Professur für Thermofluiddynamik

BACHELOR

Application of iterative solvers in themoacoustics

Autor:
Gargouri, Fares

Matrikel-No:
03665986

Betreuer:
Prof. Wolfgang Polifke, Ph. D.
Alexander Avdonin, M. Sc.

November 1, 2017

Professur für Thermofluiddynamik
Prof. Wolfgang Polifke, Ph. D.

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst zu haben. Ich habe keine anderen Quellen und Hilfsmittel als die angegebenen verwendet.

Ort, Datum	Gargouri, Fares

Acknowledgement

First, I would like to thank my country, Tunisia, as embodied by its people and institutions. It is only because it continues to maintain its scholarship program–despite economic challenges and a difficult political transition–that this work was ever made possible. I only pray I may be able to repay your generosity in the near future.

I would also like to thank my family and friends for everything. It is for you that I persist.

In the end, I would like to express my deep gratitude to my supervisor, whose patient and helpful feedback consistently improved the quality of this work, and helped me learn valuable lessons in academic writing along the way.

Abstract

The present work searches for suitable iterative solvers for problems in thermoacoustics. In particular, Helmholtz equations and discontinuous-Galarkin-discretized linearized Navier-Stokes equations (DG-LNSE) were treated. The literature related to iterative solvers is summarized, and suitable solvers were identified. Multiple solvers were found to work for the Helmholtz equation and LNSE with no heat source term. On the other hand, none of the tested solvers (GMRES, FGMRES, BiCGStab, CG) worked for the more complex LNSE problems involving a heat source term, regardless of used preconditioner. For Helmholtz eigenvalue problems, numerical investigations revealed that conjugate gradient (CG)–with the Multigrid method as a preconditioner–scales particularly slowly in terms of memory consumption with increasing problem size (number of DoF), leading to significant memory savings when compared to direct solvers. In particular, for 3D problems, CG with Multigrid is shown to solve problems one order of magnitude larger than MUMPS (a parallel direct solver) can solve with for same memory requirement. As for 2D problems, CG With Multigrid solves problems twice as large as MUMPS solves for the same memory requirements. Additionally, CG with Multigrid is as fast or faster than direct solvers for these equations. An alternative solver for Helmholtz eigenvalues was also studied, namely CG with SOR. This latter configuration was found to be effective, too, though less than CG with Multigrid. For 3D problems, CG with SOR was shown to solve problems that are five times larger than those that MUMPS can solve for the same memory requirements. For 2D problems, it solves problems that are twice as large as those that MUMPS can solve. However, for Helmholtz eigenvalue problems, CG with SOR is slower than MUMPS by one order of magnitude for 3D problems, and two orders of magnitude for 2D problems.

For DG-LNSE frequency response problems with no heat source terms, GMRES with the SOR method as preconditioner was found to be suitable. It is shown to solve problems one order of magnitude higher than MUMPS can solve for the same memory usage, while being about 3 times slower slower. For the studied DG-LNSE problems with heat source terms, no iterative solvers converged. However, this is suspected to be a result of poor choice of boundary conditions in the used test cases. Nevertheless, GMRES with SOR was the nearest solver to convergence. This area requires further investigation.

Contents

List of Tables

List of Figures

Nomenclature

Roman Symbols

$\mathbb{C}$	Set of all Complex numbers
$\mathbb{C}^{\hat{n}_1\times\hat{n}_1}$	Set of all matrices with $\hat{n}_1 \times \hat{n}_1$ Complex entries
$\mathbb{C}^{\hat{n}}$	Set of all vectors with $\hat{n}$ complex entries
$\mathcal{K}_m(\mathbf{A},\mathbf{b})$	Dimension m Krylov subspace of the problem $\mathbf{Ax}=\mathbf{b}$
$\mathbf{e}_m$	Unit vector of dimension m
$\mathbf{H}_{m+1,m}$	Non-square upper-Hessenberg matrix
$\mathbf{H}_{m,m}$	Square upper-Hessenberg matrix
$\mathbf{P}$	Preconditioner
n	Number of degrees of freedom, also referred to a problem size

Greek Symbols

λ	Eigenvalue of a matrix
ω	Angular Frequency
σ	Shift

Acronyms

BiCG	BiConjugate Gradient
BICGStab	BiConjugate Gradient Stabilized
CG	Conjugate Gradient method
DG	Discontinuous Galarkin
FGMRES	Flexible Generalized Minimal RESidual method

FRF	Frequency Response Function
FTF	Flame Transfer Function
GMRES	Generalized Minimal RESidual method
ILU	Incomplete LU factorization
MUMPS	MUltifrontal Massively Parallel Solver
SOR	Successive Over-Relaxation
SORU	Symmetric Over-Relaxation, using the upper triangular part
SSOR	Symmetric Successive Over-Relaxation

Mathematical notation

In this work, a certain style of mathematical notation was chosen to avoid ambiguity. This serves mainly to distinguish matrices from vectors and scalars, iteration step from exponent, matrix dimension from matrix entry, and so on. Table 1 shows this style of notation.

Example	Notation	Meaning
$\mathbf{A}$	Bold upper-case Latin character.	Matrix
$\mathbf{A}_{m_1,m_2}$	Matrix with comma separated subscript	Matrix with explicitly-stated dimensions.
$\mathbf{x}$	Bold lower-case Latin character	Vector
a (also: λ)	Lower-case Latin or Greek character	Scalar
$a_{i,j}$	Scalar with two comma-separated indexes as subscripts	Entry in the i-th row and j-th column of matrix $\mathbf{A}$
λ^{α}	Variable with a scalar as superscript	Power
$\mathbf{x}^{(k)}$	A variable with scalar in parentheses as superscript	Iteration step k of an iterative method
$\mathbf{x}_{\{l\}}$	A variable with scalar in braces as subscript (or occasionally superscript)	l-th multigrid level

Table 1: Table detailing the mathematical notation used throughout this work

Also consult at Appendix A for the concepts and operations used in the work.

1 Introduction

The field of thermoacoustics is of ever-increasing importance. Both in academia and industry, reliance on numerical methods is common in this field. These case a given problem into a system of linear equation system of the form $\mathbf{Ax} = \mathbf{b}$.

However, due to the proliferating application of these methods on larger and more complex models, as well as the constant drive for more robust schemes and better approximations, the task of solving these equation systems becomes increasingly challenging. The processing time increases dramatically, but even more pressing is the issue of memory requirement for solving these systems. This is particularly true when the available computational resources are limited, such as when access to high-performance supercomputers is restricted.

The straightforward way for solving linear equation systems is the use of direct solvers. These tend to be robust and applicable to a wide variety of problem classes, though at the cost of higher memory consumption. This disadvantage is particularly evident for very large systems. The focus of this work lay thus on iterative solvers instead, which allow for lower memory constraints.

Iterative methods have been the subject of fairly intensive research, especially since the early 50'. Constant theoretical advancements have produced several methods that vary in practical utility. A reoccurring theme in the literature is the need for empirical tests to establish whether a scheme is suitable for a given class of problems. This is due to the fact that these methods typically come with special constraints. An iterative solver suitable for an application is often not suitable for another. Also, the robustness of an iterative scheme can be strongly influenced by the choice of preconditioner. Therefore, an investigation to determine appropriate iterative solver-preconditioner combinations for problems in thermoacoustics is performed in this work.

This work specifically handles eigenvalue problems and problems relating to frequency response function (FRF) within the context of thermoacoustics. First, an introduction to numerical eigenvalue and frequency response algorithms is provided. Then an overview of the theory behind the iterative solvers and preconditioners that were studied is presented. Thereafter, through numerical investigation of several preconditioner-solver combinations on multiple models, this work investigates the convergence behavior of these combinations. The aim here is to find some solver that is generally applicable for all test cases. For the combinations that proved to work, the effect of increasing problem size on memory consumption are studied. This way, this work aims to push the limit of the problem size that can be solved on an ordinary work station. The scope of this study is limited to thermoacoustics. Therefore, the results are only applicable to problems from this field and problems which deliver similar equation systems. For all simulations done in this work, COMSOL Multiphysics 4.4 was used.

2 Theory

2.1 Krylov subspaces and the Arnoldi Iteration

Throughout this work, the idea of a Krylov subspace plays an important role in multiple places. The need for the concept of Krylov subspace arizes from the desire to reduce systems of linear equations that are too large to a manageable size. This is relevant for the eigenvalue algorithm, since most often, only a very small proportion of eigenvalues is of interest. This also comes very clearly into play for the GMRES iterative solver, where the algorithm essentially projects the system on a smaller Krylov subspace, solves the system, then projects the problem back onto the original dimension (This is discussed more thoroughly in Section2.6.5).

When discussing Krylov subspaces, there is often the need to find a transformation between the original system and the Kyrlov subspace projection of the system. This is most commonly achieved through the Arnoldi iteration, and is a prominent feature of the GMRES iterative solver.

2.1.1 Krylov subspaces

Consider a system of linear equations

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{b}$ and $\mathbf{x}$ are $n \times 1$ vectors. The order-m Krylov subspace of this problem is defined as the space spanned by the product of the first m powers of $\mathbf{A}$ multiplied with the initial vector $\mathbf{r}^{(0)}$ [1]. This can be expressed as

$$\mathscr{K}_r(\mathbf{A}, \mathbf{r}^{(0)}) = \text{span}\{\mathbf{r}^{(0)}, \mathbf{Ar}^{(0)}, \mathbf{A}^2\mathbf{r}^{(0)}, \ldots \mathbf{A}^{r-1}\mathbf{r}^{(0)}\} \quad .$$

where

$$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Ax}^{(0)}$$

and $\mathbf{x}^{(0)}$ is the initial estimate.

2.1.2 General introduction to the Arnoldi iteration

The Arnoldi iteration is used to calculate a projection of a matrix $\mathbf{A}$ into a Krylov subspace. This means calculating the following transformation:

$$(\mathbf{V}_{n,m})^* \mathbf{AV}_{n,m} = \mathbf{H}_{m,m} \tag{2.1}$$

This includes calculating the orthogonal basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ for the Krylov subspace, and the square upper Hessenberg matrix $\mathbf{H}_{m,m}$ [see [2],[3]]. The orthogonal basis vectors are not calculated in one go. Instead, They are calculated stepwise, while simultaneously calculating components of the matrix $\mathbf{H}_{m+1,m}$. The combined task of orthogonalization and projection can be done by relying on a standard Gramm-Schmitt process, but is usually performed using a modified Gramm-Schmitt process [4]. This is discussed in detail is Section 2.1.3. The relation between an iteration step k and the next can $k+1$ be described as follows:

$$\mathbf{A}\mathbf{V}_{n,k} = \mathbf{V}_{n,k+1}\mathbf{H}_{k+1,k} \tag{2.2}$$

2.1.3 Practical implementation of the Arnoldi iteration

The simultaneous basis orthogonalization and orthogonal projection performed by the Arnoldi algorithm usually relies of the Gramm-Schmidt procedure. This implementation delivers the following algorithm:

By standard Gram-Schmidt method

1. First, take a start vector $\mathbf{r}^{(0)}$ and normalize it by applying $\mathbf{v}_1 = \frac{1}{\|\mathbf{b}\|}$.
2. Initialize the matrices **H** and **V** such that all of their elements are 0,
3. For j from 1 to m:
4. (a) for i from 1 to j: $h_{i,j} = \mathbf{v}_i^T \mathbf{A}\mathbf{v}_j$
   (b) $\tilde{\mathbf{v}}_{j+1} = \mathbf{A}\mathbf{v}_j - \sum_{i=1}^{j} h_{i,j}\mathbf{v}_j$,
   (c) $h_{(j+1),j} = \left\|\tilde{\mathbf{v}}_{j+1}\right\|$,
   (d) $\mathbf{v}_{j+1} = \frac{1}{h_{(j+1),j}}\tilde{\mathbf{v}}_{(j+1),j}$

.

The classical Gram-Schmitt method is easy to understand and implement. However, with increasing dimension of $\mathcal{K}_m(\mathbf{A}, \mathbf{r}^{(0)}$ (i.e. with increasing m), it becomes increasingly unstable, since the vectors $\mathbf{A}^j\mathbf{v}^j$ become similar [1] [2]. The modified (or stabilized) Gramm-Schmitt procedure, as presented in the following, avoids this problem [1] .

By modified Gram-Schmidt method

1. take the vector $\mathbf{r}^{(0)}$ and normalize it by applying $\mathbf{v}_1 = \frac{1}{\|\mathbf{b}\|}$;
2. for j from 1 to m:
   (a) $\tilde{\mathbf{v}}_j = \mathbf{A}\mathbf{v}_j$;

(b) for i from 1 to j:

   i. $h_{i,j} = \mathbf{v}_i^T \tilde{\mathbf{v}}_j$;

   ii. $\tilde{\mathbf{v}}_j = \tilde{\mathbf{v}}_j - h_{i,j}\mathbf{v}_i$;

(c) $h_{(j+1),j} = \tilde{\mathbf{v}}_j$;

(d) $\mathbf{v}_{j+1} = \frac{\tilde{\mathbf{v}}_j}{h_{(i+1),i}}$;

2.2 Eigenvalue algorithm

The eigenvalue problems treated in this work arise discontinuous Galarkin methods. An explanation for discontinuous Galarkin method is beyond the scope of this study, but an adequate presentation of the subject is provided by Blom [5]. These discretizations deliver the following generalized eigenvalue problem [6]:

$$(\lambda - \lambda_0)^2 \mathbf{Eu} - (\lambda - \lambda_0)\mathbf{Du} + \mathbf{Ku} = 0 \tag{2.3}$$

The terms in Eq. (2.3) are to be understood as follows:

- **E**: Mass matrix,
- **D**: Damping matrix,
- **K**: Stiffness matrix,
- **u**: Solution vector,
- λ_0: Linearization point,
- λ: Eigenvalue.

Here, the matrices **D**, **K**, **N** and $\mathbf{N}_F$ are evaluated; $\mathbf{u}_0$ is the solution vector, and λ_0 is the linearization point. Eq (2.3) will serve as a starting point for the following discussion. As for details about the derivation of this equation, see [6, section 19.4]. Eq. (2.3) can be solved first for $\tilde{\lambda} = \lambda - \lambda_0$, i.e

$$\tilde{\lambda}^2 \mathbf{Eu} - \tilde{\lambda}\mathbf{Du} + \mathbf{Ku} = 0 \,, \tag{2.4}$$

and then solving $\lambda = \tilde{\lambda} - \lambda_0$ afterwards. The tilde in Eq. (2.4) is omitted henceforth.

To transform Eq. (2.4) into a linear eigenvalue problem, the transformation

$$\lambda \mathbf{u} = \tilde{\mathbf{u}} \tag{2.5}$$

is applied. This yields

$$\lambda \mathbf{E}\tilde{\mathbf{u}} - \mathbf{D}\tilde{\mathbf{u}} + \mathbf{Ku} = 0 \,, \tag{2.6}$$

$$\lambda \mathbf{u} = \tilde{\mathbf{u}} \,. \tag{2.7}$$

Rewriting Eq.2.6 and 2.6 as a vector equation delivers

$$\begin{pmatrix} \mathbf{K} & \lambda\mathbf{E}-\mathbf{D} \\ \lambda\mathbf{I} & -I \end{pmatrix} = \begin{pmatrix} \mathbf{u} \\ \tilde{\mathbf{u}} \end{pmatrix} = \mathbf{0}\,. \tag{2.8}$$

This can be written as

$$\left(\underbrace{\lambda\begin{pmatrix} \mathbf{0} & \mathbf{E} \\ \mathbf{I} & \mathbf{0} \end{pmatrix}}_{\mathbf{B}} - \underbrace{\begin{pmatrix} -\mathbf{K} & \mathbf{D} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}}_{\mathbf{A}} \right) \underbrace{\begin{pmatrix} \mathbf{u} \\ \tilde{\mathbf{u}} \end{pmatrix}}_{\mathbf{x}} = \mathbf{0}\,. \tag{2.9}$$

This leads to the linear eigenvalue problem

$$(\lambda\mathbf{B}-\mathbf{A})\mathbf{x} = 0\,, \tag{2.10}$$

which is equivalent to

$$\mathbf{A}\,\mathbf{x} = \lambda\mathbf{B}\,\mathbf{x}\,. \tag{2.11}$$

Note that **B** has the same dimension of **A**.

2.2.1 The implicitly restarted Arnoldi method

For the implicitly restarted Arnoldi method, some parameters need to be defined: A desired number of eigenvalues k must be chosen, and a selection criterion for these eigenmodes has to be set. Once this is done, this method, as implemented in the ARPACK library, starts by initially performing an Arnoldi factorization of size m (see Section 2.1).

$$\mathbf{A}\mathbf{V}_{n,m} = \mathbf{V}_{n,m+1}\mathbf{H}_{m+1,m} \tag{2.12}$$

where **A** is the system matrix obtained in Section 2.2.3.

To obtain a square upper Hessenberg matrix, one needs to remove the last column and the last line of $\mathbf{V}_{n,m+1}$ and $\mathbf{H}_{m+1,m}$, respectively. This yields

$$\mathbf{A}\mathbf{V}_{n,m} = \mathbf{V}_{n,m}\mathbf{H}_{m,m} + h_{m+1,m}\mathbf{v}_{m+1}\mathbf{e}_m^T \tag{2.13}$$

where $\mathbf{e}_m$ is the unit vector [3]. The last term on the right represents the residual of this process. The large eigenvalue problem in Eq.(2.11) can thus be replaced with a reduced one expressed by

$$\mathbf{H}_{m,m}\mathbf{y}_{i,m} = \lambda_i\mathbf{y}_{i,m} \tag{2.14}$$

where λ_i and $\mathbf{y}_{i,m}$ are respectively the i-th eigenvalue and eigenvector.

To solve the reduced eigenvalue problem, the following step are performed until the a sufficient number of Eigenvalues that satisfy the condition in Section 2.2.1 is reached:

1. The eigenvalues as of $\mathbf{H}_{m,m}$ are computed (This is discussed later in this Section).

2. The eigenvalues are organized in two sets according to the selection criterion: One set $\mathscr{S}_{\text{wanted}} = \{\lambda_i | i = 1,2,\dots,k\}$ for the wanted eigenvalues and another $\mathscr{S}_{\text{unwanted}} = \{\lambda_i | i = k+1, k+2,\dots,m\}$ for the unwanted eigenvalues.

3. $m-k=p$ steps of the shifted QR factorization (explained below) are performed with the unwanted eigenvalues $\mathscr{S}_{\text{unwanted}}$ as shifts to obtain

$$\mathbf{H}_{m,m}\mathbf{Q}_{m,m} = \mathbf{Q}_{m,m}\tilde{\mathbf{H}}_{m,m} \ . \tag{2.15}$$

   [see[7], Section 4.4.1].

4. The length m Arnoldi factorization [see[7]] is then multiplied from the right with $\mathbf{Q}_{m,k}$, which consists of the first k columns of $\mathbf{V}_{k,k}$, to obtain an Arnoldi factorization

$$\mathbf{A}\mathbf{V}_{n,m}\mathbf{Q}_{m,k} = \mathbf{V}_{n,m}\mathbf{Q}_{m,k}\tilde{\mathbf{H}}_{k,k} + h_{(m+1),m}\mathbf{v}_{m+1} \tag{2.16}$$

   where $\tilde{\mathbf{H}}_{k,k}$ is obtained from $\tilde{\mathbf{H}}_{m,m}$ by reducing it to exclude elements $h_{i,j}$ for which $i,j > k$. Afterwards, set $\mathbf{V}_{n,k} = \mathbf{V}_{n,m}\mathbf{Q}_{m,k}$.

5. Finally, the length k Arnoldi iteration is extended to a length m factorization.

An algorithm for the shifted QR factorization, as provided by Lehoucq *et al.* [7], is explained in the following. The initial input is the two matrices $\mathbf{H}_{m,m}$, $\mathbf{V}_{n,m}$, and a set of shifts $\{\nu_i | i = 1,2,\dots,l\}$. For these shifts, the unwanted eigenvalues $\mathscr{S}_{\text{unwanted}}$ are selected. The algorithm performs the following steps for i until $i = l$:

1. Perform a (normal) QR factorization on the matrix $[\mathbf{H}_{m,m} - \nu_i\mathbf{I}]$.

2. Compute a new $\mathbf{H}'_{m,m} = \mathbf{Q}^*_{m,m}\mathbf{H}_{m,m}\mathbf{Q}_{m,m}$ and a new $\mathbf{V}'_{n,m} = \mathbf{V}_{n,m}\mathbf{Q}_{m,m}$. These are used in the next iteration as $\mathbf{H}_{m,m}$ and $\mathbf{V}_{n,m}$, respectively.

The QR factorization

For the shifted QR factorization, a QR factorization

$$\mathbf{H} = \mathbf{QRQ}^* \ , \tag{2.17}$$

where $\mathbf{Q}$ is a unitary matrix and $\mathbf{R}$ is an upper triangular matrix, is required. Since $\mathbf{H}$ is an upper-Hessenberg matrix, this can be easily done using a series of 2×2 Givens rotations or 3×3 Householder rotations, instead of explicitly computing $\mathbf{Q}$, in a so-called bulge-chasing procedure. This is implemented in the ARPACK package. For an idea as to how such a process is done, Section 2.6.5 contains a possible implementation.

Calculating the eigenvalues of H

The Egenvalues of $\mathbf{H}_{m,m}$ are found through a (normal) QR decomposition, in the same manner as described above. It is clear from Eq. (2.17) that a QR decomposition is a similarity transformation. Moreover, it is known that eigenvalues are invariable in relation to similarity transformations. Therefore, the eigenvalues of $\mathbf{H}_{m,m}$ are those of $\mathbf{R}$, and since $\mathbf{R}$ is an upper triangular matrix, its eigenvalues are simply its diagonal entries. Note that this method of calculating the eigenvalues of $\mathbf{H}_{m,m}$, is only practical for small problems. For a larger m, other other methods to approximate $\mathbf{R}$ are used instead. These are discussed in detail in Lehoucq *et al.* [7].

After eigenvalues are obtained, the corresponding eigenvectors are calculated by solving Eq. (2.14) for $\mathbf{y}_{i,m}$. These are then transformed backwards by applying

$$\mathbf{x} = \mathbf{V}_{n,m}\mathbf{y} \tag{2.18}$$

to arrive at the eigenvectors in the original configuration.

