DISS. ETH NO. 14734

The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
ROMAN GEUS

Dipl. Informatik-Ing. ETH
born 6. September 1970
citizen of Arbon, Switzerland

accepted on the recommendation of
Prof. Dr. Walter Gander, examiner
Prof. Dr. Henk van der Vorst, co-examiner
Dr. Peter Arbenz, co-examiner

2002

Acknowledgements

This work would not have been possible without the help of many people. First of all I would like to thank Prof. Dr. Walter Gander for giving me the opportunity to do my Ph.D. thesis in his group. His guidance and encouragement were always a great help.

My special thank goes to Dr. Peter Arbenz, my supervisor during all these years. With his support he contributed a great deal to this work. He was there to answer many of my questions and to help me dealing with difficult problems. He gave me the opportunity and freedom to learn a great variety of methods and to realise my own ideas.

I'm very much obliged to Prof. Dr. Henk van der Vorst for accepting to be co-examiner of this thesis.

I'm very grateful to the three of them for reviewing the entire thesis and giving valuable suggestions.

A special thank goes to all my present and former co-workers in the group, Oliver Bröker, Oscar Chinellato, Leonhard Jaschke, Erwin Achermann and Rolf Strebel. We had many inspiring and not only scientific discussions. They all contributed a lot to a pleasant working atmosphere, together with all the other members of the institute.

Further, I want to thank the following people: Dr. Stefan Adam, who was one of the initiators of this project, for his support in the early stages of this work; Dr. Hans-Rudolf Fitze, Markus Bopp and Lukas Stingelin for providing me the mesh data of the accelerator cavities and giving me some insight into accelerator cavity design; Stefan Röllin for contributing to the optimisation and parallelisation of the sparse matrix vector multiplication; Anne Preisig for helping me with the translation of this document.

I would like to express my gratitude to all the people who dedicated their time to develop the free and open source software which was essential for realising this project.

Furthermore, I want to thank all those who have supported me in non-work related terms. In this respect I am especially grateful to Regula, my family and all my friends for the moral support during this long and interesting time.

Last but not least, I want to thank everyone else who kept asking all these years: "Have you finished your thesis yet?" Silencing that question is an extra benefit of this work.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Berechnung elektromagnetischer Schwingungen in Kavitäten von Teilchenbeschleunigern. Die zeit-harmonischen Maxwell-Gleichungen, welche diese elektromagnetischen Schwingungen beschreiben, werden mit der Methode der finiten Elemente diskretisiert und so in ein allgemeines symmetrisches Matrix-Eigenwertproblem übergeführt. Von diesem Eigenwertproblem sollen die 10 bis 20 kleinsten *positiven* Eigenwerte mit den zugehörigen Eigenvektoren berechnet werden.

Werden übliche knotenbasierte finite Elemente verwendet, so wird das Eigenwertproblem viele unphysikalische Eigenlösungen (sog. *spurious modes*) mit positiven Eigenwerten aufweisen.

In einem ersten Lösungsansatz verwenden wir weiterhin die üblichen knotenbasierten finiten Elemente und führen einen zusätzlichen Strafterm ein. Die gewünschten Eigenlösungen können dann mit der “shift-and-invert” Spektraltransformation berechnet werden.

In einer zweiten Variante verwenden wir Nédélec-Vektorelemente. Alle spurious modes haben dann den Eigenwert Null. Wir stellen verschiedene Methoden vor, welche die effiziente Berechnung der gewünschten Eigenlösungen mit positivem Eigenwert erlauben.

Als Eigenwertlöser benutzen wir einerseits das ARPACK Softwarepaket, welches die Lanczos Methode mit implizitem Neustart implementiert, und andererseits unsere eigene, für das symmetrische Eigenwertproblem optimierte Implementation der Jacobi-Davidson Methode.

Ein weiterer Schwerpunkt dieser Arbeit bildet die Optimierung der Matrix-Vektor Multiplikation für schwach-besetzte Matrizen. Wir stellen Verfahren vor, welche diese Operation substanziell beschleunigen.

Unsere Software haben wir als Python-Module implementiert, wobei die zeitkritischen Berechnungen als Erweiterungsmodule in C realisiert wurden.

Wir haben effiziente und stabile Verfahren entwickelt, um die gewünschten Eigenlösungen zu berechnen. Unsere Software ist in der Lage, grosse Eigenwertprobleme (Ordnung grösser als zwei Millionen) auf einer Einprozessormaschine zu bewältigen. Mit der Python-Implementation haben wir eine Softwareumgebung geschaffen, welche die implementierten Methoden einfach benutzen, kombinieren und erweitern lässt.

Summary

This thesis deals with the computation of electromagnetic oscillations in cavities of particle accelerators. The time-harmonic Maxwell equations, which describe these electromagnetic oscillations, are discretised using the finite element method. Of the resulting generalised symmetric matrix eigenvalue problem, the 10 to 20 smallest *positive* eigenvalues together with the associated eigenvectors have to be computed.

If ordinary node-based finite elements are used, then the eigenvalue problem will exhibit many non-physical eigensolutions (so-called *spurious modes*) with positive eigenvalues.

In a first approach we use node-based finite elements and introduce an additional penalty term. The desired eigensolutions can then be computed using a shift-and-invert spectral transformation.

In a second variant we use Nédélec vector finite elements. All spurious modes then have the eigenvalue zero. We present different methods, which allow us to efficiently compute the desired eigensolutions with positive eigenvalues.

As eigenvalue solvers we use on the one hand the ARPACK package, which implements the Implicitly Restarted Lanczos method, and on the other hand our own implementation of the Jacobi Davidson method optimised for the symmetric eigenvalue problem.

A further emphasis of this thesis is the optimisation of the matrix-vector multiplication for sparse matrices. We present techniques, which accelerate this operation substantially.

We implemented our software as Python modules. The time-critical computations were realised as extension modules written in C.

We developed efficient and stable methods for computing the desired eigensolutions. Our software is able to handle large eigenvalue problems with an order of over two million on a single-processor workstation. With the Python implementation, our methods can be used, combined and extended easily.

Notation

$\boldsymbol{A}, \ldots, \boldsymbol{Z}$	matrices
a_{ij}	entry of matrix $\boldsymbol{A}$ in the i-th row and the j-th column
$\boldsymbol{A}(i,j)$	alternate notation for a_{ij}
$\boldsymbol{A}(i,:), \boldsymbol{A}(:,j)$	i-th row, resp. j-th column of $\boldsymbol{A}$
$\boldsymbol{A}([i_1,i_2,i_3],:)$	submatrix, consisting of rows i_1, i_2 and i_3 of $\boldsymbol{A}$
$\boldsymbol{A}(:,[j_1,j_2,j_3])$	submatrix, consisting of columns j_1, j_2 and j_3 of $\boldsymbol{A}$
$\mathbf{a}, \ldots, \mathbf{z}$	vectors
$\mathbf{A}, \ldots, \mathbf{Z}$	vectors
$\langle \mathbf{a}, \mathbf{b} \rangle$	$\langle \mathbf{a}, \mathbf{b} \rangle = \int_\Omega \mathbf{a} \cdot \mathbf{b} \, \mathrm{d}\mathbf{x}$
$\mathbf{a} \cdot \mathbf{b}$	scalar dot product
$\mathbf{a}.*\mathbf{b}$	element-wise multiplication
$\mathbf{a}./\mathbf{b}$	element-wise division

Symbols

$\boldsymbol{\xi} = (\xi, \eta, \zeta)^T$	Cartesian coordinates in the unit tetrahedron
$\mathbf{x} = (x, y, z)^T$	Cartesian coordinates
L_1, L_2, L_3, L_4	simplex coordinates in a general tetrahedron, functions in x, y, z
(x_i, y_i, z_i)	coordinates of tetrahedron corners, $1 \leq i \leq 4$
N_k	scalar, global basis functions for the FEM
$N_k^{(e)}$	scalar, local basis functions for the FEM, defined in tetrahedron e
$N_k^{(0)}$	scalar, basis functions for the FEM, defined in the unit tetrahedron
$\mathbf{N}_k$	global vector basis functions for the FEM
$\mathbf{N}_k^{(e)}$	local vector basis functions for the FEM, defined in tetrahedron e
$\mathbf{N}_k^{(0)}$	vector basis functions for the FEM, defined in the unit tetrahedron
u, $\mathbf{u}$	solutions of differential equations
Ω	domain on which the differential equations are solved
Γ	boundary of Ω, surface
T_e	e-th tetrahedral element
T_0	unit tetrahedron
$\boldsymbol{A}^{(e)}$	stiffness element matrix or curl element matrix
$\boldsymbol{M}^{(e)}$	mass element matrix
$\boldsymbol{N}^{(e)}$	divergence element matrix
e	index over finite elements
i, j	indices over global degrees of freedom (DOFs)
$\boldsymbol{A}, \boldsymbol{M}$	global matrices of the generalised eigenvalue problem
$\boldsymbol{Y}$	sparse basis of the null space of $\boldsymbol{A}$
$\boldsymbol{C}$	$\boldsymbol{C} := \boldsymbol{M}\boldsymbol{Y}$
$\boldsymbol{H}$	$\boldsymbol{H} := \boldsymbol{Y}^T\boldsymbol{C}$, used in the projector $\boldsymbol{P}_{\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}}$

s	penalty parameter
α	parameter for Bespalov term
σ	shift for "Shift-and-Invert" spectral transformation
τ	target value used in the Jacobi-Davidson algorithm
$\boldsymbol{K}$	preconditioner, approximation of $\boldsymbol{A} - \sigma\boldsymbol{M}$, resp. $\boldsymbol{A} - \theta\boldsymbol{M}$
$\boldsymbol{V}$	basis of search subspace used in eigensolvers
$\boldsymbol{V_A}$	$\boldsymbol{V_A} := \boldsymbol{A}\boldsymbol{V}$
$\boldsymbol{G}$	$\boldsymbol{G} := \boldsymbol{V}^T\boldsymbol{V_A}$, interaction matrix for Rayleigh-Ritz step
$(\theta, \mathbf{u})$	Ritz pair
$\boldsymbol{W}, \boldsymbol{S}$	$[\boldsymbol{W}, \boldsymbol{S}] = \mathrm{eig}(\boldsymbol{G})$, eigenvectors $\boldsymbol{W}$ and eigenvalues $\boldsymbol{S}$ of $\boldsymbol{G}$
$\boldsymbol{Q}$	matrix of converged eigenvectors
$\boldsymbol{Q_M}$	$\boldsymbol{Q_M} := \boldsymbol{M}\boldsymbol{Q}$
$\hat{\boldsymbol{Q}}$	$\hat{\boldsymbol{Q}} := [\boldsymbol{Q}, \mathbf{u}]$
$\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$	$\hat{\boldsymbol{Q}}_{\boldsymbol{M}} := \boldsymbol{M}\hat{\boldsymbol{Q}}$
$\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$	$\hat{\boldsymbol{Q}}_{\boldsymbol{K}} := \boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}$, preconditioned eigenvectors
$\hat{\boldsymbol{F}}$	$\hat{\boldsymbol{F}} := \hat{\boldsymbol{Q}}_{\boldsymbol{K}}^T\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$, matrix used in the correction equation of JDSYM

Contents

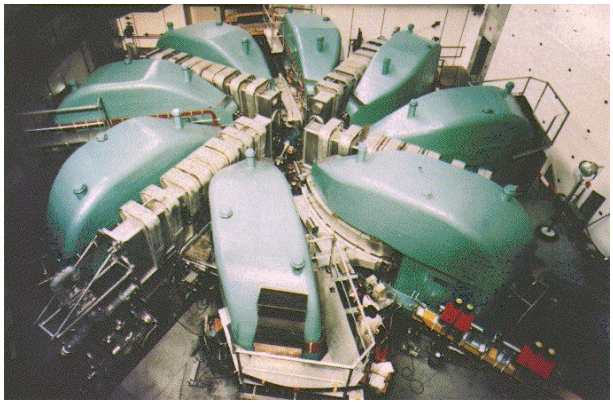
1 Introduction/Overview

In this thesis we deal with the computation of electromagnetic oscillations in cavities of particle accelerators. This problem has several aspects that have to be understood in order to develop software for efficiently calculating the desired electromagnetic fields. These aspects include mathematics, physics, computer science and numerical analysis.

The following paragraphs introduce the different aspects. The chapter continues with a list of our aims for this dissertation. The introductory chapter concludes with a paragraph describing the organisation of this thesis.

1.1 Design of Particle Accelerators

The common way to produce accelerating electromagnetic fields in cyclic accelerators like the one represented below[1] is to excite standing waves in accelerating cavities.



The mathematical model for these high frequency electromagnetic fields is an eigenvalue problem derived from the Maxwell equations [1]. Usually, the eigenfield corresponding to the fundamental mode of the cavity is used as the accelerating field. A few modes of higher order have to be analysed as well because these modes can be excited due to higher harmonic components contained in the RF (radio frequency) power fed into the cavity and also through interactions between the accelerated particles and the electromagnetic field. The RF engineer designing such an accelerating cavity therefore needs a tool to compute the fundamental and about 10 to 20 of the subsequent eigenfrequencies together with the corresponding electromagnetic eigenfields.

[1] 590 MeV Ring Cyclotron at Paul Scherrer Institute

The most interesting quantities besides the eigenfrequencies are local maxima of the eigenmodes as well as the fields on the surface that induce heat in the metallic boundary and determine the power loss from surface currents.

For our calculations we assume perfectly conducting walls and vacuum in the interior. By means of a time-harmonic ansatz for the electric and the magnetic field and by eliminating $\mathbf{H}$, one obtains the following form of Maxwell's equations [1]

$$\mathbf{rot}\,\mathbf{rot}\,\mathbf{e}(\mathbf{x}) = \lambda\,\mathbf{e}(\mathbf{x}), \qquad \mathbf{x}\in\Omega, \qquad \lambda := \omega^2/c^2, \tag{1.1a}$$

$$\operatorname{div}\mathbf{e}(\mathbf{x}) = 0, \qquad \mathbf{x}\in\Omega, \tag{1.1b}$$

$$\mathbf{n}\times\mathbf{e} = \mathbf{0}, \qquad \mathbf{x}\in\Gamma. \tag{1.1c}$$

$\mathbf{e}(\mathbf{x})$ represents the amplitude of the electric field at position $\mathbf{x}$ and c is the speed of light. We are interested in the 10 to 20 solutions of (1.1) with the lowest eigenfrequencies ω.

1.2 Applying the Finite Element Method

We discretise (1.1) using the finite element method (FEM). The interior of the cavity is approximated by a tetrahedral mesh. To implement the divergence-free condition (1.1b), two approaches, the *penalty method* and the *Nédélec finite element discretisation*, are established [45]. Both of them are investigated in this thesis.

The Penalty Method In the penalty method ordinary nodal elements are used. If the divergence-free condition was omitted, one would get the desired divergence-free eigensolutions, but also undesired solutions (spurious modes) with non-vanishing divergence, which are spread over the entire spectrum [38].

To enforce the divergence-free condition, a penalty term that increases the eigenvalue of the spurious modes [46][1][45] is added. If the penalty term is chosen large enough, the region of interest of the spectrum is free of spurious modes.

This approach yields a matrix eigenvalue problem

$$\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x},$$

where both $\boldsymbol{A}$ and $\boldsymbol{M}$ are symmetric positive definite.

The desired, smallest positive, eigenvalues are extremal and can be computed using a shift-and-invert spectral transformation.

The Nédélec finite element discretisation In this approach Nédélec vector finite elements, which were designed for solving problems containing the $\mathbf{rot}\,\mathbf{rot}$-operator, are used [54].

This approach yields a matrix eigenvalue problem

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x},$$

where both $\boldsymbol{A}$ and $\boldsymbol{M}$ are symmetric. $\boldsymbol{M}$ is positive definite, while $\boldsymbol{A}$ is only positive semi-definite. The dimension of $\boldsymbol{A}$'s null space is huge, about a sixth of the order of $\boldsymbol{A}$. The vectors in the null space of $\boldsymbol{A}$ correspond to those functions in the Nédélec finite element space that have non-zero (discrete) divergence. So, all eigensolutions not satisfying the divergence-free constraint correspond to the eigenvalue zero.

Thus, a few of the smallest *positive* eigenvalues, together with the corresponding eigenvectors, have to be computed. In contrast to the penalty method, the sought eigenvalues are no longer extremal. We present several approaches, how the desired eigensolutions can be computed efficiently. Some of them exploit the knowledge of a sparse basis of the null space.

1.3 Methods and algorithms

Numerous algorithms for solving the generalised matrix eigenvalue problem are available. However, since our matrices are sparse and can get very large (order $> 10^6$), our choice of algorithms narrows down to methods that preserve the structure of the matrices and access them only via matrix-vector multiplications. Since we are only interested in a very limited number of eigenpairs, *subspace methods* are ideal for this task.

Subspace methods are iterative methods, that build a *search subspace* from which the eigenvector approximations are selected. In the symmetric case, this is usually done using a Rayleigh-Ritz step. In each iteration the search subspace is expanded by one or more new search directions. *Restarted subspace methods* limit the dimension of the search subspace by discarding some search direction periodically.

In this thesis we focus mainly on the Jacobi-Davidson eigensolver [68]. We propose a variant called JDSYM, which is optimised for the generalised symmetric eigenvalue problem. As a second method we use is the Implicitly Restarted Lanczos algorithm (IRL). We compare the performance of both these restarted subspace methods by the means of numerical experiments.

We study a number of preconditioners, which we use to speed up the inner iterations. Apart from Jacobi, SSOR and incomplete factorisation preconditioners, we also discuss *hierarchical basis preconditioners*, that exploit the hierarchical organisation of the finite element basis functions.

We also focus on the sparse matrix-vector multiplication. This numerical kernel uses up to 80% of the total computation time, but unfortunately runs very ineffectively on modern RISC based processors. We discuss optimisation techniques, which improve the performance of this kernel and also describe its parallelisation using message-passing.

1.4 Scripting Languages in Scientific Computing

In scientific computing, software is traditionally developed using compiled languages for maximal performance. Fortran and C are the most widely used languages. However, for most applications the time-critical portion of the code, that requires the efficiency of a compiled language, can be organised in a small set of well-defined functions. Implementing the remaining part of the application using an interactive and interpreted high-level language has many advantages:

- ▷ Thanks to the high-level data types and the dynamic typing make, software is easier and faster to develop. The language assists the programmer in focusing on the problem, instead of on implementation details.
- ▷ Since the language is interpreted, turnaround times are substantially reduced.
- ▷ Interactive languages help the user to solve the problem in an explorative manner.
- ▷ High-level languages offer better possibilities for code sharing and code reuse. It is in general easier to combine modules and construct new applications.

In this thesis we study this mixed-language programming approach. All sparse and dense linear algebra routines, including iterative solvers, preconditioners, sparse matrix factorisations and the eigensolver are implemented in C. The input/output routines, the computational steering and the finite element application are implemented in Python.

Python offers some appealing features which make it ideal for our purposes: It combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. Python code can be easily interfaced to C or C++ code.

1.5 Aims of this thesis

The aim of this project is to develop software for efficiently and accurately computing electromagnetic oscillations in cavities of particle accelerators. The thesis tries to find answers to the following questions:

- ▷ Which type of finite element discretisation is best suited to solve the Maxwell eigenvalue problem?
- ▷ How can the indefinite eigenvalue problem stemming from the discretisation with Nédélec vector finite elements be solved efficiently?
- ▷ How does the performance of JDSYM compare to the performance of the well-established IRL method for our symmetric generalised eigenvalue problems?
- ▷ How big is the performance penalty that has to be paid if the mixed-language programming approach with Python is used for solving the Maxwell eigenvalue problem?

1.6 Organisation of this thesis

Chapter 2 describes the scalar Laplace eigenvalue problem. We are not really interested in its eigensolutions. Instead we illustrate the finite element method (FEM) and introduce coordinate transformations and basis functions by the means of this problem. These are then used in Chapter 3, which describes the Maxwell eigenproblem and introduces two types of finite elements: *nodal* and *Nédélec vector* elements. In the same chapter we also deal with the implementation of the boundary conditions and the divergence-free conditions of the Maxwell equations and conclude with a comparison of the finite element types.

Chapter 4 describes, how we compute the desired eigensolutions (10 to 20 of the *smallest positive* eigenvalues with the corresponding eigenvectors) of the matrix eigenvalue problem stemming from the finite element discretisation.

The eigensolvers themselves are described in Chapter 5. The Jacobi-Davidson algorithm, along with its relatives, the JOCC and the Davidson algorithms, and the Implicitly Restarted Lanczos algorithm are discussed. We thoroughly describe JDSYM, which is our implementation of the Jacobi-Davidson algorithm.

In Chapter 6 we give a brief overview over the iterative methods we used for solving the inner linear systems occurring in the eigensolvers.

In Chapter 7 we focus on the matrix-vector multiplication with sparse matrices. Several optimisation techniques, as well as the parallelisation using message-passing are presented.

In Chapter 8 the preconditioners we used to accelerate the eigen- and the linear solvers are presented and compared.

Chapter 9 describes our Python implementation, which consists of *PySparse* (extensions for supporting sparse linear algebra) and *PyFemax* (the finite element application).

The appendix contains useful reference information: We describe the storage formats we used for storing sparse matrices, the machines we ran the numerical experiments on and the analytic solutions of the scalar Laplace and the Maxwell problem for a rectangular box-shaped domain. In addition there is detailed information on the tetrahedral grids and the global finite element matrices, that we used in our experiments. Furthermore we list the formulae for calculating the various element matrices.

The *numerical experiments* are scattered throughout the document: Chapter 3 features some calculations to demonstrate the advantages and disadvantages of various types of finite elements. In Chapter 4 the solution methods for the indefinite eigenvalue problem are compared using a set of experiments. In Chapter 5 we try to find good values for JDSYM's parameters using numerical experiments. In Chapter 7 the effectiveness of the optimisation techniques for the sparse matrix-vector multiplication is investigated by the means of numerical experiments. In the same chapter we also present our results for the parallel matrix-vector multiplication. Speedup figures for six different multiprocessor systems are shown. The efficiency of the preconditioners is evaluated with experiments in Chapter 8. Finally, in Chapter 9 we investigate the overhead of the Python implementation on the basis of numerical experiments. In the same chapter we report our experience with solving large Maxwell eigenproblems.

If not mentioned otherwise, we used a Sun Enterprise E3500 for the numerical experiments (cf. Appendix B for precise specifications).

2 Problem I: 3D Laplace-Problem

The first mathematical problem to be treated is the scalar Laplace equation in a three dimensional domain. Differential equations of this kind can be used to describe e.g. oscillations in cavities [65, page 396]. Because of its simplicity, this problem is often used as a reference problem for eigenvalue solvers. In this chapter the derivation of the matrix eigenvalue problem using the *finite element method* is described.

2.1 Fundamental equations

We consider the Laplace eigenvalue problem

$$\Delta u + \lambda u = 0 \quad u : \mathbb{R}^3 \to \mathbb{R} \qquad \text{in } \Omega \tag{2.1a}$$

$$u = 0 \qquad \text{on } \partial\Omega = \Gamma \tag{2.1b}$$

in an open, simply connected, bounded three dimensional domain Ω. Typically, we assume, that the boundary is composed of polygonal faces. The Problem (2.1) is going to be transformed into a matrix eigenvalue problem using the *finite element method* (FEM).

To apply the FEM, the Equations (2.1) have to be transformed to integral form. We use *Ritz-Galerkin's method* [65, page 45] to this end.

Galerkin's method and more generally the *method of weighted residuals* can be described as follows: The desired function u shall be approximated as a linear combination of suitably chosen linearly independent functions $N_1, \ldots, N_m$, of the form

$$\tilde{u} = \sum_{k=1}^{m} u_k N_k, \quad \mathbf{u} = [u_1, \ldots, u_m]. \tag{2.2}$$

The functions N_k satisfy the homogeneous boundary conditions and thus are equal to zero on the boundary Γ because of (2.1b). Therefore, the solution $\tilde{u}$ satisfies the boundary conditions for arbitrary coefficients u_k. We call the functions N_k the *global basis functions*, since they are defined on the whole domain Ω.

If Equation (2.2) is inserted in the differential Equation (2.1a) a residual $R := \Delta\tilde{u} - \lambda\tilde{u}$ is resulting. To solve the eigenvalue problem (2.1), we apply the *method of weighted residuals*, which requires that the integral over Ω of the residual R, weighted with certain weight functions, vanishes.

Equation (2.2) contains m parameters $u_1, \ldots, u_m$. Thus the condition can be formulated for m linearly independent weight functions. From the m equations the parameters u_k can be determined.

When using the *Ritz-Galerkin method* [17] the weight functions are chosen to be the basis functions $N_1, \ldots, N_m$. If the weight functions are chosen in this way, then the residual R is orthogonal to the subspace, which is spanned by $N_1, \ldots, N_m$. The resulting approximate solution $\tilde{u}$ is optimal with respect to this subspace [17].

According to Galerkin's method we get

$$\int_\Omega R N_j \,\mathrm{d}\Omega = \int_\Omega (\Delta\tilde{u} + \lambda\tilde{u}) N_j \,\mathrm{d}\Omega = 0 \qquad \forall j = 1 \ldots m. \tag{2.3}$$

In view of applying the FEM, it is crucial to eliminate the second derivatives [65, p. 46]. We apply Green's formula[18, p. 578] to (2.3) and obtain

$$\oint_\Gamma N_j \nabla\tilde{u} \cdot \mathbf{n} \,\mathrm{d}\Gamma - \int_\Omega \nabla\tilde{u} \cdot \nabla N_j \,\mathrm{d}\Omega + \int_\Omega \lambda\tilde{u} N_j \,\mathrm{d}\Omega = 0 \qquad \forall j = 1 \ldots m. \tag{2.4}$$

Since the global basis functions N_j satisfy the homogeneous boundary conditions, the surface integral in (2.4) is equal to zero.

The discretised integral form of (2.1) is then

$$\begin{aligned} &\textit{Find } \lambda \in \mathbb{R}, \tilde{u} \in \operatorname{span}\{N_1, \dots N_m\} \textit{ such that} \\ &\int_\Omega \nabla\tilde{u} \cdot \nabla N_j \,\mathrm{d}\Omega = \lambda \int_\Omega \tilde{u} N_j \,\mathrm{d}\Omega \qquad \forall j = 1 \dots m. \end{aligned} \tag{2.5}$$

By inserting (2.2) the following *generalised eigenvalue problem* is obtained:

$$\begin{aligned} &\textit{Find } (\lambda, \mathbf{u} = [u_1, \dots, u_m]) \in \mathbb{R} \times \mathbb{R}^m \textit{such that} \\ &\sum_{i=1}^{m} u_i \int_\Omega \nabla N_i \cdot \nabla N_j \,\mathrm{d}\Omega = \lambda \sum_{i=1}^{m} u_i \int_\Omega N_i N_j \,\mathrm{d}\Omega \qquad \forall j = 1 \dots m. \end{aligned} \tag{2.6}$$

Equation (2.6) is equivalent to the *matrix eigenvalue problem*

$$\boldsymbol{A}\mathbf{u} = \lambda \boldsymbol{M}\mathbf{u}, \tag{2.7}$$

with

$$a_{ij} = \int_\Omega \nabla N_i \cdot \nabla N_j \,\mathrm{d}\Omega \quad \text{and} \quad m_{ij} = \int_\Omega N_i N_j \,\mathrm{d}\Omega. \tag{2.8}$$

$\boldsymbol{A}$ is called the global *stiffness matrix* and $\boldsymbol{M}$ is called the global *mass matrix*.

From the equations for the matrix entries (2.8) it follows immediately, that both $\boldsymbol{A}$ and $\boldsymbol{M}$ are *symmetric*. $\boldsymbol{A}$ and $\boldsymbol{M}$ are *positive definite*. The quadratic forms $\mathbf{u}^T\boldsymbol{A}\mathbf{u}$ and $\mathbf{u}^T\boldsymbol{M}\mathbf{u}$ correspond to the integrals $\int_\Omega \nabla\tilde{u}\nabla\tilde{u} \,\mathrm{d}\Omega$ and $\int_\Omega \tilde{u}^2 \,\mathrm{d}\Omega$, both of which are positive for $\tilde{u} \neq 0 \wedge (\tilde{u} = 0 \text{ on } \Gamma)$. $\mathbf{u}$ is defined by

$$\mathbf{u} = [u_1, \dots, u_m] \quad \text{with} \quad \tilde{u} = \sum_{k=1}^{m} u_k N_k.$$

2.2 Exploiting symmetries

If the domain Ω is symmetric, the later calculation effort can be reduced, by cutting the domain along its symmetry plane. The differential equation is then solved only over one half of the domain. The boundary conditions at the symmetry plane are not definite. The condition $u = 0$ as well as the condition $\frac{\partial u}{\partial n} = 0$ lead to valid, symmetric solutions of Equation (2.1). Therefore *two* eigenvalue problems with different boundary conditions must be solved in order to obtain all solutions of the original problem. Should several symmetry planes exist, then the eigenvalue problem has to be solved for all possible combinations of boundary conditions. E.g. for three symmetry planes eight eigenvalue problems have to be solved. Nevertheless it is worthwhile exploiting the symmetry planes, because firstly, the eigenvalue problems are much smaller, and secondly, the smaller eigenvalue problems are better conditioned due to a better separation of the eigenvalues. Solving all small eigenvalue problems is more economic, than solving the original large eigenvalue problem.

2.3 Description of the finite elements

In this section we describe how the domain is discretised and how we choose the ansatz functions to approximate the solution in the tetrahedral elements T_e. We also show, how the local basis functions $N_i^{(e)}$ can easily be defined with the aid of affine transformations onto the unit tetrahedron T_0.

Discretisation of the domain For the practical solution of the eigenvalue problem (2.6) the domain Ω is divided into simple sub-domains, the so-called finite elements. This makes the calculation of the integrals feasible and also simplifies the definition of the global basis functions. In this work, only tetrahedral elements are used for discretising the domain Ω, because with them arbitrary domains can be approximated easily (cf. Figs. D.2, D.3 and D.5 in the appendix).

Following the partitioning of the domain into the elements the so-called *degrees of freedom* (DOFs) are numbered consecutively. We use the terms *global basis function* and *degree of freedom* interchangeably. Each DOF is also associated with one particular entry in the eigenvectors. The index of that entry corresponds to the index k of the global basis function N_k. Since for Problem I the DOFs correspond to function values at particular nodes, the DOFs are called node variables in this case.

Coordinate transformations The formal representation as well as the numerical computation of the local basis functions $N_j^{(e)}$ are simplified, if these are defined in the unit tetrahedron T_0 (Fig. 2.1). They can then be carried over to a general tetrahedron using an affine transformation. Furthermore, the evaluation of the integrals in (2.6) is much easier, if they are transformed to the unit tetrahedron T_0.

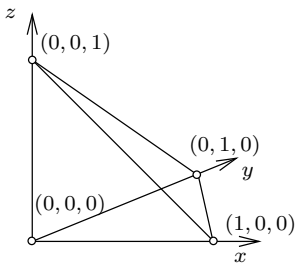


Figure 2.1: *Unit tetrahedron*

The affine transformation can be formulated through *simplex coordinates*[2] defined in the general tetrahedron shown in Fig. 2.2. In a general tetrahedron the location of a point P is determined through the four simplex coordinates L_1, L_2, L_3, L_4 [65, p 109]. Each simplex coordinate L_k is a linear function in x, y and z. Its value is 1 at the corner P_k and 0 at all other corners.

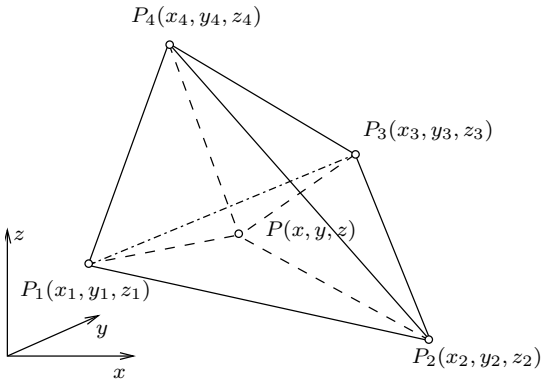


Figure 2.2: *Simplex coordinates in the tetrahedron*

[2]Simplex coordinates are often called *barycentric coordinates*.

The simplex coordinates are defined by

$$L_1 = \frac{V_1}{V}, \quad L_2 = \frac{V_2}{V}, \quad L_3 = \frac{V_3}{V} \quad \text{and} \quad L_4 = \frac{V_4}{V}, \tag{2.9}$$

where V_1 is the volume of the tetrahedron $PP_2P_3P_4$, V_2 the volume of the tetrahedron $PP_3P_4P_1$, V_3 the volume of tetrahedron $PP_4P_1P_2$, V_4 the volume of the tetrahedron $PP_1P_2P_3$, and V the volume of the entire tetrahedron with

$$V = V_1 + V_2 + V_3 + V_4. \tag{2.10}$$

From (2.9) and (2.10) we obtain

$$L_1 + L_2 + L_3 + L_4 = 1.$$

This volume can be calculated with the aid of the following determinants:

$$\begin{aligned}
V_1 &= \frac{1}{6}\begin{vmatrix} 1 & x & y & z \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}, \quad V_2 = \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x & y & z \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}, \\
V_3 &= \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x & y & z \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}, \quad V_4 = \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x & y & z \end{vmatrix}, \\
V &= \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}.
\end{aligned} \tag{2.11}$$

The relation between the basis functions $N^{(e)}$ defined in an arbitrary tetrahedron T_e and the basis functions $N^{(0)}$ defined in the unit tetrahedron T_0 is given by

$$N^{(e)}(x, y, z) = N^{(0)}(\xi(x, y, z), \eta(x, y, z), \zeta(x, y, z)),$$

with

$$\begin{aligned}
\xi(x, y, z) &= L_2 \\
\eta(x, y, z) &= L_3 \\
\zeta(x, y, z) &= L_4.
\end{aligned}$$

According to [65, p. 135] the equations for the back transformation are given by

$$\begin{aligned}
x &= x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\eta + (x_4 - x_1)\zeta \\
y &= y_1 + (y_2 - y_1)\xi + (y_3 - y_1)\eta + (y_4 - y_1)\zeta \\
z &= z_1 + (z_2 - z_1)\xi + (z_3 - z_1)\eta + (z_4 - z_1)\zeta.
\end{aligned} \tag{2.12}$$

Now let's define the ansatz functions in the unit tetrahedron. For our calculations we use two different forms: a tri-linear

$$u^{(0)}(\xi, \eta, \zeta) = c_0 + c_1\xi + c_2\eta + c_3\zeta \tag{2.13}$$

and a tri-quadratic one

$$u^{(0)}(\xi,\eta,\zeta) = c_0 + c_1\xi + c_2\eta + c_3\zeta + c_4\xi^2 + c_5\eta^2 + c_6\zeta^2 + c_7\xi\eta + c_8\xi\zeta + c_9\eta\zeta. \quad (2.14)$$

However, it is easier to describe the functions as linear combinations of so-called (local) basis functions. In this manner the continuity between two elements can easily be accomplished [64, page 58].

Linear basis functions in tetrahedra In the linear element, the node points are located at the four corners of the tetrahedron according to Fig. 2.3. The degrees of freedom (node variables) are the function values at the node points. As basis functions we are using interpolation functions with the following property:

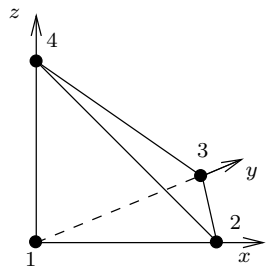


Figure 2.3: *Linear tetrahedral element*

$$\begin{aligned} N_i^{(0)}(\xi_j,\eta_j,\zeta_j) &\neq 0 \quad \text{for } j = i, \qquad \text{and} \\ N_i^{(0)}(\xi_j,\eta_j,\zeta_j) &= 0 \quad \text{for } j \neq i. \end{aligned} \quad (2.15)$$

The (ξ_j,η_j,ζ_j) are the coordinates at the node points. We define the basis functions in T_0, satisfying (2.15), by

$$\begin{aligned} N_1^{(0)}(\xi,\eta,\zeta) &= 1-\xi-\eta-\zeta \\ N_2^{(0)}(\xi,\eta,\zeta) &= \xi \\ N_3^{(0)}(\xi,\eta,\zeta) &= \eta \\ N_4^{(0)}(\xi,\eta,\zeta) &= \zeta. \end{aligned} \quad (2.16)$$

In an arbitrary finite element T_e, the basis functions are easiest expressed using the simplex coordinates defined in T_e,

$$\begin{aligned} N_1^{(e)}(x,y,z) &= L_1 \\ N_2^{(e)}(x,y,z) &= L_2 \\ N_3^{(e)}(x,y,z) &= L_3 \\ N_4^{(e)}(x,y,z) &= L_4. \end{aligned} \quad (2.17)$$

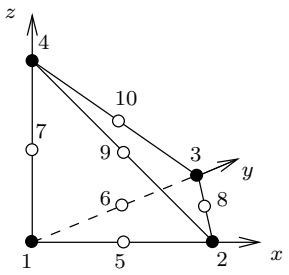


Figure 2.4: *Quadratic tetrahedral element*

Quadratic basis functions in tetrahedra For the quadratic tetrahedral element the six edge mid-points are used as node points in addition to the four corners (cf. Fig. 2.4). The node variables are again the function values in all node points. Therefore, a quadratic tetrahedral element has 10 node variables.

