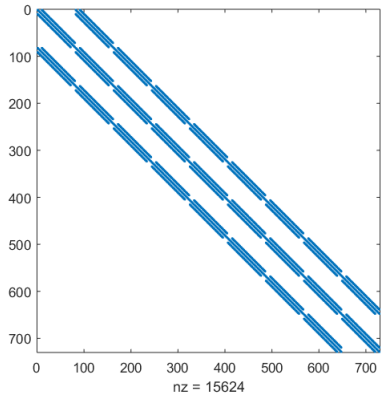


(e) Size 125×125, original form

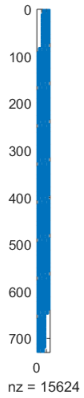


(f) Size 125×125, diagonal storage

Figure 3: Two sparse matrix storage formats



(a) Size 729×729, original form



(b) Size 729×729, diagonal storage

Figure 4: Two sparse matrix storage formats, cont.

5 Numerical tests

5.1 GLT-MG preconditioner performance

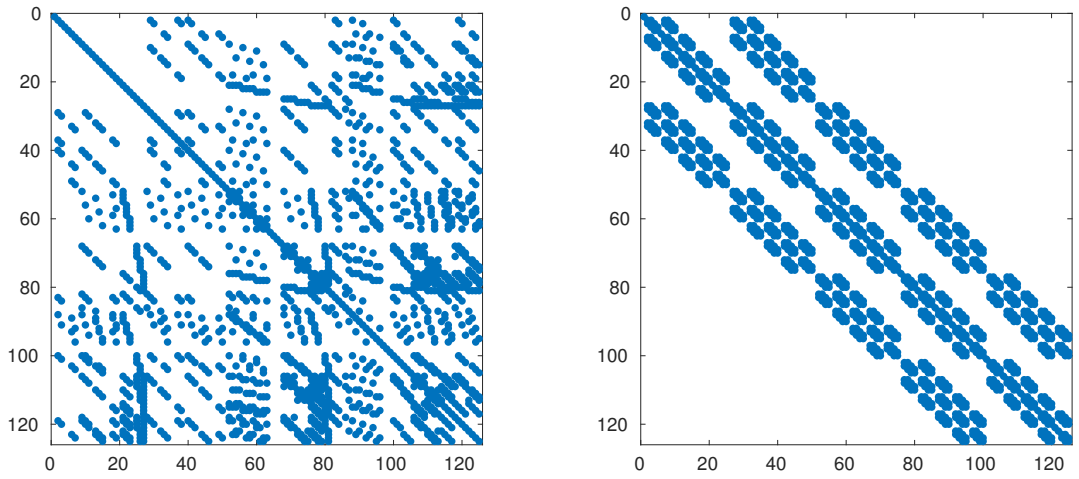
The GLT-MG performance is tested using the benchmark problem

$$\begin{aligned} -\Delta u &= f, & x \in \Omega = [0,1]^3 \subset \mathbb{R}^3 \\ u &= 0, & \text{on } \partial\Omega, \end{aligned}$$

a 3D, isotropic Laplace equation with Dirichlet boundary conditions on a unit cube domain. The problem is uniformly discretized using FEM, and the right hand side is constructed so that the solution to the discretized problem is equal to one. The discretization is done with three different number of refinements, 6,7, and 8, to produce system matrices $(A_{n\times n})$ with $n = 274625$, $n = 2146689$, and $n = 16974593$ respectively. We use $\varepsilon_{1,2,3} = 1$ in Equation 16 to get the symbol for the isotropic Laplacian.

The solution method used is a Multigrid-preconditioned CG, with a stopping criteria of 10^{-6}. The code for the numerical tests is written in C++ with the help of the C++ libraries deal.II (version 8.5.1) and Trilinos (see [1] and [2]). The tests are run in serial on a computer with an AMD A10-5745M processor and 16 GB RAM, with Ubuntu 16.04 LTS as operating system.

The version of GLT-MG implemented in this work requires that, after discretization, the system matrix is permuted according to a lexicographical ordering of the mesh points. The sparsity patterns of the original finite element matrix and the reordered one are visualized in Figure 5. The matrix assembly and reordering are done outside deal.II, the reordered matrix is exported with an element value accuracy of floats with 14 digits after the decimal



(a) after standard FEM assembly

(b) sorted in lexicographical order

Figure 5: 3D Laplace: Nonzero pattern of the system matrix

point, and is then read in deal.II from an input file. It is possible to generate the systems without a need for reordering, but that is outside the scope for this project.
The performance of GLT-MG is compared to that of the AMG preconditioner provided via the Trilinos scientific library. The parameters used for the preconditioners can be found in Appendix A. The choices of parameters are not optimal, but they are better than the default parameters for which AMG does not converge for the largest system.