Interruption condition for the implicitly restarted Arnoldi iteration

To estimate whether the eigenvalues and eigenvectors of the Hessenberg matrix are an accurate representation of those of the original problem statement at Eq.(2.11), the following formula can be used:

$$\left\| (\mathbf{A} - \lambda_i \mathbf{I})\mathbf{v}_{i,m} \right\| = h_{(m+1),m} |e_m^T \mathbf{y}_{i,m}| \tag{2.19}$$

where $\mathbf{v}_{i,m} = \mathbf{Q}_{n,m}\mathbf{y}_{i,m}$ is the obtained approximation of the eigenvector from Eq.(2.11). If the norm of the term on the right side $h_{(m+1)\,m}|e_m^T y_i^m|$ is below a certain threshold, the process is terminated. [7] [3]

2.2.2 Additional notes about the Arnoldi method

There are two extra issues regarding the Arnoldi method that have to be discussed. First, the dimension of the Krylov subspace (m) has to be chosen. This issue is not critical, as a dimension $m = 50$ was shown to be sufficient for the Helmholtz equation. This is due to the fact that the Arnoldi iteration becomes largely insensitive to changes in m around that threshold [8]. The second issue relates to large Krylov subspaces. For such spaces, the Arnoldi vectors produced by the modified Gram-Schmidt procedure are not orthogonal to machine precision. This leads to false eigenvalues [9]. In this case, explicit re-orthogonalization of the basis becomes preferable. This can be done for instance with the DKGS method, which was introduced by Daniel *et al.* [10]. However, the COMSOL version (4.4) used in this work does not support this feature.[3]

2.2.3 Shift and invert method

To solve the eigenvalue problem in Eq. (2.11), the implicitly restarted Arnoldi method described in Section 2.2.1 is used. However, this method is only appropriate for searching for the lowest an highest eigenvalues. If the eigenvalues around a specific point are of interest, the shift and invert method can be coupled with the Arnoldi method to achieve this[7]. The shift and invert method is implemented in the ARPACK library [7]. In this section, only the core idea behind this method is introduced. The publication by Lehoucq *et al.* [7] is recommended for those interested in a more complete understanding of the algorithm.

Let's define σ as the point where the solver should search for eigenvalues. σ is then called the shift. First, let's apply the spectral transformation

$$\mu = \frac{1}{\lambda - \sigma} \tag{2.20}$$

to Eq. (2.11). This yields

$$\mathbf{Ax} = \left(\sigma + \frac{1}{\mu}\right)\mathbf{Bx}\,. \tag{2.21}$$

This can be rearranged as

$$(\mathbf{A} - \sigma\mathbf{B})\mathbf{x} = \frac{1}{\mu}\mathbf{Bx}\,, \tag{2.22}$$

which in turn can be rearranged as

$$\underbrace{(\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}}_{\mathbf{C}}\mathbf{x} = \mu\mathbf{x}\,. \tag{2.23}$$

The shift and invert method consists of using the Arnoldi method (more precisely, the implicitly shifted Arnoldi method presented in Section 2.2.1) to search for the highest eigenvalues of $\mathbf{C} = (\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}$. As can be seen Eq.(2.20), the highest eigenvalues of $\mathbf{C}$ correspond to the eigenvalues of the system at Eq.(2.11) which are closest to the shift σ. After obtaining the eigenvalues μ_i, it is possible to transform the eigenvalues backwards to the original configuration by using the relation

$$\lambda_i = \sigma + \frac{1}{\mu_i}\,, \tag{2.24}$$

which immediately follows from Eq.(2.20).

The most memory-critical part of the shift and invert method is the transformation into the form at Eq.(2.23). This requires solving the system

$$(\mathbf{A} - \sigma\mathbf{B})\mathbf{C} = \mathbf{B} \tag{2.25}$$

For the matrix $\mathbf{C}$. This is done by solving

$$(\mathbf{A} - \sigma\mathbf{B})\mathbf{c}_i = \mathbf{b}_i \quad \forall i = 1,2,\ldots,n\,, \tag{2.26}$$

where $\mathbf{c}_i$ and $\mathbf{b}_i$ are the columns of $\mathbf{C}$ and $\mathbf{B}$ respectively, using any of the solvers form Sections 2.5 and 2.6

2.3 Frequency response functions

Two of the test cases treated in the present work are cases where some frequency response function (FRF) is the value of interest. Therefore, a brief introduction to this concept in provided here.

For a simple single input-single output system, a linear time-invariant system can be represented as a state-space model as follows[11]:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u \tag{2.27}$$

$$y = \mathbf{c}^T x \tag{2.28}$$

Applying Laplace transformation to Eq.(2.27) and Eq. (2.27) yields

$$s\mathbf{X}(s) = \mathbf{A}\mathbf{X}(s) + \mathbf{b}U(s)\ , \tag{2.29}$$

$$Y = \mathbf{c}^T\mathbf{X}(s)\ . \tag{2.30}$$

Note that $\mathbf{X}(s)$ is a vector, not a matrix. The upper-case notation is used here to indicate Fourier or Laplace transformation. Solving Eq. (2.29) for $\mathbf{X}(s)$ and inserting it into Eq. (2.30) yields

$$Y(s) = c^T(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{b}U(s)\ . \tag{2.31}$$

The FRF of a system can be defined as

$$\mathbf{G} := \frac{Y(\omega)}{U(\omega)}\ , \tag{2.32}$$

where $Y(\omega)$ is the Fourier transform of the output and $U(\omega)$ is that of the input. From Eq. (2.32) and Eq. (2.31), the relation

$$G(s) = c^T(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{b} \tag{2.33}$$

is obtained[11]. Eq.(2.33 can be solved using any of the solvers discussed in Sections 2.5 and 2.6.

2.4 Solving systems of linear equations

Take a system of linear equations represented as follows

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{2.34}$$

Where $\mathbf{A} \in \mathbb{C}^{n\times n}$, $\mathbf{x} \in \mathbb{C}^{n\times 1}$ and $\mathbf{b} \in \mathbb{C}^{n\times 1}$. Solving such systems is required in both eigenvalue and frequency response algorithms. This mainly applies for the systems of linear equations presented in Eq.(2.26) and Eq.(2.33). Such systems can either be solved using direct or iterative solvers. In this section, both these classes of solvers will be introduced, while only exploring iterative solvers in a detailed manner. Afterwards, an overview of preconditioners will be presented. Occasionally, some noteworthy differences are present between the preconditioned and unpreconditioned versions of the algorithms may be present. Unless explicitely otherwise stated, the unpreconditioned version will be the one that is discussed.

2.5 Direct solvers

The system described by Eq. (2.34) can be solved using direct methods. These will, in this Section, receive a very short treatment, as they lay beyond the scope of this work.

The most basic direct solver is possibly Gaussian elimination. Most direct solvers, such as the LU Decomposition and the QR Decomposition, are based, at least in part, on it [12]. Consistent innovations have been achieved in this class of solvers over the years, such as the introduction of variants of the classical factorization methors that can exploit band structures and better cope with large systems. Among these are the MUMPS [13] and PARDISO solvers[14].

A defining feature of this class of solvers is that, when neglecting round-off errors, it provides an exact solution after a finite number of steps and are typically relatively fast. However, their memory consumption increases rapidly with increasing problem size. Some estimates state that the memory requirements increases with order $\mathcal{O}(n^3)$[15]. At a certain problem size, such memory consumption is often unaffordable. Moreover, numerical schemes typically deliver a linear equation system where the matrix **A** has a pronounced, often narrow band structure. Most direct solvers do not take much of an advantage of this, resulting instead in a so-called fill in effect. This means that the entries in the matrix **A** that were originally zero acquire a non-zero value during the solution process [16].

In this work, the solver MUMPS was used as a benchmark to compare iterative solvers to. As such, a brief presentation of this solver is provided in the following

2.5.1 Multifrontal Massively Parallel Solver (MUMPS)

The MUMPS solver is a direct solver based on Gaussian elimination that was developed by Amestoy *et al.* [13]. It is a parallel, fully asynchronous solver based on a multifrontal approach that implements classical pivoting during factorization. It is designed such that it adapts to numerical work load during calculation. Moreover, it reasonably exploits existing sparsity patterns and the resulting independence of computations, as well as the independence of calculations also present in dense matrices. These features result in a high performance for this solver. Equally important is MUMPS's capacity for solving a wide range of problems. This includes symmetric, asymmetric, and indefinite matrices using **LU** or $\mathbf{LDL}^T$ factorization. For an in-depth understanding of the algorithm, the publication Amestoy *et al.* [13] is recommended.

For the purposes of this work, it might be helpful to know that MUMPS (as with other direct solvers present in COMSOL 4.4) actively exploits shared memory parallelism (such as multicore processors) according to the COMSOL 4.4 reference manual [6]. Consult the same publication for more information regarding the implementation of MUMPS.

2.6 Iterative solvers

The purpose of iterative solvers is to avoid the problems associated with direct solvers, particularly their high memory consumption. This comes at the cost of sacrificing robustness. The defining feature of iterative schemes is that they only approach the exact solution asymptotically, never truly arriving at it. Nonetheless, this can be overlooked, since the discretization process introduces its own error. It is therefore only important that the error of the iterative solver remain at lower magnitude then the discretization[17].

The most appropriate introduction to this class of solvers is are probably the Jacobi, Gauss-Seidel and SOR schemes due to their simplicity. Therefore, they shall be discussed first in the following section. However, according to Schaefer [16], they are not very effective. As a result, another class of solvers, the Krylov subspace methods, was developed. Among such solvers is the conjugate Gradient (CG) method, which is highly efficient as it only involves one matrix-vector multiplication. Yet this method reliably converges only for Hermitian, positive definite matrices. The method was thus extended to handle non-Hermitian matrices and indefinite systems. Further development of this method has delivered the biconjugate Gradients (BiCG) [18] and the Biconjugate Gradients stabilized (BiCGStab) methods [19]. Although these methods do not suffer from the drawbacks of the Conjugate Gradient Method, they do not satisfy the optimality condition[1] and converge irregularly[20]. The last Krylov subspace solver that will be discussed is the GMRES method, which is optimal and does not suffer from significant instability or robustness issues, at the cost of higher memory consumption [1].

Iterative solvers are typically not used separately, but are instead coupled with preconditioners. Mathematically, a preconditioner transforms problem statement into a form that is numerically easier to solve[21]. As discussed throughout this section, the convergence properties of iterative solvers are strongly dependent an an appropriate choice of preconditioner. This is verified by the results from this work (see tables in Appendix C.3). Therefore, instead of just discussing solvers, it is often more meaningful to discuss solver-preconditioner combinations. The latter is done often throughout this work. Preconditioners are discussed in more details in section 2.7

Another class of solvers that shows promise is the Multi-resolution methods, which can be used either as solvers or as preconditioners to other methods. Essentially solve the system on a coarser mesh using a robust solver, while using extra operations (pre- and post-smoothers) to smooth out the error produced by such an operation. This is discussed more thoroughly in subsection 2.7.6. Also see the same subsection for a brief explanation of the role of pre- and post-smoothers.

2.6.1 Classical splitting methods

These methods are also commonly referred to as stationary iterative solvers or relaxation methods. They are limited in their applicability to diagonally-dominant systems of linear equations. This means that the convergence is guaranteed only if the condition $|a_{ii}| \geq \Sigma_{i \neq j} |a_{ij}|$ is satisfied, although they may still converge otherwise [22].

Consider the Linear equation system 2.34. The system can be extended as described by equation 2.35

$$\mathbf{Bx} - \mathbf{Bx} + \mathbf{Ax} = \mathbf{b} \; , \tag{2.35}$$

[1]A global minimum for the residual is not necessarily reached

where **B** is a square matrix with suitable dimension. The general iteration rule for the step $k+1$ can thus be derived as

$$\mathbf{B}\mathbf{x}^{(k+1)} - \mathbf{B}\mathbf{x}^{(k)} + \mathbf{A}\mathbf{x}^{(k)} = \mathbf{b} \tag{2.36}$$

or

$$\mathbf{x}^{(k+1)} = \mathbf{B}^{-1}(\mathbf{B}-\mathbf{A})\mathbf{x}^{(k)} + \mathbf{B}^{-1}\mathbf{b}\,. \tag{2.37}$$

By defining

$$\mathbf{M} = \mathbf{B}^{-1}(\mathbf{B}-\mathbf{A})\,, \tag{2.38}$$

the equation changes to

$$\mathbf{x}^{(k+1)} = \mathbf{M}\,\mathbf{x}^{(k)} + \mathbf{B}^{-1}\,\mathbf{b}\,. \tag{2.39}$$

It can be observed from Eq.(2.38) and (2.39) that the matrix **B** should ideally be easily invertible. By choosing $\mathbf{B} = \mathbf{I}$, this may be achieved, but at the cost of slower convergence. On the other extreme, the choice of $\mathbf{B} = \mathbf{A}$ causes the scheme to converge after exactly one step. But this is strips the scheme of any practical or theoretical significance, as the iteration rule then simplifies to

$$\mathbf{x}^{(k+1)} = \mathbf{A}^{-1}\mathbf{b} \tag{2.40}$$

which is the same as Eq.(2.34) and does not answer the question of how to efficiently invert **A**. The best course of action is choosing a pragmatic compromise between the two approaches. This is commonly implemented by opting to assign certain values of **A** to **B**.

A can be decomposed as

$$\mathbf{A} = \mathbf{A_L} + \mathbf{A_U} + \mathbf{A_D} \tag{2.41}$$

where $\mathbf{A_L}$ and $\mathbf{A_U}$ are respectively the lower- and upper-triangular components of A, and $\mathbf{A_D}$ is the matrix of the diagonal elements of **A**. From this, the following iterative methods can be derived:

Jacobi:	$\mathbf{B} = \mathbf{A_D}$	(2.42)
Gauss-Seidel:	$\mathbf{B} = \mathbf{A_D} + \mathbf{A_L}$	(2.43)
Successive Over-Relaxation (SOR):	$\mathbf{B} = \dfrac{\mathbf{A_D} + \omega_{SOR}\mathbf{A_L}}{\omega_{SOR}}$	(2.44)
Successive Over-Relaxation U (SORU):	$\mathbf{B} = \dfrac{\mathbf{A_D} + \omega_{SOR}\mathbf{A_U}}{\omega_{SOR}}$	(2.45)
Symmetric Successive Over-Relaxation (SSOR):	$\mathbf{B} = \dfrac{\omega_{SOR}}{2-\omega_{SOR}}\left(\dfrac{\mathbf{A_D} + \omega_{SOR}\mathbf{A_L}}{\omega_{SOR}}\right)\mathbf{D}^{-1}\left(\dfrac{\omega_{SOR}\mathbf{A_U} + \mathbf{A_D}}{\omega_{SOR}}\right)$	(2.46)

.

Some versions of the Jacobi algorithm scale the matrix $\mathbf{A}_D$ by an additional factor of ω. The parameter ω_{SOR} is called the relaxation parameter. It should be chosen so that $\omega_{SOR} \in (0,2)$. A choice of $\omega_{SOR} < 1$ can be made to increase the speed of convergence of an overshooting process, or establish the convergence of otherwise diverging processes, whereas a choice of $\omega_{SOR} > 1$ tends to speed up convergence of a slow converging process. For $0 < \omega_{SOR} < 2$, the

SOR method converges if **A** is symmetric positive definite [2] [24], [6]. In general, though, these methods may or may not converge regardless of symmetry. They will, however, definitely fail if the system matrix has zeroes on its diagonals[6].

Krylov subspace Methods

A Krylov subspace is the space generated by an $n \times n$ matrix **A** and an $n \times 1$ vector **b** that spans the product of the first r powers of **A** multiplied with **b** [4]. It is expressed as

$$\mathcal{K}_r(\mathbf{A}, \mathbf{r}^{(0)}) = \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \mathbf{A}^2\mathbf{r}^{(0)}, \ldots \mathbf{A}^{r-1}\mathbf{r}^{(0)}\} .$$

where

$$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$$

and $\mathbf{x}^{(0)}$ is the initial estimate.

An iterative method that is based on the projection of the system on a Krylov subspace is called a Krylov subspace method [25]. The methods that will be discussed from this category are CG, BiCG (briefly), BiCGStab, GMRES, and the FGMRES.

2.6.2 Conjugate gradient (CG)

For the conjugate gradient method, the system in equ.2.34 should possess a matrix A that is symmetric and positive definite, though it might also occasionally converge for matrices that are not positive definite, especially if the matrix is close to positive definite [6]. the method, as described by Hestenes and Stiefel [26] can be understood as follows in this section.

Consider

$$\mathbf{A}\mathbf{x} - \mathbf{b} = 0 , \tag{2.47}$$

and define **Q** as

$$\mathbf{Q}(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} . \tag{2.48}$$

Q is called a quadratic form. Taking the first derivative of **Q** relative to **x** yields

$$\frac{\partial \mathbf{Q}(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{b} = 0 . \tag{2.49}$$

The second derivative yields

$$\frac{\partial^2 \mathbf{Q}(\mathbf{x})}{\partial \mathbf{x}^2} = \mathbf{A} . \tag{2.50}$$

Therefore, if **A** is positive definite, the solution of Eq.(2.47) represents a global minimum for the quadratic function $\mathbf{Q}(\mathbf{x})$.

It is helpful at this point to consider the geometric implications of this minimization problem. The first point of interest is the fact that the solving the equation $\mathbf{Q}(\mathbf{x}) = c$ produces an

[2] A proof for this is provided by Quarteroni *et al.* [23]

ellipsoid in $\mathbb{C}^n$. One way of solving the minimization problem in (15) is by first defining the residual $\mathbf{r}(\mathbf{x})$

$$\mathbf{r}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b} \tag{2.51}$$

and choosing an arbitrary vector $\mathbf{v}^{(0)}$ and an arbitrary search direction $\mathbf{p}^{(0)}$. The idea is to minimize $\mathbf{Q}$ along this direction, that is solving the equation

$$\frac{\partial}{\partial x}\mathbf{Q}(\mathbf{x}^{(0)} + q\mathbf{p}^{(0)}) = 0 \tag{2.52}$$

for q. Eq.(2.52) for q results in

$$q = \frac{\mathbf{p}^{(0)T}\mathbf{r}(\mathbf{x}^{(0)})}{\mathbf{p}^{(0)T}\mathbf{A}\mathbf{p}^{(0)}}. \tag{2.53}$$

Now is is possible to set

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + q^{(0)}\mathbf{p}^{(0)} . \tag{2.54}$$

A new vector $\mathbf{v}^{(1)}$ has thus been obtained. The next step is to choose a new search direction $\mathbf{p}^{(1)}$. Since the $\mathbf{Q}(\mathbf{x}) = c$ admits ellipsoids as its solution, the new search direction $\mathbf{p}^{(1)}$ and the old one $\mathbf{p}^{(0)}$ should ideally be A-conjugate, that is

$$\mathbf{p}^{(1)T}\mathbf{A}\mathbf{p}^{(0)} = 0 . \tag{2.55}$$

The new residual then reads as:

$$r^{(1)} = \mathbf{A}\mathbf{x}^{(1)} - \mathbf{b} = \mathbf{r}^{(1)} + e.\mathbf{p}^{(0)} . \tag{2.56}$$

Searching for the A-conjugate direction to $\mathbf{p}^{(0)}$ in the plane defined by the previous direction and the gradient of $\mathbf{r}^{(1)}$ in the direction of $\mathbf{v}^{(1)}$ results in

$$\mathbf{p}^{(1)} = -\mathbf{r}^{(1)} + e.\mathbf{p}^{(0)} \tag{2.57}$$

where

$$e = \frac{\mathbf{r}^{(1)}\mathbf{A}\mathbf{p}^{(0)}}{\mathbf{p}^{(0)}\mathbf{A}\mathbf{p}^{(0)}} . \tag{2.58}$$

Having obtained a new search direction $\mathbf{p}$, a new vector $\mathbf{x}$, and a new residual $\mathbf{r}$, which is smaller than the initial residual, the current iteration is concluded and the next iteration can be launched by setting $\mathbf{x}^{(0)} = \mathbf{x}^{(1)}$,$\mathbf{r}^{(0)} = \mathbf{r}^{(1)}$ and $\mathbf{p}^{(0)} = \mathbf{p}^{(1)}$, and restarting the process. Interrupt the process as soon as a certain error criterion has been met, and take the corresponding $\mathbf{x}$ vector as your approximation. [26]. A graphical example is presented in Figure 2.1. [3]

If one neglects round-off errors, this method converges after a maximum of n steps (n being the number of degrees of freedom of the system). It is for this reason that CG can theoretically be considered a direct method [27]. However this method suffers with build-up of round-off error if too many iterations are performed. Provided the matrix $\mathbf{A}$ is symmetric positive definite, CG can nevertheless be a descent iterative method, as it typically requires about half the memory required by other iterative solvers. [6]

[3]Credits: By Oleg Alexandrov, wikipedia commons, public domain.

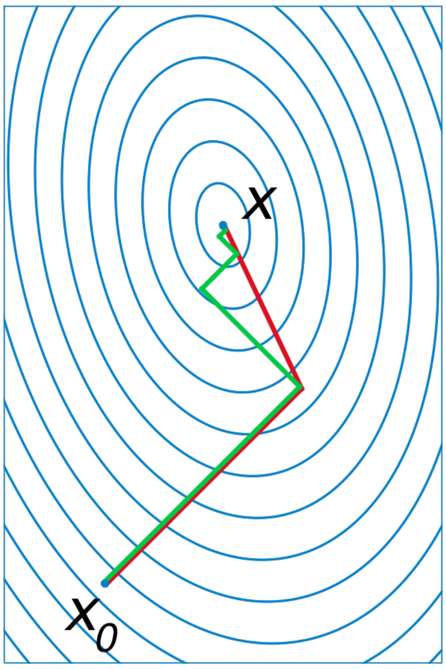


Figure 2.1: A graphical example of the conjugate gradient method (CG) for a small problem of dimension 2 (in red), along the less optimal Gradient Descent method (in green) for the same problem. The blue, concentric ellipses represent solutions for the equation $\mathbf{Q}(\mathbf{x}) = c$ for different values of c. Conjugate gradient can easily be seen as minimizing the quadratic function $\mathbf{Q}(\mathbf{x})$.

It has several other advantages, such as its suitableness for parallelization and its ability exploit sparse matrix structures [see for instance 28]. Accurate computational effort estimates are difficult to come by, but according to Van der Vorst [29], the CG-Method scales with $\mathcal{O}(n^{\frac{3}{2}})$ in 2-D cases and $\mathcal{O}(n^{\frac{4}{3}})$ for 3-D problems. Consequently, while CG is less suitable for 2-D problems, It is very effective for 3-D problems. This can be further enhanced by the choice of an appropriate preconditionner, reaching a scaling factor as low as $\mathcal{O}(n^{\frac{7}{6}})$ according to Axelsson [24] and Gustafsson [30].

Although the CG method is attractive, it has significant restrictions regarding it's applicability. Namely, it only reliably converges for positive definite matrices $\mathbf{A}$. Handling this restriction is not straightforward. Therefore, the next paragraph is dedicated to how to best tell whether $\mathbf{A}$ is positive definite.

Criteria for positive definiteness

Generally speaking, a matrix $\mathbf{A}$ is positive definite if the criterion

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > 0 \quad \forall \mathbf{x} \neq 0 \tag{2.59}$$

applies. However, this criterion is often difficult to check. Therefore, more practical criteria must be used instead. This can be achieved by restricting the discussion to symmetric matrices.

Indeed, for a symmetric matrix, positive definiteness is present if the matrix is strictly diagonally dominant with strictly positive diagonal entries, i.e.

$$a_{i,i} > 0 \quad \text{and} \quad a_{i,i} > \sum_{i \neq j}^{n} |a_{i,j}| \quad \forall i = 1,2,\ldots,n\,. \tag{2.60}$$

[31] If the matrix is diagonally dominant, but not strictly so (that is, $a_{i,i} \geq \sum_{i \neq j}^{n} |a_{i,j}|$), then the matrix is positive semi-definite. This already improves the chances of convergence for CG [6]. This theorem is important since many discretization schemes deliver such structures.