For the selection of the quadratic basis functions there are basically two possibilities to choose from:

1. *Interpolatory* basis functions according to condition (2.15)

2. *Hierarchical* basis functions [10],[9]: The linear basis functions $N_1^{(0)}$, $N_2^{(0)}$, $N_3^{(0)}$ and $N_4^{(0)}$ from (2.16) of the linear element are taken over and supplemented with quadratic basis functions $N_5^{(0)}, \dots N_{10}^{(0)}$, which are assigned to the edge mid-points. These six new basis functions satisfy the interpolation condition (2.15): Each one of the basis functions $N_5^{(0)}, \dots N_{10}^{(0)}$ has a positive value at its assigned edge mid-point and a zero value at all other edge middle points and all corners.

In the context of this work, we are using exclusively hierarchical basis functions, because we want to use their properties for preconditioning the eigenvalue system. Hierarchical basis preconditioning is described in Section 8.3.

The hierarchical basis functions defined in element T_e are given by

$$\begin{aligned} N_1^{(e)} &= L_1 & N_5^{(e)} &= L_1 L_2 \\ N_2^{(e)} &= L_2 & N_6^{(e)} &= L_1 L_3 \\ N_3^{(e)} &= L_3 & N_7^{(e)} &= L_1 L_4 \\ N_4^{(e)} &= L_4 & N_8^{(e)} &= L_2 L_3 \\ & & N_9^{(e)} &= L_2 L_4 \\ & & N_{10}^{(e)} &= L_3 L_4 \end{aligned} \tag{2.18}$$

One easily verifies that the quadratic functions $N_6^{(e)}, \dots, N_{10}^{(e)}$ satisfy the interpolation condition (2.15).

In the unit tetrahedron T_0, the basis functions $N_5^{(0)}, \ldots, N_{10}^{(0)}$ thus become

$$\begin{aligned} N_5^{(0)}(\xi,\eta,\zeta) &= \xi - \xi^2 - \xi\eta - \xi\zeta \\ N_6^{(0)}(\xi,\eta,\zeta) &= \eta - \eta^2 - \xi\eta - \eta\zeta \\ N_7^{(0)}(\xi,\eta,\zeta) &= \zeta - \zeta^2 - \xi\zeta - \eta\zeta \\ N_8^{(0)}(\xi,\eta,\zeta) &= \xi\eta \\ N_9^{(0)}(\xi,\eta,\zeta) &= \xi\zeta \\ N_{10}^{(0)}(\xi,\eta,\zeta) &= \eta\zeta. \end{aligned} \tag{2.19}$$

The basis functions $N_1^{(0)}, \ldots, N_4^{(0)}$ are the same as in (2.16).

2.4 Piece-wise representation of the solution

We consider the global representation of the solution function $\tilde{u}$ in the entire domain Ω, which is the union of all elements T_e. The function $\tilde{u}$ is composed of all functions $\tilde{u}^{(e)}$ defined in the elements T_e. Each of the global basis functions N_i is combined piece by piece from those local basis functions $N_k^{(e)}$, that are assigned to the node variable i. From this it becomes apparent, that N_i is equal to zero in all but those elements, which have node i in common. It is easily seen that the global basis functions N_i constructed in this way are continuous.

2.5 Calculation of the element matrices

In this paragraph we are dealing with the calculation of the integrals

$$\int_{T_e} N_i^{(e)} N_j^{(e)} \,\mathrm{d}\Omega \quad \text{and} \quad \int_{T_e} \nabla N_i^{(e)} \cdot \nabla N_j^{(e)} \,\mathrm{d}\Omega,$$

which are used for the calculation of the integrals over the entire domain Ω in (2.6). To compute the so-called element matrices these integrals must be evaluated for each element T_e. The entries of the so-called *mass element matrices* $\boldsymbol{M}^{(e)}$ and *stiffness element matrices* $\boldsymbol{A}^{(e)}$ are defined through

$$\begin{aligned} m_{i,j}^{(e)} &= \iiint_{T_e} N_i^{(e)} N_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \quad \text{and} \\ a_{i,j}^{(e)} &= \iiint_{T_e} \nabla N_i^{(e)} \cdot \nabla N_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \end{aligned} \tag{2.20}$$

To simplify the calculation, we formulate the integrals over T_e using integrals over T_0. We apply the coordinate transformation derived from paragraph 2.3.

According to (2.12) and by elementary rules of integral calculation, we replace the volume element $\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$ by

$$\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \begin{vmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{vmatrix} \mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta = \det \boldsymbol{J}\,\mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta.$$

The Jacobi matrix $\boldsymbol{J}$ contains all derivatives of x, y and z with respect to ξ, η and ζ. From (2.11) results, that the Jacobi determinant $J := \det \boldsymbol{J}$ can be calculated through $J = 6V$.

Mass element matrix The integrals for the mass element matrix can easily be calculated through

$$m_{i,j}^{(e)} = \iiint_{T_e} N_i^{(e)} N_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = J \iiint_{T_0} N_i^{(0)} N_j^{(0)} \,\mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta. \tag{2.21}$$

It is to be noted, that the integrals over T_0 have to be calculated only once. We store these integrals temporarily in an element matrix $\boldsymbol{M^{(0)}}$ with

$$m_{i,j}^{(0)} = \iiint_{T_0} N_i^{(0)} N_j^{(0)} \,\mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta. \tag{2.22}$$

The mass element matrices then are calculated from

$$\boldsymbol{M}^{(e)} = J\boldsymbol{M}^{(0)}.$$

The matrix $\boldsymbol{M}^{(0)}$ is printed in Appendix F.1.

Stiffness element matrix For the integrals required for the calculation of the entries of the stiffness element matrix $\boldsymbol{A}^{(e)}$ we are using the generalised chain rule [18, page 278] to convert the derivatives:

$$\begin{aligned}
\frac{\partial N^{(e)}}{\partial x} &= \frac{\partial N^{(0)}}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N^{(0)}}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial N^{(0)}}{\partial \zeta}\frac{\partial \zeta}{\partial x} \\
\frac{\partial N^{(e)}}{\partial y} &= \frac{\partial N^{(0)}}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N^{(0)}}{\partial \eta}\frac{\partial \eta}{\partial y} + \frac{\partial N^{(0)}}{\partial \zeta}\frac{\partial \zeta}{\partial y} \\
\frac{\partial N^{(e)}}{\partial z} &= \frac{\partial N^{(0)}}{\partial \xi}\frac{\partial \xi}{\partial z} + \frac{\partial N^{(0)}}{\partial \eta}\frac{\partial \eta}{\partial z} + \frac{\partial N^{(0)}}{\partial \zeta}\frac{\partial \zeta}{\partial z}.
\end{aligned} \tag{2.23}$$

The derivatives of ξ, η and ζ with respect to x, y and z are given by

$$\begin{aligned}
\frac{\partial \xi}{\partial x} &= \frac{1}{J}\begin{vmatrix} z_3 - z_2 & y_3 - y_2 \\ z_4 - z_2 & y_4 - y_2 \end{vmatrix}, & \frac{\partial \xi}{\partial y} &= \frac{1}{J}\begin{vmatrix} x_3 - x_2 & z_3 - z_2 \\ x_4 - x_2 & z_4 - z_2 \end{vmatrix}, \\
\frac{\partial \xi}{\partial z} &= \frac{1}{J}\begin{vmatrix} y_3 - y_2 & x_3 - x_2 \\ y_4 - y_2 & x_4 - x_2 \end{vmatrix}, & & \\
\frac{\partial \eta}{\partial x} &= \frac{1}{J}\begin{vmatrix} y_3 - y_1 & z_3 - z_1 \\ y_4 - y_1 & z_4 - z_1 \end{vmatrix}, & \frac{\partial \eta}{\partial y} &= \frac{1}{J}\begin{vmatrix} z_3 - z_1 & x_3 - x_1 \\ z_4 - z_1 & x_4 - x_1 \end{vmatrix}, \\
\frac{\partial \eta}{\partial z} &= \frac{1}{J}\begin{vmatrix} x_3 - x_1 & y_3 - x_1 \\ x_4 - x_1 & y_4 - x_1 \end{vmatrix}, & & \\
\frac{\partial \zeta}{\partial x} &= \frac{1}{J}\begin{vmatrix} z_2 - z_1 & y_2 - y_1 \\ z_4 - z_1 & y_4 - y_1 \end{vmatrix}, & \frac{\partial \zeta}{\partial y} &= \frac{1}{J}\begin{vmatrix} x_2 - x_1 & z_2 - z_1 \\ x_4 - x_1 & z_4 - z_1 \end{vmatrix}, \\
\frac{\partial \zeta}{\partial z} &= \frac{1}{J}\begin{vmatrix} y_2 - y_1 & x_2 - x_1 \\ y_4 - y_1 & x_4 - x_1 \end{vmatrix}. & &
\end{aligned} \tag{2.24}$$

In order to simplify the writing of the following equations we define the coordinate vector

$$\boldsymbol{\xi} = (\xi, \eta, \zeta).$$

For further transformations the scalar product is expanded:

$$\begin{aligned}
a_{ij}^{(e)} &= \iiint_{T_e} \nabla N_i^{(e)} \cdot \nabla N_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\
&= \iiint_{T_e} \frac{\partial N_i^{(e)}}{\partial x}\frac{\partial N_j^{(e)}}{\partial x} + \frac{\partial N_i^{(e)}}{\partial y}\frac{\partial N_j^{(e)}}{\partial y} + \frac{\partial N_i^{(e)}}{\partial z}\frac{\partial N_j^{(e)}}{\partial z} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z.
\end{aligned}$$

By inserting (2.23) and adapting the volume element we get an expression with integrals over T_0,

$$a_{ij}^{(e)} = \sum_{k,l=1}^{3} \iiint_{T_0} \frac{\partial N_i^{(0)}}{\partial \xi_k} \frac{\partial N_j^{(0)}}{\partial \xi_l} \, \mathrm{d}\boldsymbol{\xi} \, J \left[\frac{\partial \xi_k}{\partial x} \frac{\partial \xi_l}{\partial x} + \frac{\partial \xi_k}{\partial y} \frac{\partial \xi_l}{\partial y} + \frac{\partial \xi_k}{\partial z} \frac{\partial \xi_l}{\partial z} \right].$$

For an efficient calculation, we store the integrals that are independent of elements in the auxiliary matrices $\boldsymbol{K}_{\boldsymbol{\xi}}^{11}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{22}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{33}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{12}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{13}$ and $\boldsymbol{K}_{\boldsymbol{\xi}}^{23}$, with

$$k_{ij}^{kl} = \iiint_{T_0} \frac{\partial N_i^{(0)}}{\partial \xi_k} \frac{\partial N_j^{(0)}}{\partial \xi_l} \, \mathrm{d}\boldsymbol{\xi}. \tag{2.25}$$

The stiffness element matrix $\boldsymbol{A}^{(e)}$ is then calculated from

$$\boldsymbol{A}^{(e)} = \sum_{k,l=1}^{3} \boldsymbol{K}_{\boldsymbol{\xi}}^{kl} \, J \left[\frac{\partial \xi_k}{\partial x} \frac{\partial \xi_l}{\partial x} + \frac{\partial \xi_k}{\partial y} \frac{\partial \xi_l}{\partial y} + \frac{\partial \xi_k}{\partial z} \frac{\partial \xi_l}{\partial z} \right]. \tag{2.26}$$

Thereby

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{kl} = \boldsymbol{K}_{\boldsymbol{\xi}}^{lk^T}$$

holds. The matrices $\boldsymbol{K}_{\boldsymbol{\xi}}^{11}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{22}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{33}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{23}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{13}$ and $\boldsymbol{K}_{\boldsymbol{\xi}}^{23}$ are listed in Appendix F.1.

2.6 Construction of the global matrices

This paragraph describes how the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ of the eigenvalue problem (2.7) are constructed.

Matrix Assembly The process of constructing the global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ from the element matrices $\boldsymbol{A}^{(e)}$ and $\boldsymbol{M}^{(e)}$ is called *assembly*.

The degrees of freedom are numbered globally from $1 \ldots m$. These global numbers correspond to the row and column indices of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$.

Apart from the global numbering, the degrees of freedom are also numbered consecutively element-wise (locally) from $1 \ldots 4$ for linear elements, resp. from $1 \ldots 10$ for quadratic elements. For the assembly of the global matrices the mappings $\tau^{(e)}$ from the local numbering to the global numbering is used as follows:

For each element T_e the entries of the element matrices $\boldsymbol{A}^{(e)}$ and $\boldsymbol{M}^{(e)}$ are added up according to mappings $\tau^{(e)}$ into the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$. The individual contributions $\int_{T_e} \nabla N_i^{(e)} \cdot \nabla N_j^{(e)} \, \mathrm{dx}$ and $\int_{T_e} N_i^{(e)} N_j^{(e)} \, \mathrm{dx}$ are accumulated until finally the desired integrals $\int_\Omega \nabla N_i \cdot \nabla N_j \, \mathrm{dx}$ and $\int_\Omega N_i N_j \, \mathrm{dx}$ reside in the matrix entries.

This procedure is best illustrated using an algorithm:

$\boldsymbol{A} = \boldsymbol{0}$; $\boldsymbol{M} = \boldsymbol{0}$
for e over all elements
  for i_{loc} = 1:n_{elem}
    for j_{loc} = 1:n_{elem}
      $i = \tau^{(e)}(i_{\text{loc}})$; $j = \tau^{(e)}(j_{\text{loc}})$
      $\boldsymbol{A}(i,j) = \boldsymbol{A}(i,j) + \boldsymbol{A}^{(e)}(i_{\text{loc}}, j_{\text{loc}})$
      $\boldsymbol{M}(i,j) = \boldsymbol{M}(i,j) + \boldsymbol{M}^{(e)}(i_{\text{loc}}, j_{\text{loc}})$
    end for
  end for
end for

Here n_{elem} is the number of local DOFs, which 4 for linear and 10 for quadratic elements.

When using *quadratic elements* the global numbering is chosen in such a way, that firstly, the linear degrees of freedom (associated with the corners of the tetrahedra) get the smaller numbers than the quadratic degrees of freedom (associated with the edge mid-points), and secondly the numbering of the degrees of freedom at the corners of the tetrahedra are identical with their numbering when using linear elements. In this way the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ have a 2×2-block structure $\left(\begin{smallmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{smallmatrix}\right)\mathbf{u} = \lambda\left(\begin{smallmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{smallmatrix}\right)\mathbf{u}$, whereas the matrices $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$ correspond to the global matrices when using the linear elements.

Treatment of the boundary conditions The matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ have been assembled according to the above description without consideration of the boundary conditions. The boundary condition (2.1b) implies, that the function u vanishes at the boundary of domain Ω. This then indicates, that the DOFs u_i at nodal points on the boundary must take on the value zero. This amounts to eliminating the respective rows and columns of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$.

Storage format for matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ Since the global basis functions N_i are different from zero only in a local area, the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$, calculated according to (2.8), must have many zero elements.

It is easily seen, that for regular grids, the number of non- zero elements per line is bounded by a small constant, that is independent of the mesh size. Thus, the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are *sparse*.

Since the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are sparse and are symmetric, we are using the *Symmetric Sparse Skyline* (SSS) format [59], which stores all non-zero entries of the lower triangle of the matrix. A detailed description of the SSS data structure is given in Appendix A.3.

Practical implications with regard to the implementation During the assembly process it is not feasible to store the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ in SSS format, because then it would be very expensive to add new non-zero entries. We use the LL format, which is described in detail in Appendix A.4. The LL format, which stores every matrix row in a linked list of non-zero entries, has important advantages over the SSS format: Firstly, adding and removing elements is cheap, and secondly, the total number of non-zero entries does not have to be known beforehand. After the assembly the LL format can be converted cheaply to the SSS format in a 1-pass scheme.

For the treatment of the boundary conditions it is advantageous to identify the rows and columns that have to be eliminated prior to the assembly. This information can then be taken into consideration when assembling the matrices, both to save storing space and to prevent unnecessary computation.

3 Problem II: Electromagnetic waves in cavities

This chapter describes the second mathematical problem, which is being dealt with in this work: For the design of particle accelerators, electromagnetic fields in cavities need to be calculated. Using the finite element method, a modified form of Maxwell's equations is transformed to a matrix eigenvalue problem.

Dr. Stefan Adam, Paul Scherrer Institute (PSI), introduced us to this problem. In collaboration with Dr. Adam various approaches to the solution have been worked out.

3.1 Fundamental Equations

As already outlined in the introduction, the mathematical model for the electromagnetic fields in accelerator cavities is an eigenvalue problem derived from Maxwell's equations.

For our calculations it is assumed that the metallic surface has perfect conductivity, and that the interior of the cavity is in perfect vacuum. Under these assumptions the electromagnetic field can be described by the following (simplified) Maxwell equations [43, p.353], [48], [21, §I.4],

$$-\frac{1}{c}\frac{\partial}{\partial t}\mathbf{E}(\mathbf{x},t) + \mathbf{rot}\,\mathbf{H}(\mathbf{x},t) = \mathbf{0}, \tag{3.1a}$$

$$\operatorname{div}\mathbf{E}(\mathbf{x},t) = 0, \qquad \mathbf{x}\in\Omega, \quad t>0, \tag{3.1b}$$

$$\frac{1}{c}\frac{\partial}{\partial t}\mathbf{H}(\mathbf{x},t) + \mathbf{rot}\,\mathbf{E}(\mathbf{x},t) = \mathbf{0}, \tag{3.1c}$$

$$\operatorname{div}\mathbf{H}(\mathbf{x},t) = 0, \tag{3.1d}$$

where $\mathbf{E}$ represents the electric and $\mathbf{H}$ the magnetic field and c the speed of light. The boundary conditions result from the perfect conductivity of the surface and demand that the tangential components of $\mathbf{E}$ and the normal components of $\mathbf{H}$ disappear, therefore

$$\mathbf{n}\times\mathbf{E} = \mathbf{0}, \quad \mathbf{n}\cdot\mathbf{H} = 0, \quad \mathbf{x}\in\Gamma. \tag{3.2}$$

The vector $\mathbf{n}$ is the outer normal to Γ.

By means of a time-harmonic ansatz for $\mathbf{E}$ and $\mathbf{H}$ and by eliminating $\mathbf{H}$, one obtains the differential equation [1]

$$\mathbf{rot}\,\mathbf{rot}\,\mathbf{e}(\mathbf{x}) = \lambda\,\mathbf{e}(\mathbf{x}), \qquad \mathbf{x}\in\Omega, \qquad \lambda := \omega^2/c^2, \tag{3.3a}$$

$$\operatorname{div}\mathbf{e}(\mathbf{x}) = 0, \qquad \mathbf{x}\in\Omega, \tag{3.3b}$$

$$\mathbf{n}\times\mathbf{e} = \mathbf{0}, \qquad \mathbf{x}\in\Gamma. \tag{3.3c}$$

$\mathbf{e}(\mathbf{x})$ designates the spatial component of $\mathbf{E}(\mathbf{x},t)$ and represents the amplitude of the electric field at position $\mathbf{x}$. Solutions to (3.3) exist only for certain eigenfrequencies ω.

As described in Chapter 2, we form the solution as a linear combination of the global basis functions $\mathbf{N}_1, \ldots, \mathbf{N}_m$,

$$\mathbf{e} = \sum_{k=1}^{m} u_k \mathbf{N}_k. \tag{3.4}$$

The $\mathbf{N}_k$ satisfy the boundary conditions

$$\mathbf{n}\times\mathbf{N}_k = \mathbf{0}, \quad \mathbf{x}\in\Gamma,$$

but $\operatorname{div} \mathbf{N}_k \neq 0$ is possible.

Equation (3.3a) is transformed according to Galerkin's method into a discretised integral equation

$$\int_\Omega (\mathbf{rot}\,\mathbf{rot}\,\mathbf{e} - \lambda \mathbf{e}) \cdot \mathbf{N}_j \,\mathrm{d}\Omega = 0 \qquad \forall j = 1 \ldots m. \tag{3.5}$$

With the aid of the Nabla calculus [18, page 576] the following equivalence is derived [18, page 577, rule 3]

$$(\mathbf{rot}\,\mathbf{rot}\,\mathbf{u}) \cdot \mathbf{v} = (\mathbf{rot}\,\mathbf{u}) \cdot (\mathbf{rot}\,\mathbf{v}) + \operatorname{div}((\mathbf{rot}\,\mathbf{u}) \times \mathbf{v}). \tag{3.6}$$

According to Green's 2nd theorem [18, page 579] and (3.6) the double derivatives are eliminated from (3.5),

$$\int_\Omega (\mathbf{rot}\,\mathbf{e}) \cdot (\mathbf{rot}\,\mathbf{N}_j) \,\mathrm{d}\Omega + \oint_\Gamma [(\mathbf{rot}\,\mathbf{e}) \times \mathbf{N}_j] \cdot \mathbf{n} \,\mathrm{d}\Gamma - \lambda \int_\Omega \mathbf{e} \cdot \mathbf{N}_j \,\mathrm{d}\Omega = 0 \qquad \forall j = 1 \ldots m. \tag{3.7}$$

The surface integral in (3.7) disappears, since $\mathbf{N}_j$ satisfies the boundary conditions $\mathbf{N}_j \times \mathbf{n} = \mathbf{0}$ and therefore vector $(\mathbf{rot}\,\mathbf{e}) \times \mathbf{N}_j$ is orthogonal to the outer normal vector $\mathbf{n}$.

A simplistic approach for solving (3.3) is

$$\begin{aligned} &\textit{Find } \lambda \in \mathbb{R}, \mathbf{e} \in \operatorname{span}\{\mathbf{N}_1, \ldots \mathbf{N}_m\} \textit{ such that} \\ &\int_\Omega (\mathbf{rot}\,\mathbf{e}) \cdot (\mathbf{rot}\,\mathbf{N}_j) \,\mathrm{d}\Omega = \lambda \int_\Omega \mathbf{e} \cdot \mathbf{N}_j \,\mathrm{d}\Omega \qquad \forall j = 1 \ldots m. \end{aligned} \tag{3.8}$$

If the finite elements $\mathbf{N}_j$ would be divergence-free, then (3.8) would be a valid approximation for (3.3). It is however difficult to implement the divergence-free condition practically [46].

A simplistic procedure would be to use ordinary node based finite elements and to omit the divergence-free condition. One would then get the desired divergence-free eigensolutions, but also undesired solutions with non-vanishing divergence.

These undesired solutions (so-called spurious modes) are spread over the entire spectrum [38] and therefore cannot be identified through their eigenfrequencies. However, it is possible to eliminate these spurious modes on the basis of the divergence of their eigenfunctions. For this purpose, for all eigensolutions $(\lambda_i, \mathbf{e}_i)$ the integral $\int_\Omega (\operatorname{div} \mathbf{e})^2$ is calculated. Using a threshold value the undesired solutions are then ruled out [1].

The procedure described above has several disadvantages: Many of the computed eigensolutions are discarded after-wards. Also, the method is mathematically not well established. The method worked in preliminary experiments, but its reliability is doubtful. Furthermore, the choice of the threshold value is delicate.

For these reasons, we did not further investigate this method. In Sections 3.2 and 3.3 we now present two methods, by which the calculation of *spurious modes* is prevented completely.

3.2 Penalty Method

The penalty method is a variant to introduce the divergence-free condition in (3.8) [46][1][45]. By inserting (3.4) into the integral in (3.8) and by adding a penalty term we get the *generalised*

eigenvalue problem

$$\begin{aligned} &\textit{Find } (\lambda, \mathbf{u} = [u_1, \ldots, u_m]) \in \mathbb{R} \times \mathbb{R}^m \textit{ such that} \\ &\sum_{i=1}^{m} u_i \int_\Omega \mathbf{rot}\, \mathbf{N}_i \cdot \mathbf{rot}\, \mathbf{N}_j \,\mathrm{d}\Omega + s \sum_{i=1}^{m} u_i \int_\Omega \operatorname{div} \mathbf{N}_i \operatorname{div} \mathbf{N}_j \,\mathrm{d}\Omega \\ &\qquad = \lambda \sum_{i=1}^{m} u_i \int_\Omega \mathbf{N}_i \cdot \mathbf{N}_j \,\mathrm{d}\Omega \quad \forall j = 1 \ldots m. \end{aligned} \tag{3.9}$$

The penalty term, which is used to complement the discretised weak formulation (3.8), only affects eigensolutions, that are not divergence-free and increases their eigenvalue. If the parameter $s > 0$ is chosen large enough, the first few eigensolutions with the smallest eigenvalues are guaranteed to be divergence-free.

It is possible to estimate, how large s has to be chosen at least, such that all calculated solutions, whose eigenvalue are smaller than the given value ν, are guaranteed to be divergence-free [1]. For this, a lower bound of the smallest eigenvalue $\mu_{\min}(\Omega)$ of the negative Laplace operator $-\Delta$ on Ω needs to be computed (cf. Problem I). Then $s\,\mu_{\min}(\Omega) \geq \nu$ must hold [51].

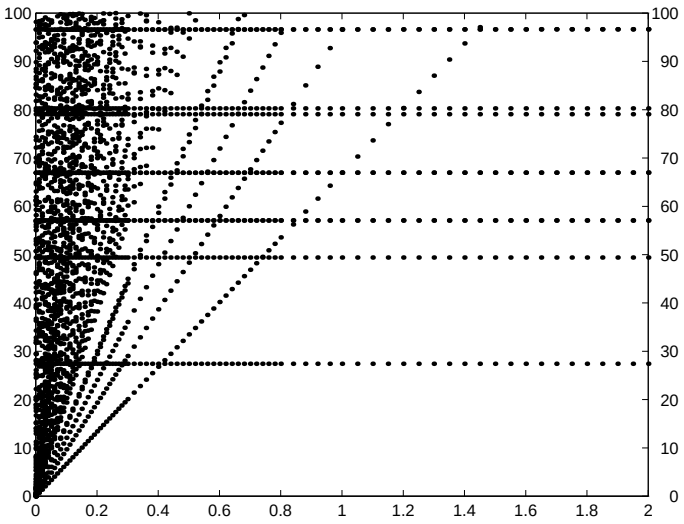


Figure 3.1: *Eigenvalues of* box8x4x6 *in the interval (0,100) computed using the penalty method for* $0 \leq s \leq 2$.

Fig. 3.1 shows the eigenvalues of the mesh *box8x4x6* (cf. Appendix D) in the interval $(0, 100)$, which were computed using the penalty method. The penalty parameter s varies from 0 to 2 on the x-axis. For small s the interval is contaminated with eigenvalues, that belong to divergence-afflicted solutions. These eigenvalues grow linearly with s, while the desired eigenvalues remain constant, i.e. are independent of s. If s is chosen sufficiently large, then the undesired eigenvalues disappear from the interval $(0, 100)$. If e.g. the eigenvalues in the interval of $(0, 100)$ are to be calculated, then $s = 1.5$ is an appropriate selection: In accordance with Appendix C.1 the smallest eigenvalue of the negative Laplace operator is $\mu_{\min}(\omega) = \frac{61}{9}\pi^2$ and $1.5\mu_{\min}(\omega) \approx 100.34$. All spurious modes are therefore larger than 100.34.

Equation (3.9) is equivalent to the *matrix eigenvalue problem*

$$\boldsymbol{A}\mathbf{u} = \lambda \boldsymbol{M}\mathbf{u}, \tag{3.10}$$

with

$$\begin{aligned} a_{ij} &= \int_\Omega \mathbf{rot}\,\mathbf{N}_i \cdot \mathbf{rot}\,\mathbf{N}_j + s\,\mathrm{div}\,\mathbf{N}_i\,\mathrm{div}\,\mathbf{N}_j\,\mathrm{d}\Omega \quad \text{and} \\ m_{ij} &= \int_\Omega \mathbf{N}_i \cdot \mathbf{N}_j\,\mathrm{d}\Omega. \end{aligned} \tag{3.11}$$

From (3.11) follows immediately, that both matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are *symmetric*.

The quadratic forms $\mathbf{u}^T\boldsymbol{A}\mathbf{u}$ and $\mathbf{u}^T\boldsymbol{M}\mathbf{u}$ match the integrals $\int_\Omega(\mathbf{rot}\,\tilde{\mathbf{e}})^2 + s(\mathrm{div}\,\tilde{\mathbf{e}})^2\,\mathrm{d}\Omega$ and $\int_\Omega \tilde{\mathbf{e}}^2\,\mathrm{d}\Omega$. $\mathbf{u}$ is defined through $\mathbf{u} = [u_1, \ldots, u_m]$ with $\tilde{\mathbf{e}} = \sum_{k=1}^m u_k\mathbf{N}_k$.

The integral $\int_\Omega \tilde{\mathbf{e}}^2\,\mathrm{d}\Omega$ is positive for $\tilde{\mathbf{e}} \neq \mathbf{0}$. Therefore $\boldsymbol{M}$ must be *positive definite*. As shown in [35, remark 3.9 and 3.10, page 47], the only function, which satisfies $\mathrm{div}\,\tilde{\mathbf{e}} = 0$ and $\mathbf{rot}\,\tilde{\mathbf{e}} = \mathbf{0}$ and the boundary conditions, is the function $\tilde{\mathbf{e}} = \mathbf{0}$. In other words: a non-disappearing, divergence-free function $\tilde{\mathbf{e}}$, which meets the boundary conditions, can not be curl-free. Therefore, the integral

$$\int_\Omega (\mathbf{rot}\,\tilde{\mathbf{e}})^2 + s(\mathrm{div}\,\tilde{\mathbf{e}})^2\,\mathrm{d}\Omega$$

is positive for $s > 0$ and hence $\boldsymbol{A}$ is *positive definite*.

3.2.1 Finite Elements for the Penalty Method

For the finite element formulation of the penalty method, we are using *node based tetrahedral elements*, as described in Chapter 2. In contrast to earlier times, the basis functions are now vector functions.

For the definition of the basis functions and their integration, we are working again in the unit tetrahedron T_0 and with the coordinate transformation introduced in Section 2.3.

We are now introducing two tetrahedral elements: one with linear basis functions and another with quadratic basis functions. The basis functions are set up *hierarchically* as in Problem I. As node points, we are again using the corners and the edge mid-points (cf. Figs. in Section 2.3). To each node point three node variables are assigned: the three field components in each coordinate direction. Thus, for the linear element, the vector valued basis functions in the tetrahedron T_0 are defined as follows:

$$\mathbf{N}_i^{(0)} = \begin{pmatrix} N_i^{(0)} \\ 0 \\ 0 \end{pmatrix},\ \mathbf{N}_{i+4}^{(0)} = \begin{pmatrix} 0 \\ N_i^{(0)} \\ 0 \end{pmatrix},\ \mathbf{N}_{i+8}^{(0)} = \begin{pmatrix} 0 \\ 0 \\ N_i^{(0)} \end{pmatrix}, \quad i = 1, \ldots 4. \tag{3.12}$$

The basis functions for the quadratic element are defined as

$$\mathbf{N}_i^{(0)} = \begin{pmatrix} N_i^{(0)} \\ 0 \\ 0 \end{pmatrix},\ \mathbf{N}_{i+10}^{(0)} = \begin{pmatrix} 0 \\ N_i^{(0)} \\ 0 \end{pmatrix},\ \mathbf{N}_{i+20}^{(0)} = \begin{pmatrix} 0 \\ 0 \\ N_i^{(0)} \end{pmatrix}, \quad i = 1, \ldots 10. \tag{3.13}$$

The N_i are the scalar hierarchical basis functions from (2.16) and (2.19) respectively, which were defined for Problem I.

3.2.2 Element matrices

For the calculation of the integrals in (3.11), mass element matrices $\boldsymbol{M}^{(e)}$, curl element matrices $\boldsymbol{A}^{(e)}$ and divergence element matrices $\boldsymbol{N}^{(e)}$ are used. They are defined as follows:

$$m_{i,j}^{(e)} = \iiint_{T_e} \mathbf{N}_i^{(e)} \cdot \mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z, \quad \text{and} \tag{3.14a}$$

$$a_{i,j}^{(e)} = \iiint_{T_e} \mathbf{rot}\, \mathbf{N}_i^{(e)} \cdot \mathbf{rot}\, \mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z, \quad \text{and} \tag{3.14b}$$

$$n_{i,j}^{(e)} = \iiint_{T_e} \operatorname{div} \mathbf{N}_i^{(e)} \operatorname{div} \mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \tag{3.14c}$$

Mass element matrix In the computation of the mass element matrix $\boldsymbol{M}^{(e)}$ the basis functions corresponding to different coordinate directions do not interact. Therefore the mass matrix has the form

$$\boldsymbol{M}^{(e)} = J \begin{pmatrix} \boldsymbol{M}^{(0)} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{M}^{(0)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{M}^{(0)} \end{pmatrix}, \tag{3.15}$$

where $\boldsymbol{M}^{(0)}$ is the element matrix defined in (2.22).

Auxiliary matrices To facilitate the calculation of the curl element matrices and the divergence element matrices, we introduce some auxiliary matrices.

In order to simplify the notation, we are using the coordinate vectors

$$\mathbf{x} = (x, y, z) \quad \text{and} \quad \boldsymbol{\xi} = (\xi, \eta, \zeta)$$

as in Chapter 2.

We define the matrices $\boldsymbol{K}_{\mathbf{x}}^{11}$, $\boldsymbol{K}_{\mathbf{x}}^{22}$, $\boldsymbol{K}_{\mathbf{x}}^{33}$, $\boldsymbol{K}_{\mathbf{x}}^{12}$, $\boldsymbol{K}_{\mathbf{x}}^{13}$ UN $\boldsymbol{K}_{\mathbf{x}}^{23}$, which contain the integrals over T_e. Their entries are calculated according to the formula

$$k_{\mathbf{x},ij}^{kl} = \iiint_{T_e} \frac{\partial N_i^{(e)}}{\partial x_k} \frac{\partial N_j^{(e)}}{\partial x_l} \,\mathrm{d}\mathbf{x}, \quad 1 \le l \le k \le 3. \tag{3.16}$$

Using the generalised chain rule [18, page 278] the matrices $\boldsymbol{K}_{\mathbf{x}}^{11}$, $\boldsymbol{K}_{\mathbf{x}}^{22}$, $\boldsymbol{K}_{\mathbf{x}}^{33}$, $\boldsymbol{K}_{\mathbf{x}}^{12}$, $\boldsymbol{K}_{\mathbf{x}}^{13}$ and $\boldsymbol{K}_{\mathbf{x}}^{23}$ can be calculated from the element independent matrices $\boldsymbol{K}_{\boldsymbol{\xi}}^{11}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{22}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{33}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{12}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{13}$ and $\boldsymbol{K}_{\boldsymbol{\xi}}^{23}$, which were defined in (2.25). First, we define the Jacobi matrix $\boldsymbol{\Upsilon}$ with the derivatives of ξ, η and ζ with respect to x, y and z,

$$\boldsymbol{\Upsilon} = \begin{pmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} & \frac{\partial \xi}{\partial z} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} & \frac{\partial \eta}{\partial z} \\ \frac{\partial \zeta}{\partial x} & \frac{\partial \zeta}{\partial y} & \frac{\partial \zeta}{\partial z} \end{pmatrix}. \tag{3.17}$$

The elements of $\boldsymbol{\Upsilon}$

$$\upsilon_{ij} = \frac{\partial \xi_i}{\partial x_j}, \quad 1 \le i, j \le 3$$

are constant and are resulting from (2.24). $\boldsymbol{\Upsilon}$ satisfies

$$\boldsymbol{\Upsilon}\,\mathrm{d}\mathbf{x} = \mathrm{d}\boldsymbol{\xi} \quad \text{and} \quad \boldsymbol{J}\,\mathrm{d}\boldsymbol{\xi} = \mathrm{d}\mathbf{x} \quad \text{and therefore} \quad \boldsymbol{\Upsilon} = \boldsymbol{J}^{-1}.$$

The matrices $\boldsymbol{K}_{\mathbf{x}}^{kl}$ can now be calculated from linear combinations of the $\boldsymbol{K}_{\boldsymbol{\xi}}^{rs}$

$$\boldsymbol{K}_{\mathbf{x}}^{kl} = \sum_{1 \le r,s \le 3} \upsilon_{rk} \upsilon_{sl} \boldsymbol{K}_{\boldsymbol{\xi}}^{rs}, \quad 1 \le l \le k \le 3. \tag{3.18}$$

Curl element matrices The curl element matrix $\boldsymbol{A}^{(e)}$ is defined in (3.14b). The integrand in (3.14b) is expanded as follows:

$$\begin{aligned}\mathbf{rot}\,\mathbf{N}_i\,\mathbf{rot}\,\mathbf{N}_j = &\frac{\partial n_{i,x}}{\partial y}\frac{\partial n_{j,x}}{\partial y} + \frac{\partial n_{i,x}}{\partial z}\frac{\partial n_{j,x}}{\partial z} - \frac{\partial n_{i,x}}{\partial y}\frac{\partial n_{j,y}}{\partial x} - \frac{\partial n_{i,x}}{\partial z}\frac{\partial n_{j,z}}{\partial x}\\ &+ \frac{\partial n_{i,y}}{\partial x}\frac{\partial n_{j,y}}{\partial x} + \frac{\partial n_{i,y}}{\partial z}\frac{\partial n_{j,y}}{\partial z} - \frac{\partial n_{i,y}}{\partial x}\frac{\partial n_{j,x}}{\partial y} - \frac{\partial n_{i,y}}{\partial z}\frac{\partial n_{j,z}}{\partial y}\\ &+ \frac{\partial n_{i,z}}{\partial x}\frac{\partial n_{j,z}}{\partial x} + \frac{\partial n_{i,z}}{\partial y}\frac{\partial n_{j,z}}{\partial y} - \frac{\partial n_{i,z}}{\partial x}\frac{\partial n_{j,x}}{\partial z} - \frac{\partial n_{i,z}}{\partial y}\frac{\partial n_{j,y}}{\partial z}.\end{aligned}$$

The $n_{i,x}$, $n_{i,y}$, $n_{i,z}$ are the components of the basis functions $\mathbf{N}_i^{(e)}$.