Figure 6 shows a plot of the solution time in logarithmic scale as a function of the system size, for AMG in red and GLT-MG in blue. Table 1 presents the number of CG iterations, solution times, and solution time-to-DoF ratios for different number of refinements of the original system. The numerical results confirm the theory, as GLT-MG gives a faster solution time for all system sizes (despite the GLT-MG code not being optimized). The GLT-MG preconditioner ensures convergence within a fixed number of iterations, and its solution time scales linearly with the system size (as can be seen in Table 1). It is worth noting that the AMG results deviate from the expected behavior for the largest system, where the ratio between solution time and system size should be nearly constant but is not. This could be caused by the parameters not being well suited for the underlying problem. A detailed study of the influence of the AMG parameters on the performance has not been done due to time constraints. We argue that this need for optimization of the AMG parameters is another point in GLT-MGs favor. GLT-MG works right away, which is a compelling property when the alternative is a time consuming optimization process with uncertain reward.

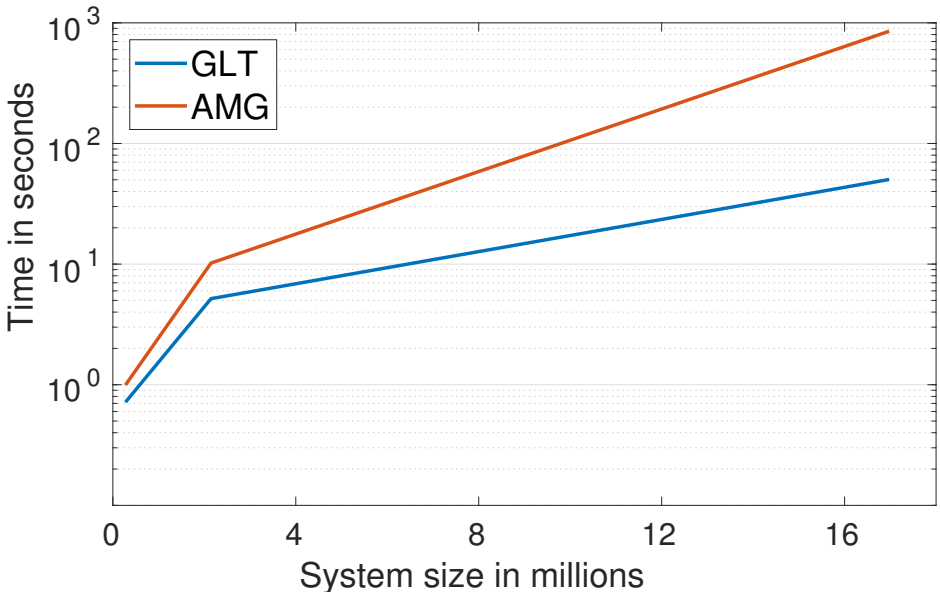


Figure 6: GLT-MG and AMG preconditioned CG: solution time as a function of the number of degrees of freedom

	CG Iterations		Solving time t_s, s		t_s/DoF, μs	
DoF (Refs)	AMG	GLT-MG	AMG	GLT-MG	AMG	GLT-MG
274625 (6)	5	3	1.00	0.72	3.62	2.61
2146689 (7)	6	3	10.2	5.17	4.75	2.41
16974593 (8)	5	3	852	50.3	50.19	2.96

Table 1: The number of CG iterations, solving time (t_s) in seconds and solving time per DoF (t_s/DoF) in microseconds for different preconditioners and system sizes.