Conjugate gradients for the Helmholtz equation

The Helmholtz equation is symmetric. Moreover, it tends to produce well-conditioned systems of linear equations, but source terms and complex boundary conditions can make it ill-conditioned. If the system is known to be well-conditioned, then it also tends to be positive definite. This is confirmed by the fact that CG worked for all the Helmholtz cases present in this study.

2.6.3 Biconjugate gradient (BiCG)

The Bi-CG Method is a modification of the classical CG-Method that aims to extend the method to non-Hermitian and indefinite systems. This method is of little practical importance, as it is surpassed by other methods in usefulness such as the biCGStab. It does, however, have a certain theoretical importance[18]. In the following, a complete description of the basic algorithm is provided. Start by choosing an initial guess $\mathbf{x}^{(0)}$, and two other vectors $\hat{\mathbf{v}}^{(0)}$ and $\hat{\mathbf{b}}$. The initial guess can be chosen arbitrarily, but a good guess could significantly speed up convergence. The same applies to $\hat{\mathbf{v}}^{(0)}$ and $\hat{\mathbf{b}}$, with the exception that the latter two are preferably chosen to be different than zero to avoid any anomalous behavior. Calculate the initial residuals

$$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)} \tag{2.61}$$

$$\hat{\mathbf{r}}^{(0)} = \hat{\mathbf{b}} - \hat{\mathbf{v}}^{(0)}\mathbf{A}^{(0)} \tag{2.62}$$

then define the new search directions

$$\mathbf{p}^{(0)} = \mathbf{r}^{(0)} \tag{2.63}$$

$$\hat{\mathbf{p}}^{(0)} = \hat{\mathbf{r}}^{(0)}\,. \tag{2.64}$$

The iteration rule for the step $k+1$ is then defined as follows

$$q^{(k)} = \frac{\hat{\mathbf{r}}^{(k)}(\mathbf{r})^{(k)}}{\hat{\mathbf{p}}^{(k)}\mathbf{A}\mathbf{p}^{(k)}}\,. \tag{2.65}$$

This definition aims to establish the biorthogonality condition, i.e.

$$\hat{\mathbf{r}}^{(k+1)T}\mathbf{r}^{k)} = \mathbf{r}^{(k+1)T}\hat{\mathbf{r}}^{(k)}$$

[18, page 80]. Then proceed with

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + q^{(k)}\mathbf{p}^{(k)} \tag{2.66}$$

$$\hat{\mathbf{x}}^{(k+1)} = \hat{\mathbf{x}}^{(k)} + q^{(k)}\hat{\mathbf{p}}^{(k)} \tag{2.67}$$

$$\mathbf{r}^{k+1} = \mathbf{r}^{(k)} - q^{(k)}\mathbf{A}\mathbf{p}^{(k)} \tag{2.68}$$

$$\hat{\mathbf{r}}^{(k+1)} = \hat{\mathbf{r}}^{(k)} - q^{(k)}\hat{\mathbf{p}}^{(k)}\mathbf{A}^T \tag{2.69}$$

$$e^{(k)} = \frac{\hat{\mathbf{r}}^{(k+1)T}\mathbf{r}^{k+1)}}{\hat{\mathbf{r}}^{(k)T}\mathbf{r}^{(k)}}\,. \tag{2.70}$$

This definition aims to establish the biconjugacy condition, i.e

$$\hat{\mathbf{p}}^{(k+1)T}\mathbf{A}\hat{\mathbf{p}}^{(k)} = \hat{\mathbf{p}}^{(k+1)T}\mathbf{A}\mathbf{p}^{(k)} = 0$$

[18] similarly to 2.55. Then proceed with

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + e^{(k)}\mathbf{p}^{(k)} \tag{2.71}$$

$$\hat{\mathbf{p}}^{(k+1)} = \hat{\mathbf{r}}^{(k+1)} + e^{(k)}\hat{\mathbf{p}}^{(k)}\,. \tag{2.72}$$

This is repeated until a desirable accuracy is reached. Although The BiCG Method can handle non-Hermitian matrices, it require around double the computational effort of the CG method, and is suffers from stability issues [18]. This method has been mentioned here since it is a helpful intermediate step to understand the BiCGStab Method. It also has other theoretical points of interest, such as its relation to Quasi-Newton methods and more [see 18]. Since the method is surpassed by BiCGStab in most important respects, it is not implemented in the COMSOL software package.

2.6.4 Biconjugate Gradient Stabilized (BiCGStab)

. Further development of the BiCG method yielded the BiCGStab method, that seeks to solve the stability issues present in BiCG and achieve stronger convergence behavior. As such, BICGStab is one of the main iterative solvers COMSOL provides [6]. In the following, a summary of the preconditioned implementation of BiCGStab is provided.

Calculate the residual

$$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)} \tag{2.73}$$

then choose an arbitrary vector $\hat{\mathbf{r}}^{(0)}$ such that $\hat{\mathbf{r}}^{(0)T}\hat{\mathbf{r}}^{(0)} \neq 0$. Define the parameters $\rho^{(0)}$, α, β, ω^0 such that

$$\rho^{(0)} = \alpha = \omega^0 = 1 \tag{2.74}$$

as well as $\mathbf{v}^{(0)}$, $\mathbf{h}$ and $\mathbf{p}^{(0)}$ where

$$\mathbf{v}^{(0)} = \mathbf{p}^{(0)} = \mathbf{0}\ . \tag{2.75}$$

The iteration rule for the step $k+1$ is then as follows:

$$\rho^{(k)} = \hat{\mathbf{r}}^{(0)T}\mathbf{r}^{(k-1)} \tag{2.76}$$

$$\beta = \frac{\rho^{(k)}}{\rho^{(k-1)}} \cdot \frac{\alpha}{\omega^{(k-1)}} \tag{2.77}$$

$$\mathbf{p}^{(k)} = \mathbf{r}^{k-1} + \beta(\mathbf{p}^{(k-1)} - \omega^{(k-1)}\mathbf{v}^{(k-1)} \tag{2.78}$$

$$\mathbf{t} = \mathbf{A}\mathbf{s} \tag{2.79}$$

$$\omega^{(k)} = \frac{\mathbf{t^T s}}{\mathbf{t^T t}} \tag{2.80}$$

$$\mathbf{x}^k = \mathbf{h} + \omega^{(k)}\mathbf{s}\ . \tag{2.81}$$

If $\mathbf{x}^k$ is sufficiently accurate, then quit. Othewise, calculate the new residual

$$\mathbf{r}^{(k)} = \mathbf{s} - \omega^k \mathbf{t}\ , \tag{2.82}$$

then proceed to the next iteration step.[19]

For each iteration, this solver requires the same amount of time and memory. This gives it an advantage over other methods such as GMRES, which requires about twice as much memory per iteration. On the other hand, it has a less regular convergence behavior than GMRES, often increasing the residual from one step to a later step by several orders of magnitude. This can affect the rate of convergence and the numerical accuracy. In the implementation in the COMSOL software package, the algorithm is modified in order to be able to detect low accuracy and stagnation; and restart the iterations accordingly, using the current appoximation as the initial guess.

Another noteworthy point is that BiCGStab requires two or three preconditioning steps per iteration when using right- or left-preconditioning, respectively; whereas CG and GMRES need both one preconditioning step per iteration.[6] [19] [32]

2.6.5 Generalized minimal residual method (GMRES)

The Generalized Minimal Residual Method (GMRES) is an extention of the the MINRES method [1]. It was developed by Saad and Schultz [1] to cope with non-Hermitian systems. It is one of the most commonly used iterative methods for solving large linear equation system [33]. As the name suggests, the reasoning behind the algorithm starts from the problem of iteratively minimizing the norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, that is,

$$\min \|\mathbf{r}\| = \min \left\|\mathbf{A}\mathbf{x} - \mathbf{r}^{(0)}\right\|\ . \tag{2.83}$$

Note here that

$$\mathbf{r}^{(0)} = \mathbf{A}\mathbf{x}^{(0)} + \mathbf{b}\ , \tag{2.84}$$

where the initial guess $\mathbf{x}^{(0)}$.

However, the since n is very large, minimizing the residual in this state is too costly, computationally. Instead, GMRES projects the problem onto a Krylov subspace $\mathscr{K}_{\tilde{m}}(\mathbf{A},\mathbf{r}^{(0)})$. This space lays within the span of $\mathbf{A}$, with a much lower dimension $\tilde{m}$ that may vary depending on implementation. This way, the minimization problem can be solved economically.

The Arnoldi algorithm is used to find an orthogonal basis $\{\mathbf{v}_1,\ldots,\mathbf{v}_n\}$ for $\mathscr{K}_{\tilde{m}}(\mathbf{A},\mathbf{r}^{(0)})$. The vectors of this basis are arranged column-wise in a Matrix $\mathbf{V}$. Additionally, $\mathbf{x}$ can be expressed as $\mathbf{x} = \mathbf{V}_{n,\tilde{m}}\mathbf{y}$. Consequently,

$$\min \|\mathbf{r}\| = \min \left\|\mathbf{A}\mathbf{V}_{n,\tilde{m}}\mathbf{y} - \mathbf{r}^{(0)}\right\| . \tag{2.85}$$

The vectors $\mathbf{v}_i$ satisfy the relation

$$\mathbf{A}\mathbf{V}_{n,\tilde{m}} = \mathbf{V}_{n,\tilde{m}+1}\mathbf{H}_{\tilde{m}+1,\tilde{m}} \, . \tag{2.86}$$

One obtains thus

$$\min \left\|\mathbf{r}^{(0)}\right\| = \min \left\|\mathbf{V}_{n,\tilde{m}+1}\mathbf{H}_{\tilde{m}+1,\tilde{m}}\mathbf{y} - \mathbf{r}^{(0)}\right\| . \tag{2.87}$$

This is equivalent to

$$\min \|\mathbf{r}\| = \min \left\|\mathbf{V}^{*}_{n,\tilde{m}+1}\mathbf{V}_{n,\tilde{m}+1}\mathbf{H}_{\tilde{m}+1,\tilde{m}}\mathbf{y} - \mathbf{Q}^{*}_{n,\tilde{m}+1}\mathbf{r}^{(0)}\right\| \tag{2.88}$$

leading to

$$\min \|\mathbf{r}\| = \min \left\|\mathbf{H}_{\tilde{m}+1,\tilde{m}}\mathbf{y} - \left\|\mathbf{r}^{(0)}\right\|\mathbf{e}_1\right\| . \tag{2.89}$$

This is based on the knowledge that $\mathbf{V}_{n,\tilde{m}+1}$ is orthonormal and $\mathbf{V}^{*}_{n,\tilde{m}+1}\mathbf{r}^{(0)} = \left\|\mathbf{r}^{(0)}\right\|\mathbf{e}_1$, where $\mathbf{e}_1 = (1,0,\ldots)^T$.

Finding an appropriate y that minimizes the residual in Eq.(2.89) is equivalent to solving the problem statement. This is a least square problem that can be solved using an appropriate solver. This is discussed in later in this section.

A simple matrix- vector multiplication $\mathbf{x} = \mathbf{V}_{n,\tilde{m}+1}\mathbf{y}$ is then sufficient to arrive at the solution. The Hessenberg matrix $\mathbf{H}_{n,\tilde{m}+1}$ determines the size of the problem to be solved.

GMRES requires the storage of the basis vectors for the Krylov subspace. This is possibly its main disadvantage, as it considerably increases memory consumption. For every iteration, the number of stored vectors increases by one, every vector being of size n. To solve this issue, a maximum number of steps m may be defined a priori, after which the process restarts. In this case, the residual from the last iteration is used to calculate a new basis for the Krylov subspace for use in the next set of iterations. Also, the last approximation is used as the initial guess for the next set of iterations [1]. This variant of the algorithm is called GMRES(m) or restarted GMRES. However, the restarted subspace and the earlier subspace are often similar, which causes the method to suffer from stagnation problems in convergence. This variant is nonetheless the default implementation of GMRES in the COMSOL software package, possibly due to it's favorable memory consumption [6].

Indeed, in practice, it is more convenient and common to use GMRES(m), as the parameter m can then be adjusted according to the estimated memory consumption. A rough algorithmic representation of this method is as follows:

1. Choose an initial guess $\mathbf{x}^{(0)}$ (usually 0) and calculate the initial residual $\mathbf{r}^{(0)}$;
2. Repeat:
   (a) Use the Arnoldi method to calculate $\mathbf{V}_{n,m+1}$ and $\mathbf{H}_{m+1,m}$ (see section 2.1);
   (b) Minimize the residual with an appropriate $\mathbf{y}^{(k)}$ (explained in the following);
   (c) Compute $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{V}_{n,m+1}\mathbf{y}$;
   (d) Calculate the new residual $\mathbf{r}^{(1)}$;
   (e) if residual norm is too large:
      i. set the new residual to the initial residual $\mathbf{r}^{(0)} = \mathbf{r}^{(1)}$;
      ii. set the obtained approximation$\mathbf{x}^{(1)}$ as the new initial guess $\mathbf{x}^{(0]}$.
   (f) if residual small enough, end process and return the final $\mathbf{x}$.

In practice, the intermediate approximations for $\mathbf{x}$ do not have to be explicitly computed, since the residual norm can be computed using the relation

$$||\mathbf{r}^{(1)}|| = h_{m+1,m}|\mathbf{e}_m\mathbf{y}| \,. \tag{2.90}$$

This saves some unnecessary calculation steps, and thus saves time [1].

GMRES is an optimal algorithm and the solution converges monotonically. However, its convergence rate strongly depends on the preconditioner. This is discussed in detail in Meister [20]. As discussed earlier in Section 2.6.2, Krylov subspace methods are generally well suited to treat 3D problems, but much less so for 2D problems. In general, GMRES requires about twice as much memory per iteration as CG[6], though this depends greatly on the structure of the problem, the condition number, and the number of iterations before GMRES is restarted. [6]

Solving the minimization problem

As discussed in this subsection thus far, solving the least square problem in Eq.(2.89), namely

$$\min \|\mathbf{r}\| = \min \left\|\mathbf{H}_{k+1,k}\mathbf{y} - \rho\mathbf{e}_1\right\| \tag{2.91}$$

where $\rho = \left\|\mathbf{r}^{(0)}\right\|$, is an important part of GMRES. There are multiple ways to achieve this. One of these methods is based on the QR decomposition, which will be discussed here (as proposed by Saad and Schultz [1]). The parameter k is chosen for the equation 2.91, since it is a good idea to orient this discussion to GMRES(m), instead of standard GMRES. Nevertheless, this discussion still holds for both.

The basic idea behind the use of QR decomposition here is to find an orthogonal $\mathbf{Q}$ such that

$$\mathbf{Q}\mathbf{H}_{k+1,k} = \mathbf{R} = \begin{pmatrix} \hat{\mathbf{R}} \\ \mathbf{0}^T \end{pmatrix} \tag{2.92}$$

where $\mathbf{R} \in \mathbb{C}^{k+1\times k}$. In the particular case of the equation 2.91, the last row of $\mathbf{R}$ is a zero row vector in $\mathbb{C}^k$. $\hat{\mathbf{R}}$ is an upper triangular matrix. Note that the structure of $\mathbf{H}_{k+1,k}$ significantly reduces the computational effort for performing this decomposition [1].

Let's define the rotation matrix $\mathbf{F}^{(j)}$, A matrix that rotates the basis vectors $\mathbf{e}_j$ and $\mathbf{e}_{j+1}$ by an angle θ. This matrix takes the shape

$$\mathbf{F}^{(j)} = \begin{array}{cl} \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & \cos(\theta_j) & -\sin(\theta_j) & & \\ & & & \cos(\theta_j) & \sin(\theta_j) & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix} & \begin{array}{l} \\ \\ \\ \leftarrow \text{row } j \\ \\ \\ \\ \\ \end{array} \end{array} \tag{2.93}$$

Let's suppose that the rotations $\mathbf{F}^{(i)}$ for i from 1 to j were applied to the matrix $\mathbf{H}^{(j)}$[4] to produce the upper triangular matrix

$$\mathbf{R}^{(j)} = \begin{pmatrix} \hat{\mathbf{R}}^{(j)} \\ \mathbf{0}^T \end{pmatrix} . \tag{2.94}$$

As with the equation 2.92, $\hat{\mathbf{R}}_j \in \mathbb{C}^{j\times j}$ [5]. At the next step, the last column and row of $\mathbf{H}^{(j+1)}$ is appended to $\mathbf{R}_j$. Therefore, in order to obtain $\mathbf{R}^{(j+1)}$, one has to start by multiplying the new column by the previous rotations. This puts the size of $\mathbf{R}^{(j+1)}$ at $(j+2)\times(j+1)$. The new rotation has the structure

$$\mathbf{R}^{(j+1)} = \begin{pmatrix} \hat{\mathbf{R}}^{(j)}_{j,j} & \mathbf{r}_{j,1} \\ \mathbf{0}_{2,j} & \mathbf{r}_{2,1} \end{pmatrix} \tag{2.95}$$

where $\mathbf{r}_{j,1}$ and $\mathbf{r}_{2,1}$ are new vectors. Note that

$$\mathbf{r}_{2,1} = \begin{pmatrix} r \\ h \end{pmatrix} . \tag{2.96}$$

The task of the next rotation will then be the elimination of the term h, in order to propagate the upper diagonal structure of $\mathbf{R}^{(j)}$. To achieve this, $\mathbf{F}^{j+1}$ should be defined as:

$$\cos(\theta_{j+1}) = \frac{r}{\sqrt{r^2+h^2}} , \tag{2.97}$$

$$\sin(\theta_{j+1}) = \frac{-h}{\sqrt{r^2+h^2}} . \tag{2.98}$$

[4]In this context, $\mathbf{H}^{(j)} \in \mathbb{C}^{j+1\times j}$, and should be understood as a reduction of the original $\mathbf{H}_{k+1,k}$, that has its characteristic structure, but where all rows and columns whose index is higher that j are omitted.

[5]The columns add rows of $\mathbf{H}^{(j)}_{(m+1,m)}$ are not affected by the rotation $\mathbf{F}^{(j)}$, and are thus omitted from $\mathbf{R}^{(j)}$.

Note that the rotations $\mathbf{F}^{(j)}$ must also be applied to $\rho\mathbf{e}_1$. At the end of this process, the transformation

$$\mathbf{Q}^{(k)}\mathbf{H}^{(k)}_{k+1,k} = \mathbf{R}^{(k)} \tag{2.99}$$

is obtained. It is useful to remember that $\mathbf{Q}^{(k)} \in \mathbb{C}^{(k+1),(k+1)}$ and $\mathbf{R}^{(k)} \in \mathbb{C}^{k+1,k}$. With the help of this transformation, it is possible to write the equation 2.91 as:

$$\min\left\|\mathbf{H}^{(k)}_{k+1,k}\mathbf{y}^{(k)} - \rho^{(k)}\mathbf{e}_1\right\| = \min\left\|\mathbf{Q}^{(k)}(\mathbf{H}^{(k)}_{k+1,k}\mathbf{y}^{(k)} - \rho\mathbf{e}_1)\right\| = \min\left\|\mathbf{R}^{(k)}\mathbf{y}^{(k)} - \mathbf{g}^{(k)}\right\| , \tag{2.100}$$

where $\mathbf{g}^{(k)} = \mathbf{Q}^{(k)}\rho\mathbf{e}_1$.

We arrive thus at Eq.(2.100). To solve the least square problem, all one needs is to remove the last row of the matrix $\mathbf{R}^{(k)}$ and the last element of $\mathbf{g}^{(k)}$. This way, a definite equation system in upper triangular form emerges. The least square solution is obtained by solving this system, which is easy, given its structure. [1].

2.6.6 Flexible Generalized Minimal Residual Method (FGMRES)

The Flexible Generalized Minimal Residual Method (FGMRES) method is an extension of GMRES that enables switching the preconditioner at every iteration step to optimize computational speed and efficiency. More specifically, a different preconditioner is used for each Arnoldi vector[34].

Compared to GMRES, FGMRES requires some additional considerations. For instance, let's look at the right-preconditioned system, namely

$$\mathbf{A}\mathbf{P}^{-1}\tilde{\mathbf{x}} = \mathbf{b} , \tag{2.101}$$

where $\tilde{\mathbf{x}} = \mathbf{P}\mathbf{x}$. To obtain the solution vector $\mathbf{x}$, one needs solve the problem $\tilde{\mathbf{x}} = \mathbf{P}\mathbf{x}$ This leads to the requirement that the operation $\mathbf{P}^{-1}\mathbf{v}$ should be easy to perform for any vector $\mathbf{v}$[34].The FGMRES saves this problem at the cost of storing a vector per iteration step. This leads FGMRES to require the same amount of memory that GMRES needs for double the number of iterations before restart [3]. A complete description of the algorithm is provided by Saad [34].

In practical implementation, flexible preconditioning is done in case the preconditioner is based on splitting methods as shown in Section 2.7.2. Namely, the relaxation parameter is adjusted automatically. On the other hand, if used with a static preconditioner, such as ILU, FGMRES is identical to right-preconditioned GMRES. Overall, GMRES and FGMRES behave fairly similarly, in practice [6]. It was deemed thus more less interesting than the simpler GMRES in this work.

2.6.7 Summary: Iterative solvers and their main characteristics

Multiple iteratiive solvers were discussed thus far in this work. Here, we provide a summary to show the main advantages and disadvantages of iterative solvers, compared to one another. A direct solver was added for comparison. This summary is presented in Table 2.1. Note that the statements in this summary are not universally valid. For example, iterative solvers can

Solver	Memoy utilization	Constraints to the system matrix **A**	speed
CG	Low	Reliable only if **A** is positive definite	Fast
BiCGStab	High	Converges for most problems	Slow
GMRES	High, but adjustable	Converges for most problems	Slow
FGMRES	High, but adjustable	Converges for most problems	Slow
MUMPS	Highest	Most reliable	Fastest

Table 2.1: A summary for the main advantages and disadvantages for each iterative solver mentioned in this work thus far. A parallel direct solver, namely MUMPS, is also included for comparison

perform faster than direct solvers if the problem is sufficiently well-conditioned, although this is rarely the case. From this table, we conclude that it is best to use CG whenever possible. Whenever CG is not a option, GMRES is to be used instead, due to its reliability, smooth convergence and adjustable memory utilization.

2.7 Preconditioners

Preconditioners are an important tool for minimizing the memory and time requirements of iterative solvers. The basic idea is multiplying the Eq. (2.34) with a matrix **P**.From the left or from the right-so called left or right preconditioning, respectively.

$$\text{Left preconditioning:} \qquad \mathbf{PAx} = \mathbf{Pb} \tag{2.102}$$

$$\text{Right preconditioning:} \qquad \mathbf{AP}^{-1}\mathbf{Px} = \mathbf{b} \tag{2.103}$$

This is done to obtain a new system that is on the one hand equivalent to the original, but in the other has a lower condition number. In the following, the concept of the condition number will be discussed. Additionally, a number of preconditioners that proved noteworthy in this study will be briefly discussed.