Due to the definition of the basis functions $\mathbf{N}_i^{(e)}$ in (3.12) and (3.13) most of the terms are zero. The curl element matrix $\boldsymbol{A}^{(e)}$ also has a 3×3-block structure,

$$\boldsymbol{A}^{(e)} = J\begin{pmatrix}\boldsymbol{K}_{\mathbf{x}}^{22}+\boldsymbol{K}_{\mathbf{x}}^{33} & -\boldsymbol{K}_{\mathbf{x}}^{12} & -\boldsymbol{K}_{\mathbf{x}}^{13}\\ -\boldsymbol{K}_{\mathbf{x}}^{21} & \boldsymbol{K}_{\mathbf{x}}^{11}+\boldsymbol{K}_{\mathbf{x}}^{33} & -\boldsymbol{K}_{\mathbf{x}}^{23}\\ -\boldsymbol{K}_{\mathbf{x}}^{31} & -\boldsymbol{K}_{\mathbf{x}}^{32} & \boldsymbol{K}_{\mathbf{x}}^{11}+\boldsymbol{K}_{\mathbf{x}}^{22}\end{pmatrix} \tag{3.19}$$

with

$$\boldsymbol{K}_{\mathbf{x}}^{lk} = \boldsymbol{K}_{\mathbf{x}}^{kl\,T}, \qquad 1\le l\le k\le 3.$$

Divergence element matrix As for the curl element matrix the integrand of the divergence element matrix $\boldsymbol{N}^{(e)}$ in (3.14c) can be expanded:

$$\begin{aligned}\operatorname{div}\mathbf{N}_i\operatorname{div}\mathbf{N}_j = &\frac{\partial n_{i,x}}{\partial x}\frac{\partial n_{j,x}}{\partial x} + \frac{\partial n_{i,x}}{\partial x}\frac{\partial n_{j,y}}{\partial y} + \frac{\partial n_{i,x}}{\partial x}\frac{\partial n_{j,z}}{\partial z}\\ &+ \frac{\partial n_{i,y}}{\partial y}\frac{\partial n_{j,x}}{\partial x} + \frac{\partial n_{i,y}}{\partial y}\frac{\partial n_{j,y}}{\partial y} + \frac{\partial n_{i,y}}{\partial y}\frac{\partial n_{j,z}}{\partial z}\\ &+ \frac{\partial n_{i,z}}{\partial z}\frac{\partial n_{j,x}}{\partial x} + \frac{\partial n_{i,z}}{\partial z}\frac{\partial n_{j,y}}{\partial y} + \frac{\partial n_{i,z}}{\partial z}\frac{\partial n_{j,z}}{\partial z}.\end{aligned}$$

Again the divergence element matrix $\boldsymbol{N}^{(e)}$ can be represented in a 3×3-block structure with the aid of the matrices $\boldsymbol{K}_{\mathbf{x}}^{11}$, $\boldsymbol{K}_{\mathbf{x}}^{22}$, $\boldsymbol{K}_{\mathbf{x}}^{33}$, $\boldsymbol{K}_{\mathbf{x}}^{12}$, $\boldsymbol{K}_{\mathbf{x}}^{13}$ and $\boldsymbol{K}_{\mathbf{x}}^{23}$:

$$\boldsymbol{N}^{(e)} = J\begin{pmatrix}\boldsymbol{K}_{\mathbf{x}}^{11} & \boldsymbol{K}_{\mathbf{x}}^{12\,T} & \boldsymbol{K}_{\mathbf{x}}^{13\,T}\\ \boldsymbol{K}_{\mathbf{x}}^{12} & \boldsymbol{K}_{\mathbf{x}}^{22} & \boldsymbol{K}_{\mathbf{x}}^{23\,T}\\ \boldsymbol{K}_{\mathbf{x}}^{13} & \boldsymbol{K}_{\mathbf{x}}^{23} & \boldsymbol{K}_{\mathbf{x}}^{33}\end{pmatrix}. \tag{3.20}$$

3.2.3 Construction of the matrix eigenvalue problem

The assembly of the global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ is done according to the same scheme as in Problem I (cf. Section 2.6). Keep in mind that by suitably numbering the degrees of freedom, a matrix eigenvalue problem with 2×2-block structure $\left(\begin{smallmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{smallmatrix}\right)\mathbf{u} = \lambda\left(\begin{smallmatrix}M_{11} & M_{12}\\ M_{21} & M_{22}\end{smallmatrix}\right)\mathbf{u}$ emerges, if *quadratic* basis functions are used. Then the matrices $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$ correspond to the global matrices using linear elements.

3.2.4 Boundary conditions

In this section we show how the boundary conditions are implemented. If all faces of Γ are planar and orthogonal to each other, and are also oriented coordinate direction, only the respective degrees of freedom need to be eliminated. In all other cases a considerable amount of effort is needed, in order to treat the boundary conditions correctly. This is one of the reasons, why vector elements, as discussed in Chapter 3.3, are being used instead of the penalty method (with node elements).

Two different types of boundary conditions may occur: At perfectly conducting surfaces of the cavity $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ holds. Here, the electric field is orthogonal to the surface. On symmetry planes either $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ or $\mathbf{e} \cdot \mathbf{n} = 0$ hold (electric field is tangential to the symmetry plane)[3].

For the implementation of the boundary conditions five cases are distinguished, and they all need to be treated differently:

1. Boundary points with $\mathbf{e} \times \mathbf{n} = \mathbf{0}$, which are located on a physical edge[4] or corner on the surface.
2. Boundary points with $\mathbf{e} \times \mathbf{n} = \mathbf{0}$, which are *not* located on a physical edge or corner on the surface.
3. Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$, which reside on exactly one symmetry plane.
4. Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$, which reside on exactly two symmetry planes.
5. Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$, which reside on three or more symmetry planes.

Boundary points with $\mathbf{e} \times \mathbf{n} = \mathbf{0}$, which are located on a physical edge or corner on the surface The electric field needs to be orthogonal to both (resp. all three) bordering surfaces. This is only possible, when the electric field disappears completely at these points. All three degrees of freedom, which correspond to the three field components in such a point, will be eliminated, i.e., the respective rows and columns are deleted from the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$.

Boundary points with $\mathbf{e} \times \mathbf{n} = \mathbf{0}$, which are *not* located on a physical edge or corner on the surface Here the electric field is orthogonal to the surface. The surface normal vector at the boundary point is generally not known. However, it can be approximated by different methods:

- ▷ using the normal vector of any adjacent surface triangle
- ▷ using an "average", formed by the normal vectors of all adjacent surface triangles
- ▷ using Newell's formula, a method used for determining the normal vector to a plane. A 3D-polygon is given by three or more vertex points, which lie (or almost lie) in this plane [27, page 476–477].

To simplify matters, we decided to use the second variant. For calculating the normal vector at a boundary point, we use the formula $\mathbf{n} := \sum_i \mathbf{n}_i / \| \sum_i \mathbf{n}_i \|_2$. The $\mathbf{n}_i$ are the normal vectors of the adjacent boundary faces.

[3] Cf. Section 2.2

[4] Physical edges are located on the surface of the domain Ω. They are not the edges, which are simply introduced by mesh generation.

Let $\mathbf{e} = (e_1\, e_2\, e_3)^T$ be the electric field and let $\mathbf{n} = (n_1\, n_2\, n_3)^T$ be the approximated surface normal vector at the boundary point. Then $e_1/n_1 = e_2/n_2 = e_3/n_3$ holds. If one of the e_i is given, the other two are resulting thereof. So there is only one degree of freedom left at such a boundary point.

So if (without loss of generality) e_1 is given, then e_2 and e_3 are obtained by

$$e_2 = \frac{n_2}{n_1} e_1 = c_2 e_1$$

and

$$e_3 = \frac{n_3}{n_1} e_1 = c_3 e_1.$$

Without loss of generality we assume that u_1 and u_2 are the degrees of freedom which correspond to e_1 and e_2. u_2 shall now be eliminated from the FEM-calculation.
The function u is approximated by

$$u = \sum_{k=1}^{n} u_k \mathbf{N}_k.$$

The $\mathbf{N}_k$ are the global ansatz functions. For each row i of the matrix eigenvalue problem

$$\sum_{s=1}^{n} a_{ij} u_j = \lambda \sum_{s=1}^{n} m_{ij} u_j$$

holds. This can be expanded to

$$\sum_{s=1}^{n} (\langle \mathbf{rot}\, \mathbf{N}_i, \mathbf{rot}\, \mathbf{N}_j \rangle - \lambda \langle \mathbf{N}_i, \mathbf{N}_j \rangle) u_j = 0.$$

Here we are using the notation $\int_\Omega \mathbf{a} \cdot \mathbf{b}\, \mathrm{d}\mathbf{x} = \langle \mathbf{a}, \mathbf{b} \rangle$.
We now introduce the condition $u_2 = c_2 u_1$ into the calculation.

$$[\langle \mathbf{rot}\, \mathbf{N}_i, \mathbf{rot}\, \mathbf{N}_1 + c_2\, \mathbf{rot}\, \mathbf{N}_2 \rangle - \lambda \langle \mathbf{N}_i, \mathbf{N}_1 + c_2 \mathbf{N}_2 \rangle] u_1 + \sum_{s=3}^{n} \cdots = 0.$$

Due to the linearity of the $\mathbf{rot}$ operator

$$[\langle \mathbf{rot}\, \mathbf{N}_i, \mathbf{rot}(\mathbf{N}_1 + c_2 \mathbf{N}_2) \rangle - \lambda \langle \mathbf{N}_i, \mathbf{N}_1 + c_2 \mathbf{N}_2 \rangle] u_1 + \sum_{s=3}^{n} \cdots = 0$$

holds.

$\mathbf{N}_1$ is now replaced by $\mathbf{N}_1 + c_2 \mathbf{N}_2$, and $\mathbf{N}_2$ is eliminated. $\mathbf{N}_1 + c_2 \mathbf{N}_2$ is linearly independent of all $\mathbf{N}_j$ for $j > 2$. Therefore, Galerkin's method continues to be valid.

The elimination of u_2 can be carried out using the matrix operations only: Matrix $\tilde{\boldsymbol{A}}$ is the matrix that results when eliminating u_2 from $\boldsymbol{A}$. $\tilde{\boldsymbol{A}}$ is computed as follows:

$$\begin{aligned}
\tilde{a}_{11} &= \langle \mathbf{rot}(\mathbf{N}_1 + c_2\mathbf{N}_2), \mathbf{rot}(\mathbf{N}_1 + c_2\mathbf{N}_2)\rangle \\
&= \langle \mathbf{rot}\,\mathbf{N}_1, \mathbf{rot}\,\mathbf{N}_1\rangle + c_2\langle \mathbf{rot}\,\mathbf{N}_1, \mathbf{rot}\,\mathbf{N}_2\rangle + c_2\langle \mathbf{rot}\,\mathbf{N}_2, \mathbf{rot}\,\mathbf{N}_1\rangle \\
&\quad + c_2^2\langle \mathbf{rot}\,\mathbf{N}_2, \mathbf{rot}\,\mathbf{N}_2\rangle \\
&= a_{11} + c_2(a_{12} + a_{21}) + c_2^2 a_{22} \\
\tilde{a}_{1j} &= \langle \mathbf{rot}(\mathbf{N}_1 + c_2\mathbf{N}_2), \mathbf{rot}\,\mathbf{N}_j\rangle \\
&= \langle \mathbf{rot}\,\mathbf{N}_1, \mathbf{rot}\,\mathbf{N}_j\rangle + c_2\langle \mathbf{rot}\,\mathbf{N}_2, \mathbf{rot}\,\mathbf{N}_j\rangle \\
&= a_{1j} + c_2 a_{2j} \\
\tilde{a}_{i1} &= \langle \mathbf{rot}\,\mathbf{N}_i, \mathbf{rot}(\mathbf{N}_1 + c_2\mathbf{N}_2)\rangle \\
&= \langle \mathbf{rot}\,\mathbf{N}_i, \mathbf{rot}\,\mathbf{N}_1\rangle + c_2\langle \mathbf{rot}\,\mathbf{N}_i, \mathbf{rot}\,\mathbf{N}_2\rangle \\
&= a_{i1} + c_2 a_{i2}
\end{aligned}$$

Thus, the c_2-fold of the second column is added to the first column, and then the c_2-fold of the second row is added to the first row, or vice versa. Afterwards the second column and the second row are eliminated from the matrix.

The correctness of this procedure shall be illustrated with entry $\tilde{a}_{11}$. For the remaining entries in the first row, resp. first column, the correctness is obvious.

$\boldsymbol{A}$' is the matrix created by adding the c_2-fold of the second row to the first row of $\boldsymbol{A}$:

$$a'_{11} = a_{11} + c_2 a_{21} \quad \text{and} \quad a'_{12} = a_{12} + c_2 a_{22}.$$

After the subsequent addition of the c_2-fold of the second column of A to its first row we have

$$\begin{aligned}
\tilde{a}_{11} &= a'_{11} - c_2 a'_{21} \\
&= a_{11} + c_2 a_{21} + c_2(a_{12} + c_2 a_{22}) \\
&= a_{11} + c_2(a_{12} + a_{21}) + c_2^2 a_{22}.
\end{aligned}$$

The matrix operations which are necessary to eliminate the degree of freedom u_2 are also valid for matrix $\boldsymbol{M}$. The derivation can be done by analogy.

This procedure can be applied to eliminate any degree of freedom. It can be shown that several degrees of freedom can be dealt with independently, according to the scheme mentioned above.

By eliminating the degrees of freedom, no new non-zero elements are created in the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$, since the three degrees of freedom of a grid point always belong to the same elements.

Pivoting When treating a boundary point, one has the choice to eliminate any one of the three field components. To ensure that the condition numbers of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are not substantially degrading, it is advisable to *not* eliminate the component showing the largest magnitude in the normal vector. By this choice, multiples with a factor less or equal than one of the rows and columns to be eliminated are added, since $\max_{n_1,n_2,n_3} |n_2/n_1| = 1$, for $n_1^2 + n_2^2 + n_3^2 = 1$ and $|n_1| > |n_2| > |n_3|$. This ensures that the elimination will not generate arbitrarily large entries in the matrix.

Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$ on exactly one symmetry plane For each boundary point of this kind three orthonormal vectors $\mathbf{n}_1$, $\mathbf{n}_2$ and $\mathbf{n}_3$ are determined. $\mathbf{n}_1$ is orthogonal to the symmetry plane; $\mathbf{n}_2$ and $\mathbf{n}_3$ are tangential to it.

Let $\mathbf{u}$ be the electric field at such a boundary point. $\mathbf{u}$ can be expressed as linear combination of $\mathbf{n}_i$:

$$\mathbf{u} = (\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3)\, \mathbf{w}.$$

The w_i are the components of $\mathbf{u}$ with respect to the basis $(\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3)$. Since the electric field is tangential to the symmetry plane, w_1 is equal to 0. For w_2 and w_3 we get

$$w_2 = \mathbf{n}_2 \mathbf{u}$$
$$w_3 = \mathbf{n}_3 \mathbf{u}.$$

The old degrees of freedom u_1, u_2 and u_3 are eliminated and replaced by w_2 and w_3. Similar to the section above, the respective linear combinations of the matrix rows and columns are formed.

Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$ on exactly two symmetry planes A boundary point of this kind is located on the line of intersection of two symmetry planes. Since the electric field must be tangential to both symmetry planes, it has only one component in the direction of the line of intersection.

Let $\mathbf{u}$ be the electric field at such a boundary point and let $\mathbf{n}$ be the normalised direction of the line of intersection. The degrees of freedom u_1, u_2, and u_3 are being replaced by the degree of freedom

$$w = \mathbf{n} \cdot \mathbf{u}.$$

Boundary points with $\mathbf{e} \cdot \mathbf{n} = 0$ on three or more symmetry planes A boundary point of this kind lies at the intersection of three or more symmetry planes. Since the electric field must be tangential to all symmetry planes, it has to disappear.

All three degrees of freedom that belong to such a boundary point are eliminated.

3.3 Mixed method

The node elements together with the penalty-method as presented in Section 3.2 have the disadvantage that the boundary conditions can only be included into the calculation with a considerable effort [45, page 231] (cf. Section 3.2.4). Also, choosing the penalty parameter s may be a problem. This motivates to use so-called *Nédélec vector elements* [54]. With vector elements the degrees of freedom are not assigned to function values at certain nodes, but to moments (integrals) over edges, faces and volumes. The elements are often called *edge elements*, as the DOFs of the lowest order basis functions are associated with edges.

We start directly from the discretised integral formulation (3.8). It turns out that by using the vector elements discussed in this section, all divergence-afflicted solutions (spurious modes) belong to the multiple eigenvalue zero and the matrix on the left hand side of (3.8) has a non-trivial null space. Furthermore, the vectors which span this null space can be calculated with little effort [4, 5]. This will be discussed further in Section 3.3.5.

3.3.1 Description of the vector elements

The degrees of freedom of the linear and the quadratic vector elements are assigned to the edges and faces of the tetrahedral element. The numbering of the nodes and edges we use is indicated in Fig. 3.2. We number the faces sequentially as follows: $(1,2,3)$, $(2,3,4)$, $(1,3,4)$, $(1,2,4)$[5].

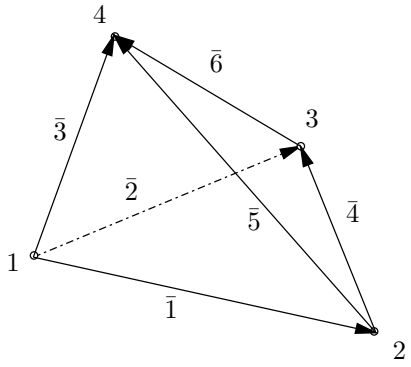


Figure 3.2: *Numbering of nodes and edges*

For the definition of the basis functions we are using the scalar functions $L_1 \ldots L_4$, as defined in (2.9), which supply the respective simplex coordinate in dependence of x, y and z.

The linear function L_i has value 1 at the tetrahedron corner i and value 0 at all other corners. The gradients $\nabla L_1 \ldots \nabla L_4$ are *constant* vectors. ∇L_i is orthogonal to the tetrahedron face $L_i = 0$, i.e. the face opposite to the corner P_i. ∇L_i points from that face in direction of the corner P_i.

The basis functions of the vector elements are defined in such a way that the tangential components of the representable field are continuous across the element boundaries. The normal components may jump [62]. For this reason, the elements are also called *tangential vector elements* [54], [78].

We use vector elements of order 1 and 2. They are described in the following two sections.

Linear vector element We define the six basis functions of the *linear vector element* as follows [66]:

$$\begin{aligned}
\mathbf{N}_1^{(e)} &= L_1 \nabla L_2 - L_2 \nabla L_1 \\
\mathbf{N}_2^{(e)} &= L_1 \nabla L_3 - L_3 \nabla L_1 \\
\mathbf{N}_3^{(e)} &= L_1 \nabla L_4 - L_4 \nabla L_1 \\
\mathbf{N}_4^{(e)} &= L_2 \nabla L_3 - L_3 \nabla L_2 \\
\mathbf{N}_5^{(e)} &= L_2 \nabla L_4 - L_4 \nabla L_2 \\
\mathbf{N}_6^{(e)} &= L_3 \nabla L_4 - L_4 \nabla L_3.
\end{aligned} \tag{3.21}$$

Here $\mathbf{N}_i^{(e)}$ is assigned to edge i. As an example $\mathbf{N}_1^{(e)}$ is shown in Fig. 3.3.

We now derive some properties of the six basis functions:

Directed degrees of freedom The basis functions of the form $L_i \nabla L_j - L_j \nabla L_i$ depend on the direction of the edge. If the orientation of the edge is reversed, i.e. the indices i and j are swapped, then the three components of the corresponding basis function change the sign.

[5]The three numbers in parentheses represent the numbers of the corners belonging to the face.

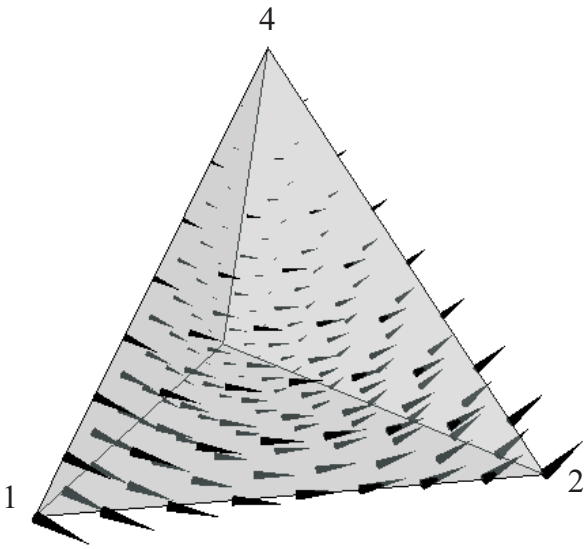


Figure 3.3: *Nedelec basis function* $\mathbf{N}_1^{(e)}$

The direction of the edges are defined both locally and globally. The direction of the local degrees of freedom is indicated in Fig. 3.2 using arrows.

The edge directions have to be considered at two places:

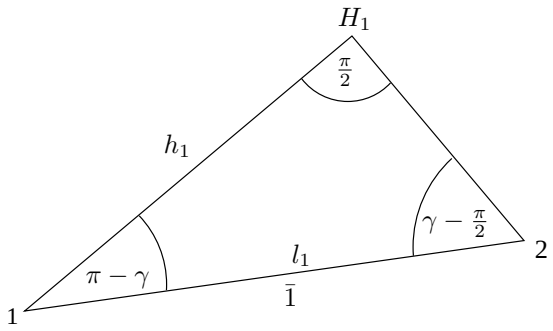
1. During the assembly of the global matrices, when contributions to the integrals from different elements are added up: If the local edge orientation does not match the global edge orientation, the element matrices have to be adjusted accordingly before adding them to the global matrices.
2. When building the sparse matrix $\boldsymbol{Y}$: The entries y_{ij} depend on the orientation of the edges (cf. Section 3.3.5).

Divergence-free interior It turns out, that the divergence of the basis functions $\mathbf{N}_1^{(e)} \dots \mathbf{N}_6^{(e)}$ vanishes in the interior of the tetrahedral element,

$$\nabla(L_i \nabla L_j - L_j \nabla L_i) = L_i \nabla\nabla L_j - L_j \nabla\nabla L_i = 0,$$

because the double derivatives of the linear functions are equal to zero.

Constant tangential component Without loss of generality we show this and the next property only for $\mathbf{N}_1^{(e)}$. Let $\mathbf{e}_1$ be the unit vector that points from node 1 in direction of node 2. Let l_1 be the length of edge $\bar{1}$. We now calculate the dot product of $\mathbf{e}_1$ and ∇L_1: Let γ be the angle between $\mathbf{e}_1$ and ∇L_1 and let h_1 be the distance between node 1 and face $(2, 3, 4)$. From the definition of the dot product follows that $\mathbf{e}_1 \cdot \nabla L_1 = \frac{1}{h_1} \cos\gamma$. We now consider the triangle $(1, 2, H_1)$, where H_1 is the point nearest to the plane $(2, 3, 4)$. The straight line from node 1 to H_1 is collinear to ∇L_1.



The figure on this page shows that $\frac{h_1}{l_1} = \sin(\pi/2 - \gamma) = -\cos\gamma$.

From the above considerations follows, that $\mathbf{e}_1 \cdot \nabla L_1 = -1/l_1$. Similar considerations show, that $\mathbf{e}_1 \cdot \nabla L_2 = 1/l_1$. Therefore $\mathbf{e}_1 \cdot \mathbf{N}_1^{(e)} = \frac{L_1+L_2}{l_1} = \frac{1-L_3-L_4}{l_1}$. Along the edge $\bar{1}$ we have $\mathbf{e}_1 \cdot \mathbf{N}_1^{(e)} = 1/l_1$, because L_3 and L_4 are equal to zero there. That means, that the basis function $\mathbf{N}_1^{(e)}$ has a *constant tangential component* along edge $\bar{1}$.

No tangential components on other edges $\mathbf{N}_1^{(e)}$ has no tangential component on the faces $(1,3,4)$ and $(2,3,4)$. It immediately follows, that $\mathbf{N}_1^{(e)}$ has no tangential component on all edges except $\bar{1}$.

Non-uniform polynomial degree Often the functions $\mathbf{N}_1^{(e)} \dots \mathbf{N}_6^{(e)}$ are called a *CT/LN-basis*, because they can represent a field with constant tangential components and linear normal components at the element boundaries [62]. Thus the representable vector fields do not have the same polynomial degree in each direction.

Quadratic vector element For the *quadratic vector element* we are using hierarchical basis functions as we have done for Problem I. Thus the first six basis functions are adopted from the linear vector element.

The basis functions $\mathbf{N}_7^{(e)} \dots \mathbf{N}_{12}^{(e)}$ are also assigned to the edges $1 \dots 6$ [66],

$$\begin{aligned}
\mathbf{N}_7^{(e)} &= L_1\nabla L_2 + L_2\nabla L_1 \\
\mathbf{N}_8^{(e)} &= L_1\nabla L_3 + L_3\nabla L_1 \\
\mathbf{N}_9^{(e)} &= L_1\nabla L_4 + L_4\nabla L_1 \\
\mathbf{N}_{10}^{(e)} &= L_2\nabla L_3 + L_3\nabla L_2 \\
\mathbf{N}_{11}^{(e)} &= L_2\nabla L_4 + L_4\nabla L_2 \\
\mathbf{N}_{12}^{(e)} &= L_3\nabla L_4 + L_4\nabla L_3.
\end{aligned} \tag{3.22}$$

As an example, $\mathbf{N}_7^{(e)}$ is shown in Fig. 3.4.

The properties of the basis functions $\mathbf{N}_7^{(e)} \dots \mathbf{N}_{12}^{(e)}$ are different from those of the first six basis functions. Their tangential component is no longer constant on their assigned edge. Together with the functions $\mathbf{N}_1^{(e)} \dots \mathbf{N}_6^{(e)}$ one would get a complete linear polynomial x, y and z for each field component.

The functions $\mathbf{N}_7^{(e)} \dots \mathbf{N}_{12}^{(e)}$ are not divergence-free in the interior of the element, since

$$\nabla(L_i\nabla L_j + L_j\nabla L_i) = L_i\nabla\nabla L_j + L_j\nabla\nabla L_i + 2\nabla L_i\nabla L_j \neq 0.$$

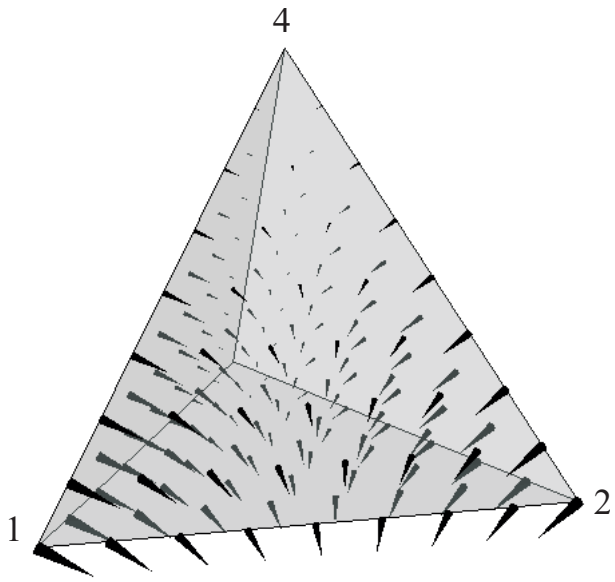


Figure 3.4: *Nedelec basis function* $\mathbf{N}_7^{(e)}$

They are however *curl-free* in the interior, since

$$\nabla \times (L_i \nabla L_j + L_j \nabla L_i) = \nabla L_i \times \nabla L_j + \nabla L_j \times \nabla L_i = 0.$$

To define the remaining basis functions $\mathbf{N}_{13}^{(e)} \ldots \mathbf{N}_{24}^{(e)}$, three out of four of the functions $L_1 \ldots L_4$ are used at a time. They are therefore assigned to a face of the tetrahedral element [66].

$$\mathbf{N}_{13}^{(e)} = \nabla L_1 \; L_2 L_3, \qquad \mathbf{N}_{17}^{(e)} = \nabla L_2 \; L_3 L_1, \qquad \mathbf{N}_{21}^{(e)} = \nabla L_3 \; L_1 L_2, \tag{3.23}$$

$$\mathbf{N}_{14}^{(e)} = \nabla L_2 \; L_3 L_4, \qquad \mathbf{N}_{18}^{(e)} = \nabla L_3 \; L_4 L_2, \qquad \mathbf{N}_{22}^{(e)} = \nabla L_4 \; L_2 L_3, \tag{3.24}$$

$$\mathbf{N}_{15}^{(e)} = \nabla L_3 \; L_4 L_1, \qquad \mathbf{N}_{19}^{(e)} = \nabla L_4 \; L_1 L_3, \qquad \mathbf{N}_{23}^{(e)} = \nabla L_1 \; L_3 L_4, \tag{3.25}$$

$$\mathbf{N}_{16}^{(e)} = \nabla L_4 \; L_1 L_2, \qquad \mathbf{N}_{20}^{(e)} = \nabla L_1 \; L_2 L_4, \qquad \mathbf{N}_{24}^{(e)} = \nabla L_2 \; L_4 L_1. \tag{3.26}$$

As an example, $\mathbf{N}_{16}^{(e)}$ is shown in Fig. 3.5.

The vector fields $\nabla L_i \; L_j L_k$, $i \neq j \neq k$, are *unidirectional*. It is orthogonal to the face opposite of corner P_i. On both faces adjacent to edge (j, k), the field has a quadratic component. On the other two faces the field vanishes.

Out of the three degrees of freedom assigned to a face, one is arbitrarily chosen and eliminated [66, page 299], since their tangential components are linearly dependent [78]. This choice cannot simply be done element-wise. It has to match with the corresponding DOF of the neighbouring element. Therefore we initially compute the element matrices for all 24 basis functions. Only after the global numbering is known, we can eliminate one degree of freedom per face.

The functions $\mathbf{N}_1^{(e)} \ldots \mathbf{N}_{24}^{(e)}$ are often referred to as a *LT/QN-basis*, since they can represent fields with linear tangential components and quadratic normal components [62].

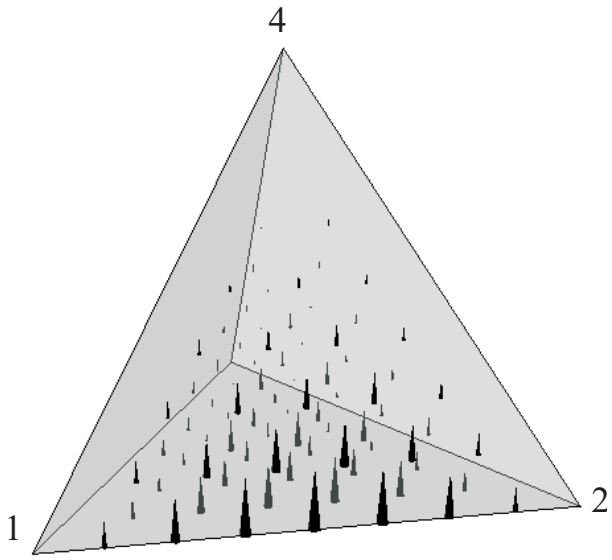


Figure 3.5: *Nedelec basis function* $\mathbf{N}_{16}^{(e)}$

3.3.2 Calculation of the element matrices

The mass element matrix $\boldsymbol{M}^{(e)}$ and the curl element matrix $\boldsymbol{A}^{(e)}$ are used to calculate the integrals in the discretised weak formulation (3.8). They are defined as follows:

$$m_{i,j}^{(e)} = \iiint_{T_e} \mathbf{N}_i^{(e)} \cdot \mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z, \quad \text{and} \tag{3.27a}$$

$$a_{i,j}^{(e)} = \iiint_{T_e} \mathbf{rot}\,\mathbf{N}_i^{(e)} \cdot \mathbf{rot}\,\mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \tag{3.27b}$$

We now describe how we compute these element matrices in practice.

Mass element matrix We exemplify the calculation of the mass element matrix for matrix entry $m_{12}^{(e)}$:

$$\begin{aligned} m_{12}^{(e)} &= \iiint_{T_e} \mathbf{N}_1^{(e)} \cdot \mathbf{N}_2^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ &= \iiint_{T_e} (L_1 \nabla L_2 - L_2 \nabla L_1) \cdot (L_1 \nabla L_3 - L_3 \nabla L_1) \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \end{aligned} \tag{3.28}$$

Expanding the formula and factoring out the constant gradients ∇L_i yields

$$\begin{aligned} m_{12}^{(e)} = {} & \nabla L_2 \cdot \nabla L_3 \iiint_{T_e} L_1 L_1 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ & + \nabla L_1 \cdot \nabla L_1 \iiint_{T_e} L_2 L_3 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ & - \nabla L_1 \cdot \nabla L_2 \iiint_{T_e} L_1 L_3 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ & - \nabla L_1 \cdot \nabla L_3 \iiint_{T_e} L_1 L_2 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \end{aligned} \tag{3.29}$$

Since the integrands in (3.29) contain only scalar quantities, the transformation of the integrals over T_e into integrals over T_0 is straight-forward. E.g.

$$\iiint_{T_e} L_2 L_3 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \frac{J}{120}.$$

J is the determinant of the Jacobi matrix, as defined on page 12. Thus for $m_{12}^{(e)}$ we get

$$m_{12}^{(e)} = \nabla L_2 \cdot \nabla L_3 \frac{J}{60} + \nabla L_1 \cdot \nabla L_1 \frac{J}{120} - \nabla L_1 \cdot \nabla L_2 \frac{J}{120} - \nabla L_1 \cdot \nabla L_3 \frac{J}{120}.$$

To simplify the calculation of $\boldsymbol{M}^{(e)}$, we store the gradients ∇L_i and their inner products. We store the constant gradients ∇L_i in the vectors $\mathbf{g}_1, \ldots, \mathbf{g}_4$:

$$\mathbf{g}_i := \nabla L_i. \tag{3.30}$$

We store the dot products of the $\mathbf{g}_i$ in $\boldsymbol{G} \in \mathbb{R}^{4\times 4}$:

$$g_{ij} := \mathbf{g}_i \cdot \mathbf{g}_j. \tag{3.31}$$

We also need to know the integrals $\iint_{T_e} L_i L_j \,\mathrm{d}\mathbf{x}$, $\iiint_{T_e} L_i L_j L_k \,\mathrm{d}\mathbf{x}$ and $\iiint_{T_e} L_i L_j L_k L_l \,\mathrm{d}\mathbf{x}$. They can all be computed using the well known volume integration formula for simplex coordinates,

$$\iiint_{T_e} L_1^p L_2^q L_3^r L_4^s \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = 6V \frac{p!q!r!s!}{(p+q+r+s+3)!}. \tag{3.32}$$

Using this information the entries of $\boldsymbol{M}^{(e)}$ can be computed efficiently. E.g. matrix entry m_{12} is then calculated by

$$m_{12} = J/120(g_{11} - g_{12} - g_{13} + 2g_{23}).$$

A complete list containing the formulas for all entries is given in Appendix F.2.1.