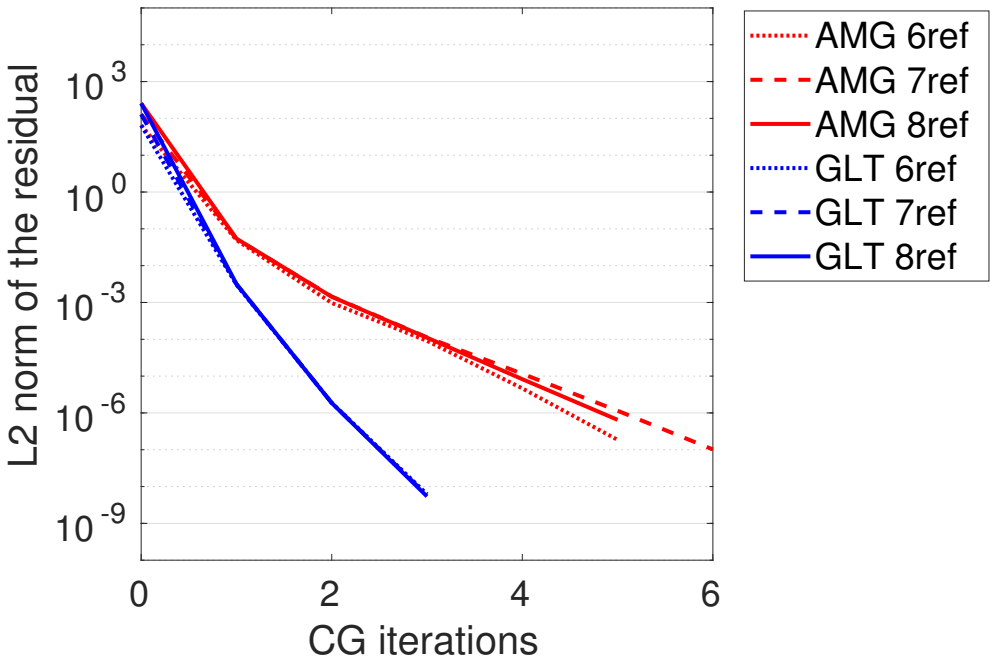


Figure 7: GLT-MG and AMG preconditioned CG: convergence history for various problem sizes (logarithmic scale)

The GLT-MG and AMG preconditioned CG methods show convergence within a number of iterations independent of the system size. Both preconditioners can be said to behave according to theory in this regard. Figure 7 visualizes the convergence history, presented in Table 2 and 3.

Refinements:	6	7	8
It 0	65.0076	129.004	257.002
It 1	0.00303404	0.00313332	0.00318410
It 2	$1.88186 \cdot 10^{-6}$	$1.86722 \cdot 10^{-6}$	$1.86027 \cdot 10^{-6}$
It 3	$6.19041 \cdot 10^{-9}$	$5.79044 \cdot 10^{-9}$	$5.38961 \cdot 10^{-9}$

Table 2: GLT-MG: norm of the residual in different CG iterations

The current GLT-MG implementation does not fully utilize the tools available in deal.II. The performance (especially the GLT-MG initialization) might improve if some of the "home made" functions, e.g. outer vector product, were replaced by preexisting tensor operations instead. Still, depending on how they are implemented in deal.II, the difference might not be that significant (most of the "home made" functions consist of straight forward implementations of standard algebraic operations). With more time, the code could be rerun on a different computer with a different processor. This would be done

Refinements:	6	7	8
It 0	65.0076	129.004	257.002
It 1	0.0496721	0.0529047	0.0542295
It 2	0.000971637	0.00144246	0.00143057
It 3	$9.07094 \cdot 10^{-5}$	0.000112855	0.000110505
It 4	$4.64606 \cdot 10^{-6}$	$1.15783 \cdot 10^{-5}$	$8.24920 \cdot 10^{-6}$
It 5	$1.85774 \cdot 10^{-7}$	$1.14680 \cdot 10^{-6}$	$6.48721 \cdot 10^{-7}$
It 6		$1.01828 \cdot 10^{-7}$	

Table 3: AMG: the norm of the residual in different CG iterations for AMG.

Memory format	Unpreconditioned CG									
	729		4913		35937		274625		2146689	
	Iter.	Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.	Time
CSR	29	0.019	46	0.066	76	0.653	131	8.367	227	112.079
CSD	29	0.019	46	0.053	76	0.384	131	8.083	227	115.404

Table 4: Comparison of the execution time (in s) for the unpreconditioned CG with CSR and CSD matrix formats

to see whether the preconditioners produce different results in relation to each other on a different computer architecture.

5.2 Sparse matrix diagonal storage tests

These results are produced using Matlab on a laptop Lenovo Thinkpad T450s (Ultrabook) with Core i7 5600U / 2.6 GHz CPU and 8 GB RAM. Both compressed sparse row-wise (CSR) and compressed sparse diagonal (CSD) formats are implemented in explicit Matlab code for a more fair comparison. The test is performed using unpreconditioned CG, with matrices generated from the same problem as in Section 5.1, a 3D isotropic Laplace equation with Dirichlet boundary conditions on a unit cube domain. The results are presented in Table 4.
As we see from Table 4 the Matlab implementation does not allow for testing the full potential of the diagonal storage format. For completeness, we include the corresponding implementations of the matrix-vector operation in Appendix B.

6 Conclusions and Future work

The performance of the GLT-MG preconditioner, implemented in C++, confirms the theory and shows a faster convergence than that of the reference AMG method for all tested system sizes (60k, 3M and 17M DoF). However, the diagonal sparse storage format implementation in Matlab didn't show a significant decrease in operation time for matrix-vector

multiplication, when compared to the compressed sparse row storage implementation using matrices from the benchmark problem.