Definition (Condition number)

The condition number is defined as the ratio between the largest and smallest singular values of a matrix. It strongly influences the convergence rate and accuracy of iterative solvers. The higher the condition number, the more iterations are needed and the more precision is lost. If the condition number is too high, then otherwise stable algorithms that are susceptible to round-off error might not converge at all. on the other hand, the lowest (theoretically) possible condition number is 1. In such a situation, the solver can theoretically find a solution without introducing error of its own, and in the case of several solvers–such as CG–might converge in as little as one step.[35]

2.7.1 Incomplete LU factorization

The LU factorization decomposes the Matrix **A** to an lower-triangular matrix **L** and an upper-triangular matrix **U** multiplicatively.

$$\mathbf{A} = \mathbf{LU} \tag{2.104}$$

There are several ways to perform this factoring [see [36]] inserting this into Eq. (2.34) yields

$$\mathbf{LUx} = \mathbf{b} \tag{2.105}$$

As a direct solver, this method is implemented by first solving the equation

$$\mathbf{Ly} = \mathbf{b} \tag{2.106}$$

which is an easy task since **L** is upper triangular, and then solving the equation

$$\mathbf{Ux} = \mathbf{y} \tag{2.107}$$

which is also easy to solve.

The LU factoring can, in principle, be used as a preconditioner. In such a case, $\mathbf{P} = \frac{1}{\omega}\mathbf{LU}$, where ω is a relaxation factor. This, however, would not be efficient, since the factoring process in itself is rather expensive. This is coupled with the fact that a complete LU decomposition generally leads to fill-in. The incomplete LU factorization (ILU) solves this by one of two possibilities [see [37]]:

- only considering elements of A that are not too small. This can be implemented in many ways. The COMSOL implementation is as follows: during the elimintion process, elements of **A** are neglected if they are lower than the euclidean norm of their corresponding column multiplied with a certain parameter called “drop Tolerance” [6] (see Appendix B for a way how this can be done).

- opting to calculate entries in **L** and **U** from A until a maximal number of entries have been calculated, at which point the algorithm is interrupted. In this case, only the entries of **L** and **U** with the highest value are calculated. The maximal number of entries to be calculated is determined by a parameter which the COMSOL software package refers to as “Fill ratio” [6].

As for the calculating the decomposition, this can be done using multiple algorithms [see 36]. In Appendix B, an implementation example is given.

2.7.2 Classical splitting methods as preconditioners

The classical iterative solvers described in Section 2.6.1 can be used as preconditioners. This applies to the Jacobi method (i.e. diagonal scaling) described in Eq. (2.42), Gauss-Seidel (Eq. (2.43)), as well as SOR, SORU , and SSOR methods described in Eq. (2.44), Eq. (2.45) and Eq. (2.46).This is done by using taking the initial guess and performing a set number of iterations from one of these methods to arrive at an improved approximation. This approximation is then used as the initial guess for the iterative solver. This is a cheap and simple trick that can also be used as a pre- or post-smoother (see Section 2.7.6). In the case of the Jacobi method, this is especially effective for very large systems [see [6]].

Equally noteworthy is that SSOR has the propety that **B** is symmetric if **A** is symmeric. This makes it particularly appropriate as a preconditioner for CG, since symmetry of the preconditioner is necessary for CG [6].

2.7.3 SOR- line, SOR gauge and SOR vector

SOR line

This is a method developed for special classes of problems, such as the treatment of boundary layers with highly anisotropic meshes, which aims to rearrange the matrix **A** to improve its band structure [see 38]. Though this is not necessarily relevant to the models handled in this work, this solver performed nevertheless adequately for these models.

SOR gauge

This preconditioner is typically used for problems in magnetostatics and similar problems. When treated with a finite element scheme, such problems give rise to systems where the matrix **A** is singular. However, These problems are solvable if the vector **b** is within the range of **A** [see [6], page 988]. Though might not always the case for Helmholtz problem, based on the results of the present study. this preconditioner is given a brief mention here since it works well for the cases studied in this work.

SOR vector

This algorithm applies SOR iterations on the original system, but also performs a SOR iterations on a projected system described by:

$$\mathbf{T}^T\mathbf{A}\mathbf{T}\mathbf{y} = \mathbf{T}^T\mathbf{b}$$

where $\mathbf{y} = \mathbf{T}\mathbf{x}$, and **T** is a discrete Gardient operator (and is orthogonal). This method can be advantageous for Helmholtz problems [see [39],[40]].

2.7.4 The Vanka algorithm

The Vanka algorithm can be understood as an SOR scheme of sorts applied to individual blocks of the system. The blocks are formed using Lagrange multiplier (for a more in-depth discussion, see Vanka [41]). This procedure can be used as a preconditioner [6]. This method was particularly developed for Navier-Stokes equation. However, for this method to provide any real advantages (or even to converge in the first place), many parameters have to be precisely adjusted. Therefore, it is not explored in-depth in this study.

2.7.5 Symmetrically-coupled Gauss-Seidel method (SCGS)

This algorithm is largely similar to the Vanka algorithm. However, contrary to the Vanka algorithm, it builds the blocks based on the degrees of freedom contained in individual elements of the mesh. This leads to the advantageous situation where the blocks are small, and their factorization can hense be stored once during initialization phase-in a similar fashion to SOR line. Other algorithms, such as Vanka's (in some of its variations) factorize at every step.

The algorithm can be modified to use either mesh elements, lines in mesh elements, or more complex schemes as a basis for forming the blocks [[6], page 984]. Apart from its usefulness as a preconditioner, this algorithm can also be used as a stand-alone solver or pre-/ post-smoother [6].

2.7.6 Multigrid methods

Multigrid methods is a vast and promising class of preconditioners/solvers [42], [43]. One can divide this class up into two categories: algebraic and geometric multigrid methods.

Geometric multigrid method creates an auxiliary set of meshes, the number of which ranges between one and several, and–whenever possible–use conducts its calculation on the coarsest levels. On the other hand, algebraic multigrid doesn't generate any actual extra meshes, but simulates this process by projecting the matrices obtained from the discretization on smaller subspaces to obtain what can be understood as virtual grid levels [6]. Since algebraic multigrid has stronger restrictions in the version of COMSOL used in this work, the discussion shall henceforth be restricted to geometric multigrid[6].

Multigrid methods provide expansive design space, since it is possible to freely adjust its components. As an example, the auxiliary meshes can be generated either independently from the fine mesh or can be derived from it using a coarsening algorithm. One may also chose to use different shape functions for the coarser meshes. Moreover, for each solver, a pre- and post-smoother may be chosen virtually at will. However, in order to obtain the best result, careful design of all solver components is recommended, sometimes even necessary. In general in geometric multigrid solvers/preconditioners are fast and memory-efficient for elliptic and parabolic equations. [6] [44]. In addition, according to Arnone *et al.* [45] and Hackbusch and Trottenberg [46], Multigrid methods can speed up the convergence of problems governed by the Navier-Stokes equations. The main advantage of Multigrid methods is that their memory consumption scales linearly with the the problem size [47].

The idea behind multigrid methods is that numerical error is smoothly distributed along the grid. A hierarchy of mesh levels is produced, and robust solver is used on the lowest grid level. Since the numerical resolution in insufficient, the direct solver produces error with a different wave number, which obscures the physical solution. High wave number error is filtered out by performing some smoothing operations.

[6]The implementation in the COMSOL 4.4, which is the version used in this work, only supports scalar partial differential equations, and does not support complex-valued system matrices [[6], page 982]

Introduction to the theoretical basis of two-grid methods

Consider the grid layer of order l, where the element size h^l changes according to the following formula:

$$\Omega_{\{l\}} = \{ j h_{\{l\}} \mid j = 1, 2, \ldots, 2^{l+1} - 1 \} \, . \tag{2.108}$$

The linear equation system to be solved on the l-th grid layer is

$$\mathbf{A}_{\{l\}} \mathbf{x}_{\{l\}} = \mathbf{b}_{\{l\}} \, . \tag{2.109}$$

Let's first consider a simple case where there is only two grids: a coarse grid and a fine grid. Such a method is called a two-grid method. Essentially, such a method consists of approximating the (smooth) long wavelength part of the vector $\mathbf{x}_{\{l\}}$ on the coarse grid. As for the short wavelength part, it is smoothed out by some iterations of a simple iterative method on the fine grid, such as the relaxation methods presented in Section 2.6.1. Inserting the formula for a general relaxation scheme (see Eq. (2.39)) delivers

$$\mathbf{x}_{\{l\}}^{(i+1)} = \mathbf{M}(\omega) \mathbf{x}_{\{l\}}^{(i)} + \mathbf{N}(\omega) \mathbf{b} \quad \text{for} \quad i = 0, 1, 2, \ldots \nu_1 \, . \tag{2.110}$$

Next, the defect $\mathbf{d}_{\{l\}} = \mathbf{A}_{\{l\}} \mathbf{x}_{\{l\}} - \mathbf{b}$ of the fine mesh is computed. A so-called restriction operation is then performed by projecting the this defect onto the coarser mesh $\Omega_{\{l-1\}}$.

$$\mathbf{d}_{\{l-1\}} = \mathbf{R}_{\{l-1\}}^{\{l\}} \mathbf{d}_{\{l\}} \tag{2.111}$$

The error on the coarser grid is calculated by solving the system

$$\mathbf{A}_{\{l-1\}} \mathbf{e}_{\{l-1\}} = \mathbf{d}_{\{l-1\}} . \tag{2.112}$$

This system can be solved by some solver from Sections 2.5 and 2.6. Interpolation on the finer mesh delivers the (unknown) error $\mathbf{e}_{\{l\}}$ on the fine mesh $\Omega_{\{l\}}$.

$$\mathbf{e}_{\{l\}} = \mathbf{P}_{\{l\}}^{\{l-1\}} \mathbf{e}_{\{l-1\}} . \tag{2.113}$$

Using the obtained error and the formula $\mathbf{A}_{\{l\}} \mathbf{e}_{\{l\}} = \mathbf{d}_{\{l\},new}$, it is possible to compute a new defect $\mathbf{d}_{\{l\},new}$ and a new solution vector $\mathbf{x}_{\{l\},new}$

$$\mathbf{x}_{\{l\},new} = \mathbf{x}_{\{l\},old} + \mathbf{e}_{\{l\}} \, . \tag{2.114}$$

Similarly to Eq. (2.110), a number ν_2 of postsmoothing operations can be performed. For pre- and post-smoothing operations, that the relaxation parameter ω in this context has a different function in comparison to relaxation parameters for standalone solvers or preconditioners. It is namely a calibration parameter that should be adjusted to deliver a low-pass filter, that ideally removes all high-wavenumber components from the error [3].

Generalization to a number m of grid layers

The process explained thus far can be extended to any number of multigrid levels. The use of m grid layers leads to a potentially very efficient solver, since now, a very coarse problem

$$\mathbf{A}_{\{l-m\}}\mathbf{e}_{\{l-m\}} = \mathbf{d}_{\{l-m\}} \tag{2.115}$$

has to be solved for $\mathbf{e}_{\{l-m\}}$ instead of Eq. (2.112). A general formulation of the multigrid method is thus obtained. One iteration of the multigrid method is called a cycle. The exact structure of a cycle is determined by the parameter γ, which stands for the number of two-grid iterations in each intermediate step. If $\gamma = 1$, then the cycle is referred to as a V-cycle. If $\gamma = 2$ or 3, then it's a so-called W- or F-cycle, respectively. Most cycles used in the literature are either V- or W-cycles. However, as Schaefer [16] notes, the explicit choice of cycle does not impact convergence significantly. [3]

Some of the literature state that multigrid methods might be inefficient for indefinite problems. This is shown to hold for the Helmholtz equation [48]. Indeed, it might even diverge, as observed in this work for the case LNSE (see Section **??**). However, multigrid methods, when used as preconditioners, proved to be very effective for the present set of Helmholtz cases. For information regarding the implementation of these solvers, consult the COMSOL 4.4 reference manual[6]. [3]

3 Overview of the test setup

First, it should be said time and memory requirements of solvers fluctuate considerably under the same parameters. This, coupled with the sheer number of freely adjustable solver/preconditioner parameters, makes it difficult to systematically study the performance of every possible combination. However, the results were consistent enough such that qualitative comparisons between solvers in terms of efficiency could be made. Therefore, it was often more meaningful to use a handful of test cases to derive rough rules of thumb about the optimal parameters of a solver-preconditioner combination. As for the amount of fluctuations, some tests were conducted to get some estimates regarding their magnitude. These are discussed in Section 3.4 and Appendix C.4. Also note that all the simulations in this work were run on COMSOL multiphisics 4.4.

In the following, an overview of the used test cases is provided

3.1 Used test cases

For this work, a total of six different numerical cases were studied. Three of the cases were Helmholtz problems, whereas the other three were LNSE problems. The used test cases are listed in Table 3.1

Case	Name	Symmetrical?
Case 1	Eigenvalues of a room (3D, Helmholtz)	yes
Case 2	Eigenmodes of a generic reheat combustor (2D, Helmholtz)	yes
Case 3	Eigenmodes of a generic reheat combustor (3D, Helmholtz)	yes
Case 4	Reflection coefficients of a swirler (3D, DG-LNSE)	no
Case 5	Eigenmodes of a laminar flame (2D, DG-LNSE)	no
Case 6	Flame transfer function of a laminar flame (2D, DG-LNSE)	no

Table 3.1: List of used test cases.

Both designations for each model used. For instance, Model 1 could be referred to as such, but might also be referred to as Eigenvalues of a room (3D, Helmholtz solver). A more detailed description of the used models is presented in the following

3.2 Default settings for solvers and preconditioners

To ensure that the tests are reasonably systematic, a set of default parameters for each solver and preconditioner was chosen, based on the theoretical understanding that was presented in Section 2.

Default solver settings In the beginning, a default set of parameters was chosen for the solvers. Table 3.2 lists these.

	Solver	Default configuration
Iterative solvers	GMRES	Number of steps before restart: 50, Preconditioning: Left, Maximum number of iterations: 100000. relative error: 10^{-6}
	FGMRES	Number of steps before restart: 50, Preconditioning: Left, Maximum number of iterations: 100000. relative error: 10^{-6}
	BiCGStab	Preconditioning: Left, Maximum number of iterations: 100000. relative error: 10^{-6}
	CG	Preconditioning: Left, Maximum number of iterations: 100000. relative error: 10^{-6}
Direct solvers	MUMPS	Memory allocation factor: 1.2 Preordering algorithm: Automatic, With row pivoting, Use pivoting: on, Pivot threshhold: 0.1. relative error: 10^{-6}

Table 3.2: Default configuration for the solvers. These are exactly the same as the default settings in COMSOL 4.4. Also note that the relative error was set to 10^{-6} when investigating the scaling behavior. For the initial tests (that were done to identify which solver-preconditioner combinations were suited for which case), the parameters described in Appendix C.2 were used

Default preconditioner settings: At the start of the tests, a default set of parameters was chosen for the preconditioners. This set is based on the default settings of the involved preconditioners in COMSOL 4.4, with modifications small modifications. These preconditioner settings will be referred to as “default” from now on. Table 3.3 lists the default settings of the preconditioners that were studied in a detailed manner. This set of preconditioners is referred to as the first set. Some other preconditioners were also studied as well, but much less thoroughly. Among these are SOR line, SOR Guage and SOR vector and others. Some have were designed for specific cases and proved to be hard to tune without a detailed knowledge of the underlying physics, such as the Vanka preconditioner. Others, such as Domain Decomposition, were fairly complex, while showing little promise in the way of potential advantages. The default settings for this second set of preconditioners are included in the Appendix C.1.

3.3 Search for suitable iterative solvers

For every test case, all available solver-preconditioner combinations were testes. This was done to establish which solvers with which preconditioners are convergent for a given case. If a solver finished successfully, its solution time and requirements in physical memory.

At the end of this process, the list of valid solvers was narrowed down to one choice iterative solver for a given model. Efforts were also made to narrow down the total list to as few solvers as possible for the sake of simplicity. In the end, the list is effectively narrowed down to two solvers. In the next step, scaling tests were conducted on each solver, as described in Section 3.4.

3.4 Scaling tests for chosen solvers

For each test case, the number of degrees of freedom (number of DOF) was gradually scaled upwards. The chosen iterative solver was tested at each scale, with time and memory requirements being documented. Additionally, a direct solver was also tested to serve as a benchmark. This was done to obtain a comparison between direct and iterative solvers in their scaling behavior.

As stated before, two solvers were chosen in the end. GMRES with SOR was chosen for case 4, whereas CG with Multigrid was chosen for cases 1, 2 and 3. But since GMRES with SOR is actually also convergent for cases 1, 2 and 3, some scaling tests were also conducted, to compare the scaling behavior of these two iterative solvers for Helmholtz problems. As another possible alternative, some tests were also made to compare the scaling behavior of CG with SOR to MUMPS. This is discussed further in Section 4.2

Error estimation

Due to the the fact that the scaling tests were often time-consuming (taking upwards of several hours per run), only a handful of runs per case could be conducted. To obtain a rough estimate of the variation in time and memory consumption under constant conditions, separate tests were conducted.

The idea behind these these tests is to take the same case and run the solver at the same problem size a set number of times (a number of 5 was chosen) for the chosen iterative solver and direct solver, and then do the same all over again for a problem size several orders of magnitude higher. Attention was also made to avoid reaching problem sizes where the tests

Preconditioner	Configuration:
Incomplete LU	Solver: Incomplete LU: Drop using: drop tolerance, Drop tolerance= 0.01, Respect pattern, Number of iterations: 1, Relaxation factor: 1.
	Solver: SPOOLS: Drop tolerance: 0.01, Pivot threshhold: 1, Preordering algorithm: Nested dissection.
SOR	Relaxation factor : 1, Number of iterations: 10 (✻).
Jacobi	Relaxation factor : 1, Number of iterations: 10 (✻).
Multigrid	General: Solver: Geometric multigrid, Number of iterations: 2, Multigrid cycle: V-cycle, Hierearchy generation method: Lower element order first (any), Number of multigrid levels: 1, Mesh coarsening factor: 2, Assemble on all levels; Presmoother: SOR: SOR, Number of iterations: 2, Relaxation factor: 1; Postsmoother: SOR: SORU, Number of iterations: 2 Relaxation factor: 1; Coarse solver: MUMPS with default settings.

Table 3.3: Used default configuration for the first set of preconditioners. Only parameters denoted by (✻) differ from the default parameters used by COMSOL

would be too time consuming. Since this is not central to this work, the results of these tests are presented and discussed in the Appendix C.4.

4 Numerical investigations for the Helmholtz eigenvalue problem

The Helmholtz equation

The first three cases are governed by the Helmholtz equation. This equation describes acoustics in a quiescent medium, and can be generally formulated as general formulation of this equation is as follows:

$$\frac{1}{\bar{\rho}\bar{c}^2}\frac{\partial^2 p}{\partial t^2} - d_a\frac{\partial p}{\partial t}\nabla.(-\frac{1}{p}(\nabla p - \mathbf{q}_d)) = Q_m \,, \tag{4.1}$$

where p is a local small variation in pressure from a stationary mean pressure p_0, $\bar{\rho}$ is the mean density of the medium, u is a local small variation in the velocity field over a stationary mean velocity u_0, $\bar{c}$ is the isentropic speed of sound, $\mathbf{q}_d$ is a dipole source term, Q_m is a monopole source term and d_a is the damping coefficient. For this class of problems, the speed of sound is is an important parameter, and it can be derived from density and pressure [**?**]. System matrices obtained form the Helmholtz equation are symmetric.

4.1 Test cases for the Helmholtz equation

4.1.1 Case 1: Eigenvalues of a room (3D, Helmholtz problem)

Case 1 is a room taken from the model library of COMSOL Multiphysics 4.4. It is essentially a room with dimensions 5 × 4 × 2.6 meters. Sound-hard boundary conditions are assumed at all boundaries. The model is governed by a Helmholtz equation with no source terms and no damping. The Helmholtz equation thus simplifies to

$$-\Delta p + \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = 0 \,. \tag{4.2}$$

A time harmonic solution of the form $p = \hat{p}e^{i\omega t}$ delivers the following governing Helmholtz equation:

$$\Delta\hat{p} + \frac{\omega^2}{c^2}\hat{p} = 0 \,. \tag{4.3}$$

The solver searches for the pressure eigenvalues of the room, i.e. the squared angular frequencies of the a time-harmonic pressure wave for which resonance occurs [see [49]].

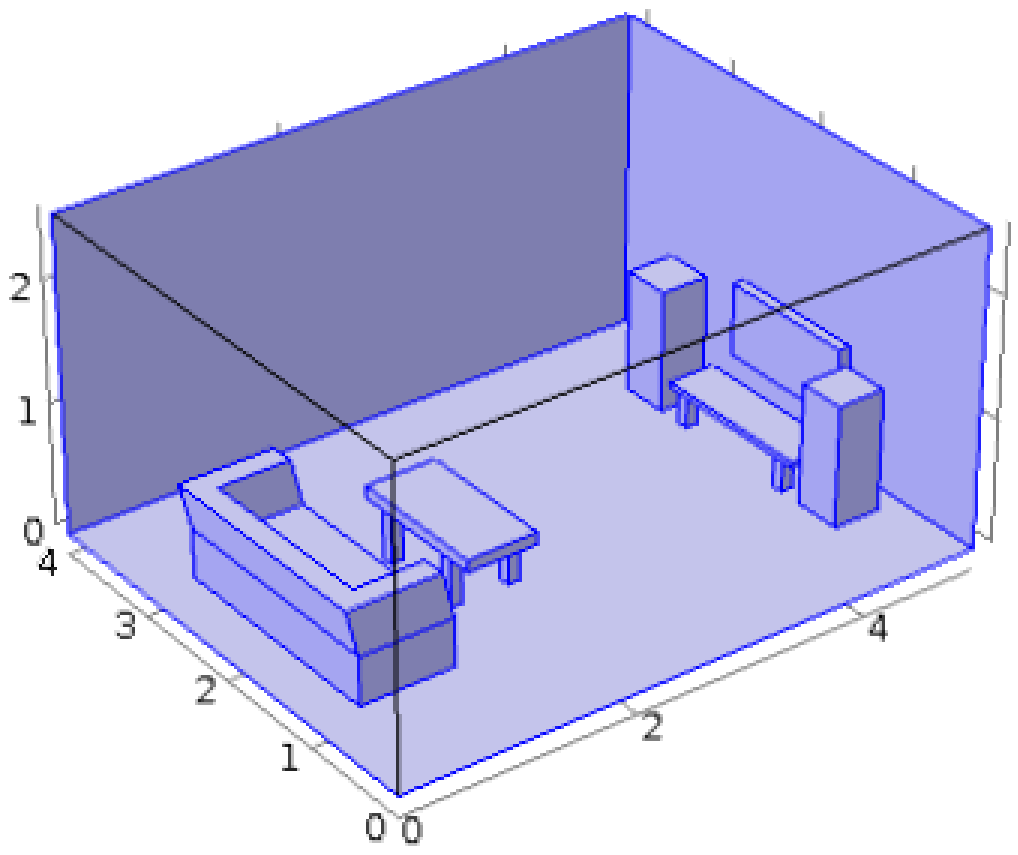


Figure 4.1: Geometry of test case 1.