Curl element matrix The curl element matrix $\boldsymbol{A}^{(e)}$ is defined by

$$a_{i,j}^{(e)} = \iiint_{T_e} \mathbf{rot}\,\mathbf{N}_i^{(e)} \cdot \mathbf{rot}\,\mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z.$$

Using the nabla calculus, the curl of the basis function $\mathbf{N}_i$ can be calculated:

$$\nabla \times (L_i \nabla L_j - L_i \nabla L_i) = 2(\nabla L_i \times \nabla L_j) \tag{3.33a}$$
$$\nabla \times (L_i \nabla L_j + L_i \nabla L_i) = 0 \tag{3.33b}$$
$$\nabla \times (\nabla L_i \; L_j L_k) = L_k(\nabla L_j \times \nabla L_i) + L_j(\nabla L_k \times \nabla L_i). \tag{3.33c}$$

As can be seen from (3.33), the vector cross products of the ∇L_i are needed to calculate $a_{ij}^{(e)}$. We store the vector cross products in $\mathbf{c}_1 \ldots \mathbf{c}_6$,

$$\begin{aligned} \mathbf{c}_1 &:= \nabla L_1 \times \nabla L_2, & \mathbf{c}_2 &:= \nabla L_1 \times \nabla L_3, \\ \mathbf{c}_3 &:= \nabla L_1 \times \nabla L_4, & \mathbf{c}_4 &:= \nabla L_2 \times \nabla L_3, \\ \mathbf{c}_5 &:= \nabla L_2 \times \nabla L_4, & \mathbf{c}_6 &:= \nabla L_3 \times \nabla L_4. \end{aligned} \tag{3.34}$$

Their inner products are stored in $\boldsymbol{C} \in \mathbb{R}^{6\times 6}$,

$$c_{ij} := \mathbf{c}_i \cdot \mathbf{c}_j. \tag{3.35}$$

For the calculation of the $a_{ij}^{(e)}$, the following special cases of (3.32) are needed,

$$\iiint_{T_e} \mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \frac{J}{6} \quad \text{and} \quad \iiint_{T_e} L_i\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \frac{J}{24}. \tag{3.36}$$

We exemplify the calculation of the curl element matrix for the entry $a_{1,13}^{(e)}$:

$$\begin{aligned} a_{1,13}^{(e)} &= \iiint_{T_e} \mathbf{rot}\,\mathbf{N}_1^{(e)} \cdot \mathbf{rot}\,\mathbf{N}_{13}^{(e)}\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ &= \iiint_{T_e} 2(\nabla L_1 \times \nabla L_2)\cdot \\ &\qquad\qquad [L_3(\nabla L_2 \times \nabla L_1) + L_2(\nabla L_3 \times \nabla L_1)]\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ &= -2\left(c_{11}\iiint_{T_e} L_3\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z + c_{12}\iiint_{T_e} L_2\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\right) \\ &= -J/12\;(c_{11}+c_{12}). \end{aligned} \tag{3.37}$$

The other matrix entries can be computed using the same scheme. A complete list containing the formulas for all entries is given in Appendix F.2.1.

3.3.3 Matrix eigenvalue problem

In this section we transform the integral Equation (3.8) into a matrix eigenvalue problem of the form $\boldsymbol{A}\mathbf{u} = \lambda \boldsymbol{M}\mathbf{u}$. The global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are defined by

$$\begin{aligned} a_{ij} &= \int_\Omega \mathbf{rot}\,\mathbf{N}_i \cdot \mathbf{rot}\,\mathbf{N}_j\,\mathrm{d}\Omega, \quad \text{and} \\ m_{ij} &= \int_\Omega \mathbf{N}_i \cdot \mathbf{N}_j\,\mathrm{d}\Omega. \end{aligned} \tag{3.38}$$

The matrix assembly is realised using the same scheme as for Problem I (cf. Section 2.6). It is to be noted that with an appropriate numbering of the degrees of freedom, the resulting matrix eigenvalue problem shows a 2×2-block structure, $\left(\begin{smallmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{smallmatrix}\right)\mathbf{u} = \lambda\left(\begin{smallmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{smallmatrix}\right)\mathbf{u}$, if *quadratic* elements are used. Due to the hierarchical basis, the $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$ matrices then coincide with the global matrices when using linear elements.

For the later discussion of the eigensolvers, the properties of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are essential. They shall be derived in this section.

From (3.38) follows immediately that $\boldsymbol{A}$ and $\boldsymbol{M}$ are symmetric. The quadratic forms $\mathbf{u}^T\boldsymbol{A}\mathbf{u}$ and $\mathbf{u}^T\boldsymbol{M}\mathbf{u}$ match the integrals $\int_\Omega(\mathbf{rot}\,\tilde{\mathbf{e}})^2\,\mathrm{d}\Omega$ and $\int_\Omega \tilde{\mathbf{e}}^2\,\mathrm{d}\Omega$. $\mathbf{u}$ is defined through $\mathbf{u} = [u_1, \ldots, u_m]$ with $\tilde{\mathbf{e}} = \sum_{k=1}^m u_k \mathbf{N}_k$.

The integral $\int_\Omega \tilde{\mathbf{e}}^2\,\mathrm{d}\Omega$ is positive for $\tilde{\mathbf{e}} \neq \mathbf{0}$. Therefore $\boldsymbol{M}$ must be *positive definite*. The integral $\int_\Omega(\mathbf{rot}\,\tilde{\mathbf{e}})^2\,\mathrm{d}\Omega$ is positive unless $\tilde{\mathbf{e}}$ is curl-free in Ω. In this case the integral is equal to zero. Thus the matrix $\boldsymbol{A}$ is only *positive semi-definite*.

3.3.4 Boundary conditions

There are two kinds of boundary conditions: On the perfectly conducting surface $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ must hold. Thus the electric field is orthogonal to the surface there. On possible symmetry planes, either $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ or $\mathbf{e} \cdot \mathbf{n} = 0$ (electric field is tangential to the symmetry plane) must hold.

Boundary condition $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ The boundary condition $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ is *forced* by setting all degrees of freedom located on such a boundary surface to zero. This eliminates all tangential components on the boundary surface, because all remaining degrees of freedom either are zero on the surface or only have components normal to it.

In practice this boundary condition is implemented by eliminating the corresponding rows and columns from the global matrices. It is advantageous to identify the degrees of freedom that are to be eliminated *before* the matrix assembly.

Boundary condition $\mathbf{e} \cdot \mathbf{n} = \mathbf{0}$ Let $(\lambda, \mathbf{e})$ be a solution of the weak form of (3.3),

$$\int_\Omega \mathbf{rot}\,\mathbf{rot}\,\mathbf{e} \cdot \mathbf{\Psi} - \lambda\,\mathbf{e} \cdot \mathbf{\Psi}\,\mathrm{d}\Omega = 0, \tag{3.39}$$

for any suitably chosen function $\mathbf{\Psi}$. Note that for this discussion we consider $\mathbf{e}$ and $\mathbf{\Psi}$ arbitrary continuous functions. They are not picked from a restricted FEM function space.

According to (3.7), Equation (3.39) is equivalent to

$$\int_\Omega \mathbf{rot}\,\mathbf{e} \cdot \mathbf{rot}\,\mathbf{\Psi} - \lambda\,\mathbf{e} \cdot \mathbf{\Psi}\,\mathrm{d}\Omega - \int_\Gamma \mathbf{rot}\,\mathbf{e} \cdot (\mathbf{n} \times \mathbf{\Psi})\,\mathrm{d}\gamma = 0, \tag{3.40}$$

for all $\mathbf{\Psi}$. If $\mathbf{\Psi}$ is chosen s.t. $\mathbf{n} \times \mathbf{\Psi} = \mathbf{0}$ on Γ, then the surface integral vanishes, and Equation (3.39) is equivalent to

$$\int_\Omega \mathbf{rot}\,\mathbf{e} \cdot \mathbf{rot}\,\mathbf{\Psi} - \lambda\,\mathbf{e} \cdot \mathbf{\Psi}\,\mathrm{d}\Omega = 0,$$

where $(\lambda, \mathbf{e})$ satisfies (3.39). Since the volume integral in (3.40) vanishes, also the surface integral must vanish, thus

$$\int_\Gamma \mathbf{rot}\,\mathbf{e} \cdot (\mathbf{n} \times \mathbf{\Psi})\,\mathrm{d}\gamma = -\int_\Gamma (\mathbf{n} \times \mathbf{rot}\,\mathbf{e}) \cdot \mathbf{\Psi}\,\mathrm{d}\gamma = 0.$$

Since the surface integral is zero for any function $\mathbf{\Psi}$,

$$\mathbf{n} \times \mathbf{rot}\,\mathbf{e} = \mathbf{0}$$

must hold and is therefore a natural boundary condition.

According to the time-harmonic Maxwell equations, the magnetic field $\mathbf{h}$ is collinear to $\mathbf{rot}\,\mathbf{e}$ [1]. Since $\mathbf{h} \cdot \mathbf{e} = 0$, the condition $\mathbf{n} \times \mathbf{rot}\,\mathbf{e} = \mathbf{0}$ implies

$$\mathbf{e} \cdot \mathbf{n} = 0,$$

which can therefore also be considered a natural boundary condition.

This means, that we do not need to enforce condition $\mathbf{e} \cdot \mathbf{n} = 0$ on the boundary. It is satisfied naturally.

Of course, in the finite element calculation $\mathbf{e} \cdot \mathbf{n} = 0$ is only satisfied approximately. However, the finer the discretisation, the better we can expect it to be satisfied.

3.3.5 Matrix form of the constraint $\operatorname{div} \mathbf{e} = 0$

Weak form of $\operatorname{div} \mathbf{e} = 0$ As there are no divergence-free finite elements, it is customary to replace the constraint

$$\operatorname{div} \mathbf{e} = 0 \tag{3.41}$$

by

$$\int_\Omega \operatorname{div} \mathbf{e}\, q \, \mathrm{d}\Omega = 0, \tag{3.42}$$

for all smooth functions q, that are zero on the boundary. (3.42) is equivalent to

$$\int_\Omega \mathbf{e} \cdot \mathbf{grad}\, q \, \mathrm{d}\Omega = 0. \tag{3.43}$$

The latter form (3.43) has the advantage, that there are no derivatives of $\mathbf{e}$ involved. To this end (3.43) is transformed into discretised integral form using the method of weighted residuals. The discretised integral form of (3.41) is

$$\int_\Omega \mathbf{e} \cdot \mathbf{grad}\, q_j \, \mathrm{d}\Omega = 0 \quad \forall j. \tag{3.44}$$

The q_j are scalar functions, that can be expressed as linear combinations of the $N_1 \dots N_m$[6]

$$q_j = \sum_{k=1}^{m} q_{jk} N_k. \tag{3.45}$$

The q_j must have the same polynomial degree as the $\mathbf{N}_k$ from (3.4).

Here we essentially work with two different kinds of finite element approaches: $\mathbf{e}$ is represented using vector basis functions $\mathbf{N}_i$ (Nédélec basis functions) and the q_j are represented using ordinary scalar basis functions N_j (Lagrange basis functions). This approach is therefore often called a *mixed method*, or more specifically a mixed *Nédélec-Lagrange finite element discretisation*.

Matrix form of the constraint $\operatorname{div} \mathbf{e} = 0$ Thus we define the matrix $\boldsymbol{C}$ as follows

$$c_{ij} = \int_\Omega \mathbf{N}_i \cdot \mathbf{grad}\, N_j \, \mathrm{d}\Omega. \tag{3.46}$$

The matrix form of the divergence-free condition is then

$$\boldsymbol{C}^T \mathbf{x} = \mathbf{0}. \tag{3.47}$$

Constructing matrix $\boldsymbol{C}$ from matrix $\boldsymbol{M}$ The matrix $\boldsymbol{C}$ could be assembled in the same way as the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$. Element matrices storing the inner products of the vector basis functions $\mathbf{N}_i^{(e)}$ and the gradients of the scalar basis functions $N_j^{(e)}$ as defined in Chapter 2 have to be computed.

We chose a different approach, which leads to a more efficient implementation. We use the fact that the gradients of the N_j lie in the space spanned by the vector basis functions $\mathbf{N}_i$ [35]. We begin by showing this property for a single finite element.

[6] The scalar functions $N_1 \dots N_m$ were defined in Chapter 2. They are the global basis functions of node elements.

Computing grad $N_k^{(e)}$ The gradients of the scalar functions $N_i^{(e)}$ from (2.17) and (2.18) are representable as linear combinations of the vector basis functions $\mathbf{N}_i^{(e)}$. The gradients of the scalar basis functions have the form ∇L_i or $\nabla(L_iL_j)$ (cf. Equation (2.18)). The necessary calculation steps are now demonstrated for ∇L_1 and $\nabla(L_1L_2)$:

$$\begin{aligned}
-\mathbf{N}_1 - \mathbf{N}_2 - \mathbf{N}_3 &= L_2\nabla L_1 - L_1\nabla L_2 + L_3\nabla L_1 - L_1\nabla L_3 \\
&\quad + L_4\nabla L_1 - L_1\nabla L_4 \\
&= \nabla L_1(L_2 + L_3 + L_4) - L_1(\nabla L_2 + \nabla L_3 + \nabla L_4) \\
&= \nabla L_1(1 - L_1) - L_1(\nabla L_2 + \nabla L_3 + \nabla L_4) \\
&= \nabla L_1 - L_1(\nabla L_1 + \nabla L_2 + \nabla L_3 + \nabla L_4) \\
&= \nabla L_1
\end{aligned}$$

and

$$\mathbf{N}_7 = L_1\nabla L_2 + L_2\nabla L_1 = \nabla(L_1L_2).$$

Equation (3.48) contains all the formulas required to show, how the gradients of all nodal basis functions can be computed using linear combinations of the vector basis functions:

$$\begin{aligned}
\nabla N_1^{(e)} &= \nabla L_1 = -\mathbf{N}_1^{(e)} - \mathbf{N}_2^{(e)} - \mathbf{N}_3^{(e)} \\
\nabla N_2^{(e)} &= \nabla L_2 = +\mathbf{N}_1^{(e)} - \mathbf{N}_4^{(e)} - \mathbf{N}_5^{(e)} \\
\nabla N_3^{(e)} &= \nabla L_3 = +\mathbf{N}_2^{(e)} + \mathbf{N}_4^{(e)} - \mathbf{N}_6^{(e)} \\
\nabla N_4^{(e)} &= \nabla L_4 = +\mathbf{N}_3^{(e)} + \mathbf{N}_5^{(e)} + \mathbf{N}_6^{(e)} \\
\nabla N_5^{(e)} &= \nabla(L_1L_2) = \mathbf{N}_7^{(e)} \\
\nabla N_6^{(e)} &= \nabla(L_1L_3) = \mathbf{N}_8^{(e)} \\
\nabla N_7^{(e)} &= \nabla(L_1L_4) = \mathbf{N}_9^{(e)} \\
\nabla N_8^{(e)} &= \nabla(L_2L_3) = \mathbf{N}_{10}^{(e)} \\
\nabla N_9^{(e)} &= \nabla(L_2L_4) = \mathbf{N}_{11}^{(e)} \\
\nabla N_{10}^{(e)} &= \nabla(L_3L_4) = \mathbf{N}_{12}^{(e)}.
\end{aligned} \tag{3.48}$$

Examining the formulas in (3.48), a scheme for calculating the $\nabla N_i^{(e)}$ can be discovered:

The gradients of the linear functions $N_1^{(e)}, \ldots, N_4^{(e)}$ are equal to the sum of the three linear vector basis functions that are associated with the edges connected to the respective node (cf. Fig. 3.2). If the edge points *to* the respective node, the vector basis function is added. If the edge points *away from* the respective node, the vector basis function is subtracted. So, e.g., inspecting Fig. 3.2, node 2 is connected to edges $\bar{1}$, $\bar{4}$ and $\bar{5}$. Edge $\bar{1}$ points to node 2, whereas edges $\bar{4}$ and $\bar{5}$ point away from node 2. According to the rule above, $\nabla N_2^{(e)}$ is equal to $+\mathbf{N}_1^{(e)} - \mathbf{N}_4^{(e)} - \mathbf{N}_5^{(e)}$.

The gradients of the quadratic basis functions $N_5^{(e)}, \ldots, N_{10}^{(e)}$ are even easier to compute, they are equivalent to the vector basis functions $\mathbf{N}_7^{(e)}, \ldots \mathbf{N}_{12}^{(e)}$.

This demonstrates that

$$\mathbf{grad}\, N_i^{(e)} \in \operatorname{span}\{\mathbf{N}_1^{(e)}, \ldots, \mathbf{N}_k^{(e)}\} \quad \forall i$$

holds for all elements e.

Going from local to global functions The rules for computing the gradients of the scalar element basis functions can be extended for the global basis functions:

The gradient of the basis function associated with mesh node j, $\mathbf{grad}\, N_j$ is a linear combination of the vector basis functions $\mathbf{N}_k$ associated with edges connected to node j. The linear factors are again either -1 or 1, determined by the direction of the edge. If the edge points *to* the node j, the factor is 1. If the edge points *away from* the node j, the factor is -1.

Analogous to the above paragraph, the gradient of the (quadratic) basis function associated with a particular edge mid-point is equal to the (quadratic) vector basis function associated with the edge, where the same mid-point is located.

Thus $\mathbf{grad}\, N_j \in \operatorname{span}\{\mathbf{N}_1, \ldots, \mathbf{N}_k\}$ for all j and furthermore

$$\mathbf{grad}\, N_j = \sum_{i=1}^{m} y_{ij} \mathbf{N}_i. \tag{3.49}$$

The factors y_{ij} are either -1, 0 or 1 depending on the rules given above. Since the N_j are associated with nodes and edge mid-points in the interior of the domain, all connected edges are also in the interior. Thus, despite certain boundary DOFs are eliminated to implement the boundary conditions, (3.49) still holds.

By inserting (3.49) into the definition of $\boldsymbol{C}$ in Equation (3.46) and considering the definition of $\boldsymbol{M}$ in Equation (3.27), it is obvious that the columns of $\boldsymbol{C}$ are linear combinations of the columns of $\boldsymbol{M}$,

$$\boldsymbol{C} = \boldsymbol{M}[\mathbf{y}_1, \ldots, \mathbf{y}_m] =: \boldsymbol{M}\boldsymbol{Y}, \quad \mathbf{y}_l = (y_{1l}, \ldots, y_{nl})^T, \tag{3.50}$$

where the factors y_{ij} form a sparse matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times n_c}$. As shown above, $\boldsymbol{Y}$ can be computed easily from the mesh data.

Sparse basis of the null space of $\boldsymbol{A}$ An important property of the mixed Nédélec-Lagrange finite element discretisation described in this section is that

$$\mathcal{N}(\boldsymbol{A}) = \mathcal{R}(\boldsymbol{Y}),$$

which is now shown:

$$\begin{aligned}
\{\boldsymbol{A}\boldsymbol{Y}\}_{ij} &= \sum_k a_{ik} y_{kj} \\
&= \sum_k \int_\Omega \mathbf{rot}\, \mathbf{N}_i \cdot \mathbf{rot}\, \mathbf{N}_k \, \mathrm{d}\Omega \, y_{kj} \\
&= \int_\Omega \mathbf{rot}\, \mathbf{N}_i \cdot \sum_k \mathbf{rot}\, \mathbf{N}_k y_{kj} \, \mathrm{d}\Omega,
\end{aligned}$$

and due to the linearity of the $\mathbf{rot}$-operator

$$= \int_\Omega \mathbf{rot}\, \mathbf{N}_i \cdot \mathbf{rot}\Big(\sum_k \mathbf{N}_k y_{kj}\Big) \, \mathrm{d}\Omega,$$

and by inserting Equation (3.49)

$$= \int_\Omega \mathbf{rot}\, \mathbf{N}_i \cdot \mathbf{rot}\, \mathbf{grad}\, q_j \, \mathrm{d}\Omega,$$

and since $\mathbf{rot}\,\mathbf{grad}\,q_j = \mathbf{0}$

$$= 0.$$

Thus we have

$$\boldsymbol{A}\boldsymbol{Y} = \mathbf{0}. \tag{3.51}$$

As shown in [35], the sparse matrix $\boldsymbol{Y}$ is a basis of the null space of $\boldsymbol{A}$.

Constrained matrix-eigenvalue problem The constrained matrix-eigenvalue problem now has the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \qquad \boldsymbol{C}^T\mathbf{x} = \mathbf{0}. \tag{3.52}$$

Since $\boldsymbol{Y}$ is a basis of the null space $\mathcal{N}(\boldsymbol{A})$, the constraint $\boldsymbol{C}^T\mathbf{x} = \boldsymbol{M}\boldsymbol{Y}^T\mathbf{x} = \mathbf{0}$ forces $\mathbf{x}$ to $\boldsymbol{M}$-orthogonal to $\mathcal{N}(\boldsymbol{A})$. The eigensolutions of (3.52) are therefore eigensolutions of $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x}$ with *positive* eigenvalues.

Methods for computing the smallest positive eigenvalues of (3.52) are presented in Section 4.2.

3.4 Linear or quadratic elements?

In Chapters 3.2 and 3.3 linear and quadratic node and vector elements were described. This section deals with the question, whether linear or quadratic elements should be preferred.

Given a mesh, linear elements have the advantage that they yield global matrices with smaller order. Furthermore the number of non-zeros per matrix row is smaller. The computation time is therefore substantially smaller. On the other hand, with quadratic elements more accurate solutions are computed due to their superior approximation properties. Furthermore, quadratic elements make it possible to use the efficient hierarchical preconditioner (cf. Section 8.3).

Because the linear basis functions are contained in the quadratic ones, it is evident that solutions computed with quadratic elements are always at least as accurate as solutions computed with linear elements, if the same mesh is used[7].

In the following numerical experiments we compare the ratio of computation time and the accuracy of the calculated eigenvalues for all element types.

To this end we use a rectangular brick-shaped domain $\Omega = (0, 5.2)\times(0, 3.3)\times(0, 0.77)$ for which we know the analytic eigensolutions (cf. Appendix C.2 on page 157). We discretise the domain Ω using different refinements and compute the ten lowest positive eigenvalues. Using the analytic solution we compute relative accuracy of the 10th largest eigenvalue λ_{10} and compare it with the spent computation time.

For this experiment we use the JDSYM eigenvalue solver and a stopping tolerance of $\varepsilon = 10^{-8}$ (cf. Section 5.22 on page 65). We chose the SSOR preconditioner as described in Section 8.1.2 for this calculation.

Fig. 3.6 shows the results for nodal elements[8]. The two curves indicate that quadratic node elements are far superior for all mesh sizes.

[7] This observation is however only exact for nodal elements. Due to the constraint $\operatorname{div}\mathbf{e} = 0$, the function space spanned by the linear basis functions is not contained in elements the function space spanned by the quadratic basis functions with vector elements.

[8] For the penalty parameter we chose $s = 1.5$.

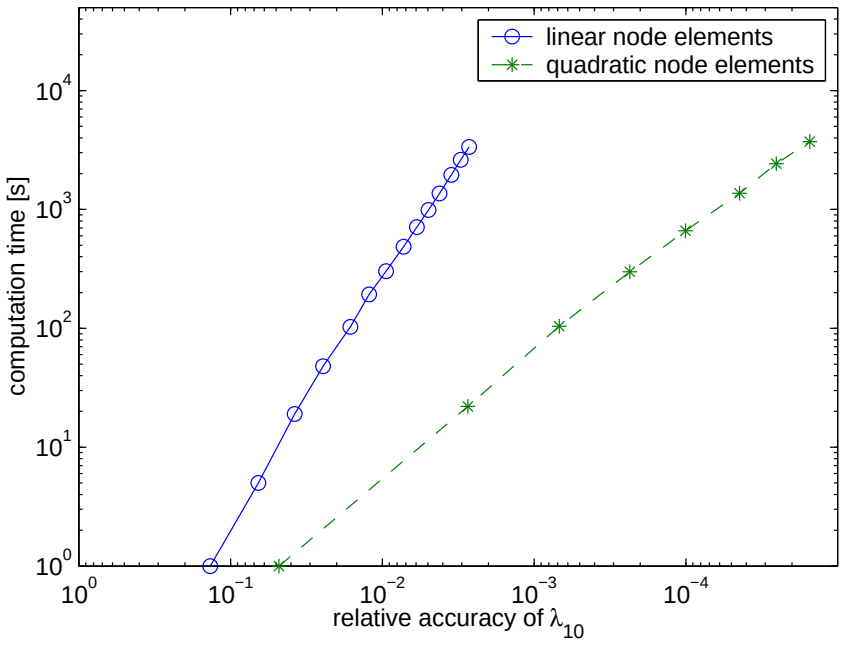


Figure 3.6: *Relative accuracy of λ_{10} vs computation time for node elements*

Fig. 3.7 shows the results for vector elements. To compute the smallest positive eigenvalues, we used the AD-method (cf. Section 4.2.3). The two curves show that quadratic elements are superior to linear elements for all relevant mesh sizes. Only for very coarse meshes, linear elements seem to be the better choice.

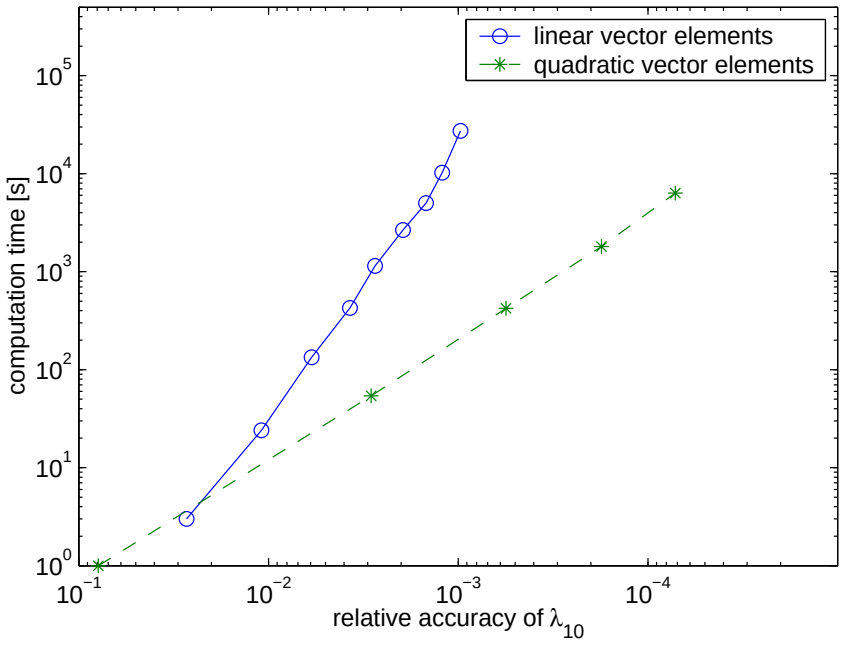


Figure 3.7: *Relative accuracy of λ_{10} vs computation time for vector elements*

3.4.1 Upshot

The results in Figs. 3.6 and 3.7 clearly indicate, that quadratic elements are to be preferred over linear elements. The possibility to use the hierarchical basis preconditioner instead of the SSOR-preconditioner makes the advantage even greater.

3.5 Comparison of FEM approaches

In Sections 3.2 and 3.3, two different approaches to discretise Maxwell's problem (3.8) using the FEM were introduced.

The *penalty method* uses ordinary node elements and leads to a matrix eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \tag{3.53}$$

with symmetric positive-definite matrices $\boldsymbol{A}$ and $\boldsymbol{M}$.

The *mixed method* uses vector elements and leads to a matrix eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \quad \boldsymbol{C}^T\mathbf{x} = \mathbf{0}, \tag{3.54}$$

with symmetric positive semi-definite matrix $\boldsymbol{A}$ and positive-definite matrix $\boldsymbol{M}$.

Speed To compare both methods, we calculate ten eigensolutions using four different meshes and compare the execution times. We use quadratic elements and the JDSYM eigensolver (cf. Section 5.1.2) for this experiment. For vector elements, the matrix eigenvalue problem (3.54) is solved using the AD-method as described in Section 4.2.3.

Quadratic node elements (penalty method)				
Mesh	it_{in}	it_{out}	t_{tot}	t_{eig}
box8x4x6	5.1	72	17.0	15.5
boxcav16x10x3	4.5	67	128.0	118.5
copcav18	4.9	70	215.0	201.5
cop20k	4.0	63	570.0	520.9
Quadratic vector elements				
box8x4x6	4.5	83	24.0	22.9
boxcav16x10x3	3.7	72	174.0	163.5
copcav18	5.1	78	319.0	302.1
cop20k	4.1	80	1181.0	1090.4

Table 3.1: *Comparison of execution times with penalty and mixed method*
The penalty parameter s was set to 1.5 for all meshes.

The results in Tab. 3.1 show that the ten eigensolutions are computed much faster with the penalty method. The matrix eigenvalue problem (3.53) can be handled more efficiently than (3.54). The reason for this is, that additional effort is needed to enforce the constraint $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$ (cf. Section 4.2).

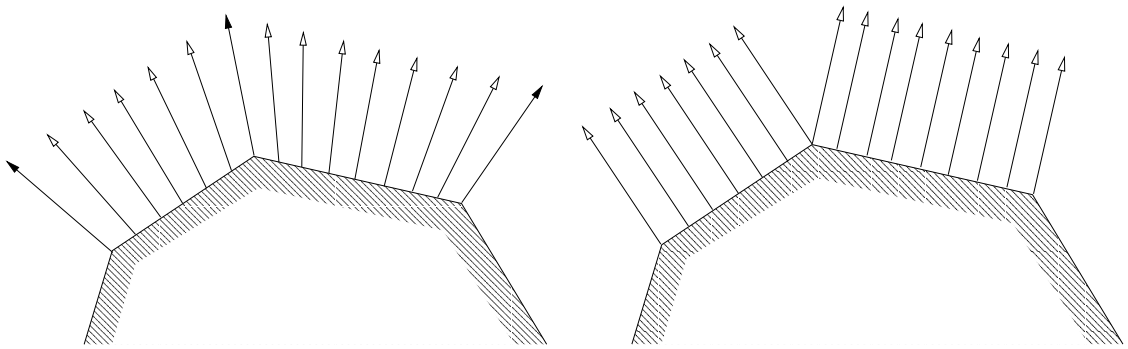
Tab. 3.2 shows eigenvalues calculated for rectangular brick-shaped domain (mesh *boxcav16x10x3*) using both approaches, together with the analytic eigenvalues. The numbers indicate that both FEM-approaches show similar accuracy[9].

[9] Actually the eigenvalues computed using vector finite elements are about one digit more accurate than the

	Node elements	Vector elements	Analytic
λ_1	53.79846641	53.79779566	53.79784076
λ_2	73.39956507	73.39645992	73.39657161
λ_3	95.31908131	95.30824191	95.30992408
λ_4	97.69334032	97.68127749	97.68216391
λ_5	107.61501301	107.59418189	107.59568152
λ_6	123.96242187	123.92506944	123.92922552
λ_7	125.46833267	125.42374390	125.42559190
λ_8	139.34042652	139.27131895	139.28485761
λ_9	146.88245386	146.78857970	146.79314322
λ_{10}	148.04689261	147.95031359	147.96324075

Table 3.2: *Eigenfrequencies in MHz of* boxcav16x10x3*, calculated using the penalty and the mixed method compared to analytically calculated eigenvalues.*

Boundary conditions The following difficulties arise when implementing the boundary conditions with the penalty method:



(a) (b)

Figure 3.8: *Boundary conditions for* (a) *vector- and* (b) *node elements*

Electrical field on a surface The boundary condition $\mathbf{n} \times \mathbf{e} = \mathbf{0}$ from (3.3) demands, that the electrical field $\mathbf{e}$ is perpendicular to the surface, i.e. collinear to the surface normal vector $\mathbf{n}$.

With node elements, the electrical field is defined through the x-, y- and z-components at the node points. We approximate the surface normal vector $\mathbf{n}$ at surface node points using the mesh geometry data and force the electrical field to be collinear to $\mathbf{n}$ (cf. Section 3.2.4) at these points. The electrical field is interpolated between the surface node points. As can be seen from Fig. 3.8a, the interpolated electrical field is not orthogonal to the surface.

With vector elements, the degrees of freedom themselves are vector functions. All degrees of freedom on the surface are eliminated. The remaining degrees of freedom are either zero on the surface or are orthogonal to it. Since the representable field is allowed to be discontinuous

eigenvalues computed using node elements. This can be explained with the fact that the dimension of the eigenvalue problem generated using vector elements is about 50% larger than the dimension of the eigenvalue problem generated using node elements (cf. Tab. E.1).

at the element boundaries, it is possible to have the electrical field orthogonal to the surface everywhere (cf. Fig. 3.8b).

Treatment of physical edges on the surface Another problem arises when dealing with physical edges on the surface. Note that here we do not mean edges of the tetrahedral mesh lying on the surface.

When using node elements, such edges require special treatment (cf. Section 3.2.4). Because the electrical field cannot be orthogonal to two faces, it must vanish at node points lying on such edges. The field components are set to zero here.

With vector elements however, no special treatment is required.

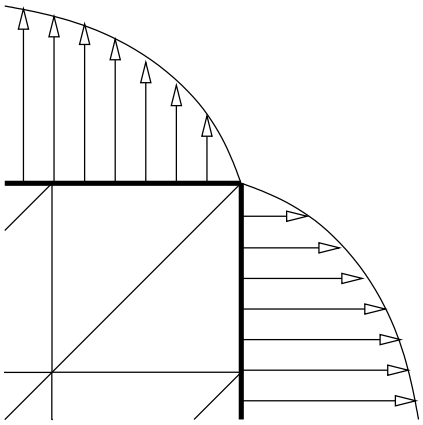
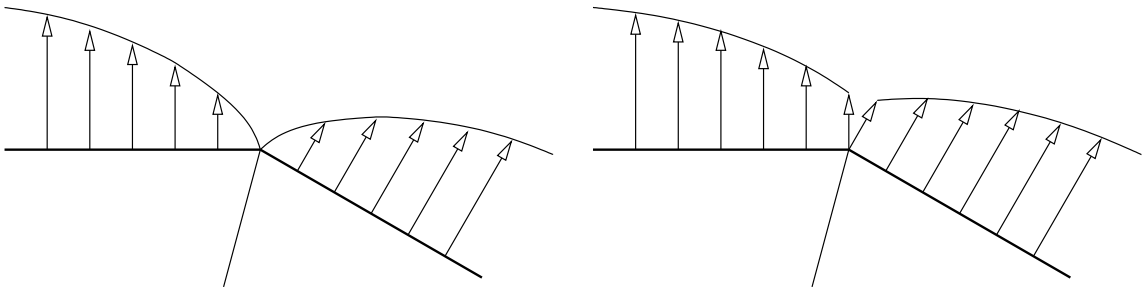


Figure 3.9: *Electrical field on a 90° edge*

If both faces adjacent to the edge are orthogonal, the electrical field can be represented correctly with both node and vector elements, i.e. the properties of the analytic solution can be represented (cf. Appendix C.2).



(a) (b)

Figure 3.10: *Boundary conditions of node and vector elements*

If the adjacent faces form an obtuse angle, vector elements can represent a more realistic solution. As can be seen from Fig. 3.10a, the field disappears completely on the edge when using node elements. Vector elements are more flexible: the field is orthogonal to both faces, but does not have to disappear on the edge (cf. Fig. 3.10b). The field representable by node elements is too smooth.

Due to the reasons above, the electrical field representable by node elements is unrealistic for arbitrary domains Ω. To illustrate this effect, we compute the ten smallest positive eigenvalues for an "eight-shaped" cavity (mesh *copcav18*, cf. Appendix D) with both types of elements.

	Node elements	**Vector elements**
λ_1	57.00939678	58.11875417
λ_2	90.29582738	97.71888041
λ_3	111.51167826	104.73392285
λ_4	125.11387346	130.67271534
λ_5	129.24351844	140.82439688
λ_6	149.47719254	150.60510255
λ_7	152.21375981	165.05269329
λ_8	156.47921848	169.36294623
λ_9	169.22317313	184.98521572
λ_{10}	175.86034665	187.45850060

Table 3.3: *Eigenvalues of* copcav18 *calculated using quadratic node and vector elements.* The analytic eigenvalues for this domain are not available.

The results in Tab. 3.3 show the deviations in the eigenvalues calculated using node and vector elements. These deviations can be traced back primarily to the fact, that the electrical field cannot be accurately represented on the surface of mesh *copcav18* when using node elements. For the rectangular brick-shaped mesh *boxcav16x10x3* the field can be represented correctly, which manifests itself in the accurate eigenvalues listed in Tab. 3.2.