Looking forward, there are several steps that can be taken to further improve the performance of GLT-MG. The first step could be to continue the work on finding a more multiplication-efficient storage and incorporate in the GLT-MG C++ code. An implementation of the current CSD format could perform better in C++, and even if it doesn't, there's the possibility of implementing a format even more customized to fit Toeplitz and Block Toeplitz matrices. (The diagonals can be handled more efficiently if they're known to be constant diagonals or block constant diagonals. The current CSD implementation is not taking that into account.) The second step would be to implement a matrix-free version of the code so that it relies entirely on the symbol to recalculate elements instead of storing them. A matrix-free code should be very well suited for modern computer architectures, where a high ratio between computation and memory references is known to be beneficial.

7 Acknowledgements

We would like to thank our supervisors Maya Neytcheva and Ali Dorostkar. Thank you Maya Neytcheva, for your patience with us and your commitment to this project, going above and beyond to make this project what it is, and always keeping an open door. Ali Dorostkar, thank you for providing comfort and invaluable know-how in all matters regarding deal.II.

References

[1] Deal.II. `http://www.dealii.org/about.html`. Accessed: 2018-01-14.

[2] Trilinos. `https://www.trilinos.org`. Accessed: 2018-01-14.

[3] J. R. Bull, A. Dorostkar, S. Holmgren, A. Kruchinina, M. Neytcheva, D. Nikitenko, N. Popova, P. Shvets, A. Teplov, V. Voevodin, and Vl. Voevodin. Multidimensional performance and scalability analysis for diverse applications based on system monitoring data. In *Parallel Processing and Applied Mathematics :*, Lecture Notes in Computer Science. Springer, 2018. to appear.

[4] J. Demmel. *Applied numerical linear algebra.* SIAM, 1997.

[5] M. Donatelli, A. Dorostkar, M. Mazza, M. Neytcheva, and S. Serra-Capizzano. Function-based block multigrid strategy for a two-dimensional linear elasticity-type problem. volume 74, pages 1015 – 1028. 2017. SI: SDS2016 – Methods for PDEs.

[6] M. Donatelli, C. Garoni, C. Manni, S. Serra-Capizzano, and H. Speleers. Symbol-based multigrid methods for Galerkin B-spline isogeometric analysis. *SIAM Journal on Numerical Analysis*, 55(1):31–62, 2017.

[7] A. Dorostkar. Function-based Algebraic Multigrid method for the 3D Poisson problem on structured meshes. Technical Report 2017-022, Department of Information Technology, Uppsala University, October 2017. Updated 2017-10-26.

[8] A. Dorostkar, M. Neytcheva, and S. Serra-Capizzano. Spectral analysis of coupled PDEs and of their Schur complements via Generalized Locally Toeplitz sequences in 2D. *Computer Methods in Applied Mechanics and Engineering*, 309:74 – 105, 2016.

[9] G Fiorentino and S Serra-Capizzano. Multigrid methods for Toeplitz matrices. *Calcolo*, 28(3-4):283–305, 1991.

[10] C. Garoni and S. Serra-Capizzano. Generalized Locally Toeplitz Sequences. In *Generalized Locally Toeplitz Sequences: Theory and Applications*, pages 4–5. Springer, 2017.

[11] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2nd edition, 2003.

Appendix A: Parameter choice for AMG and GLT-MG

AMG is run with the parameters
elliptic = true
higher order elements = true
smoother sweeps = 2
aggregation threshold = 0.02
smoother type = "Jacobi".

GLT-MG is run with three V-cycles per iteration, one pre-smoothing step with Jacobi, and one post-smoothing step with weighted Jacobi using the weight 2/3.

Appendix B: Matlab codes for matrix-vector multiplications using the CSR and CSD formats

Algorithm CSR

```
function [y] = matvec_csr(x,AI,AJ,AV,n1,n2)
% Multiplication with a matrix in
% sparse compressed row-wise storage
% The matrix is assumed of size(n1xn2)
```

```
y = zeros(n2,1);
for k=1:n1
     y(k) = 0;
     kf = AI(k);
     kl = AI(k+1)-1;
     if (kl >= kf),
         for l=kf:kl
             y(k) = y(k) + AV(l) * x(AJ(l));
         end
     end
end
```

Algorithm CSD

```
function [y] = diag_matvec(x,A,vd)
% A(size(n,vd) matrix in a compressed diagonal storage
% vd = [-q,-q+1, ... -1,0,1,2,...p] vector describing the positions of the
% diagonals
% x the vector to multiply with

nd = length(vd); % number of diagonals
n  = length(x); % the size of the matrix
y  = zeros(n,1);
for k=1:nd
    cd = vd(k);  % the current diagonal
    z  = zeros(n,1);
    if cd<=0,
            z(abs(cd)+1:n,1)=x(1:n-abs(cd));
    else
            z(1:n-cd) = x(cd+1:n,1);
    end
    y = y+A(:,k).*z;
end
```