.

4.1.2 Case 2: Eigenmodes of a 2D generic reheat combustor (2D, Helmholtz problem)

Case 2 handles a 2D generic reheat combustor. This setup was derived from a 3D reheat combustor studied by Zellhuber [50]. It is composed of a 2D inlet channel and an area jump. This model features passive flame by changing thermodynamical properties like pressure, density and temperature through the flame. Figure 4.3 represents the temperature mean field which indicates the flame position.

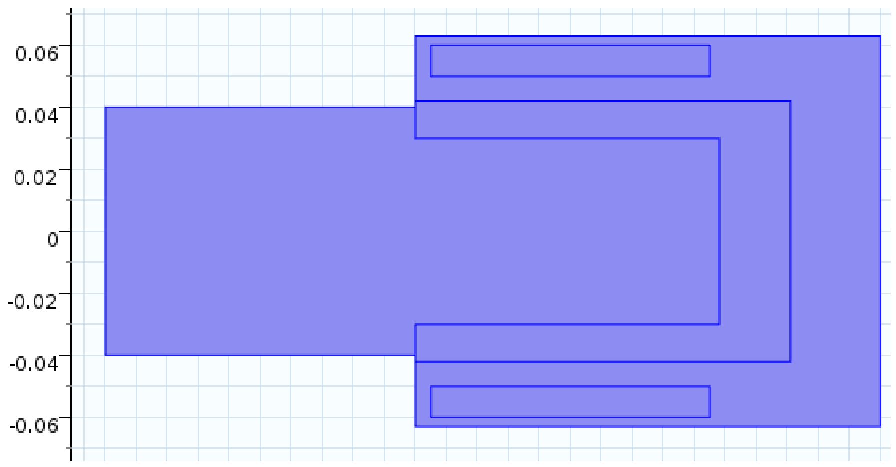


Figure 4.2: Geometry of test case 2.

.

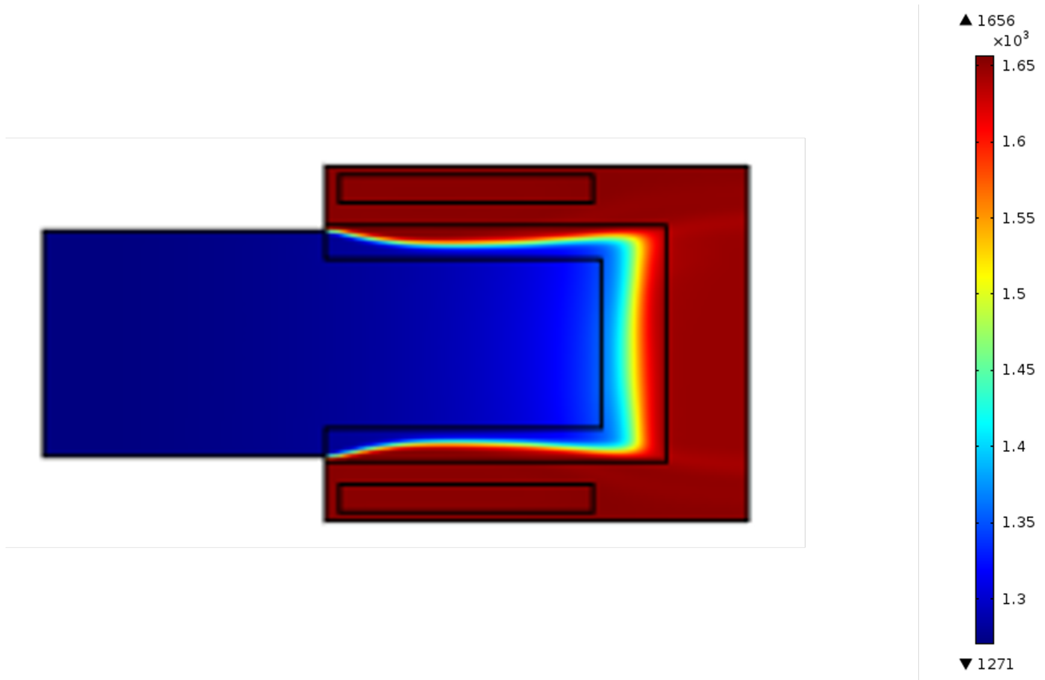


Figure 4.3: The temperature mean field (in Kelvin) that results from the flame in test case 2.

.

4.1.3 Case 3: Eigenmodes of a 3D generic reheat combustor (3D, Helmholtz problem)

Case 3 is a 3D generic reheat combustor with rectangular cross-section area. Similarly to case 2, it is also a Helmholtz case with passive flame. One exception is that case 3 has a square profile. The geometry is shown in Figure 4.4. Figure 4.5 represents the temperature mean field which indicates the flame position.

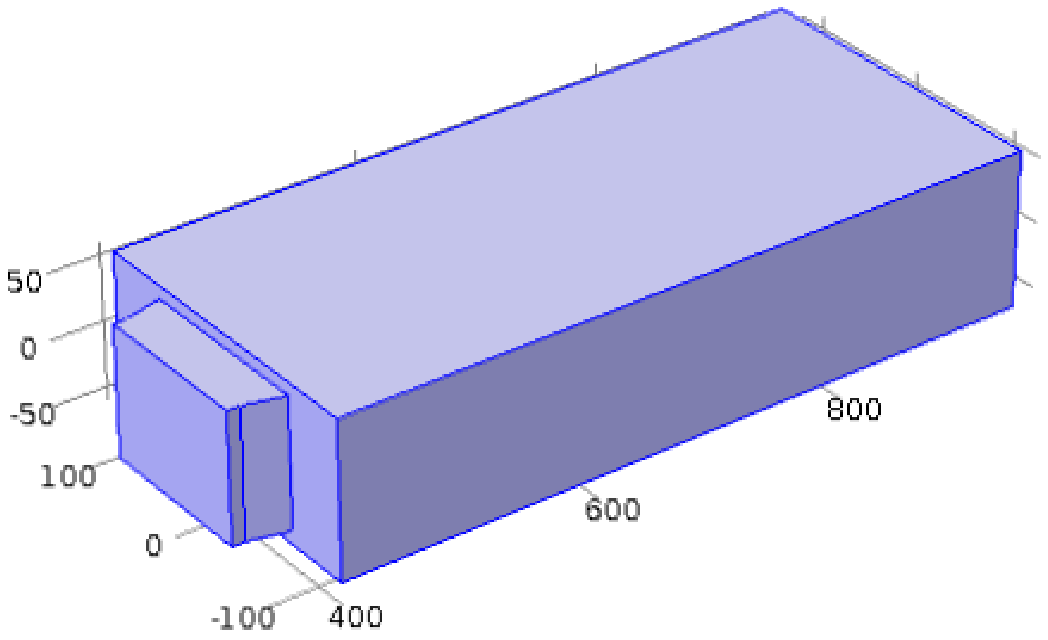


Figure 4.4: Geometry of test case 3.

.

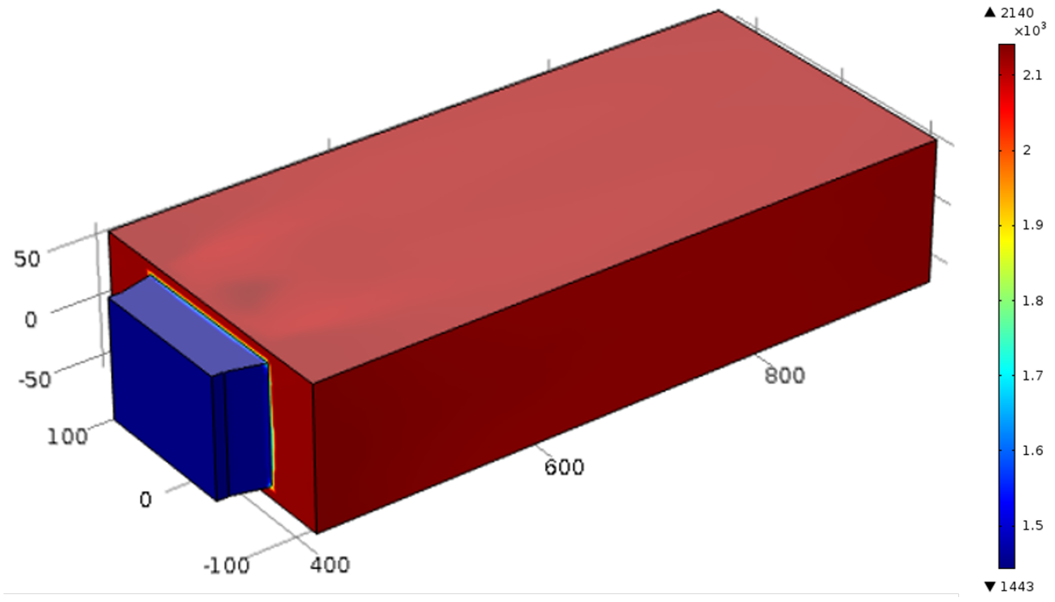


Figure 4.5: The temperature mean field (in Kelvin) that results from the flame in test case 3.

.

4.2 Results and discussion

For the Helmholtz test cases, the results from the initial investigation of all solver-preconditioner combinations, as well as the scaling tests, are presented and discussed in this section. The results from the initial investigation are summarized in tables that are included in Appendix C.3, whereas the results from the scaling tests are plotted in Figures 4.8, 4.9 and 4.10. The plot points in the memory consumption vs problem size curves were best fitted using a linear trendline. This is despite the fact that some power law was to be expected, theoretically. For these tests, however, a power law provided a significantly worse fit. Table 4.2 provides a list of the obtained correlations, as fitted using both linear and power fit. The table additionally provides the coefficient of determination for every fit. This measure can be understood as the ratio of variation accounted for by the trendline to variation unaccounted for[52]. It ranges from 0 to 1, with 1 being the ideal case, and can be thus used to compare the quality of the linear and power fits. The most significant issue with a power fit is that it tends to be significantly far off for higher problem sizes. This can be verified by using the fitted relations provided in Table 4.2.

Throughout the discussion, an upper limit on RAM of 14 GB is assumed, although the actual limit of the hardware considered in this work is 16 GB. This latter limit is considered a hard upper limit, since the operating system itself needs about 2 GB of RAM to run smoothly.

As for time requirements of solvers as a function of problem size, these are plotted alongside the memory plots in Figures 4.8, 4.9 and 4.10. The relations are emphasized by power trendlines. The numerical expressions is not of interest, since only a rough idea about the time requirements is needed. They are therefore not presented here.

This class of problems (Helmholtz eigenvalue problems), which covers the models 1, 2 and 3 in table 3.1, is notable for admitting most iterative solver-preconditioner combinations. Since the resulting matrices are symmetrical and (likely to be) positive definite, it is possible to use CG here. This is an advantage, since CG is known to be one of the most efficient solvers available.

From the start, an interesting anomaly was observed. Namely, the eigenvalue solver implemented in COMSOL multiphysics apparently might become very ill-conditioned when explicitly searching for eigenvalues near zero. As a result, when explicitly searching for eigenvalues around zero for certain models, almost all iterative solvers either fail or deliver highly distorted eigenmodes (whereas direct solvers work with no apparent issues). This was especially noticed for case 1 (eigenvalues of a room). However, this can be fixed quiet easily, by searching for eigenvalues around a frequency that is slightly above zero (such as, say, 20 Hz), when low frequency eigenmodes are of interest.

Once the above-mentioned issue is taken into consideration, it is observed that most iterative solvers converge robustly. A list of all tested iterative solvers and the outcome is included in Appendix C.3. Most noteworthy is the conjugate gradient method, which is known to require significantly less memory and time than other methods, independent of preconditioner. Although the difference between solver performance is often not visible in the tables at Appendix C.3, this is probably due to the relatively small problem size. However, we know from the literature that CG is the most memory-efficient of the present iterative solvers, and should be used when possible. The performance of CG for these problems would thus be examined in detail. As for the preconditioner, SOR and Multigrid performed similarly, and better than most other preconditioners. The multigrid preconditioner offers much more room for further improvement than SOR, it is shown by the literature to be very efficient in terms of memory and time. On the other hand, the SOR preconditioner is a tried-and-true approach that should not be overlooked. It was decided to first examine the scaling behavior of CG with multigrid. This was done for the test cases 1, 2 and 3. In addition, another round of tests was preformed to examine the scaling behavior of CG with SOR, in case any unexpected flaws in multigrid preconditioning are discovered at a later point. These tests were conducted on test cases 2 and 3.

CG (as implemented in COMSOL 4.4) offers no parameters to modify, except for the maximum number of iterations. This parameter doesn't affect convergence in any way, but the process is interrupted if this number of iterations is reached. So it should be set to a high value if the problem at hand is expected to require a high number of iterations.

In the following, the optimal parameters for the Multigrid preconditioner and the SOR preconditioner are discussed

4.2.1 Optimal parameters for the multigrid preconditioner

The default implementation of MG in COMSOL 4.4 was used, since they were found to be effective. Some brief attempts at increasing the mesh coarsening factor were made, but there were no immediately noticeable improvements in memory savings over the default value of 2. However, a high coarsening factor for very small problems was observed to cause the solver to fail, as the process of generating the coarse mesh breaks down. Moreover, as discussed in the next paragraph, it was revealed later that Multigrid with the default settings is very efficient as it is. Since these attempts where made at a relatively small n, it is possible that actual differences can be observed at higher n (around $n = 10^6$). As discussed in Section 2.7.6, the convergence behavior of Multigrid is largely insensitive to cycle type, so it makes sense to just use the simpler V-cycle. Figure 4.6 is a basic representation of the default structure of the Multigrid solver. This structure was used in the scaling tests.

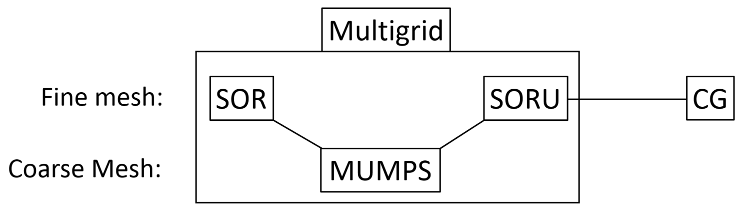


Figure 4.6: Default design of the multigrid preconditioner

.

Further investigation went into finding a way to use an iterative solver as the coarse solver instead of MUMPS. This was initially unsuccessful, since using an iterative solver as the coarse solver made caused the solver-preconditioner combination to stagnate. The reasons for this are unknown. In a later stage, it was discovered that changing the post-smoother to an iterative solver (namely non-preconditioned CG) remedied this problem. Figure 4.7 is a representation of the modified solver structure. Some preliminary tests show this setting to converge reliably. Based on these tests, This setting is expected to be moderately slower than the default solver structure but also less memory consuming. Nevertheless, no systematic scaling tests were done to confirm this prediction.

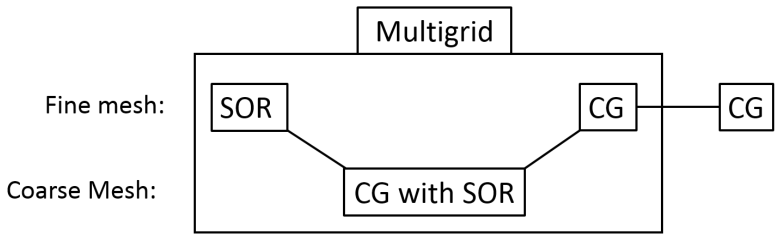


Figure 4.7: Modified design of the multigrid preconditioner

.

4.2.2 Optimal parameters for the SOR preconditioner

For the SOR preconditioner, There are three variants to choose from: SOR, SORU and SSOR (see Section 2.7.2). The SSOR version is chosen, since the latter is theoretically slightly more robust than standard SOR. There are two additional parameters to adjust: the relaxation parameters and the number of SOR iterations. Since the SOR preconditioner is stable for this case, a high relaxation number could be used. However, the more conservative choice of $\omega_{sor} = 1$ is made. A choice of 0.9 is slightly safer, but a value lower than around 0.7 offers no known advantages.

As for the ideal number of iterations of SOR, it depends significantly on the relaxation parameter. The lower the relaxation parameter, the more iterations are needed. As discussed in Section 5.2.1, SOR begins to stagnate after a few steps. For CG, better preconditioning doesn't reduce the memory utilization, and only serves to guarantee convergence and speed it up. In fact, it is more meaningful to use as little as 2 iterations of SOR, despite this leading to more CG iterations. Overall, using 2 SOR iterations was shown to save time while causing no robustness issues.

4.2.3 Scaling tests: Comparison between MUMPS and CG with Multigrid

Scaling tests on CG with multigrid on these cases revealed that this combination is very effective for 3D cases. Indeed, as can be seen in Figures 4.8 and 4.10 (right), CG with Multigrid manages to reach up a full order of magnitude in problem size compared to direct solvers. Also, for sparse problems of higher size (starting from around $n = 10^5$), CG with Multigrid is, on occasion, even slightly faster than MUMPS, as can be seen in Figures 4.8 and 4.10 (left).

However, the results were significantly less impressive for the 2D Helmholtz case. Although the memory requirements of CG scaled less strongly than MUMPS, the difference is small. Using the linear fits from Table 4.2, it can be estimated that with around 14 GB of RAM, CG with Multigrid can solve problems 2.00 times the size of the problems solvable by MUMPS. This is dwarfed in comparison to the memory savings in the 3D cases (case 1 and 3). In fact, when limited to 14 GB of RAM and using the same method, CG with Multigrid can handle sizes that are roughly 10 times larger than problems solvable by MUMPS. This disparity between performance in 2D and 3D cases is consistent with the findings in Section 2.6.2.

Case	solver	Linear fit		power fit	
		resulting fit	coefficient of determination	resulting fit	coefficient of determination
case1	CG with Multigrid	$M = 2 * 10^{-6} * n + 1.6178$	0.9459	$M = 0.0531 * n^{0.3178}$	0.8839
	MUMPS	$M = 2 * 10^{-5} * n + 0.738$	0.9981	$M = 0.0112 * n^{0.5036}$	0.9277
case2	CG with Multigrid	$M = 1 * 10^{-6} * n + 1.2652$	0.9962	$M = 0.1127 * n^{0.2523}$	0.8335
	MUMPS	$M = 2 * 10^{-6} * n + 1.314$	0.9937	$M = 0.0894 * n^{0.2814}$	0.8468
case3	CG with Multigrid	$M = 2 * 10^{-6} * n + 6.6746$	0.8708	$M = 1.9752 * n^{0.1163}$	0.5798
	MUMPS	$M = 2 * 10^{-5} * n + 6.5339$	0.9871	$M = 1.5292 * n^{0.1588}$	0.7792

Table 4.1: Fitted relations for memory (M) in Gigabytes of RAM, as a function of number of DoF (n) for CG with Multigrid vs MUMPS. Notice how the coefficient of determination of the linear model (left) is clearly closer to 1 than the power model (right).

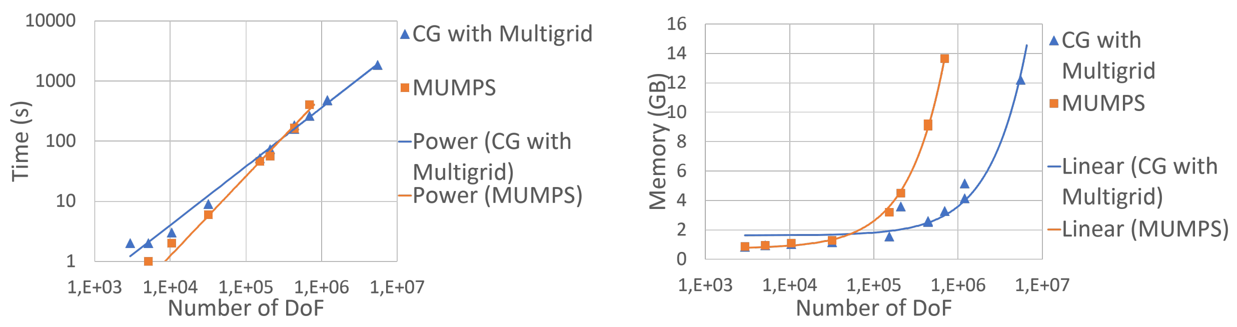


Figure 4.8: Scaling behavior of CG with Multigrid and MUMPS for **case 1**, in terms of time in seconds (left) and memory consumption in GB (right). The linear fit in the memory curve is clearly more accurate.

.

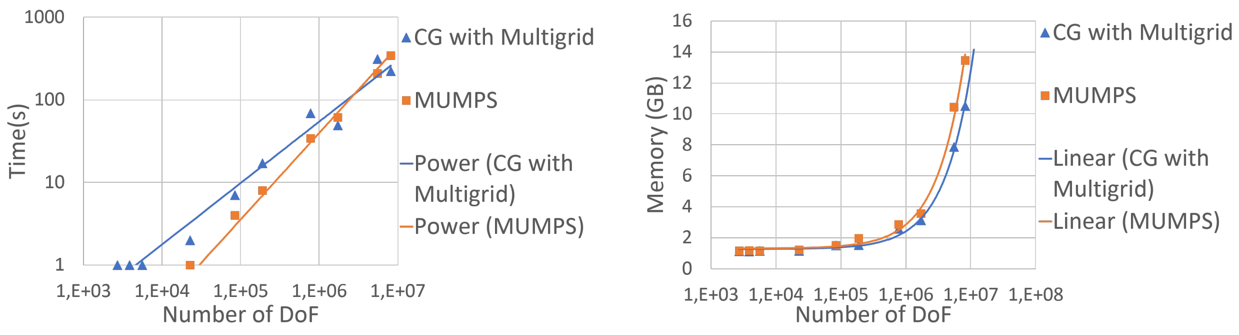


Figure 4.9: Scaling behavior of CG with Multigrid and MUMPS for **case 2**, in terms of time in seconds (left) and memory consumption in GB (right). The linear fit in the memory curve is clearly more accurate.

.

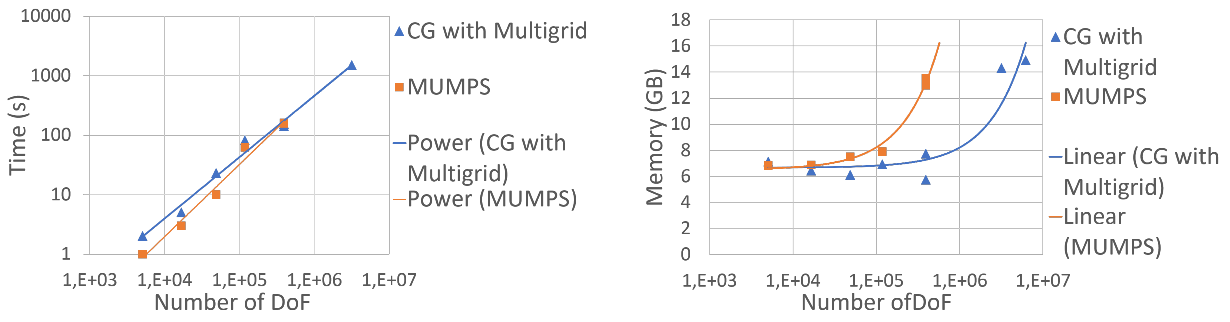


Figure 4.10: Scaling behavior of CG with Multigrid and MUMPS for **case 3**, in terms of time in seconds (left) and memory consumption in GB (right). The linear fit in the memory curve is clearly more accurate.