Conclusion Discretisations with node elements lead to matrix eigenvalue problems, that can be solved more efficiently, than discretisations stemming from vector elements. However, finding a suitable value for the penalty parameter s may be costly. For arbitrary domains Ω, it is not possible to implement the boundary conditions correctly with node elements. This can lead to substantial inaccuracies.

Node elements can only be used safely, if all surfaces of Ω are planar, i.e. Ω represents a polyhedron, and all adjacent faces of this polyhedron form acute angles ($\leq 90°$). Examples for such polyhedra are the rectangular cuboid or the equilateral tetrahedron. If Ω does not satisfy these properties, vector elements must be used.

4 Matrix eigenvalue problems

Two mathematical problems were introduced in Chapters 2 and 3. Both of them result in a generalised matrix eigenvalue problem. The emerging matrix eigenvalue problems can be categorised into two classes: *positive definite eigenvalue problems* and *indefinite eigenvalue problems with constraints*.

To calculate the desired eigensolutions we use on the one hand an implementation of the *Jacobi-Davidson algorithm* called JDSYM, which was developed in the course of this dissertation. On the other hand, we use the widely-known ARPACK package, which implements the Implicitly Restarted Lanczos method (IRL).

This chapter discusses, how the two types of eigenvalue problems can be solved efficiently using JDSYM and ARPACK. The eigenvalue solvers themselves are discussed in Chapter 5.

4.1 Positive definite eigenvalue problems

Positive definite eigenvalue problems

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \quad \boldsymbol{A}, \boldsymbol{M} > 0, \tag{4.1}$$

result from Problem I and Problem II, if node elements are used. Because $\boldsymbol{A}$ as well as $\boldsymbol{M}$ are symmetric positive-definite, the eigenvalue problem (4.1) has all real and positive eigenvalues. According to the problem statements in Chapters 2 and 3, 10 to 20 of the smallest eigenvalues together with corresponding eigenvectors of (4.1) are to be computed.

To compute these eigensolutions we use the *shift-and-invert* spectral transformation with ARPACK. The transformed eigenvalue problem is

$$(\boldsymbol{A} - \sigma \boldsymbol{M})^{-1}\boldsymbol{M}\mathbf{x} = \mu \mathbf{x}, \quad \mu := \frac{1}{\lambda - \sigma}, \quad \boldsymbol{A}, \boldsymbol{M} > 0. \tag{4.2}$$

The shift σ is chosen smaller than the smallest eigenvalue of (4.1) (cf. Section 4.3 on page 55). With this choice of σ, all linear systems to be solved in ARPACK are positive-definite.

In JDSYM the expansion of the search space is computed by solving the so-called correction equation (5.24) which has the form[10]

$$\boldsymbol{P}(\boldsymbol{A} - \sigma \boldsymbol{M})\boldsymbol{P}^T\mathbf{v} = \boldsymbol{P}\mathbf{r}. \tag{4.3}$$

The linear systems to be solved are not necessarily positive-definite, since the shift σ is not constant in JDSYM. Usually the shift is set to the Ritz value closest to a given target value τ, which is chosen a little smaller than the smallest eigenvalue of (4.1) (cf. Section 5.1.2).

The symmetric positive definite eigenvalue problem (4.2) can be solved in a straight-forward manner without special considerations. Due to the appropriate choice of σ and τ respectively, ARPACK and JDSYM converge to the desired smallest positive eigenvalues and corresponding eigenvectors.

4.2 Indefinite eigenvalue problem

The second kind of eigenvalue problems of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \qquad \boldsymbol{A}, \boldsymbol{M} \in \mathbb{R}^{n\times n} \tag{4.4a}$$

$$\boldsymbol{Y}^T\boldsymbol{M}\mathbf{x} = \boldsymbol{C}^T\mathbf{x} = \mathbf{0}, \qquad \boldsymbol{C} \in \mathbb{R}^{n\times n_c} \tag{4.4b}$$

[10] The correction equation is discussed in Section 5.1.2. The matrix $\boldsymbol{P}$ is a projector.

emerges from Problem II, if vector finite elements are used. Here $\boldsymbol{A}$ is symmetric positive semi-definite and $\boldsymbol{M}$ symmetric positive-definite. The eigenvalues of (4.4) are all real and non-negative. According to the problem statement in Chapter 3, we have to compute 10 to 20 of the smallest *positive* eigenvalues together with corresponding eigenvectors of (4.1). Because (4.4a) shows the eigenvalue 0 with high multiplicity[11], the desired eigenvalues are located in the interior of the spectrum of (4.4a).

The following discussion is going to demonstrate, that the eigenvalue problem (4.4) cannot be solved using the shift-and-invert approach on $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x}$ in a straight-forward manner.

For the *shift-and-invert* transformed eigenvalue problem

$$(\boldsymbol{A} - \sigma\boldsymbol{M})^{-1}\boldsymbol{M}\mathbf{x} = \frac{1}{\lambda - \sigma}\mathbf{x}, \quad \boldsymbol{A} \geq 0,\ \boldsymbol{M} > 0, \tag{4.5}$$

the shift σ must be chosen in the interior of the spectrum, in order that the absolute values of the transformed desired eigenvalues become large. However, then the linear systems to be solved become indefinite.

The transformed zero eigenvalues have an absolute value of $\frac{1}{|\sigma|}$. If σ is chosen near the desired eigenvalues, then the eigenvector approximations in ARPACK and JDSYM will always have substantial components in the unwanted null space. This severely deteriorates the convergence to the desired eigensolutions.

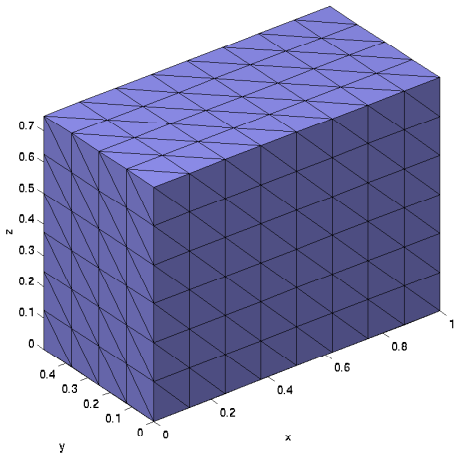


Figure 4.1: *mesh* box8x4x6

We use a cuboid with dimensions $[1.0 \times 0.5 \times 0.75]$ to illustrate these difficulties. The five smallest eigenvalues together with the corresponding eigenvectors shall be computed using linear vector elements. The cuboid is discretised using 1152 tetrahedral elements (cf. Fig. 4.1). The resulting eigenvalue problem has order $n = 1050$ and null space of dimension $\dim(\mathcal{N}(\boldsymbol{A})) = 105$. The five smallest positive eigenvalues are $\lambda_1 \approx 27.3$, $\lambda_2 \approx 48.8$, $\lambda_3 \approx 56.5$, $\lambda_4 \approx 56.6$ and $\lambda_5 \approx 67.1$.

Usually the shift σ is chosen as an approximation of the smallest positive eigenvalue. Fig. 4.2 shows all eigenvalues λ_i smaller than 70 on the x-axis and the corresponding transformed eigenvalues $\nu_i = \frac{1}{\lambda_i - \sigma}$ for $\sigma = 22$ on the y-axis. Since ν_1 is by far the largest eigenvalue, an eigensolver will converge rapidly to eigenvalue λ_1. The components in direction of the corresponding eigenvector $\mathbf{x}_1$ are strongly amplified. It is however tougher to compute the other

[11] multiplicity n_c

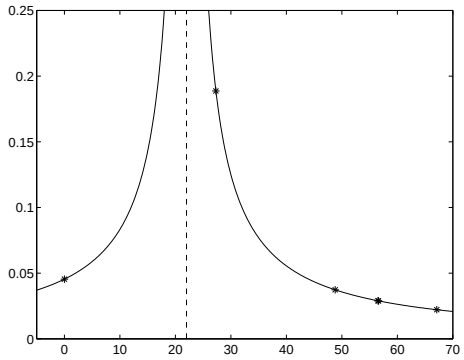


Figure 4.2: *Absolute values of shift-and-invert transformed eigenvalues for* $\sigma = 22$

non-zero eigenvalues. Their transformed eigenvalues are smaller in magnitude than the transformed zero-eigenvalues. The high dimension of the null space makes it even more difficult.

The shift-and-invert spectral transformation is unable to calculate the desired smallest positive eigenvalues of (4.4a). Sections 4.2.1 to 4.2.5 discuss five methods that accomplish this. All these techniques (with the exception of EIGSOLV in Section 4.2.5) make use of an important feature of the mixed Nédélec-Lagrange FEM-discretisation (cf. Section 3.3.5)

$$\boldsymbol{C}^T\mathbf{x} = \mathbf{0} \iff \mathbf{x} \perp_{\boldsymbol{M}} \mathcal{N}(\boldsymbol{A}). \tag{4.6}$$

Eigenvectors that satisfy the constraint $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$ are associated with *positive* eigenvalues and vice-versa.
The property (4.6) can be used in the following ways:

- ▷ If the constraint $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$ is enforced during the solution of the eigenvalue problem, then the desired eigenvalues become extremal. Therefore the shift-and-invert approach can be used as in Section 4.1.
- ▷ If the eigensolutions $(\mathbf{x}_i, \lambda_i)$ with positive eigenvalues λ_i are computed by any means, then these solutions automatically satisfy the constraint $\boldsymbol{C}^T\mathbf{x}_i = \mathbf{0}$.

In the following the five methods for computing the desired solutions of (4.4) are presented.

4.2.1 Direct projection method (DIRPROJ)

The desired eigenpairs $(\lambda, \mathbf{x})$ (with positive λ) satisfy $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$. We use the $\boldsymbol{M}$-orthogonal projector

$$\boldsymbol{P}_{\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}} = \boldsymbol{I} - \boldsymbol{Y}\boldsymbol{H}^{-1}\boldsymbol{C}^T, \qquad \boldsymbol{H} := \boldsymbol{Y}^T\boldsymbol{C}, \tag{4.7}$$

to keep the search space of the eigensolver in $\mathcal{N}(\boldsymbol{C}^T)$.

The direct projection method uses the property, that both $\boldsymbol{M}$ and $\boldsymbol{A} - \sigma\boldsymbol{M}$ map $\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}$ one-to-one onto $\mathcal{R}(\boldsymbol{Y})^{\perp}$. Since the eigensolver only performs operations like vector updates, dot products and matrix multiplications with the matrices mentioned above, it is sufficient to make the initial subspace satisfy the condition (4.6) using the projector $\boldsymbol{P}_{\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}}$.

ARPACK as well as JDSYM solve linear systems involving the shifted operator $\boldsymbol{A} - \sigma\boldsymbol{M}$ (cf. JDSYM's correction equation (5.24) and equation (5.43) for ARPACK). These equations have to be solved subject to the linear constraint $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$. I.e., in both cases we have to solve systems of the form

$$(\boldsymbol{A} - \sigma\boldsymbol{M})\mathbf{x} = \mathbf{b}, \quad \boldsymbol{C}^T\mathbf{x} = \mathbf{0}. \tag{4.8}$$

Typically, a preconditioner is used to accelerate the linear solver. After invoking the preconditioner, the projector $\boldsymbol{P}_{\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}}$ must be applied, since otherwise the preconditioned vector would no longer be in $\mathcal{N}(\boldsymbol{C}^T)$.

The actual implementation of this method simply inserts projections (4.7) at certain places in the algorithm. As stated above, it is essential to project the initial subspace and the preconditioned vectors. Due to rounding errors, additional projections are needed: We project the right hand sides of the linear systems (4.8). Otherwise, the convergence would deteriorate, since the components in direction of $\mathcal{R}(\boldsymbol{Y})$ would never disappear from the residual vector. Additionally, we project the approximate solution $\mathbf{x}$ of the linear system (4.8).

Without those additional projections, the method is unstable and does not converge in certain cases.

4.2.2 Simplified augmented system (SAUG)

In contrast to the direct projection method, the linear system (4.8) is solved differently with the SAUG method [6].

The linear system (4.8) can be viewed at as an augmented system

$$\begin{bmatrix} \boldsymbol{A} - \sigma \boldsymbol{M} & \boldsymbol{C} \\ \boldsymbol{C}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{x}' \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}. \tag{4.9}$$

The matrix in (4.9) is non-singular if $\sigma < \lambda_1$. For any $\mathbf{x}$ satisfying $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$, the vector $\boldsymbol{A} - \sigma\boldsymbol{M}\mathbf{x}$ is orthogonal to $\mathcal{R}(\boldsymbol{Y})$. Therefore, if $\boldsymbol{Y}^T\mathbf{b} = \mathbf{0}$, then the solution $\mathbf{x}$ from (4.8) satisfies the second equation of (4.9). Since $\boldsymbol{C}$ has full rank and due to (4.8), the first equation of (4.9) holds if and only if $\mathbf{x}' = 0$.

The symmetric indefinite matrix in (4.9) has an order of $n + n_\mathrm{c}$. It has n positive and n_c negative eigenvalues. The advantage of solving the augmented system (4.9) instead of the original system (4.8) is, that no explicit constraints have to be considered. On the other hand, the augmented system (4.9) has more unknowns. Also, to enforce the constraint $\boldsymbol{C}^T\mathbf{x} = \mathbf{0}$, the augmented system has to be solved accurately.

It can be shown that this method is very similar to the direct projection method with respect to the actual implementation: If a preconditioner of the form

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{C} \\ \boldsymbol{C}^T & \mathbf{0} \end{bmatrix}, \quad \text{with} \quad \boldsymbol{K} \approx \boldsymbol{A} - \sigma\boldsymbol{M}$$

is used, then the implementation is the same, with the exception of the preconditioner. Instead of $\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{H}^{-1}\boldsymbol{C}^T$, the more complicated term $\boldsymbol{I} - \boldsymbol{K}^{-1}\boldsymbol{C}(\boldsymbol{C}^T\boldsymbol{K}^{-1}\boldsymbol{C})^{-1}\boldsymbol{C}^T\boldsymbol{K}^{-1}$ has to be used for the preconditioner [6].

This augmented method can be simplified and made more efficient: As shown in [6], the method is modified by using a cheaper preconditioner of the form

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{C}_1 \\ \boldsymbol{C}_1^T & \mathbf{0} \end{bmatrix}, \qquad \boldsymbol{C}_1 = \boldsymbol{M}_1\boldsymbol{Y}. \tag{4.10}$$

$\boldsymbol{M}_1$ is the 2×2 block diagonal matrix[12], constructed from $\boldsymbol{M}$. The initial guess $\mathbf{x}_0$ must satisfy $\boldsymbol{C}^T\mathbf{x}_0 = \mathbf{0}$ and the right hand side $\mathbf{b}$ has to be in the range of the operator, i.e. $\boldsymbol{Y}^T\mathbf{b} = \mathbf{0}$,

[12] The 2×2 block structure results from the hierarchical construction of the basis functions as explained in Section 3.3.3. For linear elements $\boldsymbol{M}_1$ is simply equal to $\boldsymbol{M}$.

for the above simplifications to be valid. If the Conjugate Gradient (CG) method is applied to solve (4.9) using the preconditioner (4.10), the method is simplified in such a way, that all operations with $\boldsymbol{C}$ and $\boldsymbol{C}_1$ become superfluous.

Hence in effect, the standard preconditioned CG method is applied to the system $(\boldsymbol{A} - \sigma\boldsymbol{M})\mathbf{x} = \mathbf{b}$. As with the the direct projection method, certain vectors are kept in $\mathcal{R}(\boldsymbol{Y})^{\perp_{\boldsymbol{M}}}$ using projections. But in contrast to the DIRPROJ method, the projection in the preconditioning step is omitted. In effect, no projections at all are necessary in the inner iteration loop. The name "Simplified augmented system" refers to the underlying idea and not to the actual implementation of the method.

Since the system matrix is not positive definite, it is unclear whether this method is stable. The constraint $\boldsymbol{C}^T\mathbf{x}' = \mathbf{0}$ is not satisfied exactly until the iterative method has converged. If the iteration is stopped earlier, as e.g. with the Jacobi-Davidson method, the constraint has to be enforced explicitly after the iteration (by projection).

Even if the stability of the SAUG method could not be proven, the experimental results in Sections 4.2.6, 4.3 and 8.4 show very good results anyway. And even if the validity of the SAUG method has only been derived for the CG iteration, it also worked well with other iterative methods like SYMMLQ, MINRES, QMRS and CGS.

4.2.3 AD method (AD)

The AD method by Arbenz and Drmač [3] is a technique for accurately and efficiently calculating the Cholesky factors of a symmetric positive semi-definite matrix, if a basis of the null space of that matrix is known. The insights gained from this method can be used for solving the matrix eigenvalue problem (4.4).

Let $\boldsymbol{Y} \in \mathbb{R}^{n\times n_c}$ be the matrix, whose columns span the null space of $\boldsymbol{A}$ and let $\boldsymbol{C} = \boldsymbol{M}\boldsymbol{Y}$ (cf. Section 3.3.5). Let $\boldsymbol{W}$ be the matrix

$$\boldsymbol{W} := \begin{bmatrix} \boldsymbol{I}_{n-n_c} & \boldsymbol{Y}_1 \\ \mathbf{0} & \boldsymbol{Y}_2 \end{bmatrix}, \qquad \boldsymbol{Y} = \begin{bmatrix} \boldsymbol{Y}_1 \\ \boldsymbol{Y}_2 \end{bmatrix}, \quad \boldsymbol{Y}_2 \in \mathbb{R}^{n_c\times n_c}.$$

We assume, that $\boldsymbol{Y}_2$ is invertible. If that should not be the case, the matrix has to be permuted accordingly. The following equations hold:

$$\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{W} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \boldsymbol{Y}_1^T & \boldsymbol{Y}_2^T \end{bmatrix} \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \boldsymbol{Y}_1 \\ \mathbf{0} & \boldsymbol{Y}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

and

$$\boldsymbol{W}^T\boldsymbol{M}\boldsymbol{W} = \begin{bmatrix} \boldsymbol{M}_{11} & \boldsymbol{C}_1 \\ \boldsymbol{C}_1^T & \boldsymbol{H} \end{bmatrix}$$

with

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_1 \\ \boldsymbol{C}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_{11} & \boldsymbol{M}_{12} \\ \boldsymbol{M}_{21} & \boldsymbol{M}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{Y}_1 \\ \boldsymbol{Y}_2 \end{bmatrix}, \qquad \boldsymbol{H} = \boldsymbol{Y}^T\boldsymbol{M}\boldsymbol{Y} = \boldsymbol{Y}^T\boldsymbol{C}.$$

The matrices $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$ are positive definite. By using the 2×2-block LDL^T-factorisation

$$\boldsymbol{W}^T\boldsymbol{M}\boldsymbol{W} = \begin{bmatrix} \boldsymbol{M}_{11} & \boldsymbol{C}_1 \\ \boldsymbol{C}_1^T & \boldsymbol{H} \end{bmatrix} = \boldsymbol{P}^T \begin{bmatrix} \boldsymbol{S} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{H} \end{bmatrix} \boldsymbol{P}, \qquad \boldsymbol{P} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \boldsymbol{H}^{-1}\boldsymbol{C}_1^T & \boldsymbol{I} \end{bmatrix}$$

with the Schur complement $\boldsymbol{S} = \boldsymbol{M}_{11} - \boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T$, and in consideration of

$$\boldsymbol{P}^{-T}\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{W}\boldsymbol{P}^{-1} = \boldsymbol{W}^T\boldsymbol{A}\boldsymbol{W},$$

we get the matrix eigenvalue problem

$$\begin{bmatrix} \boldsymbol{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \lambda \begin{bmatrix} \boldsymbol{S} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{H} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} \tag{4.11}$$

with

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \boldsymbol{P}\boldsymbol{W}^{-1}\mathbf{x}.$$

In the matrix eigenvalue problem (4.11) $\mathbf{y}_1$ and $\mathbf{y}_2$ are decoupled. Since $\lambda > 0$, $\mathbf{y}_2 = \mathbf{0}$ holds, and hence only the $(1,1)$-block has to be considered. Thus, the positive eigenvalues of (4.4) are the eigenvalues of

$$\boldsymbol{A}_{11}\mathbf{y}_1 = \lambda(\boldsymbol{M}_{11} - \boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T)\mathbf{y}_1 = \lambda\boldsymbol{S}\mathbf{y}_1. \tag{4.12}$$

The matrix $\boldsymbol{S} = \boldsymbol{M}_{11} - \boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T$ is positive-definite. If $\mathbf{y}_1$ is an eigenvector of (4.12), then because of

$$\boldsymbol{P}^{-1} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ -\boldsymbol{H}^{-1}\boldsymbol{C}_1^T & \boldsymbol{I} \end{bmatrix},$$

the corresponding eigenvector of (4.4) is calculated by

$$\mathbf{x} = \boldsymbol{W}\boldsymbol{P}^{-1}\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 - \boldsymbol{Y}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T\mathbf{y}_1 \\ -\boldsymbol{Y}_2\boldsymbol{H}^{-1}\boldsymbol{C}_1^T\mathbf{y}_1 \end{bmatrix} = (\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{H}^{-1}\boldsymbol{C}^T)\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{0} \end{bmatrix}. \tag{4.13}$$

Implementation The global matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}$ are constructed as usual. Afterwards n_c rows from $\boldsymbol{Y}$ are selected, which then form the non-singular matrix $\boldsymbol{Y}_2$.

The basic procedure is the following: For each column j of the sparse matrix $\boldsymbol{Y}$ we identify the largest row index i_j of all non-zero entries in that column, i.e. $i_j = \max(i \,|\, y_{ij} \neq 0)$. If all i_j are distinct, then $\boldsymbol{Y}_2$ can be built from the rows $i_1, \ldots, i_{n_c}$. $\boldsymbol{Y}_2$ is non-singular, since it can be brought into triangular shape using column permutations.

In general the i_j will not be all distinct. Therefore the algorithm needs to be extended slightly: As stated above, for each column the largest row index of all non-zero entries in that column is identified. If that row was already eliminated for an earlier column, a Gauss elimination step is performed, which eliminates the non-zero entry with largest row-index in the current column. This is repeated until a row index is identified, that has not yet been eliminated.

After the calculation of the row indices $\mathbf{i} = (i_1, \ldots, i_{n_c})$ the matrix $\boldsymbol{Y}_2 = \boldsymbol{Y}(\mathbf{i}, :)$ is constructed. Afterwards the rows $i_1, \ldots, i_{n_c}$ are eliminated from the matrices $\boldsymbol{Y}$ and $\boldsymbol{C}$, which yields the matrices $\boldsymbol{Y}_1$ and $\boldsymbol{C}_1$. From $\boldsymbol{A}$ and $\boldsymbol{M}$ the corresponding rows *and* columns are eliminated to form $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$. This elimination of rows and columns can be done in place.

In the eigenvalue solver the matrix vector product

$$\mathbf{y} \leftarrow \boldsymbol{M}\mathbf{x}$$

is replaced by

$$\mathbf{y} \leftarrow (\boldsymbol{M}_{11} - \boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T)\mathbf{x}.$$

It is not advisable to construct the matrix $\boldsymbol{M}_{11} - \boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T$ explicitly, since $\boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T$ is not sparse.

The calculated eigenvectors are transformed back according to (4.13).

```
function i = AD_ELIM(n, n_c, Y)
    g = zeros(n, 1)
    i = zeros(n_c, 1)
    E = sparse(n, n_c)
    for j = 1 to n_c
        c = Y(:, j)
        Select i_j = max(i | c_i ≠ 0)
        while g(i_j) ≠ 0
            k = g(i_j)
            α = −E(i_k, k)/c(i_j)
            c = αc + E(:, k)
            Select i_j = max(i | c_i ≠ 0)
        end
        E(:, j) = c
        g(i_j) = j
    end
end
```

Algorithm 4.1: *Selection of rows*
This procedure computes the indices i_j of the rows, which are selected from the matrix $\boldsymbol{Y}$ to form the non-singular matrix $\boldsymbol{Y}_2$.

Remarks If IRL is used as the eigensolver, then it could be beneficial to use the shift $\sigma = 0$. In this way, no expensive operations with $\boldsymbol{C}_1\boldsymbol{H}^{-1}\boldsymbol{C}_1^T$ have to be performed in the inner iterations. Even with JDSYM, it could be beneficial to set $\tau = 0$. However, due to the dynamic shift, not all correction equations will be solved with $\theta = 0$.

Tab. 4.1 compares the performance of the AD method, if σ and τ are set in different ways. In one case σ and τ are set to zero, and in the other case σ and τ are set a little smaller than the smallest positive eigenvalue.

The results in Tab. 4.1 clearly indicate that it is advantageous to set $\sigma = 0$ or $\tau = 0$, respectively.

4.2.4 Bespalov term (BESPALOV)

In [14] Bespalov suggests to solve the eigenvalue problem

$$(\boldsymbol{A} + \boldsymbol{C}\boldsymbol{S}\boldsymbol{C}^T)\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \quad \boldsymbol{A} \geq 0,\ \boldsymbol{M}, \boldsymbol{S} > 0$$

instead of (4.5). Since $\boldsymbol{S}$ is positive definite, the zero eigenvalues are moved to the right in the spectrum. The desired eigensolutions are not affected, since they are in $\mathcal{N}(\boldsymbol{C}^T)$. This idea is similar to the penalty method described in Section 3.2.

As suggested by Bespalov, we choose $\boldsymbol{S} = \alpha \boldsymbol{I}$ with $\alpha > 0$. The parameter α must be chosen large enough, such that shifted zero-eigenvalues do not interfere with the desired ones. However, as α is increased, the condition of the linear systems to be solved becomes more and more ill-conditioned. In addition, inaccuracies in the calculated solutions may emerge.

Choosing a suitable α is a difficult matter. The parameter α depends on the fineness of the mesh, since $\|\boldsymbol{A}\|$ grows with the mesh size, but $\|\boldsymbol{M}\|$ and $\|\boldsymbol{C}\|$ do not. We have not been able to find a good heuristic, and thus we choose α empirically.

Grid	σ *resp.* τ	it_{in}	it_{out}	t_{tot}	t_{eig}
JDSYM					
boxcav16x10x3	0.0	3.7	35	88.0	77.4
boxcav16x10x3	1.15	3.7	35	114.0	103.6
copcav18	0.0	5.0	37	157.0	140.9
copcav18	1.4	4.6	36	201.0	185.1
cop20k	0.0	4.0	39	595.0	507.6
cop20k	5.0	4.1	39	829.0	741.1
ARPACK					
boxcav16x10x3	0.0	28.0	26	156.0	145.4
boxcav16x10x3	1.15	28.2	26	281.0	271.0
copcav18	0.0	35.2	26	262.0	246.0
copcav18	1.4	39.5	26	605.0	588.4
cop20k	0.0	28.1	26	792.0	704.1
cop20k	5.0	28.8	26	1795.0	1705.7

Table 4.1: *AD method: Choice of* σ
it_{in} is the average number of inner iterations per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh the generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem.

α	it_{in}	it_{out}	t_{tot}	t_{eig}	t_{11}
10^0	*convergence to wrong eigensolutions*				
10^1	*no convergence*				
10^2	*no convergence*				
10^3	*no convergence*				
$5 \cdot 10^3$	*no convergence*				
10^4	374	72	36401	36334	9548
$5 \cdot 10^4$	339	64	30180	30115	7956
10^5	571	63	48586	48652	12741
$5 \cdot 10^5$	572	68	52474	52407	13879
10^6	641	58	47948	47883	12607
$5 \cdot 10^6$	766	61	64807	64742	17133
10^7	708	81	81943	81878	21503

Table 4.2: *Bespalov method: choice of parameter* α

Tab. 4.2 shows the influence of parameter α on the convergence of the Bespalov method. For this experiment, we use the mesh *cop10k_Xv2* (cf. Appendix D) and quadratic vector elements and calculate the smallest five positive eigenvalues and the corresponding eigenvectors to an accuracy of $\varepsilon = 10^{-4}$. The boundary condition $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ is enforced on all symmetry planes. The JDSYM eigensolver is used for this experiment.

If $\alpha \leq 5 \cdot 10^3$, then either the wrong eigensolutions are computed, or JDSYM doesn't converge at all. The zero eigenvalues were not shifted far enough. For $\alpha \geq 10^4$ JDSYM converges to the desired eigensolutions. If α is chosen too large, then the condition of the matrix

of the correction equation worsens and the number of inner iterations increases accordingly. This manifests itself in longer computation times.

Implementing the Bespalov method is rather straight-forward: In the eigensolver, each matrix-vector multiplication with $\boldsymbol{A}$ is replaced by a multiplication with $\boldsymbol{A} + \alpha \boldsymbol{C}\boldsymbol{C}^T$.

4.2.5 Eigensolver (EIGSOLV)

Instead of transforming the eigenvalue problem or introducing projections onto $\mathcal{N}(\boldsymbol{C}^T)$, convergence to the desired eigensolutions can also be ensured by extending the capabilities of the eigenvalue solvers themselves accordingly.

JDSYM can avoid convergence to zero eigenvalues by adaptive adjustment of the target value τ and clever sorting of the Ritz values (cf. Appendix 5.1.5). ARPACK can be configured to calculate only eigenvalues larger than σ. This is done by selecting the shifts for QR-sweeps accordingly (cf. Appendix 5.2.1).

The EIGSOLV method has the advantage that *no* expensive operations with $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$ are necessary. These matrices are not required by the EIGSOLV method and thus they don't need to be constructed. As a consequence, the EIGSOLV method consumes less memory than the other methods. In particular, not needing to store the Cholesky factor of $\boldsymbol{H}$ saves a lot of space.

4.2.6 Experimental results

In this section, the above methods are compared by means of computational examples. For these experiments, we use the four meshes *boxcav16x10x3*, *copcav18*, *cop10k* and *cop20k*[13] (cf. Appendix D) and compute the smallest five positive eigenvalues and the corresponding eigenvectors using quadratic vector elements up to an accuracy of at least 10^{-4}. The experiments are performed for both eigensolvers (ARPACK and JDSYM) using the 2LevJACssor preconditioner (cf. Section 8.3).

JDSYM					
Method	it_{in}	it_{out}	t_{tot}	t_{eig}	t_c
DIRPROJ	3.7	35	104.0	94.7	14.2
SAUG	4.9	33	**79.0**	**69.2**	16.2
AD	3.7	35	87.0	76.5	18.3
BESPALOV	439.9	63	9988.0	9955.6	2464.8
EIGSOLV	7.6	45	141.0	136.1	29.9
ARPACK					
DIRPROJ	31.0	26	296.0	285.3	56.2
SAUG	30.8	26	187.0	177.3	58.5
AD	28.0	26	**154.0**	**144.1**	66.3
BESPALOV	4057.0	26	32823.0	32790.7	9641.3
EIGSOLV	35.3	26	223.0	218.6	67.6

Table 4.3: *Comparison of methods for the indefinite eigenvalue problem with mesh* boxcav16x10x3.

[13] For the meshes *cop10k* and *cop20k* the condition $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ is enforced on all symmetry planes.

JDSYM					
Method	it_{in}	it_{out}	t_{tot}	t_{eig}	t_c
DIRPROJ	4.8	39	204.0	188.4	34.5
SAUG	6.5	35	**144.0**	**128.5**	39.6
AD	5.0	37	148.0	132.6	33.4
BESPALOV	753.5	66	25995.0	25939.7	6139.2
EIGSOLV	11.5	51	307.0	299.7	94.3
ARPACK					
DIRPROJ	38.8	26	610.0	593.4	138.2
SAUG	37.3	26	342.0	326.4	137.7
AD	35.2	26	**251.0**	**235.2**	115.1
BESPALOV	7625.7	26	83371.0	83316.1	22123.2
EIGSOLV	40.7	26	379.0	371.0	149.0

Table 4.4: *Comparison of methods for the indefinite eigenvalue problem with mesh* copcav18.

JDSYM					
Method	it_{in}	it_{out}	t_{tot}	t_{eig}	t_c
DIRPROJ	3.9	39	218.0	198.1	38.8
SAUG	5.1	36	**160.0**	**140.6**	43.7
AD	3.9	39	**160.0**	141.2	27.9
BESPALOV	339.5	63	12217.0	12155.5	2735.2
EIGSOLV	8.8	49	297.0	287.9	88.5
ARPACK					
DIRPROJ	30.3	26	553.0	532.4	128.8
SAUG	29.7	26	335.0	315.8	130.8
AD	29.2	26	**239.0**	**220.9**	95.8
BESPALOV	2929.2	26	39238.0	39174.5	10196.7
EIGSOLV	32.9	26	372.0	362.0	146.8

Table 4.5: *Comparison of methods for the indefinite eigenvalue problem with mesh* cop10k.

Tabs. 4.3 to 4.6 summarise the results. it_{in} is the average number of inner iterations per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem. t_c is the time spent for operations with the matrices $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$.

When comparing the total computation times t_{tot}, it is evident, that the SAUG and AD methods are fastest for all grids when using the JDSYM eigensolver. SAUG is usually a little (at most 10% in our experiments) faster than the AD method.

The DIRPROJ and EIGSOLV methods are both considerably slower. Compared to DIRPROJ, SAUG is very efficient, because it doesn't require any operations with the matrices $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$ in the inner iteration. On the other hand the average number of inner iterations it_{in} is higher for SAUG.

When using the ARPACK eigensolver, SAUG and AD are again the fastest methods. However, in contrast to JDSYM, here AD is considerably faster than SAUG.

JDSYM					
Method	it_{in}	it_{out}	t_{tot}	t_{eig}	t_c
DIRPROJ	4.0	40	798.0	709.5	144.5
SAUG	5.1	36	**526.0**	**437.2**	153.2
AD	4.0	39	579.0	495.2	112.0
BESPALOV	402.3	66	47014.0	46614.4	9755.0
EIGSOLV	9.6	48	952.0	915.9	337.1
ARPACK					
DIRPROJ	29.5	26	1973.0	1881.0	479.3
SAUG	28.7	26	1093.0	1002.8	476.6
AD	28.1	26	**760.0**	**676.6**	322.1
BESPALOV	3354.8	26	135063.0	134668.2	31984.4
EIGSOLV	32.7	26	1163.0	1126.2	538.2

Table 4.6: *Comparison of methods for the indefinite eigenvalue problem with mesh* cop20k.

The shift applied to matrix $\boldsymbol{A}$ when using the BESPALOV method worsens the condition of the linear systems. Very high inner iteration counts are the consequence. As a result, the BESPALOV method is so slow, that it is useless for even modestly sized meshes.

If the fastest methods for both eigensolvers are compared, it becomes clear, that JDSYM is faster than ARPACK for all meshes. The difference in timings can be mainly attributed to the fact that JDSYM allows for inaccurate solutions of the correction equation, whereas ARPACK requires an accurate solution of the shifted operator.

The EIGSOLV method is considerably slower than the respective fastest method for all meshes and eigensolvers. However, its key advantage over the other methods is its modest memory consumption. The EIGSOLV method does not rely on the matrices $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$, and therefore a lot of memory can be saved. To investigate the memory consumption of the various methods, we determined the memory footprint of the whole eigensolver application for two large meshes *cop20k* and *cop40k* as shown in Tab. 4.7.

mesh *cop20k*		
Preconditioner	*Footprint for SAUG*	*Footprint for EIGSOLV*
2LevJACssor	441.3 MB	183.5 MB
diag	407.2 MB	149.5 MB
mesh *cop40k*		
2LevJACssor	887.2 MB	370.5 MB
diag	795.4 MB	276.7 MB

Table 4.7: *Memory footprint of the eigensolver application*
The table shows the memory footprint of the eigensolver application, comparing the SAUG and the EIGSOLV methods when used with the JDSYM eigensolver and the 2-level or the diagonal preconditioner. The memory footprint was obtained using the UNIX `top` command. It incorporates all private and shared memory regions used by the application.

The data in Tab. 4.7 clearly show that the EIGSOLV method uses less than half of the

memory required by the other methods[14].

4.2.7 Conclusions

Out of the five presented methods to calculate the smallest positive eigenvalues of the indefinite eigenproblem, the AD and SAUG methods are the fastest.

The SAUG method, combined with the JDSYM eigensolver, is the fastest way to compute the desired eigensolutions. However, the stability of the SAUG method has not been proven, and it was derived using the Conjugate Gradients algorithm, which may not be applied on indefinite linear systems. To make the SAUG method work reliably together with the JDSYM eigensolver, additional projections had to be incorporated in the implementation of JDSYM, to keep the search space orthogonal to $\mathcal{R}(\boldsymbol{C}^T)$.

The AD method, combined with the JDSYM eigensolver, is usually a little (up to 10%) slower than the SAUG method. This method leads to a generalised positive-definite eigenvalue problem, which (provably) yields the desired eigensolutions. The AD method poses no special requirements on the eigensolver, and thus it can be used together with the standard JDSYM algorithm. However, more effort (both in terms of the implementation and the computational cost) is necessary, to construct the global finite element matrices. The AD method has the disadvantage, that the right-hand-side matrix can not be stored explicitly, since it is almost full. This causes difficulties when constructing certain types of preconditioners.