.

Comparaison between CG with Multigrid and GMRES with SOR

Although it was assumed that memory and time requirements of GMRES with SOR would scale faster than those of CG with Multigrid, scaling tests to compare the two were nonetheless conducted, starting from the simple case "eigenvalues of a room". As expected, the memory requirements of GMRES increased faster than those of CG with increasing problem size, albeit slightly. But a much bigger difference was observed in the time requirements: whereas CG was remarkably fast at high problem sizes, GMRES experiences a steep slowdown. The results of these tests can be seen in Figure 4.11.

Based on these results, it was deemed unnecessary to conduct the same tests for the two other Helmholz cases (case 2 and 3). CG with Multigrid clearly outclasses GMRES with SOR in the two important aspects for these cases.

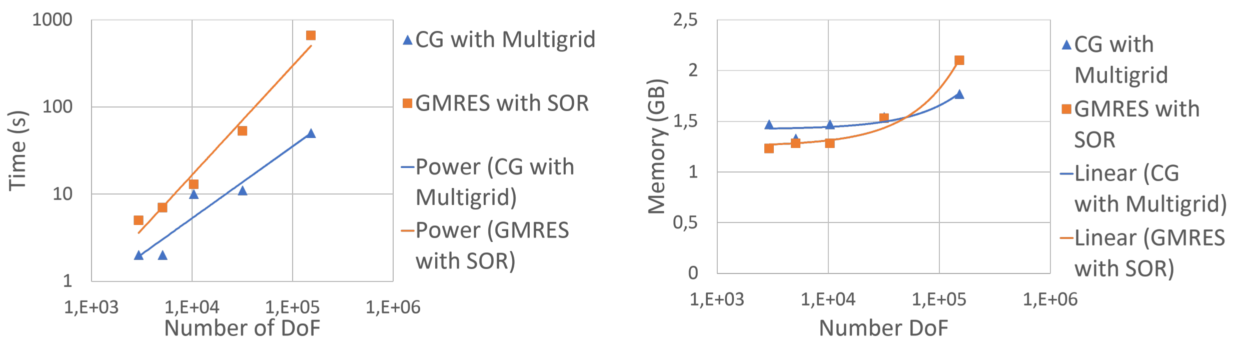


Figure 4.11: Scaling behavior of CG with Multigrid and GMRES with SOR for **case 1**, in terms of time in seconds (left) and memory consumption in GB (right). The tests were concluded prematurely, since it had become clear that CG with Multigrid is superior.

.

4.2.4 Scaling tests: Comparison between MUMPS and CG with SOR

Case	solver	Linear fit		power fit	
		resulting fit	coefficient of determination	resulting fit	coefficient of determination
case2	CG with SOR	$M = 1 * 10^{-6} * n + 0.9636$	0.9991	$M = 0.0778 * n^{0.2791}$	0.8425
	MUMPS	$M = 2 * 10^{-6} * n + 1.1067$	0.9923	$M = 0.0761 * n^{0.2875}$	0.8301
case3	CG with SOR	$M = 2 * 10^{-6} * n + 6.4135$	0.9981	$M = 2.1106 * n^{0.1109}$	0.8.239
	MUMPS	$M = 2 * 10^{-5} * n + 6.12$	0.9999	$M = 1.3775 * n^{0.158}$	0.8994

Table 4.2: Fitted relations for memory (M) in Gigabytes of RAM, as a function of number of DoF (n) for CG with Multigrid vs MUMPS. Notice how the coefficient of determination of the linear model (left) is clearly closer to 1 than the power model (right).

Scaling tests on CG with SOR revealed the combination to achieve noteworthy memory savings when compared to MUMPS for the 3D test case. It is namely possible to solve problems around 5 times larger than those that MUMPS can solve for the same memory consumption, as can be seen in Figure 4.13 (right). However, CG with SOR is somewhere between 10 and 100 times slower than MUMPS at the same number of DoF. This can be seen in Figure 4.13 (left).

For the 2D test case, the results were less impressive. As shown in Figure 4.12 (right), CG with SOR was able to solve problems around twice as large as those that MUMPS could solve for the same memory consumption. However, this comes at the cost of CG with SOR being more than two orders of magnitude slower than MUMPS. This is shown in Figure 4.12 (left).

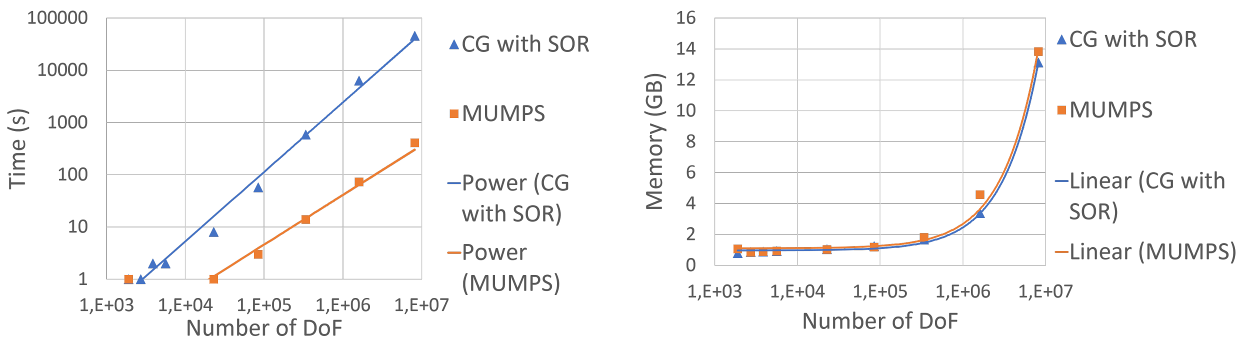


Figure 4.12: Scaling behavior of CG with SOR, compared to MUMPS, for **case 2**. This is expressed in terms of time in seconds (left) and memory consumption in GB (right).

.

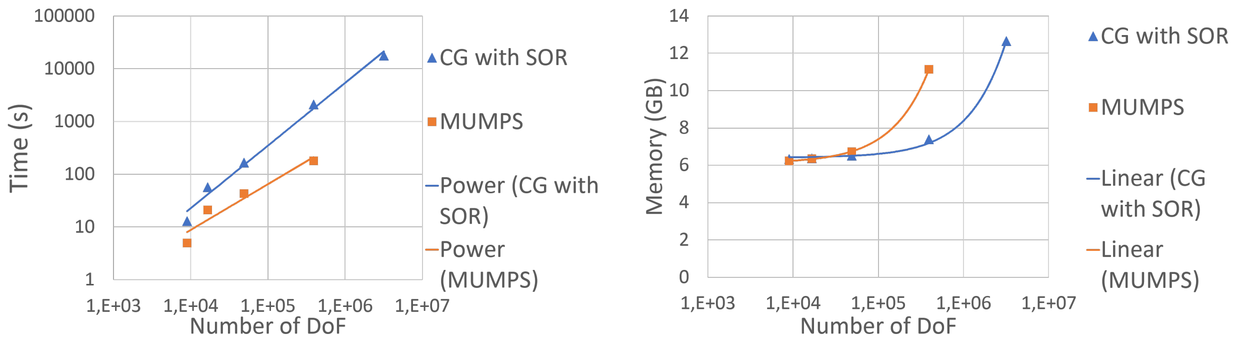


Figure 4.13: Scaling behavior of CG with SOR, compared to MUMPS, for **case 3**. This is expressed in terms of time in seconds (left) and memory consumption in GB (right).

.

4.2.5 Summary: Solver recommendation for Helmholtz eigenvalue problems

Three solvers settings were tested overall. These are summed up here, beginning from the most effective to least effective. For details regarding the memory savings and time consumption, see the discussion in the Section 4.2.

1st choice: CG with Multigrid

The default Multigrid preconditioner was used for the scaling tests. The solver configuration is shown in detail in Figure 4.14. See Section 4.2.3 for the scaling behavior of this solver configuration.

A more optimal configuration was later discovered. This configuration avoids using direct solvers as coarse solvers. It is expected to be moderately slower but more memory-efficient than CG with the default Multigrid. The solver configuration is shown in detail in Figure 4.15.

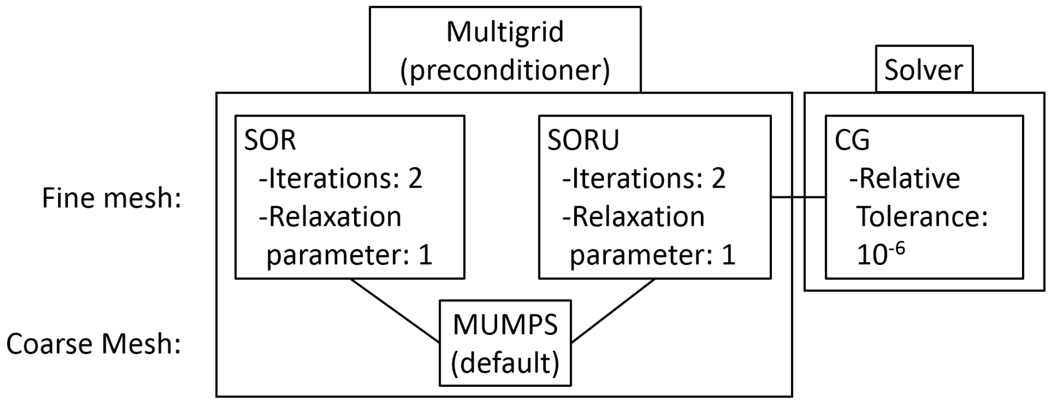

Multigrid (preconditioner)
Solver
Fine mesh:
Coarse Mesh:
SOR -Iterations: 2 -Relaxation parameter: 1
SORU -Iterations: 2 -Relaxation parameter: 1
CG -Relative Tolerance: 10^{-6}
MUMPS (default)

Figure 4.14: Configuration of CG with default multigrid preconditioner in detail

.

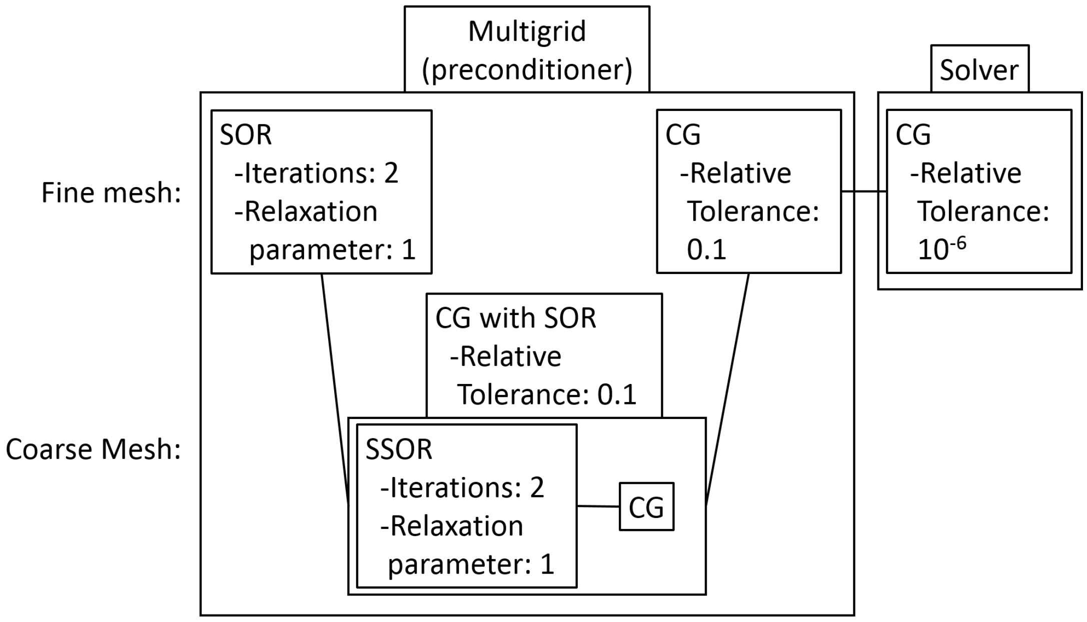

Multigrid (preconditioner)
Solver
Fine mesh:
Coarse Mesh:
SOR -Iterations: 2 -Relaxation parameter: 1
CG -Relative Tolerance: 0.1
CG -Relative Tolerance: 10^{-6}
CG with SOR -Relative Tolerance: 0.1
SSOR -Iterations: 2 -Relaxation parameter: 1
CG

Figure 4.15: Configuration of CG modified Multigrid preconditioner in detail[1]

.

2nd choice: CG with SOR

This configuration uses the tried-and-true SOR preconditioning. Figure 4.16 shows the configuration in detail. See Section 4.2.4 for the scaling behavior of this solver configuration.

[1]CG with SOR as coarse solver, as well as CG as post-smoother, can be implemented in COMSOL using the Krylov preconditioner node

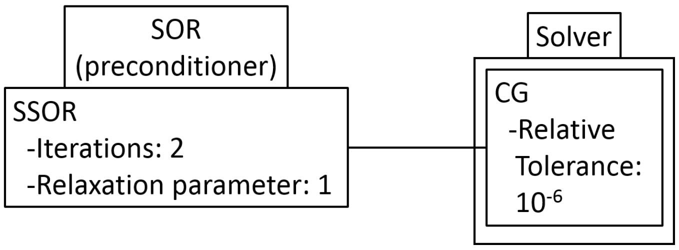


Figure 4.16: Configuration of CG with SOR in detail

.

3rd choice: GMRES with SOR

This configuration has the advantage that it can be used for a wider variety of problem classes. However, it's less optimal then the previous two for Helmholtz eigenvalue problems. Figure 4.17 shows the configuration in detail. See Section 4.2.3 for the scaling behavior of this solver configuration.

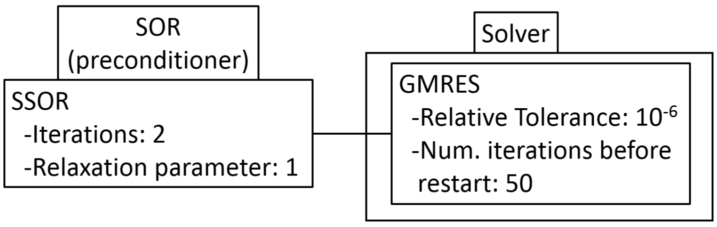


Figure 4.17: Configuration of GMRES with SOR in detail [2]

.

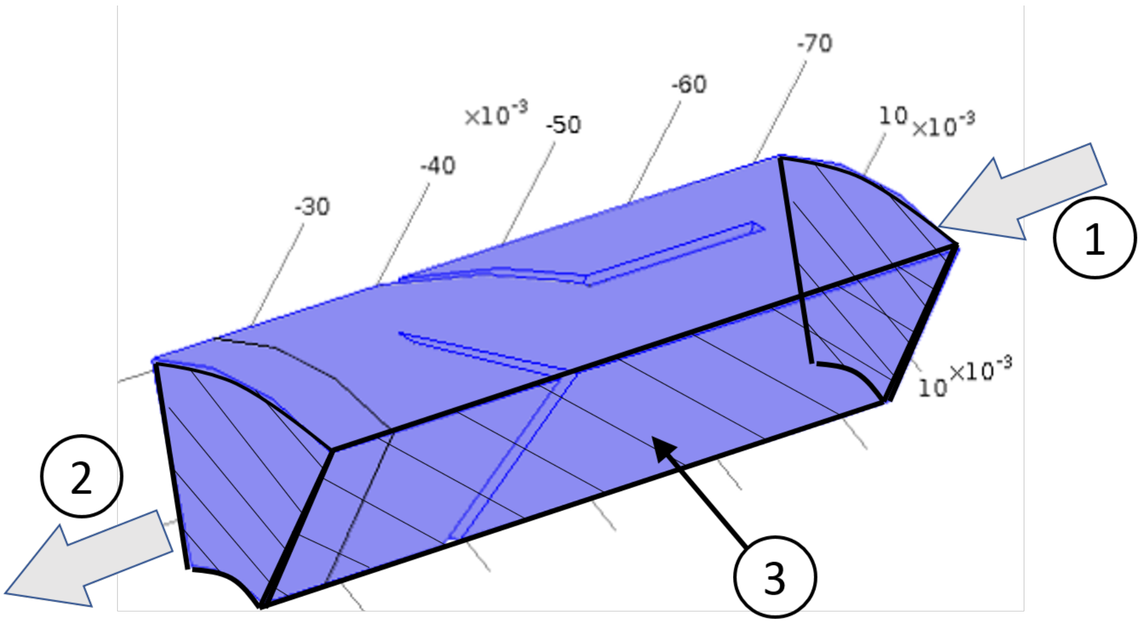
[2]5 SOR iterations were used in the scaling tests. But a number of 2 was found to be more efficient upon further investigation.

5 Numerical investigations for the linearized Navier-Stokes equations (LNSE)

5.1 Test cases for LNSE

5.1.1 Case 4: Reflection coefficients of a swirler (3D, DG-LNSE)

Case 4 studies a swirler of a BRS Burner Tay-Wo-Chong *et al.* [51]. Since the swirler features a structure that is rotationally symmetrical, only 1/6 of the cross-section is represented. The geometry can be seen in Figure 5.1. Here, we compute the reflection coefficients at the inlet. This is done over several frequencies to obtain a FRF.



Boundary conditions:

1: Inlet, non-reflecting

2: Outlet, non-reflecting

3: Periodic boundary condition (on both sides)

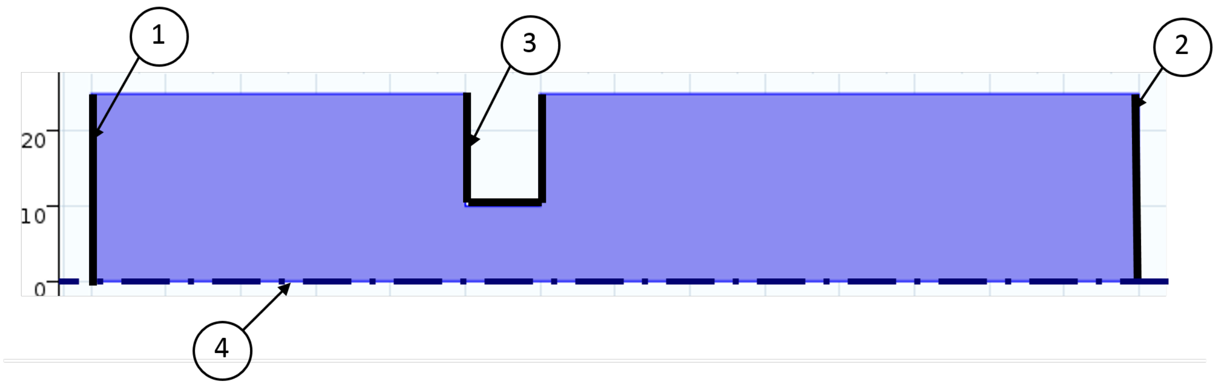
All walls: non-slip adiabatic walls

Figure 5.1: Geometry of the test case 4.

.

5.1.2 Cases 5 and 6: Eigenmodes and flame transfer function of laminar flame

Cases 5 and 6 are LNSE cases, that are discretized with on a discontinuous Galarkin (DG) method[5]. Here, we look at a 2D perfectly premixed laminar flame. The heat release rate is modeled with linearized one-step irreversible Arrhenius equation. In case 5, we a study the eigenmodes of the flame. In case 6, we compute the flame transfer function (FTF). The input is the inlet velocity, and the output is the global heat release rate. For more information about the FTF, consult for example Tay-Wo-Chong *et al.* [51]. Since the geometry is symmetric, only half of the structure is modeled. The geometry of the model is shown in Figure 5.2



Boundary conditions:
1: For FTF: velocity inlet, for eigenvalues: non-reflective boundary
2: For FTF: pressure outlet, for eigenvalues: non-reflective boundary
3: Non-slip walls with fixed temperature
4: Symmetry
Remaining boundaries: non-slip walls

Figure 5.2: Geometry of test cases 5 and 6..

.

5.2 Results and discussion

Similarly to the Helmholtz test cases, the results for the LNSE cases are presented and discussed here. This includes the the initial investigation of all solver-preconditioner combinations, as well as the scaling tests. The results from the initial investigation are summarized in tables that are included in Appendix C.3, whereas the results from the scaling tests are plotted in Figure 5.3. Table 5.1 provides a list of the obtained correlations, as fitted using both linear and power fit. The time requirements of solvers as a function of problem size, these are plotted alongside the memory plots in Figure 5.3[1].

[1]See Section 4.2 for more details

Case	solver	Linear fit		power fit	
		resulting fit	coefficient of determination	resulting fit	coefficient of determination
case4	GMRES with SOR	$M = 3E-06 * n + 4.9861$	0.9861	$M = 1.0403 * n^{0.1501}$	0.7862
	MUMPS	$M = 3E-05 * n + 4.1789$	0.9734	$M = 0.2062 * n^{0.313}$	0.8228
case5	no working iterative solver was found				
case6	no working iterative solver was found				

Table 5.1: List of obtained expressions for memory consumption as a function of problem size (M) in Gigabytes of RAM, as expressed by number of degrees of freedom (n) for the **LNSE** cases. Notice how the coefficient of determination of the linear model (left) is clearly closer to 1 than the power model (right).

5.2.1 Frequency response function of LNSE without heat source term (case 4)

This test case, where the reflection coefficients of a swirler were studied, admitted less solvers than the Helmholtz cases. However, it still offers a host of solvers to chose from. The most notable solvers are listed in Table 5.2. Since this problem is not symmetrical, CG wasn't expected to converge, and this assumption held true. The Multigrid preconditioner didn't work either, so the SOR preconditioner was used instead.

solver	preconditioner	configuration	time	Physical Memory
GMRES	SOR	type: SSOR relaxation factor: 0.9 5 iterations	00:11:01	4.63 GB
BiCGStab	SOR	type: SSOR relaxation factor: 0.9 5 iterations	00:14:38	4.36 GB

Table 5.2: Solver settings with least memory consumption and reasonable solution time for **case 4**.

Optimal solver

From this list, GMRES with SOR was chosen, due to its more regular convergence behavior. The solver converges without much delay for a number of iterations before reset of 50. Increasing this number could speed-up convergence, but would definitely increase memory consumption for larger problems. Smaller problems are insensitive to this value, as observing the convergence diagrams indicates that the solver could then converge before it had performed 50 iterations.

Concerning the SOR preconditioner, the SSOR version is preferred, since the latter is theoretically slightly more robust than standard SOR. The SOR preconditioner is stable for this case, therefore a high relaxation parameter could be advantageous. Nevertheless, a conservative choice of 0.9 or 1 is recommended, with 0.9 being the safer choice, though slightly slower. Any relaxation parameter below around 0.7 offers no known advantages, and needs to be offset by a higher number of iterations.