DIRPROJ is always considerably slower than the SAUG method and therefore there is no need to consider it. The BESPALOV method is ridiculously slow and thus can also be discarded.

However, if memory is an issue (memory will most certainly be an issue if large meshes are used) then EIGSOLV is the method of choice. EIGSOLV uses less than half of the memory required by the other methods, and thus allows to solve much larger problems. Although the EIGSOLV method is considerably slower than AD or SAUG for smaller and for medium-sized problems presented in this section, this is not the case for larger problems, as the results in Section 8.4 indicate.

[14]The DIRPROJ and the AD method require about the same amount of memory, since they all make use of $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$. The BESPALOV method requires less space, because it doesn't use $\boldsymbol{H}$ or $\boldsymbol{H}^{-1}$.

4.3 Choice of the shift

For the shift[15], two choices seem reasonable:

Positive shift, close to desired eigenvalues We choose the shift close to, and smaller than the smallest positive eigenvalue. With this choice, the convergence in the outer iteration is accelerated, since the desired eigenvalues are close to the shift and therefore the components in direction of the associated eigenvectors are amplified.

Shift equal to zero Here the shift is set to zero. The advantage is, that the number of multiplications with $\boldsymbol{M}$ is substantially reduced in this way. In ARPACK, no multiplications with $\boldsymbol{M}$ are needed at all in the inner iteration, since the shift is constant. Due to the dynamic shift in JDSYM, the multiplications with $\boldsymbol{M}$ are not totally eliminated in the inner iteration, but nevertheless their number is still substantially reduced.

The advantage is especially significant with the AD method, since expensive operations with $\boldsymbol{M}_{11} - \boldsymbol{C}_1 \boldsymbol{H}^{-1} \boldsymbol{C}_1^T$ can be saved.

Notice that if the shift is set to zero, the linear systems become singular (with the exception of the AD method and the BESPALOV method). The direct projection method operates in the positive definite subspace of the system matrix and can be applied despite the singularity. The simplified augmented method could be problematic, since no projections onto the positive definite subspace are carried out. Our practical experience showed no difficulties however. If the EIGSOLV method is used, the shift must not be set to zero, because the inner iteration operates also in the null space of the system matrix.

As described above, the DIRPROJ, SAUG, BESPALOV and AD methods can be used with a shift equal to zero. However, the zero shift may not be used for the preconditioner. If e.g. the 2-level hierarchical basis preconditioner is constructed with a shift equal to zero, a singular preconditioner will result. In this case the methods DIRPROJ, SAUG and BESPALOV fail too. As a consequence, the shift for the preconditioner must not be zero. We set it positive, smaller than the smallest positive eigenvalue of $\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}$. The AD method does not suffer from these problems, since the matrix $\boldsymbol{A}_{11}$ is positive definite.

Experimental results Since it is a priori not clear, which choice of the shift is better, we try to decide by running numerical experiments. We use three different meshes (cf. Appendix E) and calculate the five smallest positive eigenvalues and the corresponding eigenvectors to a accuracy of at least 10^{-4}. The calculations were performed using quadratic node- and vector elements and both eigensolvers (ARPACK and JDSYM). For all runs the 2LevJACssor preconditioner was used (cf. Section 8.3).

The results for the quadratic *node* elements are summarised in Tab. 4.8. The numbers show, that choosing a zero shift is usually a bit faster. However, it makes not much difference how the shift is chosen. Since matrix $\boldsymbol{A}$ is three times more dense than matrix $\boldsymbol{M}$, the savings made possible by the zero shift are not great.

The results for the quadratic *vector* elements are summarised in Tab. 4.9. We report results for the AD method, which particularly benefits from the choice $\sigma = 0$, and the SAUG method. The numbers show clear cut, that a zero shift is the better choice. Since matrix $\boldsymbol{M}$ is denser than matrix $\boldsymbol{A}$ when using vector elements, the savings are remarkable, especially for the AD method.

[15] The shift is the parameter σ in ARPACK, and the parameter τ in JDSYM

JDSYM					
Mesh	σ, τ	it_{in}	it_{out}	t_{tot}	t_{eig}
boxcav16x10x3	0.0	4.5	34	66.0	56.8
boxcav16x10x3	1.15	3.9	32	64.0	55.4
copcav18	0.0	4.8	35	108.0	94.8
copcav18	1.4	5.0	35	117.0	104.2
cop20k	0.0	3.8	34	310.0	263.7
cop20k	5.0	4.0	33	339.0	291.8
ARPACK					
boxcav16x10x3	0.0	33.2	26	182.0	173.4
boxcav16x10x3	1.15	33.8	26	187.0	178.3
copcav18	0.0	42.6	26	356.0	343.4
copcav18	1.4	52.8	26	445.0	431.5
cop20k	0.0	29.0	26	831.0	783.6
cop20k	5.0	29.4	26	859.0	811.5

Table 4.8: *Influence of the shift when solving the positive-definite eigenvalue problem*
Quadratic node elements are used for this experiment. it_{in} is the average number of inner iterations per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem.

JDSYM						
Mesh	*Method*	σ, τ	it_{in}	it_{out}	t_{tot}	t_{eig}
boxcav16x10x3	SAUG	0.0	4.9	33	82.0	71.9
boxcav16x10x3	SAUG	1.15	4.7	34	93.0	82.8
boxcav16x10x3	AD	0.0	3.7	35	87.0	76.7
boxcav16x10x3	AD	1.15	3.7	35	115.0	105.0
copcav18	SAUG	0.0	6.5	35	148.0	131.8
copcav18	SAUG	1.4	6.2	35	163.0	146.3
copcav18	AD	0.0	5.0	37	156.0	140.0
copcav18	AD	1.4	4.6	36	202.0	185.8
cop20k	SAUG	0.0	5.1	36	538.0	446.4
cop20k	SAUG	5.0	5.0	36	590.0	496.4
cop20k	AD	0.0	4.0	39	589.0	502.0
cop20k	AD	5.0	4.1	39	840.0	752.7
ARPACK						
boxcav16x10x3	SAUG	0.0	30.8	26	191.0	181.0
boxcav16x10x3	SAUG	1.15	30.0	26	220.0	209.1
boxcav16x10x3	AD	0.0	28.0	26	153.0	142.2
boxcav16x10x3	AD	1.15	28.2	26	283.0	272.8
copcav18	SAUG	0.0	37.3	26	354.0	337.5
copcav18	SAUG	1.4	37.0	26	401.0	383.4
copcav18	AD	0.0	35.2	26	260.0	243.9
copcav18	AD	1.4	39.5	26	612.0	595.7
cop20k	SAUG	0.0	28.7	26	1125.0	1031.3
cop20k	SAUG	5.0	29.6	26	1304.0	1208.0
cop20k	AD	0.0	28.1	26	789.0	702.3
cop20k	AD	5.0	28.8	26	1772.0	1684.4

Table 4.9: *Influence of the shift when solving the indefinite eigenvalue problem*
Quadratic vector elements are used for this experiment. it_{in} is the average number of inner iterations per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem.

5 Eigensolvers

To compute the desired eigensolutions for the matrix eigenvalue problems discussed in the previous chapter, we use two kinds of eigensolvers. The first method is an implementation of the *Jacobi-Davidson* algorithm, called JDSYM, which was developed in the course of this dissertation. The second method is the *Implicitly Restarted Lanczos* method (IRL). We use the publicly available software package ARPACK, which implements the IRL method.

In the following sections, both algorithms are discussed and compared to each other. In the case of JDSYM, the implementation and various variants are discussed in detail.

5.1 Jacobi-Davidson algorithm

5.1.1 Description of the algorithm

The Jacobi-Davidson algorithm was suggested in 1996 by Sleijpen and Van der Vorst [68]. Because the algorithm borrows ideas from both the *Davidson* and the *Jacobi's Orthogonal Component Correction* algorithm, these methods are presented first.

Davidson method The Davidson method is an iterative subspace method for calculating eigensolutions. It is particularly well-suited for diagonally dominant matrices $\boldsymbol{A}$.

Let $\operatorname{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_j\}$ be a search subspace, in which the matrix $\boldsymbol{A}$ has a Ritz value θ_j and a corresponding Ritz vector $\mathbf{u}_j$. The matrix $\boldsymbol{V} \in \mathbb{R}^{n \times j}$ stores the orthogonal basis vectors $\mathbf{v}_1$, $\ldots$, $\mathbf{v}_j$.

The various subspace methods for calculating eigensolutions differ mainly in the way they expand the search subspace. The idea behind the Davidson method is the following:

If the desired eigenvalue λ would be known beforehand, then an optimal expansion of the search subspace $\mathbf{z}$ satisfies

$$\boldsymbol{A}(\mathbf{u}_j + \mathbf{z}) = \lambda(\mathbf{u}_j + \mathbf{z})$$

and thus is calculated by

$$(\boldsymbol{A} - \lambda \boldsymbol{I})\mathbf{z} = \lambda \mathbf{u}_j - \boldsymbol{A}\mathbf{u}_j.$$

With this choice of $\mathbf{z}$, the residual would vanish and the method would converge to the exact eigenvector in the next iteration step. Since the wanted eigenvalue is not known, the approximation θ_j is used instead. Besides that, to keep the cost for solving the linear system of equations low, Davidson proposes to use the diagonal $\boldsymbol{D}_A$ as an approximation for $\boldsymbol{A}$. In the end, this leads to the following equation for the expansion of the search subspace

$$(\boldsymbol{D}_A - \theta_j \boldsymbol{I})\tilde{\mathbf{z}} = -\mathbf{r}_j \tag{5.1}$$

with $\mathbf{r}_j := \boldsymbol{A}\mathbf{u}_j - \theta_j \mathbf{u}_j$. The vector $\tilde{\mathbf{z}}$ is then orthogonalised against $\mathbf{v}_1, \ldots, \mathbf{v}_j$ and added to the search subspace as the new direction $\mathbf{v}_{j+1}$. This process is repeated, until $\|\mathbf{r}_j\|$ falls below a given bound ε, i.e. $(\theta_j, \mathbf{u}_j)$ is accepted as a converged eigenpair.

The Davidson method is often used in quantum chemistry applications. In these applications the matrices are large, symmetric and strongly diagonally dominant, and therefore $\boldsymbol{D}_A$ is quite a good approximation for $\boldsymbol{A}$. It is reported that for these applications the Davidson method is substantially faster than the Lanczos method [22]. Alg. 5.1 shows a straight-forward implementation of the Davidson method. Alg. 5.2 shows an orthogonalisation routine needed by Alg. 5.1.

function [λ, $\mathbf{u}$] = DAVIDSON($\boldsymbol{A}$, $\mathbf{v}_0$, ε)

$\boldsymbol{D}_A = \mathsf{diag}(\mathsf{diag}(\boldsymbol{A}))$;
$\mathbf{v}_0 = \mathbf{v}_0/\|\mathbf{v}_0\|$; $\mathbf{v}_{\boldsymbol{A}} = \boldsymbol{A}\mathbf{v}_0$;
$\boldsymbol{V} = \mathbf{v}_0$; $j = 1$; *initialise search subspace*
$\boldsymbol{G} = \mathbf{v}_0^T\mathbf{v}_{\boldsymbol{A}}$; *initialise projected matrix* $\boldsymbol{G}$
while 1
  $[\boldsymbol{W}, \boldsymbol{S}] = \mathsf{eig}(\boldsymbol{G})$;
  $[\theta, \mathrm{imax}] = \mathsf{max}(\mathsf{diag}(\boldsymbol{S}))$; *compute largest Ritz value* θ
  $\mathbf{u} = \boldsymbol{V}\boldsymbol{W}(:, \mathrm{imax})$; *and associated Ritz vector* $\mathbf{u}$
  $\mathbf{r} = \boldsymbol{V}_{\boldsymbol{A}}\boldsymbol{W}(:, \mathrm{imax}) - \theta\mathbf{u}$; *compute residual* $\mathbf{r}$ *(without accessing* $\boldsymbol{A}$*)*
  if $\|\mathbf{r}\| < \varepsilon$, break; end if *stopping criterion*
  $\mathbf{z} = -(\boldsymbol{D}_A - \theta\boldsymbol{I})^{-1}\mathbf{r}$ *compute correction* $\mathbf{z}$
  $\mathbf{z} = \mathsf{mgs}(\boldsymbol{V}, \mathbf{z})$; $\mathbf{z} = \mathbf{z}/\|\mathbf{z}\|$; *orthonormalise* $\mathbf{z}$ *against* $\boldsymbol{V}$ *using modified Gram-Schmidt*
  $\mathbf{v}_{\boldsymbol{A}} = \boldsymbol{A}\mathbf{z}$;
  $\boldsymbol{V} = [\boldsymbol{V}, \mathbf{z}]$; *add* $\mathbf{z}$ *to search subspace*
  $\boldsymbol{V}_{\boldsymbol{A}} = [\boldsymbol{V}_{\boldsymbol{A}}, \mathbf{v}_{\boldsymbol{A}}]$;
  $\boldsymbol{G} = [\boldsymbol{G}, \boldsymbol{V}^T\mathbf{v}_{\boldsymbol{A}}; \mathbf{v}_{\boldsymbol{A}}^T\boldsymbol{V}, \mathbf{v}_{\boldsymbol{A}}^T\mathbf{z}]$; *update* $\boldsymbol{G}$
  $j = j + 1$;
end while
$\lambda = \theta$;
end function

Algorithm 5.1: *Davidson method*
This routine is a straight-forward implementation of the Davidson method without restart and without the possibility to compute more than one eigenpair.
DAVIDSON computes the largest eigenvalue λ of the matrix $\boldsymbol{A}$ and the corresponding eigenvector $\mathbf{u}$. $\mathbf{v}_0$ is the initial subspace. ε is the tolerance for the stopping criterion.

```
function u = MGS(Q, u)
    m = size(Q, 2)
    for i = 1:m
        s = u^T Q(:, i)
        u = u − sQ(:, i)
    end for
end function
```

Algorithm 5.2: *Modified Gram-Schmidt orthogonalisation*
The routine MGS orthogonalises the vector $\mathbf{u}$ vs. the columns of $\boldsymbol{Q}$ using the modified Gram-Schmidt approach.

The method can be extended for matrices that are not diagonally dominant, by computing the expansion vector $\tilde{\mathbf{z}}$ using the equation

$$(\boldsymbol{K} - \theta_j \boldsymbol{I})\tilde{\mathbf{z}} = -\mathbf{r}_j, \tag{5.2}$$

where $\boldsymbol{K} - \theta_j \boldsymbol{I}$ is an approximation of $\boldsymbol{A} - \theta_j \boldsymbol{I}$ [53]. Morgan and Scott [53] noticed however, that this generalised Davidson method suffers from the weird property that the approximation $\boldsymbol{K}$ must not be too good, because otherwise the method could stagnate or even break down. This problem becomes obvious, if $\boldsymbol{K}$ is set equal to $\boldsymbol{A}$. If $\boldsymbol{K} = \boldsymbol{A}$, the search subspace $\boldsymbol{V}$ is not expanded at all, which leads to immediate stagnation of the method.

Jacobi's orthogonal component correction method (JOCC) In 1846 Jacobi proposed a method for the calculation of eigenvalues and eigenvectors of symmetric diagonal dominant matrices [44].

Let $\boldsymbol{A}$ be a symmetric diagonally dominant matrix, whose diagonal entry $a_{1,1}$ is the entry with the largest modulus. Then $\alpha = a_{1,1}$ is an approximation for the eigenvalue with the largest modulus and $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$ is an approximation for the corresponding eigenvector.

The eigenvalue problem to be solved is

$$\boldsymbol{A} \begin{bmatrix} 1 \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \alpha & \mathbf{b}^T \\ \mathbf{b} & \boldsymbol{F} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{z} \end{bmatrix} = \lambda \begin{bmatrix} 1 \\ \mathbf{z} \end{bmatrix}. \tag{5.3}$$

Let λ (close to α) and $\mathbf{u} = (1, \mathbf{z}^T)^T$ be the desired eigenvalue and eigenvector. The matrix eigenvalue problem (5.3) is equivalent to

$$\lambda = \alpha + \mathbf{b}^T \mathbf{z} \tag{5.4}$$

$$(\boldsymbol{F} - \lambda \boldsymbol{I})\mathbf{z} = -\mathbf{b}. \tag{5.5}$$

Jacobi suggested, to solve (5.5) iteratively, using the Jacobi iteration for linear systems (cf. Section 8.1.1). The Jacobi iteration is modified, such that λ is updated in each iteration step according to (5.4). $\mathbf{z}_0 = \mathbf{0}$ is used as the initial guess. Alg. 5.3 shows an implementation of the JOCC method.

Alg. 5.3 is not identical to the method originally suggested by Jacobi. In the original work, λ is updated only in every second step and using the diagonal system the increment $\Delta\mathbf{z}$ is calculated, instead of solving for $\mathbf{z}$ directly. With regard to the derivation of the Jacobi-Davidson method, the representation in Alg. 5.3 is better suited for our purposes.

```
function [λ, u] = JOCC(A)
    — extract parts of matrix —
    n = size(A, 1);
    F = A(2:n, 2:n);  d = diag(F);
    b = A(2:n, 1);
    α = A(1, 1);
    — initialisation —
    z = zeros(n − 1, 1);
    — iteration —
    while not converged
        λ = α + b^T z;                      update λ
        z = (d.*z − Fz − b)./(d − λ);       Jacobi iteration step
    end while
    u = [1; z];
end function
```

Algorithm 5.3: *Jacobi's orthogonal component correction*

Comparison of the Davidson method with the JOCC method

Search subspace The Davidson method chooses the Ritz pair $(\lambda, \mathbf{u})$ as the current eigenpair approximation. $\mathbf{u}$ is in the space spanned by all corrections computed so far. The JOCC method on the other hand calculates the current eigenvector approximation as a simple linear combination of the initial vector $\mathbf{u}_1$, the old eigenvector approximation $\mathbf{u}_k$ and a correction $[0; \mathbf{z}]$.

Orthogonalisation JOCC calculates corrections, that are orthogonal to $\mathbf{u}_1 = \mathbf{e}_1$. The orthogonalisation results implicitly from the method. In contrast, the corrections calculated by the Davidson method are explicitly orthogonalised to the whole search space $\boldsymbol{V}$ (therefore they are also orthogonal to $\mathbf{v}_1$).

Eigenvalue approximations The Davidson method uses a Ritz value as the eigenvalue approximation, whereas JOCC computes the eigenvalue approximation according to Equation (5.4) using an approximation for $\mathbf{z}$.

The following problems can occur when using the Davidson method:

1. Cancellation effects caused by the explicit orthogonalisation will render the correction ineffective, if it has large components in direction of the search space ran$(\boldsymbol{V})$.

2. Approximating the matrix $\boldsymbol{A}$ by its diagonal $\boldsymbol{D}$ makes sense for strongly diagonally dominant matrices. For other matrices, this approximation is not accurate enough. If the generalised Davidson method is used, another weakness of the Davidson approach manifests itself: If the approximation of the eigenvalue θ is very accurate, then the system matrix in the correction equation $\boldsymbol{K} - \theta\boldsymbol{I}$ becomes almost singular. This results in ineffective corrections, or if an iterative solver is used, in slow convergence.

3. If the current approximation is far away from the wanted eigenvalue, a poor correction $\mathbf{z}$ is computed. Therefore the rate of convergence degrades.

The basic Jacobi-Davidson algorithm The Jacobi-Davidson algorithm suggested in 1996 by Sleijpen and Van der Vorst [68] is quite similar to the generalised Davidson method, but without inheriting its weaknesses. The most important difference is the way the correction $\mathbf{z}$ is calculated. Here Jacobi's idea to keep the correction orthogonal to the eigenvector approximation, is adopted. As in JOCC, this is not made sure by explicit post-orthogonalisation, but by incorporating the orthogonality requirement into the correction equation. In that sense, the Jacobi-Davidson algorithm is a combination of the JOCC method and the Davidson method.

Instead of solving

$$(\boldsymbol{A} - \theta \boldsymbol{I})\mathbf{z} = -\mathbf{r}$$

approximately like in the generalised Davidson method (cf. Equation (5.2)), the correction equation for the Jacobi-Davidson method is formulated as

$$(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)(\boldsymbol{A} - \theta \boldsymbol{I})(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)\mathbf{z} = -\mathbf{r} \quad \text{and} \quad \mathbf{z} \perp \mathbf{u}, \tag{5.6}$$

whereupon the first equation may be solved approximately.

The correction $\mathbf{z}$ can be computed in two ways [68]:

1. $\mathbf{z}$ is calculated from

   $$(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)(\boldsymbol{K} - \theta \boldsymbol{I})(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)\mathbf{z} = -\mathbf{r} \quad \text{and} \quad \mathbf{z} \perp \mathbf{u} \tag{5.7}$$

   by a direct method (e.g. Gauss-elimination). $\boldsymbol{K}$ is an approximation of $\boldsymbol{A}$, with which linear systems are cheaper to solve. $\boldsymbol{K}$ must exist explicitly as a matrix.

   Since the correction $\mathbf{z}$ must satisfy $\mathbf{z} \perp \mathbf{u}$,

   $$(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)\mathbf{z} = \mathbf{z} \tag{5.8}$$

   must hold. Due to (5.8) the projection on the right in (5.7) can be omitted. The projection on the left is replaced by adding a vector collinear to $\mathbf{u}$. Thus we get

   $$(\boldsymbol{K} - \theta \boldsymbol{I})\mathbf{z} - \varepsilon \mathbf{u} = -\mathbf{r}. \tag{5.9}$$

   The correction $\mathbf{z}$ is calculated from

   $$\mathbf{z} = (\boldsymbol{K} - \theta \boldsymbol{I})^{-1}(\varepsilon \mathbf{u} - \mathbf{r}). \tag{5.10}$$

   The value of ε is given by the condition $\mathbf{z} \perp \mathbf{u}$:

   $$\varepsilon = \frac{\mathbf{u}^T(\boldsymbol{K} - \theta \boldsymbol{I})^{-1}\mathbf{r}}{\mathbf{u}^T(\boldsymbol{K} - \theta \boldsymbol{I})^{-1}\mathbf{u}}. \tag{5.11}$$

   The augmented correction equation with $\boldsymbol{A}$ approximated by $\boldsymbol{K}$

   $$\begin{bmatrix} \boldsymbol{K} - \theta \boldsymbol{I} & \mathbf{u} \\ \mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ -\varepsilon \end{bmatrix} = \begin{bmatrix} -\mathbf{r} \\ 0 \end{bmatrix} \tag{5.12}$$

   is equivalent to (5.9) together with the orthogonality constraint $\mathbf{u}^T\mathbf{z} = \mathbf{0}$.

2. (5.6) is solved iteratively by e.g. SYMMLQ, CGS, GMRES, etc. The iteration is stopped, as soon as the residual norm falls below a given bound.

   It has to be ensured, that the iterative method operates in the subspace $\mathcal{R}(\mathbf{u})^\perp$. Otherwise the linear system may become singular. This is done by choosing the initial subspace orthogonal to $\mathbf{u}$ and keeping all relevant vectors in the iterative method in $\mathcal{R}(\mathbf{u})^\perp$.

In this dissertation we deal solely with the second approach, because it offers important advantages over the first one:

- ▷ The quality of the approximation can be controlled easily by adjusting the stopping criterion. Thus, it is possible to dynamically adjust the approximation. This can substantially improve the convergence of the Jacobi-Davidson algorithm (cf. Section 5.1.2).
- ▷ The first method is in some sense a special case of the second method: The approximation $\boldsymbol{K} - \theta\boldsymbol{I}$ can be viewed as a preconditioner for the second method.

---

function $[\lambda, \mathbf{u}] = \mathrm{JD}(\boldsymbol{A}, \mathbf{v}_0, \varepsilon)$
  — *initialise search subspace and Rayleigh-Ritz procedure* —
  $\mathbf{v}_0 = \mathbf{v}_0/\|\mathbf{v}_0\|;\ \mathbf{v}_{\boldsymbol{A}} = \boldsymbol{A}\mathbf{v}_0;$
  $\boldsymbol{V} = \mathbf{v}_0;\ j = 1;$
  $\boldsymbol{G} = \mathbf{v}_0^T\mathbf{v}_{\boldsymbol{A}};$
  $\mathbf{u} = \boldsymbol{V};\ \theta = \boldsymbol{G};$
  $\mathbf{r} = \mathbf{v}_{\boldsymbol{A}} - \theta\mathbf{u};$
  while 1,
    — *expanding the search subspace* —
    Correction equation: Solve (approximately) for $\mathbf{z}$
      $(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)(\boldsymbol{A} - \theta\boldsymbol{I})(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T)\mathbf{z} = -\mathbf{r}$ and
      $\mathbf{z}^T\mathbf{u} = 0$
    $\mathbf{z} = \mathrm{mgs}(\boldsymbol{V}, \mathbf{z});\ \mathbf{z} = \mathbf{z}/\|\mathbf{z}\|;$ *orthonormalise* $\mathbf{z}$ *against* $\boldsymbol{V}$ *using modified Gram-Schmidt*
    $\boldsymbol{V} = [\boldsymbol{V}, \mathbf{z}];\ \mathbf{v}_{\boldsymbol{A}} = \boldsymbol{A}\mathbf{z};$
    — *compute Ritz pairs* —
    $\boldsymbol{G} = [\boldsymbol{G}, \boldsymbol{V}^T\mathbf{v}_{\boldsymbol{A}}; \mathbf{v}_{\boldsymbol{A}}^T\boldsymbol{V}, \mathbf{v}_{\boldsymbol{A}}^T\mathbf{z}];$ *update* $\boldsymbol{G}$
    $[\boldsymbol{W}, \boldsymbol{S}] = \mathrm{eig}(\boldsymbol{G});$ *solve projected eigenproblem*
    $[\theta, \mathrm{imax}] = \max(\mathrm{diag}(\boldsymbol{S}));$ *select largest Ritz value*
    $\mathbf{u} = \boldsymbol{V}\boldsymbol{W}(:, \mathrm{imax});$ *and associated Ritz vector*
    $\mathbf{r} = \boldsymbol{A}\mathbf{u} - \theta\mathbf{u};$ *compute residual*
    $j = j + 1;$
    if $\|\mathbf{r}\| < \varepsilon$, break; end if *stopping criterion*
  end while
  $\lambda = \theta;$
end function

Algorithm 5.4: *Basic Jacobi-Davidson algorithm*
This subroutine is a straight forward implementation of the Jacobi-Davidson algorithm without restarts and without the possibility to compute more than one eigenvalue.
JD computes the largest eigenvalue λ of the matrix $\boldsymbol{A}$ and the corresponding eigenvector $\mathbf{u}$. $\mathbf{v}_0$ is the initial subspace and ε is the tolerance for the stopping criterion.

Alg. 5.4 shows a Matlab implementation of a straight forward variant of the Jacobi-Davidson method. Since the largest eigenvalue (Ritz value) is extracted from the projected eigenproblem, the method converges to the largest eigenvalue of the matrix $\boldsymbol{A}$ and its associated eigenvector. Alg. 5.4 does not support the calculation of more than one eigensolution and also restarts (for

limiting the memory requirements) are not implemented. Furthermore, many other details that are critical for an efficient implementation are missing. The fully-fledged algorithm for the symmetric eigenvalue problem of the form $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x}$ is discussed in Section 5.1.2.

5.1.2 Jacobi-Davidson algorithm for the symmetric eigenvalue problem $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x}$

Several papers on the Jacobi-Davidson algorithm for the generalised eigenvalue problem $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}$ with *arbitrary* matrices exist [67], [26]. We are however interested in an efficient implementation of the Jacobi-Davidson algorithm for the symmetric eigenvalue problem.

The work described in this section has been published in [5]. A very similar approach was later published in [8, Section 5.6]. The differences between the two approaches are described in Section 5.1.6.

This section deals with the formulation and implementation of the Jacobi-Davidson algorithm for the generalised eigenvalue problem

$$\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x} \tag{5.13}$$

with real, symmetric matrix $\boldsymbol{A}$ and real, symmetric and positive definite matrix $\boldsymbol{M}$.

The generalised eigenvalue problem (5.13) has the same eigensolutions as the standard eigenvalue problem

$$\boldsymbol{M}^{-1}\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}. \tag{5.14}$$

Properties of the eigensolutions Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be symmetric and $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ be symmetric positive definite.

Then two matrices $\boldsymbol{X} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{\Lambda} \in \mathbb{R}^{n\times n}$ exist, such that

$$\boldsymbol{X}^T\boldsymbol{A}\boldsymbol{X} = \boldsymbol{\Lambda} \quad \text{and} \quad \boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X} = \boldsymbol{I} \tag{5.15}$$

holds. $\boldsymbol{\Lambda}$ is a diagonal matrix holding the eigenvalues λ_k of $\boldsymbol{M}^{-1}\boldsymbol{A}$ on its diagonal. The columns $\mathbf{x}_k$ of $\boldsymbol{X}$ are the associated eigenvectors of $\boldsymbol{M}^{-1}\boldsymbol{A}$.

Description of the eigensolutions to be calculated The task of the algorithm is to compute a *partial eigenvalue decomposition* of dimension $k_{\max}$ for the eigenvalue problem (5.13). So, we are looking for a $\boldsymbol{Q} \in \mathbb{R}^{n\times k_{\max}}$ and a diagonal matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{k_{\max}\times k_{\max}}$, such that

$$\boldsymbol{A}\boldsymbol{Q} = \boldsymbol{M}\boldsymbol{Q}\boldsymbol{\Lambda} \quad \text{with} \quad \boldsymbol{Q}^T\boldsymbol{M}\boldsymbol{Q} = \boldsymbol{I}. \tag{5.16}$$

The algorithm shall compute the $k_{\max}$ eigenvalues (together with associated eigenvectors), that are closest to a given target value τ.

The converged eigensolutions $(\mathbf{q}_k, \lambda_k)$ are required to satisfy

$$\|\mathbf{r}_k\|_2 := \|\boldsymbol{A}\mathbf{q}_k - \lambda_k\boldsymbol{M}\mathbf{q}_k\|_2 < \varepsilon, \tag{5.17}$$

i.e. the 2-norm of the associated residual $\mathbf{r}_k$ is smaller than a given bound ε.

Organisation of the algorithm The following sections describe the implementation details of the JDSYM algorithm, which computes eigensolutions of the symmetric eigenvalue problem (5.13). Many ideas were adopted from the JDQR and JDQZ algorithms described in [26].

Rayleigh-Ritz step As described in Section 5.1.1 for the basic Jacobi-Davidson algorithm, in each iteration step an approximation $\mathbf{u}$ for the eigenvector is computed from the search subspace $\mathcal{R}(\boldsymbol{V})$. The Galerkin condition requires orthogonality to a test subspace. According to the Rayleigh-Ritz approach we choose the test subspace identical to the search subspace $\mathcal{R}(\boldsymbol{V})$ and get

$$\mathbf{r} = \boldsymbol{A}\mathbf{u} - \theta\boldsymbol{M}\mathbf{u} \perp \mathcal{R}(\boldsymbol{V}), \quad \mathbf{u} \in \mathcal{R}(\boldsymbol{V}). \tag{5.18}$$

Since the eigenvectors of (5.13) are $\boldsymbol{M}$-orthogonal, it is convenient to have the columns of $\boldsymbol{V}$ $\boldsymbol{M}$-orthogonal as well,

$$\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{V} = \boldsymbol{I}.$$

In this way, the *projected eigenvalue problem* derived from Equation (5.18)

$$\boldsymbol{G}\mathbf{w} := \boldsymbol{V}^T\boldsymbol{A}\boldsymbol{V}\mathbf{w} = s\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{V}\mathbf{w} = s\mathbf{w} \tag{5.19}$$

becomes a standard eigenvalue problem. This small dense eigenvalue problem is solved using the symmetric QR-method (LAPACK Routine `dsyev`), yielding

$$\boldsymbol{G}\boldsymbol{W} = \boldsymbol{W}\boldsymbol{S}, \tag{5.20}$$

where $\boldsymbol{W} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_j]$ is the matrix containing the eigenvectors $\mathbf{w}_i$ of $\boldsymbol{G}$, and $\boldsymbol{S} = \mathrm{diag}(s_1, s_2, \ldots, s_j)$ is the matrix containing the eigenvalues s_i of $\boldsymbol{G}$.

Selection of Ritz values By selecting the appropriate Ritz value as the current eigenvalue approximation, the Jacobi-Davidson iteration can be controlled to converge to a specific eigensolution. Since we are interested in eigenvalues close to the target value τ, we choose the Ritz value closest to τ as the approximation for the eigenvalue to be calculated.

It turns out to be of advantage to sort the eigensolutions stored in $\boldsymbol{W}$ and $\boldsymbol{S}$, such that

$$|s_1 - \tau| \leq |s_2 - \tau| \leq \ldots \leq |s_j - \tau|. \tag{5.21}$$

With this ordering, the implementation of the restart and the calculation of more than one eigensolution are simplified (cf. page 70).

The Ritz pair $(\mathbf{u}, \theta)$ is then calculated by

$$\mathbf{u} = \boldsymbol{V}\mathbf{w}_1 \quad \text{and} \quad \theta = s_1.$$

The associated residual then is

$$\mathbf{r} = \boldsymbol{A}\mathbf{u} - \theta\boldsymbol{M}\mathbf{u}.$$

Calculating the residual has two purposes: On the one hand, the residual is used in the correction equation, and on the other hand, it is used in the convergence criterion for the eigensolution approximations $(\mathbf{u}, \theta)$.

Convergence criterion for eigensolutions The Ritz pair $(\mathbf{u}, \theta)$ is accepted as a converged eigensolution, if the residual norm falls below a given bound,

$$\|\mathbf{r}\|_2 = \|\boldsymbol{A}\mathbf{u} - \theta\boldsymbol{M}\mathbf{u}\|_2 < \varepsilon. \tag{5.22}$$

If this condition is satisfied, a series of operations have to be performed, before the calculation of the next eigenpair can be started:

- ▷ Vector $\mathbf{u}$ is appended to the matrix containing the converged eigenvectors: $\boldsymbol{Q} := [\boldsymbol{Q}, \mathbf{u}]$. $\boldsymbol{Q}$ is empty initially.
- ▷ In order to start the search for the next eigenvector, the search subspace must be adjusted, such that it is $\boldsymbol{M}$-orthogonal to the converged eigenvector $\mathbf{u}$ (and to the other converged eigenvectors stored in $\boldsymbol{Q}$). Therefore we span the new search subspace using the Ritz vectors $\boldsymbol{VW}(:,2) \ldots \boldsymbol{VW}(:,j)$, and thus

$$\boldsymbol{V}_{\text{new}} := \boldsymbol{VW}(:,[2,\ldots,j]).$$

- ▷ Due to the change of $\boldsymbol{V}$, also $\boldsymbol{G}$ must be adjusted,

$$\boldsymbol{G}_{\text{new}} = \boldsymbol{V}_{\text{new}}^T \boldsymbol{A} \boldsymbol{V}_{\text{new}} = \boldsymbol{W}(:,2{:}j)^T \boldsymbol{G}_{\text{old}} \boldsymbol{W}(:,2{:}j) = \boldsymbol{S}(2{:}j,2{:}j).$$

- ▷ For this reason also, the eigenvectors $\boldsymbol{W}$ of $\boldsymbol{G}$ change,

$$\boldsymbol{W}_{\text{new}} = \boldsymbol{I}.$$

The correction equation For the generalised eigenvalue problem (5.13), the correction equation (5.6) becomes [5][8, Section 5.6]

$$\begin{gathered}(\boldsymbol{I} - \boldsymbol{M}\mathbf{u}\mathbf{u}^T)(\boldsymbol{A} - \theta\boldsymbol{M})(\boldsymbol{I} - \mathbf{u}\mathbf{u}^T\boldsymbol{M})\mathbf{z} = -(\boldsymbol{I} - \boldsymbol{M}\mathbf{u}\mathbf{u}^T)\mathbf{r} = -\mathbf{r} \\ \text{with} \qquad (\boldsymbol{I} - \mathbf{u}\mathbf{u}^T\boldsymbol{M})\mathbf{z} = \mathbf{z}.\end{gathered} \tag{5.23}$$

In view of the solution of the correction equation using iterative methods, the formulation in (5.23) proves itself to be advantageous, since the matrix is symmetric. The matrix in (5.23) maps $\mathcal{R}(\mathbf{u})^{\perp_{\boldsymbol{M}}}$ onto $\mathcal{R}(\mathbf{u})^{\perp}$.

The projections on the right hand sides in (5.23) drop out, since $\mathbf{r} \perp \mathbf{u}$ as shown in (5.18).