As for the ideal number of iterations of SOR, it depends on the relaxation parameter. The lower the relaxation parameter, the more iterations are needed. Some tests were made to obtain an estimate for the ideal number of iterations , and the results are shown in Table 5.3. One weakness of SOR that was revealed is that it begins to stagnates after a few iterations. This means that using too many steps just slows down the process and offers no benefits. In fact, it is better to use just enough iterations to guarantee that the iterative solver converges, and instead do more iterations of the solver. One exception to this is GMRES for smaller problems, namely because using less iterations of GMRES before restarting it (due to improved conditioning) saves memory[2]. The scaling tests were conducted using 5 SOR iterations, although further investigation at a later stage proved a 2 SOR iterations to lead to comparable memory savings while being significantly faster.

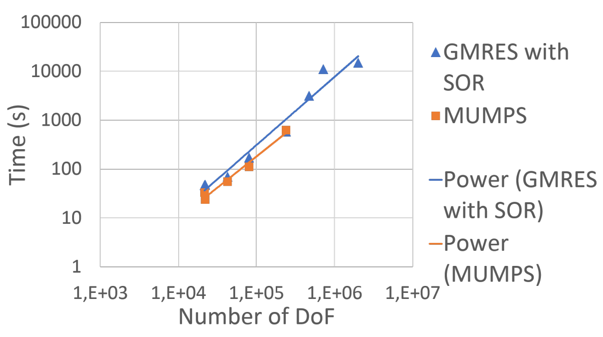
Number of SSOR iterations	Time	Memory consumption
50	00:59:58	4.23 GB
5	00:14:38	4.36 GB
2	00:21:00	5.82 GB

Table 5.3: Influence of number of SOR iterations on memory and time consumption, as shown through time and memory consumption of BiCGStab with SOR for different numbers of iteration for the SOR preconditioner. These tests were run on case 4, at $n = 238540$. The relaxation factor was set to 0.9.

Scaling tests: comparison between GMRES with SOR and MUMPS

GMRES offers significant memory savings, as can be seen in Figure 5.3 (right). This, however, comes at the cost of very rapidly increasing costs in terms of time, as is shown in Figure 5.3 (left). Using the linear fit from Table 5.1, it can be estimated that with around 14 GB of RAM, GMRES with SOR can solve problems that are around 10 times larger than those solvable by CG with Multigrid for the same memory.

[2]This is particularly the case for eigenvalue problems, where the eigenvalue algorithm itself restarts constantly the solver after few iterations



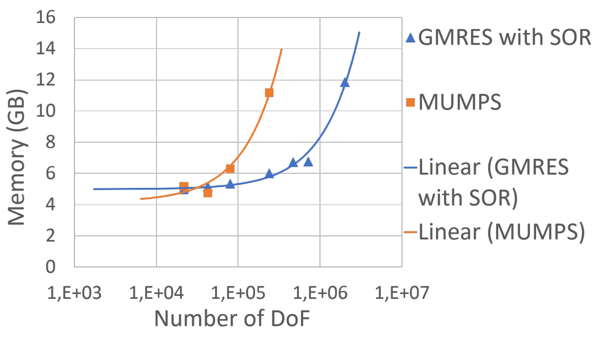


Figure 5.3: Scaling behavior of GMRES with SOR compared to MUMPS for **case 4**, in terms of time in seconds (left) and memory consumption in GB (right). The linear fit in the memory curve is clearly more accurate.

5.2.2 Eigenvalues and frequency response function of LNSE with heat source terms (case 5 and case 6)

For both these cases, all attempts to find an appropriate iterative solver have failed. This is despite the fact that direct solvers work for them with no apparent issues. Interestingly, when handling the eigenvalue problem (case 4), the solvers would indeed converge for most solver-preconditioner combinations (except for CG, which diverges immediately). But the returned eigenvalues are spurious

As for the flame transfer function case (case 5), GMRES looks most promising, but it stagnates some orders of magnitude short of convergence. BiCGStab might also hold some promise, but it's highly oscillating behavior makes an accurate assessment of it's potential difficult to make. It also occasionally suddenly diverges after seemingly steadily converging for some time. So in the end, GMRES makes more sense to explore further, since it is more reliable, overall, while providing comparable memory savings. These cases were abandoned at this point.

5.2.3 Summary: Solver recommendation for LNSE cases

GMRES with SOR

This solver configuration is effective for LNSE cases with no heat source terms. See Section 5.2.1 for the scaling behavior of this solver setting. This setting is also promising for LNSE with heat source terms, as long as the problems are well-conditioned. Figure 5.4 shows the recommended solver configuration in detail.

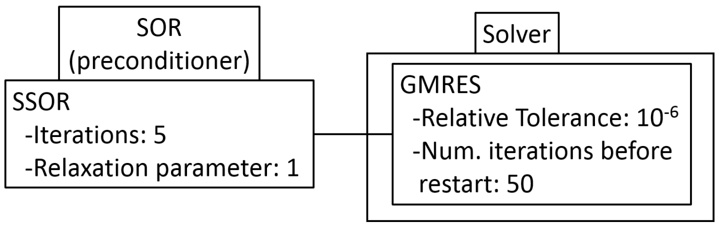


Figure 5.4: Configuration of GMRES with SOR in detail [3]

.

[3] 5 SOR iterations were used in the scaling tests. Upon further investigation, a number as low as 2 was revealed to moderately speed up convergence with no memory penalties.

6 Conclusions and outlook

Based on the results, iterative solvers can be used to save memory for Helmholtz eigenvalue problems. Three solvers were shown to achieve this. The most effective of these was CG with Multigrid. For 3D problems, CG with Multigrid was found to be capable of solving problems that are roughly 10 times larger than direct solvers can solve for the same memory. It was also slightly faster than direct solvers for large problems for both 2D and 3D cases. However, its memory savings were less significant for 2D cases. In such cases, for the same memory utilization, CG with Multigrid was capable of solving problems that were twice as large as the ones direct solvers can solve. An alternative Solver that was studied is CG with SOR. Here, the simple and reliable SOR preconditioning was tested. For 3D cases, CG with SOR proved capable of solving problems 5 times larger than direct solvers could solve for the same memory. However, This drops to a factor of 2 for 2D problems. CG with SOR was observed to be significantly slower than MUMPS. For larger problems, it is slower than direct solvers by 1 order of magnitude for 3D cases, and 2 orders of magnitude for 2D cases. Finally, GMRES with SOR was briefly studied for Helmholtz cases, but preliminary tests showed it to be less interesting than the previous two. The specifically recommended parameters for each of these solvers are summarized in Section 4.2.5.

For frequency response function problems governed by LNSE with no heat source terms, GMRES with SOR is recommended. This choice provides significant memory savings, at the cost of more time consumption. To keep the memory consumption to a minimum, it is recommended that the number of iterations before restart is kept to a minimum. 50 iterations before restart are sufficient for GMRES to converge, even at large problem sizes. A summary of recommended parameters for this solver-preconditioner combination can be found in Section 5.2.3.

For frequency response function problems governed by LNSE, when a heat source term is present, there might be no appropriate iterative solver. This can be seen in cases 4 and 5. This is due to the poor condition, possibly as a result of poor choice of boundary conditions. Nevertheless, the solver that shows most promise is GMRES with SOR. As for the solver and preconditioner settings, the same settings recommended above (for LNSE cases without source terms) are probably appropriate here, too.

Further improvements could be made to the Multigrid preconditioner for Helmholtz cases. Its default settings are indeed effective as they are, but the alternative settings shown in Figure 4.15 promises further reduction in memory consumption. On the other, as discussed in subsection 2.7.6, the convergence behavior of Multigrid is largely insensitive to cycle type, so it makes sense to use the V-cycle, as it is the simplest one. But it could be possible to improve memory savings by increasing the number of grid layers and adjusting coarsening factors for higher numbers of DoF. This also seems to be a promising topic for future exploration.

Another area which warrants further study is LNSE cases with heat source terms. Here, further work could be done to find which boundary conditions are responsible for the poor conditioning. These boundary conditions could then potentially be relaxed or changed completely. Such changes could allow this class of problems to be solved using iterative solvers.

Appendices

A Linear Algebra

This work requires a certain level of understanding of linear algebra. The concepts that were cental to the subject matter, such as eigenvalue problems, were explained in the main body. In this appendix, the more basic concepts of linear algebra that should be known to the reader are recalled. Please note that only concepts that were used in this work are present here.

A.1 Matrices and vectors

The mathematical relations in this work are formulated for complex matrices. This choice was made since it is more convinient to do so when discussing eigenvalues [54].

Definition of a matrix: A complex $n_1 \times n_2$ matrix $\mathbf{A}$ is an array of complex numbers $a_{i,j}$ such that $i = 1, 2, \ldots, n_1$ and $j = 1, 2, \ldots, n_2$.

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}. \tag{A.1}$$

This example represents the particular case of a square matrix

A.1.1 Vectors

Definition of a vector: A vector in $\mathbb{C}^n$ (or $\mathbb{C}^{n\times 1}$) is a matrix with one column and n rows.

Vector norm: In this work, the norm of a vector $\mathbf{v}$ in $\mathbb{C}^n$, $\|\mathbf{v}\|$ is defined as:

$$\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} |v_i|^2}\,. \tag{A.2}$$

This definition is often referred to as euclidean norm or L_2 norm.

Orthogonal vectors: Two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{C}^n$ are orthogonal if $\mathbf{u}^T\mathbf{v} = \mathbf{v}^T\mathbf{u} = 0$.

Orthonormal vectors: Two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{C}^n$ are orthonormal if they are orthogonal and are unit vectors, i.e. $\|\mathbf{v}\| = \|\mathbf{u}\| = 1$.

A.2 Matrix operations

Several matrix operations were used in this work. These operations are:

- Addition of two matrices $\mathbf{A} = \mathbf{B} + \mathbf{C}$. This operation is defined solely for matrices of the same dimentions. It is defined as:

$$a_{i,j} = b_{i,j} + c_{i,j} \tag{A.3}$$

- Multiplication of a scalar α and a matrix $\alpha\mathbf{A}$

$$\{\alpha\mathbf{A}\}_{i,j} = \alpha a_{i,j} \tag{A.4}$$

- Multiplication of two matrices $\mathbf{A} = \mathbf{BC}$. This operation is defined only for the case where $\mathbf{B} \in \mathbb{C}^{n_1 \times n_2}$ and $\mathbf{C} \in \mathbb{C}^{n_3 \times n_4}$ such that $n_3 = n_4 = m$. Matrix product is defined as

$$a_{i,j} = \sum_{l=1}^{m} b_{i,l} c_{l,j} \tag{A.5}$$

- Taking the transpose of a matrix $\mathbf{A} \to \mathbf{A}^T$. This operation is defined as

$$\{\mathbf{A}^T\}_{i,j} = a_{j,i} \tag{A.6}$$

- Taking the conjugate transpose of a matrix $\mathbf{A} \to \mathbf{A}^*$. This operation is defined as:

$$\mathbf{A}^* = \overline{\mathbf{A}}^T \,. \tag{A.7}$$

  Here, $\overline{\mathbf{A}}$ is the matrix $\mathbf{A}$, with each element replaced by its complex conjugate[1].

[54]

A vector in $\mathbb{C}^3$ can be considered as a 3×1 matrix. Therefore, a scalar product between two vectors in $\mathbb{C}^3$, $\mathbf{u}$ and $\mathbf{v}$, often represented as $\mathbf{u} \cdot \mathbf{v}$ or $< \mathbf{u}, \mathbf{v} >$, can be instead represented using matrix operations as

$$\text{(scalar product)} \qquad \mathbf{u}^T\mathbf{v} \,. \tag{A.8}$$

This latter notation is the only one used in this work.

A.3 Special types of matrices

There are several types of matrices that are of interest. This either because their structure gives rise to special types of eigenvalues, or because these structures are a very common product of numerical discretization schemes.

[1]the complex conjugate of a complex number $a + ib$, where a and b are real, is $a - ib$.

- Identity matrix: $\mathbf{I}$. They are defined as $\{\mathbf{I}\}_{i,j} = 1 \quad \forall i = j$ and $\{\mathbf{I}\}_{i,j} = 0 \quad \forall i \neq j$;
- Diagonal matrix: $\mathbf{A}$ is diagonal if $a_{i,j} = 0 \quad \forall i \neq j$;
- Symmetric matrices: $\mathbf{A} = \mathbf{A}^T$,
- Hermtian matrices : $\mathbf{A} = \mathbf{A}^*$,
- Positive definite matrices: $\text{Re}\{\mathbf{x}^*\}\mathbf{A}\mathbf{x} > 0 \quad \forall \mathbf{x} \in \mathbb{C}^n$. For real matrices, this reduces to $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$.
- Diagonal matrix:$a_{i,j} = 0 \quad \forall i \neq j$.
- Upper triangular matrix: $a_{i,j} = 0 \quad \forall i > j$.
- Lower triangular matrix: $a_{i,j} = 0 \quad \forall i < j$.
- Non-square upper Hessenberg Matrix $\mathbf{H}_{m+1,m}$: $\mathbf{H}_{ij} = 0 \quad \forall i > j+1$, that is:

$$\begin{pmatrix} h_{1,1} & h_{1,2} & \dots & \dots & h_{1,m-1} & h_{1,m} \\ h_{2,1} & h_{2,2} & \dots & \dots & h_{2,m-1} & h_{2,m} \\ 0 & \ddots & & \dots & h_{3,m-1} & h_{3,m} \\ \vdots & \ddots & \ddots & & \vdots & \vdots \\ \vdots & & \ddots & \ddots & \vdots & \vdots \\ \vdots & & & \ddots & h_{m,m-1} & h_{m,m} \\ 0 & \dots & \dots & \dots & 0 & h_{m+1,m} \end{pmatrix}. \tag{A.9}$$

- Upper Hessenberg matrix $\mathbf{H}_{m,m}$:

$$\begin{pmatrix} h_{1,1} & h_{1,2} & \dots & \dots & h_{1,m-1} & h_{1,m} \\ h_{2,1} & h_{2,2} & \dots & \dots & h_{2,m-1} & h_{2,m} \\ 0 & \ddots & & \dots & h_{3,m-1} & h_{3,m} \\ \vdots & \ddots & \ddots & & \vdots & \vdots \\ \vdots & & \ddots & \ddots & \vdots & \vdots \\ 0 & \dots & \dots & 0 & h_{m,m-1} & h_{m,m} \end{pmatrix}. \tag{A.10}$$

  The structure of a Hessenberg matrix is interesting because it is almost triangular.

- Banded matrices : A matrix $\mathbf{A}$ has a band structure, if

$$a_{i,j} = 0 \quad \forall i > j + m_l \quad \text{and} \quad \forall j < i + m_u \,. \tag{A.11}$$

  Here, $m_l + m_u + 1$ is called the bandwidth.

- Orthonormal matrix: $\mathbf{A}^*\mathbf{A} = \mathbf{I}$ where $\mathbf{A}$ can be a rectangular matrix. Note here that the literature is not consistent on this notation.
- Unitary matrix: $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A} = \mathbf{I}$. The matrix $\mathbf{A}$ can't be rectangular, and can only be a square matrix.

[54]

B Incomplete LU decomposition

Suppose a drop tolerance t and a fill ratio f are given, and we want to compute the incomplete LU decomposition of a square matrix $\mathbf{A}$. One way to implement this is as follows:

1. for i from 1 to n:
   (a) *this part is only performed when using fill ratio*:
      i. count the non-zero entries in $\mathbf{v}$. Let's call this number $m_{\neq 0}$;
      ii. use $m_{\neq 0}$ and f to compute $m = m_{\neq 0} f$;
   (b) calculate $\mathbf{v} = \mathbf{e}_i^T \mathbf{A}$, where

$$\mathbf{e}_i = \begin{pmatrix} \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix} \leftarrow \text{row } i$$

   (c) for j from 1 to $i-1$:
      i. compute $v_j = \frac{v_j}{a_{j,j}}$;
      ii. *this step is only performed if using a drop tolerance*: If $v_j < t$ then drop v_j (this means $v_j = 0$);
      iii. if $v_j \neq 0$, then calculate $\mathbf{v} = -\mathbf{v} - v_i \mathbf{e}_j^T \mathbf{U}$;
2. *this step is only performed if using fill ratio*:
   Here, the number m, as defined earlier, comes in play. Drop all the smaller elements of $\mathbf{v}$, keeping only the largest m elements;
3. for j from 1 to $i-1$: $l_{i,j} = v_j$;
4. for j from i to n: $u_{i,j} = v_j$;

[55]

The above algorithm was formulated such that it is simultaneously compatible with the drop rules "drop tolerance" and "fill ratio". Indeed, according to Karypis and Kumar [55], it is possible to apply both rules simultaneously by following this formulation. Nevertheless, the COMSOL software package doesn't support this feature. Moreover, the COMSOL software package implements two exceptions to the drop rules stated above [6]:

- Diagonal elements are never dropped;
- If an element of the original matrix **A** is non-zero, the algorithm may be set such that is never drops such an element. COMSOL applies this if the check box “respect pattern” is checked.

C Results from secondary tests

C.1 Default parameters of the extra preconditioners

Preconditioner	Configuration:
Vanka	Main: Variable: pressure (p) (✻), Block solver: direct, Number of iterations: 10 (✻), Relaxation factor = 0.7 (✻); Secondary: Number of iterations: 10 (✻), Relaxation factor: 1.
SCGS	Main: Number of iterations: 2, Relaxation factor: 0.7 (✻), Block solver: Direct, stored factorisation (✻), Method: Mesh element lines and vertices, Vertex relaxation factor: 0.5; Secondary: Number of iterations: 1 iteration, relaxation factor : 0.5.
SOR Line	Main: Sweep type: SSOR, Number of iterations: 2, Relaxation factor: 0.7 (✻), Line based on: Mesh, Multivariable method: coupled, Blocked version. Secondary: Number of iterations: 1, Relaxation factor: 0.7 (✻).
SOR Gauge	Main: Solver: SSOR gauge, Number of iterations: 10 (✻), Relaxation factor: 0.7 (✻), Blocked version, Variables: pressure (p) (✻), Secondary: Number of secondary solvers: 1;
SOR Vector	Main: Solver: SSOR verctor, Number of iterations: 10 (✻), Relaxation factor: 0.7 (✻), Blocked version, Variables: pressure (p); Secondary: Number of secondary solvers: 1.

Preconditioner	Configuration:
Domain Decomposition	General: Solver: Multiplicative Schwarz, Number of iterations: 1, Number of subdomains: 2, Maximum nDOF per subdomain: 100 000, Maximum number of nodes per subdomain: 1, Additional overlap: 1, Overlap method: Matrix based, Hierarchy generation method: Lower element order first, Mesh coarsening factor: 2, Assemble on all levels, Use subdomain coloring; Coarse solver: MUMPS Domain solver: MUMPS with default settings
Krylov preconditioner	Using GMRES as preconditioner with: Number of iterations before restart: 50, Preconditioning: Left, Termination technique: Fixed number of iterations (10) Preconditioner for GMRES is ILU with: Drop using: Tolerance, Drop tolerance:1.

Table C.1: Defaut parameters of second set of preconditioners. Parameters that are different from the original default parameters implemented in COMSOL are marked by (✻).

C.2 The default parameters of the studied cases used in the solver search

In order to accelerate the process of searching for working solver-preconditioner combination, some parameters in the eigenvalue solver (for cases 1,2 and 3) and frequency domain solver (for case 4) were changed from the values present in the original settings[1]. Test cases 5 and 6 are omitted, as no iterative solver was found to work for either of them

C.2.1 Test case 1

- Search for eigenvalues around: $20Hz$
- Number of desired eigenvalues: 6
- Relative tolerance: 10^{-6}
- Number of degrees of freedom: 2958

C.2.2 Test case 2

- Search for eigenvalues around: $20Hz$
- Number of desired eigenvalues: 4
- Relative tolerance: 10^{-2}
- Number of degrees of freedom: 2718

C.2.3 Test case 3

- Search for eigenvalues around: $0Hz$
- Number of desired eigenvalues: 4
- Relative tolerance: 10^{-2}
- Number of degrees of freedom 48880

C.2.4 Test case 4

- Relative tolerance: 10^{-6}
- Number of degrees of freedom 238540

[1]as provided by the professorship for thermo-fluid dynamics in the Technical University of Munich

C.3 Overview of iterative solvers and their respective results

The purpose of this paper is just to find one or two iterative solvers that work for all models. Therefore, multiple descent candidates have been overlooked, as studying all solvers thoroughly is beyond the scope of this paper. Nonetheless, for each model,a complete list of the recorded behavior all solver-preconditioner combination is presented. This could hopefully serve as a guide for whoever wishes to explore alternative solutions. Note: all preconditioning is left preconditioning, unless stated otherwise. Additionally, all solvers and preconditioners use the default parameters, as described in Tables 3.2, 3.3 and C.1, unless otherwise stated.

C.3.1 Test case 1

Solver	preconditioner	Time	Physical memory
GMRES	Incomplete LU (standard)	00:00:08	1.08 GB
	Incomplete LU (SPOOLES)	00:00:07	1.04 GB
	SOR	00:00:04	1.09 GB
	Jacobi	00:00:04	1.09 GB
	Vanka	00:01:07	1:16 GB
	SCGS	00:00:05	1.16 GB
	SOR Line	00:00:03	1.16 GB
	SOR Gauge	00:00:08	1.15 GB
	SOR Vector	00:00:08	1.18 GB
	Multigrid	00:00:01	1.19 GB
	Domain Decomposition	00:00:03	1.22 GB
	Krylov Preconditioner	no convergence	
FGMRES	Incomplete LU (standard)	00:00:05	1.22 GB
	Incomplete LU (SPOOLES)	00:00:07	1.25 GB
	SOR	00:00:07	1.27 GB
	Jacobi	00:00:04	1.26 GB
	Vanka	00:01:01	1.34 GB
	SCGS	00:00:05	1.33 GB
	SOR Line	00:00:03	1.34 GB
	SOR Gauge	00:00:07	1.36 GB
	SOR Vector	00:00:07	1.36 GB
	Multigrid	00:00:01	1.37 GB
	Domain Decomposition	00:00:02	1.37 GB
	Krylov Preconditioner	00:00:05	1.37 GB
BiCGStab	Incomplete LU (standard)	00:00:13	1.36 GB
	Incomplete LU (SPOOLES)	00:00:14	1.38 GB
	SOR	00:00:07	1.39 GB
	Jacobi	00:00:07	1.38 GB
	Vanka	00:02:22	1.38 GB
	SCGS	00:00:04	1.41 GB
	SOR Line	00:00:04	1.42 GB
	SOR Gauge	00:00:16	1.42 GB
	SOR Vector	00:00:16	1.44 GB
	Multigrid	00:00:02	1.44 GB
	Domain Decomposition	00:00:03	1.46 GB
	Krylov Preconditioner	00:00:20	1.48 GB
Conjugate Gradient	Incomplete LU (standard)	00:00:12	1.48 GB
	Incomplete LU (SPOOLES)	00:00:07	1.48 GB
	SOR	00:00:05	1.44 GB
	Jacobi	00:00:03	1.50 GB
	Vanka	no convergence	
	SCGS	no convergence	
	SOR Line	00:00:02	1.54 GB
	SOR Gauge	00:00:07	1.56 GB
	SOR Vector	00:00:07	1.57 GB
	Multigrid	00:00:07	1.56 GB
	Domain Decomposition	no convergence	
	Krylov Preconditioner	no convergence	
Preconditioner	Multigrid	00:00:01	1.76 GB
	Domain Decomposition	00:00:02	1.77 GB
	Krylov Preconditioner	00:00:02	1.78 GB
	Remaining preconditioners	no convergence	

Table C.2: Overview of solver behavior, case 1

C.3.2 Test case 2

Solver	Preconditioner	Time	Physical Memory
GMRES	Incomplete LU (standard)	00:00:01	1.1 GB
	Incomplete LU (SPOOLS)	00:00:03	1.09 GB
	SOR	00:00:02	1.09 GB
	Jacobi	00:00:02	1.1 GB
	Vanka	00:00:05	1.12 GB
	SCGS	00:00:06	1.13 GB
	SOR Line	00:00:03	1.14 GB
	SOR Gauge	00:00:03	1.16 GB
	SOR Vector	00:00:03	1.17 GB
	Multigrid	00:00:01	1.15 GB
	Domain Decomposition	00:00:01	1.14 GB
	Krylov Preconditioner	00:01:45	1.24 GB
FGMRES	Incomplete LU (standard)	00:00:01	1.15 GB
	Incomplete LU (SPOOLS)	00:00:03	1.21 GB
	SOR	00:00:02	1.21 GB
	Jacobi	00:00:02	1.21 GB
	Vanka	00:00:04	1.19 GB
	SCGS	00:00:06	1.21 GB
	SOR Line	00:00:03	1.23 GB
	SOR Gauge	00:00:03	1.2 GB
	SOR Vector	00:00:03	1.2 GB
	Multigrid	00:00:01	1.23 GB
	Domain Decomposition	00:00:01	1.23 GB
	Krylov Preconditioner	00:00:01	1.23 GB
BiCGStab	Incomplete LU (standard)	00:00:01	1.21 GB
	Incomplete LU (SPOOLS)	00:00:07	1.21 GB
	SOR	00:00:05	1.24 GB
	Jacobi	00:00:02	1.25 GB
	Vanka	00:00:10	1.24 GB
	SCGS	00:00:02	1.22 GB
	SOR Line	00:00:02	1.26 GB
	SOR Gauge	00:00:06	1.25 GB
	SOR Vector	00:00:06	1.26 GB
	Multigrid	00:00:01	1.23 GB
	Domain Decomposition	00:00:01	1.26 GB
	Krylov Preconditioner	distorted result	
Conjugate Gradient	Incomplete LU (standard)	00:00:01	1.27 GB
	Incomplete LU (SPOOLS)	00:00:03	1.25 GB
	SOR	00:00:02	1.27 GB
	Jacobi	00:00:01	1.28 GB
	Vanka	00:00:05	1.25 GB
	SCGS	distorted result	
	SOR Line	00:00:01	1.34 GB
	SOR Gauge	00:00:03	1.3 GB
	SOR Vector	00:00:03	1.3 GB
	Multigrid	00:00:01	1.31 GB
	Domain Decomposition	00:00:01	1.31 GB
	Krylov Preconditioner	00:00:02	1.32 GB
Use preconditioner	Incomplete LU	00:00:13	1.31 GB
	SOR	00:00:02	1.3 GB
	Jacobi	00:00:02	1.29 GB
	Vanka	00:01:33	1.39 GB
	SCGS	no convergence	
	SOR Line	no convergence	
	SOR Gauge	no convergence	
	SOR Vector	no convergence	
	Multigrid	00:00:01	1.25 GB
	Domain Decomposition	00:00:01	1.26 GB
	Krylov Preconditioner	00:00:02	1.26 GB

Table C.3: Overview of solver behavior, case 2.

C.3.3 Test case 3

Solver	Preconditioner	Time	Physical memory
GMRES	Incomplete LU (standard)	00:00:31	6.61 GB
	Incomplete LU (SPOOLS)	00:01:25	3.93 GB
	SOR	00:00:39	6.55 GB
	Jacobi	00:01:21	6.54 GB
	Vanka	00:02:21	6.64 GB
	SCGS	00:02:32	6.64 GB
	SOR Line	00:01:08	3.88 GB
	SOR Gauge	00:01:20	6.41 GB
	SOR Vector	00:01:22	7.15 GB
	Multigrid	00:00:08	7.13 GB
	Domain Decomposition	00:00:19	6.27 GB
	Krylov Preconditioner	00:46:18	6.65 GB
FGMRES	Incomplete LU (standard)	00:00:26	6.63 GB
	Incomplete LU (SPOOLS)	00:01:41	4.06 GB
	SOR	00:01:01	5.8 GB
	Jacobi	00:01:33	6.33 GB
	Vanka	00:01:51	6.39 GB
	SCGS	00:02:17	3.81 GB
	SOR Line	00:01:13	6.43 GB
	SOR Gauge	00:01:43	6.48 GB
	SOR Vector	00:01:26	6.37 GB
	Multigrid	00:00:06	6.46 GB
	Domain Decomposition	00:00:26	4.04 GB
	Krylov Preconditioner	00:01:00	6.47 GB
BiCGStab	Incomplete LU (standard)	00:00:46	6.5 GB
	Incomplete LU (SPOOLS)	00:03:13	6.62 GB
	SOR	00:01:54	4 GB
	Jacobi	00:02:07	6.43 GB
	Vanka	00:05:13	3.79 GB
	SCGS	00:01:12	6.34 GB
	SOR Line	00:01:31	4.06 GB
	SOR Gauge	00:03:45	6.43 GB
	SOR Vector	00:03:15	7.01 GB
	Multigrid	00:00:29	6.43 GB
	Domain Decomposition	00:00:26	6.63 GB
	Krylov Preconditioner	no convergence	
Conjugate Gradient	Incomplete LU (standard)	00:00:34	3.87 GB
	Incomplete LU (SPOOLS)	00:01:32	6.61 GB
	SOR	00:01:20	3.64 GB
	Jacobi	00:01:07	3.57 GB
	Vanka	00:03:02	3.65 GB
	SCGS	no convergence	
	SOR Line	00:00:41	3.76 GB
	SOR Gauge	00:01:56	3.6 GB
	SOR Vector	00:01:48	3.62 GB
	Multigrid	00:00:09	3.69 GB
	Domain Decomposition	00:00:23	3.98 GB
	Krylov Preconditioner	00:01:39	3.68 GB
Use preconditioner	Incomplete LU	00:17:01	3.82 GB
	Vanka	00:57:00	6.46 GB
	Multigrid	00:00:11	3.59 GB
	Domain Decomposition	00:00:17	3.99 GB
	Krylov Preconditioner	00:00:55	5.44 GB
	Remaining	no convergence	

Table C.4: Overview of solver behavior, case 3.

C.3.4 Test case 4

Solver	Preconditioner	Time	Physical memory
GMRES	Incomplete LU (standard)	no convergence	
	Incomplete LU (SPOOLS)	no convergence	
	SOR	00:11:01	4.63 GB
	Jacobi	no convergence	
	Vanka	00:38:16	5.89 GB
	SCGS	no convergence	
	SOR Line	00:08:51	6.96 GB
	SOR Gauge	00:09:38	6.74 GB
	SOR Vector	00:38:16	5.89 GB
	Multigrid	no convergence	
	Domain Decomposition	00:17:20	9.26 GB
	Krylov Preconditioner	no convergence	
FGMRES	Incomplete LU (standard)	no convergence	
	Incomplete LU (SPOOLS)	no convergence	
	SOR	00:17:50	5.99 GB
	Jacobi	no convergence	
	Vanka	01:23:42	5.97 GB
	SCGS	no convergence	
	SOR Line	00:16:55	5.96 GB
	SOR Gauge	00:20:16	5.97 GB
	SOR Vector	01:25:55	5.88 GB
	Multigrid	no convergence	
	Domain Decomposition	00:18:17	10.7 GB
	Krylov Preconditioner	01:01:40	4.55 GB
BiCGStab	Incomplete LU (standard)	01:29:47	4.45 GB
	Incomplete LU (SPOOLS)	no convergence	
	SOR (relaxation factor: 0.9; 5 iterations)	00:14:38	4.36 GB
	Jacobi	no convergence	
	Vanka	no convergence	
	SCGS	no convergence	
	SOR Line	00:20:45	5.9 GB
	SOR Gauge	00:22:43	6.22 GB
	SOR Vector	03:09:27	4.26 GB
	Multigrid	no convergence	
	Domain Decomposition	00:38:03	9.69 GB
	Krylov Preconditioner	no convergence	
Conjugate Gradient	All preconditioners	no convergence	
Use preconditioner	All preconditioners	no convergence	

Table C.5: Overview of solver behavior, case 4.

C.3.5 Test case 5

No working iterative solvers were found for this case.

C.3.6 Test case 6

No working iterative solvers were found for this case.

C.4 Estimating the fluctuation in memory utilization

As stated in subsection 3.4, it was deemed interesting to briefly study to amount of variation in solution time and memory consumption under identical conditions. Since this is not central to this work, it was decided to include this discussion in the appendix.

C.4.1 Memory leak

At the beginning, it was suspected that some memory leak issues with COMSOL Multiphysics 4.4 might be present. Since this would interfere with any error estimates, some time was dedicatied to study this issue. Based on total of 24 of runs on case 1 (eigenmodes of a room), it was observed that for $nDoF = 2958$, using the CG with Multigrid (with default settings), the memory consumption increased by around $0.03GB$ per run, while the time requirements remained constant. However, once COMSOL Multiphysics 4.4 was shut down and restarted, the accumulated leak seems to vanish, and start accumulating again. This can clearly be seen in Figure C.1. However, the gradual increase in memory consumption does seem to slow down or level after a certain number of runs, as can be seen towards the end of the curve. This latter finding is not certain, the memory consumption in other tests did not stop increasing.

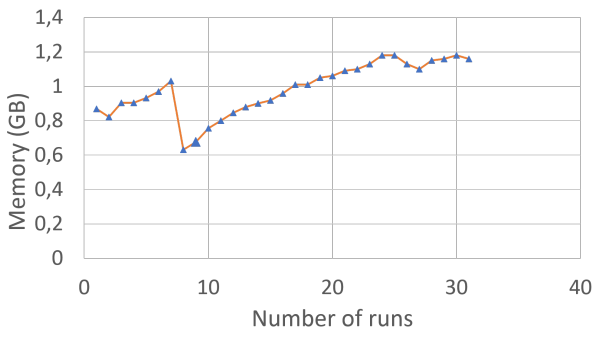


Figure C.1: The change in the memory consumption for case 1 (eigenvalues of a room), for increasing number of runs under identical conditions. COMSOL Multiphysics was shut down and launched anew around run number 7. Notice how this translates into the disappearance of the accumulated increase in memory consumption.

.

This was not studied any further than this. Since the presence of such issues was suspected early on, the tests discussed thus far were done so as to minimize the interference of such effects. For instance, tests for the scaling tests of direct and iterative solvers were conducted in a strictly alternating fashion. Additionally, tests involving a given case were conducted in a single session for the sake of consistency. However, if a more detailed study is to be conducted, it is advised to take this issue into consideration. It should also be stated that the tests for error estimation were conducted after running a simple model somewhere between 30 and 40 times in hope of minimizing the interference from this effect. However, this might not have much of a mitigating effect, as can be seen in for instance Table C.2, where a steady skew upwards in the memory consumption of solvers can still be observed afterwards.

C.4.2 Error in the scaling tests

The results of the error estimates are presented in Tables C.6, C.7, C.8 and C.9. It can be observed that the solution time is fairly constant, only varying by a some seconds at high problem sizes. An exact relation between problem size and the amount of variation in time is not of interest, here.

The more important matter is the variation in memory consumption. The standard deviation appers be fairly high, reaching upwards of 1.27 GB for case 3, at a memory consumption of 6.9 GB. The results indicate that this deviation is dependent on the problem size and/or the average memory requirements of the problem, though Table C.9 suggests that problem size is the more relevant parameter. Indeed, in that table, it can be seen that for GMRES with SOR, an increase of problem size by about an order of magnitude, at near constant average memory consumption, caused an increase in the deviation in memory consumptio by a similar amount.

Nevertheless, due to the small sample sizes, these results should be taken with caution. For instance, the results for the error estimate tests of MUMPS in case 4 (Table C.9), would indicate that the deviation in memory consumption of MUMPS scales inversely with increasing problem size. This is in contradiction with the results from the other cases, and is more likely to be the result of the small sample size. In any case, the reader should feel free to interpret these values to estimate the maximal problem size that can be safely solved by the suggested solvers. More robust tests with larger sample sizes are needed to obtain a more accurate data regarding the scaling behavior and the variation in the memory consumption. Automating any such tests is highly recommended, even if it would require significant investments in time and effort.

Test cases 5 and 6 admitted no working iterative solvers.

nDoF			runs					average	standard deviation
2958	CG with Multigrid	Time (s)	2	2	2	2	2	2	0.0
		Physical memory (GB)	1.18	1.17	1.16	1.17	1.17	1.17	0.0063
	MUMPS	Time (s)	0	0	0	0	0	0	0.0
		Physical memory (GB)	1.19	1.17	1.16	1.17	1.19	1.176	0.0120
153076	CG with Multigrid	Time (s)	53	52	53	53	53	52.8	0.4
		Physical memory (GB)	1.86	1.85	1.94	1.86	1.94	1.89	0.0410
	MUMPS	Time (s)	45	45	45	45	45	45	0.0
		Physical memory (GB)	3.04	3.31	3.33	3.32	3.36	3.272	0.1172

Table C.6: Analysis of typical deviation in solution time and memory consumption for **case 1**

nDoF	solver		runs					average	standard deviation
2718	CG with Multigrid	Time (s)	1	1	1	1	1	1	0.00
		Physical memory (GB)	1.24	1.25	1.25	1.27	1.29	1.26	0.0179
	MUMPS	Time (s)	0	0	0	0	1	0.2	0.40
		Physical memory (GB)	1.21	1.25	1.26	1.29	1.31	1.264	0.0344
775356	CG with Multigrid	Time (s)	73	68	68	70	55	66.8	6.18
		Physical memory (GB)	2.48	2.53	2.53	2.56	2.57	2.534	0.0314
	MUMPS	Time (s)	37	34	34	36	38	35.8	1.60
		Physical memory (GB)	2.55	2.84	2.93	2.94	2.94	2.84	0.1498

Table C.7: Analysis of typical deviation in solution time and memory consumption for **case 2**

nDoF	solver		runs					average	standard deviation
367	CG with Multigrid	Time (s)	1	0	0	0	0	0.2	0.40
		Physical memory (GB)	6.3	6	6.42	6.47	6.29	6.296	0.1633
	MUMPS	Time (s)	0	0	0	0	0	0	0.00
		Physical memory (GB)	6.39	5.42	5.84	5.58	6.29	5.904	0.3817
509793	CG with Multigrid	Time (s)	210	192	197	197	195	198.2	6.18
		Physical memory (GB)	8.07	5.32	8.12	7.59	5.4	6.9	1.2712
	MUMPS	Time (s)	230	236	228	246	237	235.4	6.31
		Physical memory (GB)	14.14	12.94	14.12	14.33	14.1	13.926	0.4998

Table C.8: Analysis of typical deviation in solution time and memory consumption for **case 3**

nDoF	solver		runs					average	standard deviation
17540	GMRES with SSOR	Time (s)	33	32	33	32	32	32.4	0.49
		Physical memory (GB)	5.06	5	5.05	5.01	5	5.024	0.0258
	MUMPS	Time (s)	30	28	28	28	29	28.6	0.80
		Physical memory (GB)	3.64	5.06	5.07	5.1	5.08	4.79	0.5752
238400	GMRES with SSOR	Time (s)	908	834	835	838	845	852	28.26
		Physical memory (GB)	5.7	5.88	6.14	4.52	5.23	5.494	0.5705
	MUMPS	Time (s)	585	582	587	591	573	583.6	6.05
		Physical memory (GB)	11.15	11.25	11.31	11.29	11.19	11.238	0.0601

Table C.9: Analysis of typical deviation in solution time and memory consumption for **case 4**

Bibliography

[1] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3): 856–869, 1986.

[2] Y. Saad. Variations on arnoldi's method for computing eigenelements of large unsymmetric matrices. *Linear algebra and its applications*, 34:269–295, 1980.

[3] J. Gikadi. *Prediction of Acoustic Modes in Combustors using Linearized Navier-Stokes Equations in Frequency Space*, chapter 4, pages 60–84. PhD Thesis, Universitätsbibliothek der TU München, 2014.

[4] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of computation*, 37(155):105–126, 1981.

[5] C. Blom. *Discontinuous Galerkin Method on tetrahedral elements for aeroacoustics.* PhD thesis, University of Twente, Enschede, 2003.

[6] C. Inc. *COMSOL Muntiphysics Reference Manual*, version 4.4 edition, 2013.

[7] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods.* SIAM, 1998.

[8] C. Sensiau, F. Nicoud, M. Van Gijzen, and J. Van Leeuwen. A comparison of solvers for quadratic eigenvalue problems from combustion. *International journal for numerical methods in fluids*, 56(8):1481–1487, 2008.

[9] E. Åkervik. *Global stability and feedback control of boundary layer flows.* PhD thesis, KTH, 2008.

[10] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the gram-schmidt factorization. *Mathematics of Computation*, 30(136):772–795, 1976.

[11] J. Lunze. *Regelungstechnik 1 - Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen.* Springer-Verlag, Berlin Heidelberg New York, 2016. ISBN 978-3-662-52678-1.

[12] T. A. Davis. *Direct methods for sparse linear systems.* SIAM, 2006.

[13] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[14] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[15] L. N. Trefethen and D. Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[16] M. Schaefer. *Numerik im Maschinenbau.* Springer-Verlag, Berlin Heidelberg New York, 2013. ISBN 978-3-642-58416-9.

[17] B. Wohlmuth. Numerical treatement of partial differential equations (mse). `https://www.moodle.tum.de/pluginfile.php/942360/mod{_}resource/content/1/Gesamtvolltext.pdf`, 2016. Lecture, slide nr 112, available on demand.

[18] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Numerical Analysis: Proceedings of the Dundee Conference on Numerical Analysis, 1975*, pages 73 – 89, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg. doi: 10.1007/BFb0080116.

[19] H. A. Van der Vorst . Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[20] A. Meister. Iterative verfahren. *Numerik linearer Gleichungssysteme*, pages 69–206, 2011.

[21] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[22] H. R. Schwarz and N. Köckler. *Numerische mathematik.* Springer-Verlag, 8th edition, 2011.

[23] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[24] O. Axelsson. Iterative solution methods. *Cambridge University*, 1994.

[25] Y. Saad. *Numerical methods for large eigenvalue problems*, chapter Krylov subspace methods, pages 125 – 162. Manchester University Press, 1992.

[26] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

[27] W. Bunse and A. Bunse-Gerstner. *Numerische lineare Algebra*, chapter The Method of conjugate gradients, pages 148–165. Teubner Studienbücher, 1985. ISBN 3-519-02067-X.

[28] G. R. Di Brozolo and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11 (2):223–239, 1989.

[29] H. A. Van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13. Cambridge University Press, 2003.

[30] I. Gustafsson. A class of first order factorization methods. *BIT Numerical Mathematics*, 18(2):142–156, 1978.

[31] G. H. Golub and C. F. V. Loan. *Matrix Computations* -. JHU Press, London, 1996. ISBN 978-0-801-85414-9.

[32] A. Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.

[33] K. Morikuni, L. Reichel, and K. Hayami. Fgmres for linear discrete ill-posed problems. *Applied Numerical Mathematics*, 75:175–187, 2014.

[34] Y. Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[35] D. A. Belsley, E. Kuh, and R. E. Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*, volume 571. John Wiley & Sons, 2005.

[36] A. Schwarzenberg-Czerny. On matrix factorization and efficient least squares solution. *Astronomy and Astrophysics Supplement Series*, 110:405, 1995.

[37] Y. Saad. *Iterative methods for sparse linear systems*, volume 3, chapter Section 10.3, page 87. Los Alamitos, CA: IEEE Computer Society, c1994-c1998., 1996.

[38] V. John. Higher order finite element methods and multigrid solvers in a benchmark problem for the 3d navier–stokes equations. *International Journal for Numerical Methods in Fluids*, 40(6):775–798, 2002.

[39] R. Beck and R. Hiptmair. Multilevel solution of the time-harmonic maxwell's equations based on edge elements. *International journal for numerical methods in engineering*, 45 (7):901–920, 1999.

[40] R. Hiptmair. Multigrid method for maxwellś equations. *SIAM Journal on Numerical Analysis*, 36(1):204–225, 1998.

[41] S. P. Vanka. Block-implicit multigrid solution of navier-stokes equations in primitive variables. *Journal of Computational Physics*, 65(1):138–158, 1986.

[42] A. Brandt. Multi-level adaptive technique (mlat) for fast numerical solution to boundary value problems. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, pages 82–89. Springer, 1973.

[43] R. P. Fedorenko. The speed of convergence of one iterative process. *USSR Computational Mathematics and Mathematical Physics*, 4(3):227–235, 1964.

[44] W. Hackbusch. *Multi-grid methods and applications*, volume 4. Springer Science & Business Media, 2013.

[45] A. Arnone, M.-S. Liou, and L. A. Povinelli. Integration of navier-stokes equations using dual time stepping and a multigrid method. *AIAA journal*, 33(6):985–990, 1995.

[46] W. Hackbusch and U. Trottenberg. *Multigrid methods: proceedings of the conference held at Köln-Porz, November 23-27, 1981*, volume 960. Springer, 2006.

[47] P. Wessling. *An introductio to multigrid methods.* John Wiley and Sons, Chichester, 1992.

[48] G. C. Hackbusch, W. Iterative lösung großer schwach besetzter gleichungssysteme. stuttgart, bg teubner 1991. 382 s., dm 42,-isbn 3-519-02372-5 (leitfaden der angewandten mathematik und mechanik 69, teubern-studienbücher: Mathematik). *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 74(2):96–96, 1994.

[49] I. COMSOL. *COMSOL Multiphysics Model library manual.* COMSOL Inc., 4.4 edition, 2013. chapter: Eigenmodes of a Room.

[50] M. P. G. Zellhuber. *High Frequency Response of Auto-Ignition and Heat Release to Acoustic Perturbations.* PhD thesis, Universitätsbibliothek der TU München, 2013.

[52] H. Draper, N.R.; Smith. *Applied Regression Analysis.* John Wiley, 3 edition, 1998. ISBN ISBN 0-471-17082-8.

[51] L. Tay-Wo-Chong, T. Komarek, R. Kaess, S. Foller, and W. Polifke. Identification of flame transfer functions from les of a premixed swirl burner. *ASME Turbo Expo 2010: Power for Land, Sea, and Air,* pages 623–635, 2010.

[54] Y. Saad. *Numerical methods for large eigenvalue problems*, chapter Background in Matrix Theory and Linear Algebra, pages 1 – 27. Manchester University Press, 1992.

[55] G. Karypis and V. Kumar. Parallel threshold-based ilu factorization. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 28–28. IEEE, 1997.