If more than one eigenpair is to be calculated, the correction $\mathbf{z}$ must be orthogonal not only to $\mathbf{u}$, but also to all already converged eigenvectors $\mathbf{q}_1 \ldots \mathbf{q}_k$. Therefore the following extended correction equation is used:

$$\begin{gathered}(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^T)(\boldsymbol{A} - \theta\boldsymbol{M})(\boldsymbol{I} - \hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^T)\mathbf{z} = -(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^T)\mathbf{r} \\ \text{with} \qquad (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^T)\mathbf{z} = \mathbf{z},\end{gathered} \tag{5.24}$$

with $\hat{\boldsymbol{Q}} := [\mathbf{q}_1, \ldots, \mathbf{q}_k, \mathbf{u}] = [\boldsymbol{Q}, \mathbf{u}]$ and $\hat{\boldsymbol{Q}}_{\boldsymbol{M}} := \boldsymbol{M}\hat{\boldsymbol{Q}}$. In contrast to (5.23), the projector on the right hand side cannot be removed in (5.24), since according to inequality (5.40), $\mathbf{r}$ is only approximately orthogonal to $\hat{\boldsymbol{Q}}$.

Instead of using (5.24), the correction $\mathbf{z}$ could also be computed using (5.23). But then $\mathbf{z}$ has to be orthogonalised against $\boldsymbol{Q}$ afterwards (*implicit deflation*). Experimental results in [26] show, that the *explicit deflation*[16] in (5.24) improves the condition of the linear system, and therefore iterative solvers converge in less iteration steps. Our experimental results indicate, that the reduction of iteration steps outweighs the increased cost due to the projections in almost in all cases.

[16] In explicit deflation, the computation continues with the deflated matrix $(\boldsymbol{I} - \boldsymbol{Q}_{\boldsymbol{M}}\boldsymbol{Q}^T)(\boldsymbol{A} - \theta\boldsymbol{M})(\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}_{\boldsymbol{M}}^T)$ after convergence of eigenvectors. In implicit deflation, the computation continues with original matrix $(\boldsymbol{A} - \theta\boldsymbol{M})$. The projection occurs just before adding the new direction to the search subspace.

We solve (5.24) approximately using iterative methods (more specifically Krylov subspace methods) as already mentioned in Section 5.1.1. The methods we implemented are SYMMLQ, MINRES, QMRS and CGS (cf. Chapter 6).

These methods cannot be applied in a straight-forward manner, since the system matrix in (5.24) doesn't map $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$ onto itself. In the following section it is shown, how this issue can be resolved elegantly when using a preconditioner.

Preconditioning the correction equation To improve the convergence speed of the iterative solver, we choose a preconditioner of the form

$$(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\boldsymbol{K}(\boldsymbol{I} - \hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}) \tag{5.25}$$

for (5.24). $\boldsymbol{K}$ is a symmetric matrix, which approximates $\boldsymbol{A} - \tau\boldsymbol{M}$ and is cheap to "invert".

Thus, in each preconditioning step an equation of the form

$$(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\boldsymbol{K}(\boldsymbol{I} - \hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\mathbf{c} = \mathbf{b} \quad \text{and} \quad \hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\mathbf{c} = \mathbf{0} \tag{5.26}$$

is solved. The right hand side $\mathbf{b}$ is orthogonal to $\hat{\boldsymbol{Q}}$, since $\mathbf{b}$ is constructed by multiplying with $\boldsymbol{A}$ and $\boldsymbol{M}$.

Since $\mathbf{c}$ is required to satisfy $\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\mathbf{c} = \mathbf{0}$, Equation (5.26) is simplified to

$$(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\boldsymbol{K}\mathbf{c} = \mathbf{b}. \tag{5.27}$$

The projection $(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})$ adds components in direction of the columns of $\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$. We therefore write (5.27) as follows:

$$\boldsymbol{K}\mathbf{c} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\boldsymbol{\epsilon} = \mathbf{b} \tag{5.28}$$

or equivalently

$$\mathbf{c} = \boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}\boldsymbol{\epsilon} + \boldsymbol{K}^{-1}\mathbf{b}. \tag{5.29}$$

From the orthogonality requirement for $\mathbf{c}$

$$\mathbf{0} = \hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\mathbf{c} = \hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}\boldsymbol{\epsilon} + \hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\boldsymbol{K}^{-1}\mathbf{b}$$

with $\hat{\boldsymbol{Q}}_{\boldsymbol{K}} := \boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$ and $\hat{\boldsymbol{F}} := \hat{\boldsymbol{Q}}_{\boldsymbol{K}}^{T}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$ follows

$$\boldsymbol{\epsilon} = -\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{K}}^{T}\mathbf{b}.$$

By inserting this into (5.29) we get the solution of (5.26)

$$\begin{aligned}\mathbf{c} &= \boldsymbol{K}^{-1}(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{K}}^{T})\mathbf{b} \\ &= (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}\mathbf{b}.\end{aligned} \tag{5.30}$$

The preconditioner (5.30) maps $\mathcal{R}(\hat{\boldsymbol{Q}})^{\perp}$ back onto $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$. Thus the preconditioner serves two purposes: Firstly it improves the condition of the correction equation, and secondly it causes the preconditioned vector to be in the correct subspace.

The following table shows how a Krylov subspace method is used, to solve the correction Equation (5.24) to a given accuracy.

$$\begin{array}{ll} \text{system matrix:} & (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})(\boldsymbol{A} - \theta\boldsymbol{M}) \\ \text{preconditioner:} & (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1} \\ \text{right hand side:} & -(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\mathbf{r} \\ \text{initial vector:} & \mathbf{0} \end{array}$$

The projection to the right of $\boldsymbol{A} - \theta\boldsymbol{M}$ in (5.24) drops out, since the preconditioner maps the iteration vector and the search direction onto $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$. The initial vector $\mathbf{0}$ satisfies the constraint trivially.

Both the system matrix and the preconditioner are symmetric. However, because of the shift θ or τ respectively, they can become indefinite. For this reason, the QMRS iterative solver [30][29] is particularly well suited (cf. Chapter 6).

Even if no preconditioner is to be used, the projection onto $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$ must still be performed in the preconditioning step:

$$\begin{array}{ll} \text{system matrix:} & (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})(\boldsymbol{A} - \theta\boldsymbol{M}) \\ \text{preconditioner:} & (\boldsymbol{I} - \hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}) \\ \text{right hand side:} & -(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\mathbf{r} \\ \text{initial vector:} & \mathbf{0} \end{array}$$

An alternative formulation for the correction equation is obtained, by left-multiplying Equation (5.24) by the matrix $(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}$. Together with the identity [25]

$$\begin{aligned} &(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}(\hat{\boldsymbol{Q}}_{\boldsymbol{K}}^{T}\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T}) \\ &= \boldsymbol{K}^{-1} - \boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T} \\ &\quad - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\boldsymbol{K}^{-1} + \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T}\boldsymbol{K}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T} \\ &= \boldsymbol{K}^{-1} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{Q}}^{T} \\ &\quad - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{K}}^{T} + \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{Q}}^{T} \\ &= (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1} \end{aligned}$$

this yields the preconditioned correction equation

$$\begin{aligned} (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}(\boldsymbol{A} - \theta\boldsymbol{M})\mathbf{z} &= -(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}\mathbf{r} \\ \text{with} \qquad (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\mathbf{z} &= \mathbf{z}. \end{aligned} \tag{5.31}$$

Here the preconditioner is incorporated into the system matrix. With this formulation of the correction equation the system matrix is not symmetric. Only Krylov subspace methods for general non-symmetric matrices can be used, as e.g. CGS or BiCG-Stab.

The following table shows how a Krylov subspace method is used, to solve the correction equation (5.31).

$$\begin{array}{ll} \text{system matrix:} & (\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}(\boldsymbol{A} - \theta\boldsymbol{M}) \\ \text{preconditioner:} & \boldsymbol{I} \\ \text{right hand side:} & -(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}\mathbf{r} \\ \text{initial vector:} & \mathbf{0} \end{array}$$

Since the preconditioner is incorporated into the system matrix, the corresponding software routine is called as if no preconditioner would be used.

If no preconditioner is to be used, the iteration vectors must still be kept in the correct subspace $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$. We replace $(\boldsymbol{I}-\hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}$ by $(\boldsymbol{I}-\hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})$ and obtain a simplified unpreconditioned form of the correction equation.

$$\begin{aligned}
&\text{system matrix:} && (\boldsymbol{I}-\hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})(\boldsymbol{A}-\theta\boldsymbol{M})\\
&\text{preconditioner:} && \boldsymbol{I}\\
&\text{right hand side:} && -(\boldsymbol{I}-\hat{\boldsymbol{Q}}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\mathbf{r}\\
&\text{initial vector:} && \mathbf{0}
\end{aligned}$$

Choice of the preconditioner $\boldsymbol{K}$ The task of the preconditioner is to improve the convergence rate of the Krylov subspace method, which is employed to approximately solve the correction equation (5.24). $\boldsymbol{K}$ is a symmetric matrix, that approximates $\boldsymbol{A}-\theta\boldsymbol{M}$ and is cheap to "invert".

Since θ is updated in every step of the Jacobi-Davidson iteration, the preconditioner $\boldsymbol{K}$ and the preconditioned eigenvectors $\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$ should be updated too. However, for almost all preconditioners these operations are very expensive. Thus, we construct the preconditioner only once in the beginning, namely in such a way that it approximates $\boldsymbol{A}-\tau\boldsymbol{M}$. In [26], Fokkema et al. show that the convergence rate is almost as high as if $\boldsymbol{K}$ and $\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$ were updated in each iteration step.

The various preconditioners investigated in this dissertation are discussed in Chapter 8.

Stopping criterion and selection of the initial guess for solving the correction equation The stopping criterion and the initial guess are both crucial ingredients of a Krylov subspace method. They are discussed in this section.

The more accurate the eigenvalue approximation is, the better the expansion of the search space potentially is. In the extreme case, where $\theta=\lambda$, the subspace expansion is optimal, i.e. the desired eigenvector will lie in the new search subspace. Whereas if the eigenvalue approximation is very imprecise, the calculated correction will generally be ineffective and will not lead to a substantial reduction of the residual, as a general rule. In this case it makes no sense to solve the correction equation accurately.

For this reason, we use a variable stopping criterion of the form

$$\|\tilde{\mathbf{r}}_i\|_2 < \gamma^{-i_{\text{step}}}\|\tilde{\mathbf{r}_0}\|_2,$$

as suggested by Fokkema et al. in [26]. Here $\tilde{\mathbf{r}}_0$ is the initial residual and $\tilde{\mathbf{r}}_i$ is the residual associated with the current guess of the inner iteration. i_{step} is the number of JD iterations, which were already executed for calculating the next eigenpair. In other words, i_{step} is a counter that is incremented in each JD iteration, and reset to zero every time a new eigenpair is found. Thus, with the progression of the JD-iterations the correction equation is solved more and more accurately, until the eigenpair has converged.

The initial guess for the iterative method must lie in $\mathcal{R}(\hat{\boldsymbol{Q}}_{\boldsymbol{M}})^{\perp}$. Since no cheaply obtainable estimate for the solution $\mathbf{z}$ is available, we use the zero vector as the initial guess for the inner iteration.

Choosing the shift in the correction equation As discussed in the previous section, the calculated correction is ineffective if the shift in the correction equation is far away from the desired eigenvalue. With this insight, the choice of the shift can improved: If θ, the Ritz value closest to τ, is a bad approximation for the desired eigenvalue, the target value τ itself, being a better approximation, should be used as the shift instead of θ.

This is implemented as suggested by Fokkema et al. in [26]: If the residual norm from Equation (5.22) is larger than a given bound ε_{tr}, we choose τ as the shift, otherwise the Ritz value θ is used.

This strategy has two advantages: Firstly, the expansion of the search space is improved (especially in the first few iteration steps), and secondly, convergence to the eigenvalues closest to τ is enforced.

Restart In each Jacobi-Davidson iteration a new expansion of the search subspace is computed from the correction equation. Thus, in each JD iteration the dimension of the search subspace increases by one.

There are two reasons to restrict the dimension of the search subspace: Firstly this limits the memory consumption of the algorithm and secondly, it also limits the computational cost of a Jacobi-Davidson iteration step.

Thus, we define the maximal dimension of the search subspace $j_{\max}$. As soon as the maximal dimension is reached, a so-called *restart* is undertaken, and thereby the search subspace dimension is reduced to $j_{\min}$.

This gives rise to the question, which $j_{\min}$ search directions should be kept. Since we are interested in eigenvalues closest to τ, we choose the $j_{\min}$ Ritz vectors with associated Ritz values closest to τ, as the basis for the new search space. Due to the ordering (5.21), the Ritz values $s_1, \ldots, s_{j_{\min}}$ are closest to τ.

The following steps are undertaken to perform the restart:
Set j the to new search subspace dimension,

$$j \leftarrow j_{\min}.$$

Replace matrix $\boldsymbol{V}$ by the first j Ritz vectors,

$$\boldsymbol{V}_{\text{new}} := \boldsymbol{V}\boldsymbol{W}(1{:}j, :).$$

Down-size matrix $\boldsymbol{S}$,

$$\boldsymbol{S}_{\text{new}} := \boldsymbol{S}(1{:}j, 1{:}j).$$

Due to the change in $\boldsymbol{V}$, $\boldsymbol{G}$ must be updated also:

$$\boldsymbol{V}_{\text{new}}^T \boldsymbol{A} \boldsymbol{V}_{\text{new}} = \boldsymbol{G}_{\text{new}} := \boldsymbol{S}.$$

Therewith the eigenvectors $\boldsymbol{W}$ of $\boldsymbol{G}$ need to be updated too:

$$\boldsymbol{W}_{\text{new}} := \boldsymbol{I}.$$

The actions for implementing the restart are almost the same as the actions performed when a new eigenpair has converged (cf. page 65).

Orthogonalisations in the Jacobi-Davidson algorithm To ensure the stability and efficiency of the Jacobi-Davidson method, it is essential to keep the columns of $\boldsymbol{Q}$ and $\boldsymbol{V}$ $\boldsymbol{M}$-orthogonal. Thus, it must be guaranteed, that

$$\boldsymbol{Q}^T\boldsymbol{M}\boldsymbol{Q} = \boldsymbol{I} \tag{5.32}$$
$$\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{V} = \boldsymbol{I} \tag{5.33}$$
$$\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{Q} = \boldsymbol{0}. \tag{5.34}$$

Additional orthogonalisations are necessary in the iterative method for solving the correction equation.

For the inner iteration we use *classical Gram-Schmidt orthogonalisations*, i.e. operations of the form

$$\mathbf{y} = (\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}_{\boldsymbol{M}}^T)\mathbf{x} \quad \text{and} \quad \mathbf{y} = (\boldsymbol{I} - \boldsymbol{Q}_{\boldsymbol{M}}\boldsymbol{Q}^T)\mathbf{x}.$$

These type of orthogonalisation is the cheapest, but also the most inaccurate. In the inner iteration they are used for both the system matrix and the preconditioner (cf. Alg. 5.9).

Afterwards, the correction $\mathbf{z}$ computed in the inner iteration is orthogonalised more accurately using the *modified Gram-Schmidt method*. This orthogonalisation can be implemented efficiently, since we have the matrix $\boldsymbol{Q}_{\boldsymbol{M}} = \boldsymbol{M}\boldsymbol{Q}$ available (cf. Alg. 5.5).

---

function $\mathbf{u}$ = MGSM($\boldsymbol{Q}$, $\boldsymbol{Q}_{\boldsymbol{M}}$, $\mathbf{u}$)
  $m = \mathsf{size}(\boldsymbol{Q}, 2)$
  for $i = 1{:}m$
    $s = \mathbf{u}^T\boldsymbol{Q}_{\boldsymbol{M}}(:, i)$
    $\mathbf{u} = \mathbf{u} - s\boldsymbol{Q}(:, i)$
  end for
end function

Algorithm 5.5: *Modified $\boldsymbol{M}$-orthogonal Gram-Schmidt orthogonalisation*
The routine MGSM $\boldsymbol{M}$-orthogonalises the vector $\mathbf{u}$ vs. the columns of $\boldsymbol{Q}$ using the modified Gram-Schmidt approach. $\boldsymbol{Q}_{\boldsymbol{M}}$ is equal to $\boldsymbol{M}\boldsymbol{Q}$.

The vector $\mathbf{z}$ is the extension of the search space and therefore it must be $\boldsymbol{M}$-orthogonalised against all existing search directions, i.e. against the columns of $\boldsymbol{V}$. Since we do not explicitly store $\boldsymbol{M}\,\boldsymbol{V}$, modified Gram-Schmidt orthogonalisation would be too expensive, because many multiplications with $\boldsymbol{M}$ would be required. Therefore we use the *iterative classical Gram-Schmidt* method (cf. Alg. 5.6), which is cheaper in this case and also more accurate than the modified Gram-Schmidt method [16].

The JDSYM algorithm In this section, Alg. 5.7, our implementation of the Jacobi-Davidson algorithm, optimised for the symmetric, generalised eigenvalue problem (5.13) is presented in a Matlab-like notation. Alg. 5.8 ... 5.10 are auxiliary routines called by Alg. 5.7.

Note, that in Alg. 5.7, the matrix $\hat{\boldsymbol{Q}}$ corresponds to matrix $\boldsymbol{Q}$, expanded by an additional vector. Accordingly, the matrices $\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$, $\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$ and $\hat{\boldsymbol{F}}$ are related to $\boldsymbol{Q}_{\boldsymbol{M}}$, $\boldsymbol{Q}_{\boldsymbol{K}}$ and $\boldsymbol{F}$. This notation was chosen to keep the Matlab-like code short and simple. In an optimised version of the code, these matrices would share the same memory space.

function $[\mathbf{u}, r_1]$ = ICGSM($\boldsymbol{Q}$, $\boldsymbol{M}$, $\mathbf{u}$)
    $\alpha = 0.5;\ it_{\max} = 3;\ it = 1$
    $\mathbf{u}_{\boldsymbol{M}} = \boldsymbol{M}\mathbf{u}$
    $r_0 = \sqrt{\mathbf{u}^T\mathbf{u}_{\boldsymbol{M}}}$
    while 1
        $\mathbf{u} = \mathbf{u} - \boldsymbol{Q}(\boldsymbol{Q}^T\mathbf{u}_{\boldsymbol{M}})$
        $\mathbf{u}_{\boldsymbol{M}} = \boldsymbol{M}\mathbf{u}$
        $r_1 = \sqrt{\mathbf{u}^T\mathbf{u}_{\boldsymbol{M}}}$
        if $r_1 > \alpha\, r_0$ or $it \geq it_{\max}$ then break end if
        $it = it + 1;\ r_0 = r_1$
    end while
    if $r_1 \leq \alpha\, r_0$ then
        warning('loss of orthogonality')
    end if
end function

Algorithm 5.6: *Iterative Classical $\boldsymbol{M}$-orthogonal Gram-Schmidt orthogonalisation*
The routine ICGSM $\boldsymbol{M}$-orthogonalises the vector $\mathbf{u}$ vs. the columns of $\boldsymbol{Q}$ using the Iterative Classical Gram-Schmidt method.

function [$\boldsymbol{\lambda}$, $\boldsymbol{Q}$] = JDSYM($\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{K}$, $\mathbf{v}_0$, τ, ε, $k_{\max}$, $j_{\min}$, $j_{\max}$, ε_{tr}, $\text{it}_{\max}$, γ)
Initialisation according to Alg. 5.8
while $it < it_{\max}$,
— *Projected eigenproblem* —
$[\boldsymbol{W}, \boldsymbol{S}] = \text{eig}(\boldsymbol{G})$
$[\boldsymbol{W}, \boldsymbol{S}] = \text{sorteig}(\boldsymbol{W}, \boldsymbol{S}, \tau)$
while 1
— *Ritz approximation* —
$\mathbf{u} = \boldsymbol{V}\boldsymbol{W}(:,1);\ \theta = \boldsymbol{S}(1,1);\ \mathbf{u}_{\boldsymbol{M}} = \boldsymbol{M}\mathbf{u};\ \mathbf{u}_{\boldsymbol{K}} = \boldsymbol{K}^{-1}\mathbf{u}_{\boldsymbol{M}}$
$\mathbf{r} = \boldsymbol{A}\mathbf{u} - \theta\mathbf{u}_{\boldsymbol{M}}$
$\hat{\boldsymbol{Q}} = [\boldsymbol{Q}, \mathbf{u}];\ \hat{\boldsymbol{Q}}_{\boldsymbol{K}} = [\boldsymbol{Q}_{\boldsymbol{K}}, \mathbf{u}_{\boldsymbol{K}}];\ \hat{\boldsymbol{Q}}_{\boldsymbol{M}} = [\boldsymbol{Q}_{\boldsymbol{M}}, \mathbf{u}_{\boldsymbol{M}}]$
$\hat{\boldsymbol{F}} = [\boldsymbol{F}, \boldsymbol{Q}_{\boldsymbol{M}}^T\mathbf{u}_{\boldsymbol{K}}; \mathbf{u}_{\boldsymbol{K}}^T\boldsymbol{Q}_{\boldsymbol{M}}, \mathbf{u}_{\boldsymbol{M}}^T\mathbf{u}_{\boldsymbol{K}}]$
if not ($\|\mathbf{r}\|_2 < \varepsilon$ and ($j > 1$ or $k = k_{\max} - 1$)) then break end if
— *Convergence* —
$\boldsymbol{\lambda} = [\boldsymbol{\lambda}, \theta]$
$\boldsymbol{Q} = \hat{\boldsymbol{Q}};\ \boldsymbol{Q}_{\boldsymbol{M}} = \hat{\boldsymbol{Q}}_{\boldsymbol{M}};\ \boldsymbol{Q}_{\boldsymbol{K}} = \hat{\boldsymbol{Q}}_{\boldsymbol{K}};\ \boldsymbol{F} = \hat{\boldsymbol{F}}$
$\boldsymbol{V} = \boldsymbol{V}\boldsymbol{W}(:, 2{:}j);\ \boldsymbol{S} = \boldsymbol{S}(2{:}j, 2{:}j);\ \boldsymbol{G} = \boldsymbol{S}; \boldsymbol{W} = \boldsymbol{I}_j$
$j = j - 1;\ k = k + 1;\ i_{\text{step}} = 1$
if $k = k_{\max}$ then return endif
end while
if $j = j_{\max}$ then
— *Restart* —
$j = j_{\min}$
$\boldsymbol{V} = \boldsymbol{V}\boldsymbol{W}(:, 1{:}j);\ \boldsymbol{S} = \boldsymbol{S}(1{:}j, 1{:}j);\ \boldsymbol{G} = \boldsymbol{S}; \boldsymbol{W} = \boldsymbol{I}_j$
end if
— *Expansion of the subspace* —
if $\|\mathbf{r}\|_2 < \varepsilon_{\text{tr}}$ then $\sigma = \theta$ else $\sigma = \tau$ end if
Calculate the correction $\mathbf{z}$ according to Alg. 5.9
$\mathbf{z} = \text{mgsm}(\hat{\boldsymbol{Q}}, \hat{\boldsymbol{Q}}_{\boldsymbol{M}}, \mathbf{z})$
$[\mathbf{z}, r] = \text{icgsm}(\boldsymbol{V}, \mathbf{z});\ \mathbf{z} = \mathbf{z}/r$
$\boldsymbol{V} = [\boldsymbol{V}, \mathbf{z}];\ \mathbf{z}_{\boldsymbol{A}} = \boldsymbol{A}\mathbf{z};\ \boldsymbol{G} = [\boldsymbol{G}, \boldsymbol{V}^T\mathbf{z}_{\boldsymbol{A}}; \mathbf{z}_{\boldsymbol{A}}^T\boldsymbol{V}, \mathbf{z}^T\mathbf{z}_{\boldsymbol{A}}];$
$j = j + 1;\ i_{\text{step}} = i_{\text{step}} + 1;\ it = it + 1$
end while
end function

Algorithm 5.7: *Jacobi-Davidson algorithm*
JDSYM is an implementation of the Jacobi-Davidson method. JDSYM calculates eigensolutions $(\lambda_k, \mathbf{q}_k)$ of the generalised eigenvalue problem $\boldsymbol{A}\mathbf{q} = \lambda\boldsymbol{M}\mathbf{q}$

$n = \text{size}(\boldsymbol{A}, 1); it = 0; i_{\text{step}} = 0$
$j = 1; \mathbf{v}_0 = \mathbf{v}_0/\|\mathbf{v}_0\|; \boldsymbol{V} := \mathbf{v}_0; \boldsymbol{G} = \mathbf{v}_0^T(\boldsymbol{A}\mathbf{v}_0)$
$k = 0; \boldsymbol{Q} = \text{zeros}(n, 0); \boldsymbol{Q}_{\boldsymbol{M}} = \text{zeros}(n, 0); \boldsymbol{Q}_{\boldsymbol{K}} = \text{zeros}(n, 0); \boldsymbol{\lambda} = []$

Algorithm 5.8: *Initialisation of JDSYM*
This code sequence belongs to Alg. 5.7 and is given here due to space constraints.

Use iterative method for symmetric matrices according to scheme

System matrix: $(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})(\boldsymbol{A} - \sigma\boldsymbol{M})$

Preconditioner: $(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}$

Right hand side: $-(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}^{T})\mathbf{r}$

or use iterative method for general matrices according to scheme

System matrix: $(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}(\boldsymbol{A} - \sigma\boldsymbol{M})$

Preconditioner: $\boldsymbol{I}$

Right hand side: $-(\boldsymbol{I} - \hat{\boldsymbol{Q}}_{\boldsymbol{K}}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{Q}}_{\boldsymbol{M}}^{T})\boldsymbol{K}^{-1}\mathbf{r}$

with initial vector $\mathbf{0}$.
Stop after reduction of the residual norm by a factor of $\varepsilon_{\text{lin}} = \gamma^{-i_{\text{step}}}$ or until $it_{\text{max,lin}}$ iterations have been executed.

Algorithm 5.9: *Solution of the correction equation*
This code sequence belongs to Alg. 5.7 and is given here due to space constraints.

function $[\boldsymbol{W}, \boldsymbol{S}]$ = SORTEIG($\boldsymbol{W}$, $\boldsymbol{S}$, τ)
  $[\mathbf{t}_{\text{dummy}}, \mathbf{i}] = \mathsf{sort}(|\mathsf{diag}(\boldsymbol{S}) - \tau|)$
  $\boldsymbol{S} = \boldsymbol{S}(\mathbf{i}, \mathbf{i})$
  $\boldsymbol{W} = \boldsymbol{W}(:, \mathbf{i})$
end function

Algorithm 5.10: *SORTEIG*
SORTEIG sorts the eigenpairs in $(\boldsymbol{S}, \boldsymbol{W})$ such, that $|\boldsymbol{S}(i,i) - \tau| \leq |\boldsymbol{S}(i+1) - \tau|$ for $i = 1, \ldots, j-1$.

Important properties of the matrices in the JDSYM algorithm The eigenvectors of (5.13) are $\boldsymbol{M}$-orthogonal to each other. Thus

$$\boldsymbol{Q}^T\boldsymbol{M}\boldsymbol{Q} = \boldsymbol{I} \tag{5.35}$$

holds. As a result of the way the search space is constructed in Alg. 5.7

$$\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{V} = \boldsymbol{I} \tag{5.36}$$

holds. In Alg. 5.7 the search space is kept $\boldsymbol{M}$-orthogonal to the already converged eigenvectors. Therefore

$$\boldsymbol{V}^T\boldsymbol{M}\boldsymbol{Q} = \boldsymbol{0} \tag{5.37}$$

must hold. Due to the orthogonality constraint, on which the Rayleigh-Ritz approach is based on,

$$\boldsymbol{V}^T\mathbf{r} = \boldsymbol{V}^T(\boldsymbol{A}\mathbf{u} - \theta\boldsymbol{M}\mathbf{u}) = \boldsymbol{0} \tag{5.38}$$

holds. Since $\mathbf{u}$ lies in $\mathcal{R}(\boldsymbol{V})$,

$$\mathbf{u}^T\mathbf{r} = \mathbf{0} \tag{5.39}$$

must hold. The relation between $\mathbf{r}$ and $\boldsymbol{Q}$ is less obvious. From the definition of $\mathbf{r}$ follows

$$\boldsymbol{Q}^T\mathbf{r} = \boldsymbol{Q}^T\boldsymbol{A}\mathbf{u} - \theta\boldsymbol{Q}^T\boldsymbol{M}\mathbf{u}.$$

Since $\mathbf{u} \in \mathcal{R}(\boldsymbol{V})$ and because of Equation (5.37) we have $\mathbf{u} \perp_{\boldsymbol{M}} \boldsymbol{Q}$, or rewritten $\boldsymbol{Q}^T\boldsymbol{M}\mathbf{u} = \mathbf{0}$.

Now, let's assume that the eigenvectors stored in $\boldsymbol{Q}$ are exact to machine precision. Let $\boldsymbol{Q}^\perp$ be the matrix of eigenvectors not contained in $\boldsymbol{Q}$. So $\mathbf{u} \in \mathcal{R}(\boldsymbol{Q}^\perp)$ holds. Let $\mathbf{t}$ store the components of $\mathbf{u}$ in direction of the columns of $\boldsymbol{Q}^\perp$.

$$\begin{aligned}\boldsymbol{Q}^T\boldsymbol{A}\mathbf{u} &= \boldsymbol{Q}^T\boldsymbol{A}[\boldsymbol{Q}, \boldsymbol{Q}^\perp][\mathbf{0}, \mathbf{t}]^T \\ &= (\boldsymbol{Q}^T\boldsymbol{A}\boldsymbol{Q})\mathbf{0} + \boldsymbol{Q}^T\boldsymbol{A}\boldsymbol{Q}^\perp\mathbf{t}\end{aligned}$$

and because of (5.15)

$$\begin{aligned}&= \boldsymbol{\Lambda}\mathbf{0} + \mathbf{0}\mathbf{t} \\ &= \mathbf{0}.\end{aligned}$$

However, the eigenpairs $(\mathbf{q}_k, \lambda_k)$ calculated by Alg. 5.7 are not exact to machine precision, and thus the above calculation is not valid. Instead the eigenpairs $(\mathbf{q}_k, \lambda_k)$ satisfy

$$\|\boldsymbol{A}\mathbf{q}_k - \lambda_k\boldsymbol{M}\mathbf{q}_k\|_2 < \varepsilon \quad \forall k,$$

for the given bound ε (cf. Section 5.1.2). In the following estimation we take into account that the eigenvectors $\boldsymbol{Q}$ are not exact and denote these with $\tilde{\boldsymbol{Q}}$. Accordingly we define $\tilde{\boldsymbol{Q}}^\perp \in \mathbb{R}^{n\times n-k}$ with $(\tilde{\boldsymbol{Q}}^\perp)^T\boldsymbol{M}\tilde{\boldsymbol{Q}}^\perp = \boldsymbol{I}$ and $\tilde{\boldsymbol{Q}}^T\boldsymbol{M}\tilde{\boldsymbol{Q}}^\perp = \mathbf{0}$. Then we have $\operatorname{rank}([\tilde{\boldsymbol{Q}}, \tilde{\boldsymbol{Q}}^\perp]) = n$.

We define

$$\boldsymbol{R} := \boldsymbol{A}\tilde{\boldsymbol{Q}} - \boldsymbol{M}\tilde{\boldsymbol{Q}}\boldsymbol{\Lambda}$$

with

$$\|\boldsymbol{R}\|_F \leq \sqrt{k}\,\varepsilon.$$

By transposition and by multiplication with $\tilde{\boldsymbol{Q}}^{\perp}$ from the right we get

$$\boldsymbol{R}^T\tilde{\boldsymbol{Q}}^{\perp} = \tilde{\boldsymbol{Q}}^T\boldsymbol{A}\tilde{\boldsymbol{Q}}^{\perp} - \boldsymbol{\Lambda}\tilde{\boldsymbol{Q}}^T\boldsymbol{M}\tilde{\boldsymbol{Q}}^{\perp} = \tilde{\boldsymbol{Q}}^T\boldsymbol{A}\tilde{\boldsymbol{Q}}^{\perp}.$$

Since $\mathcal{R}(\boldsymbol{V}) \subseteq \mathcal{R}(\tilde{\boldsymbol{Q}}^{\perp})$, $\mathbf{u}$ can be represented by its components in $\mathcal{R}(\tilde{\boldsymbol{Q}}^{\perp})$ (analogous to above): $\mathbf{u} = \tilde{\boldsymbol{Q}}^{\perp}\mathbf{t}$. This finally yields the following estimation:

$$\begin{aligned}
\|\tilde{\boldsymbol{Q}}^T\mathbf{r}\|_2 &= \|\tilde{\boldsymbol{Q}}^T(\boldsymbol{A}\mathbf{u} - \theta\boldsymbol{M}\mathbf{u})\|_2 = \|\tilde{\boldsymbol{Q}}^T\boldsymbol{A}\mathbf{u}\|_2 \\
&= \|\tilde{\boldsymbol{Q}}^T\boldsymbol{A}\tilde{\boldsymbol{Q}}^{\perp}\mathbf{t}\|_2 \\
&= \|\boldsymbol{R}^T\tilde{\boldsymbol{Q}}^{\perp}\mathbf{t}\|_2 \\
&\leq \|\boldsymbol{R}\|_F\|\tilde{\boldsymbol{Q}}^{\perp}\mathbf{t}\|_2 \\
&\leq \sqrt{k}\,\varepsilon\|\mathbf{u}\|_2.
\end{aligned} \tag{5.40}$$

Due to $\mathbf{u}^T\boldsymbol{M}\mathbf{u} = 1$ we have

$$1/\sqrt{\|\boldsymbol{M}\|_2} \leq \|\mathbf{u}\|_2 \leq \sqrt{\|\boldsymbol{M}^{-1}\|_2}.$$

Computational cost and memory requirements The memory requirements of Alg. 5.7 are governed by the matrices $\hat{\boldsymbol{Q}}$, $\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$, $\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$ and $\boldsymbol{V}$. They occupy $(3k_{\max} + j_{\max})\,n$ memory locations[17]. All other matrices are $k_{\max} \times k_{\max}$ or $j_{\max} \times j_{\max}$ at the most, and therefore can be neglected for realistic problem sizes. The memory taken up by the matrices $\boldsymbol{A}$, $\boldsymbol{M}$ and the preconditioner $\boldsymbol{K}$ is not taken into account here.

For the analysis of the computational cost we only consider operations that depend on the matrix order n.

Tab. 5.1 shows the computational cost for the Jacobi-Davidson iteration, subdivided into the various parts of Alg. 5.7. The Ritz approximation and the expansion of $\boldsymbol{V}$ are carried out in every JD iteration step. The other parts are carried out when necessary.

Tab. 5.2 shows the computational cost for applying the operator and preconditioner in one inner iteration. Operations executed by the iterative solver itself, are not counted. Usually these are a few DAXPY- and DDOT operations per iteration. The cost of these operations is small, when compared to the costs shown in Tab. 5.2. E.g., the QMRS iterative solver executes 3 DAXPY- and 3 DDOT-operations per iteration step. The CGS iteration executes (apart from two multiplications with the system matrix and two preconditioning steps) 2 DDOTs and 6 DAXPYs in each iteration.

5.1.3 Choosing suitable parameters for the Jacobi-Davidson algorithm

The performance of the JDSYM algorithm (Alg. 5.7) depends on the values of many parameters. Only for some parameters a good choice is obvious or already given by the underlying problem: $k_{\max}$ is the number of requested eigensolutions and ε is the requested error tolerance. The parameter $it_{\max}$ can safely be chosen very large, such that all requested eigenvalues are computed. The parameter τ determines which eigensolutions JDSYM will converge to. However, if extremal eigenvalues are computed, there is still some freedom for choosing τ.

[17] By the term *memory location* we refer to the space taken up by one floating point number (usually 8 bytes for a double precision number)

Operation	$y \leftarrow Ax$	$y \leftarrow Mx$	$y \leftarrow K^{-1}x$	MEMOP($n,\ldots$)	DGEMM($n,\ldots,\ldots$)	DGEMV($n,\ldots$)	DAXPY(n)	DDOT(n)
Ritz approx.	1	1	1	3		$j+k$	1	2
Expansion of $\boldsymbol{V}$	1	2		2		$2(k+1)$ $+j$	$k+1$	$k+1$ $+3$
Convergence				j	$j(j-1)$			
Restart				j_{max}	$j_{\text{max}}j_{\text{min}}$			

Table 5.1: *Computational cost of various parts of the JDSYM algorithm*
The MEMOP column shows the number of memory copy and memory write operations, that are not part of the other listed operations.

Variant	$y \leftarrow Ax$	$y \leftarrow Mx$	$y \leftarrow K^{-1}x$	DGEMV($n,\ldots$)	DAXPY(n)
Symmetric form	1	1	1	$4(k+1)$	1
Non-symmetric form	1	1	1	$2(k+1)$	1

Table 5.2: *Computational cost of an inner iteration step*

For the other parameters j_{min}, j_{max}, ε_{tr}, $it_{\text{max,lin}}$ and γ the best choice is not evident. This is also true for the shift σ, which is used to construct the preconditioner $\boldsymbol{K}$. Since it is difficult to describe the impact of these parameters on the execution time of JDSYM theoretically, we ran a series of numerical experiments to better understand their influence on the performance of JDSYM and to find good choices for them.

We implemented a simple optimisation scheme: We start with reasonable values for all parameters and then optimise each parameter individually, one after the other. This is repeated over and over again. To optimise one parameter a simple procedure is undertaken: The eigenvalue problem is solved for (about 10) different values of this parameter. The value which yields the best execution time is then used for subsequent calculations.

We performed this optimisation for the Maxwell eigenvalue problem (Problem II) using vector finite elements with the SAUG and the EIGSOLV approach. For all calculations we used the Prec2LevJACssor preconditioner. The optimisation was carried out for four different grids: *copcav2*, *boxcav16x10x3*, *cop10k* and *copcav18* and for $k_{\text{max}} = 5$, 10 and 20. Some parameters are fixed: We set $\varepsilon = 10^{-6}$ and $it_{\text{max}} = 1000$. The optimisation was run for the parameters τ, σ, j_{max}, j_{min}, ε_{tr}, $it_{\text{max,lin}}$ and γ.

The amount of data generated by these experiments is large, since the eigenvalue problems are solved hundreds of times. We abstain from reporting raw data. Instead we discuss our experience for choosing each parameter.

Target τ We did not optimise this parameter for EIGSOLV, since this method adjusts τ dynamically. The initial value is set close to but smaller than λ_1.

When using the SAUG method the parameter τ is chosen in the range $[0, (\lambda_1 + \lambda_{k_{\max}})/2]$. Choosing τ larger will result in convergence to wrong eigensolutions. Choosing $\tau = 0$ has the advantage, that the number of multiplications with the matrix $\boldsymbol{M}$ is reduced. In most cases $\tau = 0$ seems to be the best choice. In the other cases the optimal value is close to but smaller than $(\lambda_1 + \lambda_{k_{\max}})/2$.

For $k_{\max} = 20$ and the two larger grids τ has a large impact on the performance of the algorithm. In these cases setting τ to zero results in a gain of over 100% in execution time compared to the worst case. For smaller $k_{\max}$ the parameter τ has only a limited impact on the execution time of the eigensolver. Then one can gain about 15% if τ is chosen optimally.

Shift for the preconditioner σ We choose the parameter σ from the range $(0, (\lambda_1 + \lambda_{k_{\max}})/2]$. The choice $\sigma = 0$ is not possible, since this would result in a singular preconditioner. σ has a large impact on the performance of the eigenvalue solver especially for the larger grids, where fluctuations can amount to more than 100%.

For the EIGSOLV method the optimal value is always close to but smaller than $(\lambda_1 + \lambda_{k_{\max}})/2$. For $k_{\max} = 5$ and $k_{\max} = 10$ this is also true with SAUG in most cases. For $k_{\max} = 20$ with SAUG a value close to but smaller than λ_1 seems to be the better choice.

It turns out, that the optimal values of both σ and τ are often close together.

Minimal and maximal dimension of the search space $j_{\min}$ and $j_{\max}$ The parameter $j_{\max}$ is chosen in the range $j_{\min} + 1, \ldots, 6k_{\max}$ and $j_{\min}$ is chosen in the range $1, \ldots, j_{\max} - 1$. The experimental results show, that the execution time is almost insensitive to the choice of $j_{\min}$ and $j_{\max}$.

When using SAUG the performance degrades if $j_{\min}$ is very small ($j_{\min} < k_{\max}/4$) or if $j_{\min}$ and $j_{\max}$ are very close ($j_{\max} - j_{\min} \leq 3$). With EIGSOLV the algorithm is still efficient for very small values of $j_{\min}$. In some cases $j_{\min} = 1$ yields the best results.

In all cases except the ones mentioned above the execution time only fluctuates by a few percents. So there is no point in choosing these parameters large, as this would waste memory. A reasonable choice is $j_{\min} = k_{\max}/2$ and $j_{\max} = k_{\max}$. With these settings, the memory requirements of the JDSYM algorithm amount to $4k_{\max}$ floating point numbers (cf. page 76 ff).

Tracking parameter ε_{tr} The parameter ε_{tr} is chosen in $[\varepsilon, 10^2]$. The choice of ε_{tr} can have a large impact on the performance of JDSYM and result in a gain of up to 80% in execution time, compared to the worst choice. Unfortunately the optimal value depends heavily on the grid, $k_{\max}$ and the solution method (SAUG or EIGSOLV).

For SAUG the optimal value is in the range $[\varepsilon, 10^{-4}]$. For EIGSOLV the optimal value is larger, usually in the range $[1, 100]$. There are however many exceptions to this rule. Therefore we suggest to use $\varepsilon_{\text{tr}} = 10^{-3}$ if no better value is known.

Maximal number of inner iterations $it_{\text{max,lin}}$ The parameter $it_{\max,\text{lin}}$ is chosen in the interval $2, \ldots, 500$.

With the SAUG method and $k_{\max} = 5$ or $k_{\max} = 10$ the optimal value seems to be approximately 20. However the performance of JDSYM seems to be almost indifferent to the value $it_{\max,\text{lin}}$ for any value larger than 15 in this case. For $k_{\max} = 20$ the best results were obtained

with $it_{\text{max,lin}} = 5$. For the two large grids the gain amounts to almost 90%, compared to the worst choice.

With EIGSOLV the optimal value for $it_{\text{max,lin}}$ tends to be larger. It is usually around 50 for all values of k_{max}. The performance seems to be almost indifferent to the choice of $it_{\text{max,lin}}$ as long as it is chosen larger than 15 for any k_{max}.

It must be noted, that setting $it_{\text{max,lin}}$ very small can be dangerous, especially if ill-conditioned inner systems have to be solved. The corrections computed in the inner iteration may be ineffective, and JDSYM may fail to converge.

Decay parameter γ We choose ε_{tr} in $[1.1, 5.0]$. The execution time is not very sensitive to the value of ε_{tr}; it fluctuates by 15% at the most. For SAUG the optimal choice for γ is in the range $[1.5, 4.0]$. For EIGSOLV the optimal value is in the range $[2.4, 3.6]$.

Summary In this section we showed, that choosing good parameters for JDSYM is not a simple task. For most parameters, the optimal choice depends on the grid, the method for solving the indefinite eigenproblem and the number of eigenvalues to be computed. We have not examined the influence of the preconditioner, but it certainly has an impact on $it_{\text{max,lin}}$. Further, it has to be noted, that the guidelines given in this section cannot be generalised to other problems.

5.1.4 Block Jacobi-Davidson Algorithm for the generalised symmetric eigenvalue problem

In this section we consider a block version of the JDSYM algorithm discussed in Section 5.1.2. The eigenvalue problem is expected to have the same form and properties as for JDSYM.

With the conventional Jacobi-Davidson algorithm problems can arise, if the desired eigenvalues are multiple or if they are very close together: If the current eigenpair approximation $(\lambda, \mathbf{u})$ is very accurate, the system matrix of the correction equation becomes ill-conditioned. If λ is simple and well separated from the rest of the spectrum, then working in the orthogonal complement remedies the ill condition. If λ is a multiple eigenvalue then this is not enough; the constrained system can be almost singular, too. Though the projections in the correction equation remove components in one "bad" direction, the other "bad" directions stemming from the multiple eigenvalue remain. In fact the system matrix can become almost singular. We suspect that this may result in an unsatisfactory convergence behaviour of the iterative method.

The block version of the algorithm presented in this section tries to fix these issues. The idea is the following: The block algorithm is parametrised by an additional parameter: the block size n_{b}. Now n_{b} Ritz pairs are computed in each JD iteration, instead of only one. For each of these n_{b} Ritz pairs a search direction is computed from the correction equation. The projections in the correction equation are adjusted in such a way, that they keep the solutions orthogonal to *all* Ritz vectors. These modification prevent the system matrix from becoming nearly singular.

The following list describes the changes made for the block version in detail:

Initialisation The JD algorithm is started with a search space of dimension n_{b}.

Projected eigenvalue problem no changes

Ritz approximation n_{b} Ritz pairs with Ritz values closest to τ are calculated. n_{b} new columns or rows and columns respectively are calculated for $\hat{\boldsymbol{Q}}$, $\hat{\boldsymbol{Q}}_{\boldsymbol{M}}$, $\hat{\boldsymbol{Q}}_{\boldsymbol{K}}$ and $\hat{\boldsymbol{F}}$.

Convergence There are two variants: In the first case, the iteration continues until all n_b Ritz pairs fulfil the convergence criterion. In the second case the Ritz pairs are treated individually and removed from the search space as soon as they satisfy the convergence criterion.

Restart no changes

Subspace expansion The correction equation is solved n_b times with n_b different shifts and right hand sides. This results in n_b new search directions.

Correction equation $\hat{\boldsymbol{Q}}$ includes the selected n_b Ritz vectors. The projections in the system matrix and in the preconditioner keep the new search directions $\boldsymbol{M}$-orthogonal to all n_b Ritz vectors.

Experimental results We now try to confirm the advantages of the block algorithm by means of a concrete example.

Because Problems I and II do generally not yield eigenvalue problems with multiple eigenvalues, we artificially create such an eigenvalue problem: We use a small eigenvalue problem $\boldsymbol{A}\mathbf{u} = \lambda\boldsymbol{M}\mathbf{u}$ stemming from Problem II and construct two 5×5 block diagonal matrices $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{M}}$, by using the original matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ as diagonal blocks. This yields a matrix eigenvalue problem

$$\tilde{\boldsymbol{A}}\mathbf{u} = \lambda\tilde{\boldsymbol{M}}\mathbf{u}, \tag{5.41}$$

with all eigenvalues being fivefold. This eigenvalue problem is now solved with the Jacobi-Davidson block algorithm using several different block sizes n_b. In this experiment we calculate 30 eigenpairs using the parameters $j_\mathrm{min} = 30$, $j_\mathrm{max} = 40$ and no preconditioner.

n_b	t_eig	$it_\mathrm{inner,\ avg}$
1	28.86	19.12
2	29.51	19.62
3	30.81	20.52
4	30.85	20.55
5	31.79	20.04
6	38.75	23.31
7	41.01	24.39
8	42.62	24.74
9	44.75	24.95
10	40.55	22.32

Table 5.3: *Experiment with the JD block algorithm*
n_b is the block size, t_eig is the total computational time for computing 30 eigenpairs, $it_\mathrm{inner,\ avg}$ is the average number of inner iterations for solving the correction equation.

Tab. 5.3 shows the results for different block sizes n_b. The row with $n_\mathrm{b} = 1$ corresponds to the non blocked JDSYM algorithm (Alg. 5.7). It is apparent that the block algorithm shows no improvement when compared to the standard algorithm with this problem. The standard JDSYM algorithm has no difficulties computing the multiple eigenvalues. The misgivings expressed before came out to be causeless. This behaviour has been observed with other matrix eigenvalue problems, too.

It remains to be clarified why the standard JDSYM algorithm has no difficulties computing multiple eigenvalues. Two effects are responsible for this:

- ▷ By analogy to the inverse power method [36, page 408], components in direction of the eigenvectors associated with a multiple eigenvalue λ_1 close to the shift σ will be amplified. The correction $\mathbf{z}$ will be orthogonal to the Ritz vector $\mathbf{u}$, but still have components in direction of the eigenspace associated with λ_1. So, while the convergence to the first eigenvector of λ_1 is underway, the components in direction of the remaining eigenvectors associated with λ_1 are also accumulated in the search space. Typically, the convergence to the first eigenvector is slow, but once it has converged, the other eigenvectors associated with λ_1 will quickly follow.

- ▷ Since the system matrix of the correction equation becomes nearly singular, certain components of the residual are reduced only slowly. However, since only modest accuracy is required for the correction, usually only few inner iterations are needed to satisfy the stopping criterion, anyhow.

- ▷ The right hand side of the correction equation is orthogonal to $\mathcal{R}(\boldsymbol{V}_M)$. Close to convergence one can expect that the solution $\mathbf{z}$ has only small components on direction of the (approximate) eigenvectors corresponding to λ.

5.1.5 Solving the indefinite eigenvalue problem

If Problem II (time-independent Maxwell equations) is solved using vector finite elements as described in Section 3.3, then matrix eigenvalue problems of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}, \quad \boldsymbol{A}, \boldsymbol{M} \in \mathbb{R}^{n\times n} \tag{5.42}$$

emerge. $\boldsymbol{A}$ is symmetric positive semi-definite with a null space of high dimension, and $\boldsymbol{M}$ is symmetric positive definite. The task is to compute the smallest *positive* eigenvalues and associated eigenvectors of (5.42). This section describes, how the JDSYM algorithm can be extended to avoid convergence to the unwanted zero eigenvalues.
Three steps are necessary:

1. The target value τ is chosen near, but smaller than the smallest positive eigenvalue of (5.42).

2. If an eigenpair converges in the JD algorithm, the target value τ is set equal to the just converged eigenvalue. Thus, τ is increased and is now further away from zero and also nearer to the next desired eigenvalue.

3. As described earlier the Ritz pairs computed in the Rayleigh-Ritz step are ordered according to equation (5.21) on page 65 by their distance from τ. With this ordering, the Ritz value closest to τ is used as the eigenvalue approximation.

   The ordering must be adjusted, such that all Ritz values smaller than the current target τ are moved all the way to the bottom. In this way, convergence to the null space is prevented effectively, since now no undesired Ritz value is chosen as eigenvalue approximation and also because the Ritz vectors with large components in direction of the null space are eliminated when the next restart happens.

In the numerical experiments we conducted, this method worked effectively for all grids we tested. Experimental results are given in Section 4.2.6.

5.1.6 Related work

In [8, Section 5.6] Sleijpen and van der Vorst present the application of the Jacobi-Davidson approach to the generalised Hermitian eigenvalue problem. Their algorithm, which is of course also applicable to the generalised symmetric eigenvalue problem (5.13), shares many similarities with JDSYM (Alg. 5.7). The differences are the following:

- ▷ The algorithm presented in [8, Section 5.6] stores the matrices $\boldsymbol{AV}$ and $\boldsymbol{MV}$ explicitly. Thus, it requires storage for $(3k_{\max} + 3j_{\max})\, n$ floating point numbers, whereas JDSYM only needs space for $(3k_{\max} + j_{\max})\, n$ floating point numbers (cf. page 76 ff).
- ▷ The algorithm in [8, Section 5.6] does not project the right-hand-side of the correction equation onto $\mathcal{R}(\boldsymbol{Q})^{\perp}$.
- ▷ The use of iterative solvers for symmetric linear systems like SYMMLQ, MINRES or QMRS is not studied in [8, Section 5.6].
- ▷ The authors suggest the use of harmonic Petrov values instead of Ritz values if interior eigenvalues are computed. JDSYM uses the techniques described in Section 5.1.5 to do the same.

In [32] Genseberger and Sleijpen present a new variation of the Jacobi-Davidson algorithm. In contrast to the standard algorithm the new JDV method keeps the correction $\mathbf{z}$ implicitly orthogonal to *all* search directions. In this way the authors seek to obtain a better-conditioned system matrix in the correction equation. This approach is similar to our block approach, both in terms of the motivation and also in terms of the implementation. However, the results presented in [32] indicate, that JDV is inferior to the standard Jacobi-Davidson algorithm.

5.2 Implicitly restarted Lanczos algorithm (IRL)

The standard Lanczos method as e.g. described in [57][37][36, p. 470 ff.] is one of the best known algorithms for computing eigensolutions of sparse symmetric eigenvalue problems. The number of required iterations taken by the Lanczos method can be very high, depending on the matrix properties. A high memory consumption for storing the basis vectors of the search space and a large computational cost for the orthogonalisations are the outcome. Sorensen suggested the "Implicitly Restarted Lanczos" method (IRL method), which deals with this problem by employing a clever variant of the restart [49]. The Fortran 77 software package ARPACK by Lehoucq and Sorensen implements the IRL method. The IRL method is outlined below:

The Lanczos iteration with the shift-and-invert spectral transformation

$$\begin{aligned} \mathbf{q}_{j+1}\beta_{j+1} = \mathbf{r}_{j+1} &= (\boldsymbol{A} - \sigma\boldsymbol{M})^{-1}\boldsymbol{M}\mathbf{q}_j - \mathbf{q}_j\alpha_j - \mathbf{q}_{j-1}\beta_j, \\ \text{with} \quad \alpha_j = \mathbf{q}_j^T\boldsymbol{M}(\boldsymbol{A} - \sigma\boldsymbol{M})^{-1}\boldsymbol{M}\mathbf{q}_j \quad &\text{and} \quad \mathbf{q}_i^T\boldsymbol{M}\mathbf{q}_j = \delta_{ij}, \end{aligned} \tag{5.43}$$

is executed for $j = 1, 2, \ldots$, until $j = j_{\max}$. Here, $j_{\max}$ is the maximal dimension of the search space. For stability reasons complete reorthogonalisation is performed, i.e. each new $\mathbf{q}_{j+1}$ is $\boldsymbol{M}$-orthogonalised to the whole search space. The search space $\boldsymbol{V}_j$ is spanned by the vectors $\mathbf{q}_1, \ldots, \mathbf{q}_j$. Let

$$\boldsymbol{V}_j = [\mathbf{q}_1, \ldots, \mathbf{q}_j],$$

and let

$$\boldsymbol{T}_j = \text{tridiag}\begin{pmatrix} & \beta_2 & \beta_3 & \cdots & \beta_j \\ \alpha_1 & \alpha_2 & \cdots\cdots & & \alpha_j \\ & \beta_2 & \beta_3 & \cdots & \beta_j \end{pmatrix}. \tag{5.44}$$

After $j_{\max}$ iterations the Lanczos relation

$$(\boldsymbol{A}-\sigma\boldsymbol{M})^{-1}\boldsymbol{M}\boldsymbol{V}_{j_{\max}} = \boldsymbol{V}_{j_{\max}}\boldsymbol{T}_{j_{\max}} + \mathbf{r}_{j_{\max}+1}\mathbf{e}_{j_{\max}}^T \tag{5.45}$$

is satisfied. Now $k = j_{\max} - k_{\max}$ sweeps of the QR algorithm with the shifts $\mu_1, \ldots, \mu_k$ are applied on $\boldsymbol{T}_j$, such that $\hat{\boldsymbol{T}} = \hat{\boldsymbol{Q}}^T\boldsymbol{T}_j\hat{\boldsymbol{Q}}$ with $\hat{\boldsymbol{Q}} = \boldsymbol{Q}_1 \ldots \boldsymbol{Q}_k$, where $\boldsymbol{Q}_i$ represents the QR sweep with shift μ_i. After multiplication with $\hat{\boldsymbol{Q}}$ from the right we get

$$(\boldsymbol{A}-\sigma\boldsymbol{M})^{-1}\boldsymbol{M}\boldsymbol{V}_{j_{\max}}\hat{\boldsymbol{Q}} = \boldsymbol{V}_{j_{\max}}\hat{\boldsymbol{Q}}(\hat{\boldsymbol{Q}}^T\boldsymbol{T}_j\hat{\boldsymbol{Q}}) + \mathbf{r}_{j_{\max}+1}\mathbf{e}_{j_{\max}}^T\hat{\boldsymbol{Q}}. \tag{5.46}$$

Since the orthogonal matrices $\boldsymbol{Q}_i$ have Hessenberg form, the matrix $\hat{\boldsymbol{Q}}$ has k sub-diagonals. Because of this only the last $k+1$ columns of $\mathbf{r}_{j_{\max}+1}\mathbf{e}_{j_{\max}}^T\hat{\boldsymbol{Q}}$ are non-zero.

Let $\hat{\boldsymbol{V}}_{k_{\max}}$ the matrix consisting of the first $k_{\max}$ columns of $\boldsymbol{V}_{j_{\max}}\hat{\boldsymbol{Q}}$ and let $\hat{\boldsymbol{T}}_{k_{\max}}$ be the diagonal block of $\hat{\boldsymbol{T}} = \hat{\boldsymbol{Q}}^T\boldsymbol{T}_j\hat{\boldsymbol{Q}}$ consisting of the first $k_{\max}$ rows and columns and let $\hat{\mathbf{r}}_{k_{\max}+1}$ be the $k_{\max}+1$-th column of $\beta_{k_{\max}+1}\boldsymbol{V}\hat{\boldsymbol{Q}}$ added to $k_{\max}$-th column of $\mathbf{r}_{j_{\max}+1}\mathbf{e}_{j_{\max}}^T\hat{\boldsymbol{Q}}$. Then we can write

$$(\boldsymbol{A}-\sigma\boldsymbol{M})^{-1}\boldsymbol{M}\hat{\boldsymbol{V}}_{k_{\max}} = \hat{\boldsymbol{V}}_{k_{\max}}\hat{\boldsymbol{T}}_{k_{\max}} + \mathbf{r}_{k_{\max}+1}\mathbf{e}_{k_{\max}}^T. \tag{5.47}$$

Equation (5.47) corresponds to the Lanczos relation, that one would get, if $k_{\max}$ steps of the iteration (5.43) where performed with the start vector $\hat{\boldsymbol{V}}_{k_{\max}}\mathbf{e}_1$. Since the information of $j_{\max} - k_{\max}$ search directions is lost at each restart, the IRL method converges slower than the Lanczos method.

In the above discussion, the choice of the shifts was not considered. In the ARPACK package, when using standard settings, they are set to the k eigenvalues of $\hat{\boldsymbol{T}}_{k_{\max}}$ with the smallest magnitude[18]. If the shifts $\mu_1, \ldots, \mu_k$ are equal to eigenvalues of $\hat{\boldsymbol{T}}_{k_{\max}}$, they are called *exact*, or *perfect shifts*. With perfect shifts all components in direction of the associated Ritz vectors are eliminated. Other variants for choosing the shifts are possible [50]. However, the ARPACK strategy has provided satisfactory results in our practical experiments.

The memory requirements of Alg. 5.11 add up to $2nj_{\max}$ floating point numbers. This corresponds to the maximal size of matrices $\boldsymbol{V}$ and $\boldsymbol{V}_{\boldsymbol{M}}$. Alongside memory is needed for the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$. To solve the linear system of equations with $\boldsymbol{A}-\sigma\boldsymbol{M}$ efficiently, it may be beneficial to store this matrix separately[19].

5.2.1 Solving the indefinite eigenvalue problem

If Problem II (time-independent Maxwell equations) is solved using vector finite elements as described in Section 3.3, then matrix eigenvalue problems of the form

$$\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{M}\mathbf{x}, \quad \boldsymbol{A}, \boldsymbol{M} \in \mathbb{R}^{n\times n} \tag{5.48}$$

emerge. $\boldsymbol{A}$ is symmetric positive semi-definite with a null space of high dimension, and $\boldsymbol{M}$ is symmetric positive definite. The task is to compute the smallest *positive* eigenvalues and

[18] The eigenvalues of $\hat{\boldsymbol{T}}_{k_{\max}}$ with the smallest magnitude correspond to the Ritz values of the original problem, that are furthest away from σ.

[19] In contrast to the JD algorithm this is feasible here, since the shift σ is constant.

associated eigenvectors of (5.48). This section describes, how the ARPACK software can be configured to avoid convergence to the unwanted null space.
Three measures are necessary:

1. The shift σ is chosen near, but smaller than the smallest positive eigenvalue of (5.48).

2. ARPACK must be called using the parameter `WHICH = 'LA'`. In this way ARPACK is instructed to converge to eigensolutions with associated eigenvalues closest to σ, but larger than σ.

   The parameter `WHICH = 'LA'` affects the way the shifts $\mu_1, \ldots, \mu_k$ for the QR sweeps are chosen: Instead of selecting the Ritz values with smallest magnitude, the algebraically smallest Ritz values are chosen. As mentioned above, all components in direction of the k associated Ritz vectors are eliminated from the search space $\boldsymbol{V}$. Thus, IRL will converge to the eigensolutions with the algebraically largest transformed eigenvalues. This corresponds to the desired eigenvalues, that are closest to shift σ and also larger than σ. However, slow convergence is possible.

In the numerical experiments we conducted, this method worked effectively for all grids we tested. Experimental results are given in Section 4.2.6.

function$[\boldsymbol{V}, \boldsymbol{\lambda}, it] = \mathsf{irl}(\boldsymbol{A}, \boldsymbol{M}, k_{\max}, j_{\max}, \mathbf{q}, \sigma, \varepsilon)$
  $n = \mathsf{size}(\boldsymbol{A}, 1);\ it = 1;\ j = 1$
  — Initialisation of the Lanczos iteration —
  $\mathbf{q} = \mathbf{q}/\mathsf{sqrt}(\mathbf{q}^T \boldsymbol{M} \mathbf{q});\ \mathbf{q}_{\boldsymbol{M}} = \boldsymbol{M}\mathbf{q}$
  $\boldsymbol{V} = [\mathbf{q}];\ \boldsymbol{V}_{\boldsymbol{M}} = [\mathbf{q}_{\boldsymbol{M}}]$
  solve $(\boldsymbol{A} - \sigma \boldsymbol{M})\mathbf{z} = \mathbf{q}_{\boldsymbol{M}}$
  $\boldsymbol{\alpha} = [\mathbf{z}^T \mathbf{q}_{\boldsymbol{M}}];\ \boldsymbol{\beta} = [0]$
  $\mathbf{r} = \mathbf{z} - \alpha_1 \mathbf{q}$
  while 1
    $j = j + 1;\ it = it + 1$
    — Lanczos iteration and complete reorthogonalisation —
    $\beta_j = \mathsf{sqrt}(\mathbf{r}^T \boldsymbol{M} \mathbf{r})$
    $\mathbf{q} = \mathbf{r}/\beta_j;\ \mathbf{q}_{\boldsymbol{M}} = \boldsymbol{M}\mathbf{q}$
    $\boldsymbol{V} = [\boldsymbol{V}, \mathbf{q}];\ \boldsymbol{V}_{\boldsymbol{M}} = [\boldsymbol{V}_{\boldsymbol{M}}, \mathbf{q}_{\boldsymbol{M}}]$
    solve $(\boldsymbol{A} - \sigma \boldsymbol{M})\mathbf{z} = \mathbf{q}_{\boldsymbol{M}}$
    $\mathbf{h} = \boldsymbol{V}_{\boldsymbol{M}}^T \mathbf{z};\ \mathbf{r} = \mathbf{z} - \boldsymbol{V}\mathbf{h}$
    $\mathbf{c} = \boldsymbol{V}_{\boldsymbol{M}}^T \mathbf{r};\ \mathbf{r} = \mathbf{r} - \boldsymbol{V}\mathbf{c};\ \mathbf{h} = \mathbf{h} + \mathbf{c}$
    $\alpha_j = h_j$
    — Eigenvalue decomposition of the tridiagonal matrix T_j —
    $[\boldsymbol{\theta}, \boldsymbol{S}] = \mathsf{qrsym}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
    $[\boldsymbol{\theta}, ii] = \mathsf{sort}(-\boldsymbol{\theta}); \boldsymbol{\theta} = -\boldsymbol{\theta}; \boldsymbol{S} = \boldsymbol{S}(:, ii)$
    — Convergence criterion —
    if $k_{\max} \leq j$ and $\mathsf{all}(\beta_j |\boldsymbol{S}(j, 1{:}k_{\max})| < \varepsilon)$, break, end
    — Implicit restart —
    if $j = j_{\max}$
      $[\hat{\boldsymbol{Q}}, \boldsymbol{\alpha}, \boldsymbol{\beta}] = \mathsf{perfectshiftqr}(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}(k_{\max} + 1{:}j_{\max}))$
      $\mathbf{r} = \boldsymbol{V}\hat{\boldsymbol{Q}}(:, k_{\max} + 1)\beta_{k_{\max}+1} + \mathbf{r}\hat{q}_{j_{\max}, k_{\max}}$
      $\boldsymbol{V} = \boldsymbol{V}\hat{\boldsymbol{Q}}(:, 1{:}k_{\max});\ \boldsymbol{V}_{\boldsymbol{M}} = \boldsymbol{V}_{\boldsymbol{M}}\hat{\boldsymbol{Q}}(:, 1{:}k_{\max})$
      $\boldsymbol{\alpha} = \boldsymbol{\alpha}(1{:}k_{\max});\ \boldsymbol{\beta} = \boldsymbol{\beta}(1{:}k_{\max});$
      $j = k_{\max}$
    end
  end
  $\boldsymbol{V} = \boldsymbol{V}\boldsymbol{S}(:, 1{:}k_{\max});\ \boldsymbol{\lambda} = \sigma + 1./\boldsymbol{\theta}(1{:}k_{\max})$

Algorithm 5.11: *IRL*
Implicitly restarted Lanczos algorithm with complete reorthogonalisation for the generalised symmetric eigenvalue problem $\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x}$, with $\boldsymbol{A}$, $\boldsymbol{M}$ symmetric and $\boldsymbol{M} > 0$, using the shift-and-invert spectral transformation

5.3 Comparison of eigensolvers

The section briefly discusses the common features and differences of both the JDSYM and the IRL methods. Numerical results comparing the performance of both solvers are given in Section 4.2.6.

Both methods compute an $\boldsymbol{M}$-orthogonal basis for the search subspace and perform restarts to limit the dimension of this subspace. The IRL method computes a Krylov subspace, but JDSYM does not. Both methods solve a linear system of equations in each iteration. IRL solves a system of the form $(\boldsymbol{A} - \sigma \boldsymbol{M})\mathbf{z} = \mathbf{q}_{\boldsymbol{M}}$ with a constant σ to high accuracy[20]. Here high accuracy means, that a more stringent stopping criterion is used for the linear system than for the eigenvalue solver. JDSYM solves the correction equation with a variable shift in each iteration. The correction equation includes expensive projection and is usually solved only approximately. The memory consumption of both methods is comparable.

Comparing the computational cost of both methods is difficult, since the number of inner and outer iterations is a priori unknown. For this reason we restrict ourselves to compare the performance of the two methods by the means of numerical experiments.

The experimental results reported in Sections 4.2.6 and 8.4 show that JDSYM is faster than ARPACK in all cases. This can be mainly attributed to the fact that JDSYM allows for inaccurate solutions of the correction equation, whereas ARPACK requires an accurate solution of the shifted operator.

[20] The ARPACK documentation [50] states, that the convergence test for the inner iteration must be more stringent than the accuracy requirements for the eigenvalue approximations.

6 Iterative methods

In this chapter the iterative methods used in the eigensolvers are briefly introduced. It is not the intention of this chapter to give very detailed description or even to derive these methods. For a more thorough look at iterative methods see e.g. [11].

In the simplest case, the linear systems of equations have the form

$$\boldsymbol{A}_\sigma \mathbf{x} = \mathbf{b}, \tag{6.1}$$

with symmetric $\boldsymbol{A}_\sigma = \boldsymbol{A} - \sigma \boldsymbol{M}$. If the indefinite eigenvalue problem is solved, the linear system may include additional terms with the matrices $\boldsymbol{C}$, $\boldsymbol{Y}$ and/or $\boldsymbol{H}$, depending on the method chosen (cf. Section 4.2). If JDSYM's correction equation (5.24) is solved, additional projections onto $\mathcal{R}(\boldsymbol{Q})^{\perp_{\boldsymbol{M}}}$ are necessary.

When using the IRL eigensolver, the shift σ can be chosen in such a way, that the system matrix is positive definite. On the other hand, in JDSYM the shift is determined by the eigensolver, and thus the system matrix may become indefinite. For one form of the correction equation the linear system may even become non-symmetric (cf. page 68).

All methods presented in this chapter fall into the category of *Krylov subspace methods*. So, the approximate solution after the k-th iteration $\mathbf{x}^{(k)}$ lies in the Krylov subspace

$$\mathcal{K}^k(\boldsymbol{A}_\sigma, \mathbf{r}^{(0)}) = \operatorname{span}\{\mathbf{r}^{(0)}, \boldsymbol{A}_\sigma \mathbf{r}^{(0)}, \ldots, \boldsymbol{A}_\sigma^{k-1} \mathbf{r}^{(0)}\},$$

with $\mathbf{r}^{(0)} = \mathbf{b} - \boldsymbol{A}_\sigma \mathbf{x}^{(0)}$, where $\mathbf{x}^{(0)}$ is the initial guess.

Method	*Mult*	*Prec*	*Daxpy*	*Ddot*	*Storage*
CG	1	1	3	2	$4n$
SYMMLQ	1	1	9	2	$5n$
MINRES	1	1	3	3	$7n$
QMRS	1	1	5	2	$6n$
CGS	2	2	7	3	$8n$

Table 6.1: *Computational cost per iteration*
The table shows the number of matrix-vector multiplications, the number of preconditioner invocations, the number of vector updates, the number of inner products per iteration and the memory requirements for each iterative method. The storage space is given in the number of floating-point numbers to be stored. The solution vector $\mathbf{x}$ and the right-hand side $\mathbf{b}$ are not counted.

Tab. 6.1 summarises the computational costs per iteration and also the memory requirements for the various iterative methods.

6.1 CG

The Conjugate Gradients method by Hestenes and Stiefel [39] is over 50 years old, but still the method of choice for symmetric positive definite linear systems.

The CG method derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors. These vectors are the residuals of the iterates. They are also the gradients of a quadratic functional, the minimisation of which is equivalent to solving the linear system. CG is an extremely effective method when the coefficient matrix is symmetric positive definite, since storage for only a limited number of vectors is required (cf. Tab. 6.1).

The CG method implements the *Ritz-Galerkin approach* which requires, that the residuals are orthogonal to the Krylov subspace, thus $\mathbf{b} - \boldsymbol{A}_\sigma \mathbf{x}^{(k)} \perp \mathcal{K}^k(\boldsymbol{A}_\sigma, \mathbf{r}^{(0)})$.

We use the CG method for Problem I and also for Problem II if node finite elements are used, because in this cases the system matrix is positive definite. In the indefinite case (Problem II with vector finite elements) CG can only be employed if the method can operate in the positive definite subspace, as it is the case for the DIRPROJ, the BESPALOV and the AD method (cf. Sections 4.2.1, 4.2.4, 4.2.3)

However, the CG method can not be used together with JDSYM. Due to the variable shift σ, it is not guaranteed that the system matrix is positive definite. Thus for JDSYM, one of the other methods below has to be used.

6.2 MINRES and SYMMLQ

MINRES and SYMMLQ by Paige and Saunders [56] are computational alternatives for CG for coefficient matrices that are symmetric but possibly indefinite. The preconditioner is still required to be positive definite.

MINRES is a *minimum residual approach*, that requires $\|\mathbf{b} - \boldsymbol{A}_\sigma \mathbf{x}^{(k)}\|_2$ to be minimal over $\mathcal{K}^k(\boldsymbol{A}_\sigma, \mathbf{r}^{(0)})$. On the other hand SYMMLQ is a *minimum error approach* that minimises $\|\mathbf{x} - \mathbf{x}^{(k)}\|_2$ over $\boldsymbol{A}_\sigma \mathcal{K}^k(\boldsymbol{A}_\sigma, \mathbf{r}^{(0)})$. SYMMLQ generates the same iterates as CG, if the coefficient matrix is symmetric positive definite.

We employ MINRES or SYMMLQ with JDSYM, but only in the cases we are sure, that the preconditioner is positive definite, or when we operate in a subspace in which the preconditioner is positive definite (DIRPROJ, BESPALOV and AD methods).

We obtain slightly better results with MINRES, because the cost per iteration is smaller (cf. Tab. 6.1), and because for SYMMLQ the factor $\boldsymbol{A}_\sigma$ in front of the Krylov subspace may lead to a delay in convergence proportional to the condition number of $\boldsymbol{A}_\sigma$ [69].

6.3 QMRS

QMRS (Quasi Minimal Residual Simplified) is a variation of the QMR-method by Freund and Nachtigal [28] for $\boldsymbol{J}$-symmetric matrices[21]. As shown in [30] a linear system with $\boldsymbol{J}$-symmetric coefficient matrix can be constructed from the symmetric linear system (6.1) preconditioned by symmetric *indefinite* matrix. Thus, the QMRS method solves linear systems with symmetric indefinite matrix and symmetric indefinite preconditioner.

The QMR and the QMRS methods follow the *Petrov-Galerkin approach*, which requires, that the residual $\mathbf{b} - \boldsymbol{A}_\sigma \mathbf{x}^{(k)}$ is orthogonal to some suitable k-dimensional subspace other than $\mathcal{K}^k(\boldsymbol{A}_\sigma, \mathbf{r}^{(0)})$. In fact, for QMR-type methods the residual is orthogonal to $\mathcal{K}^k(\boldsymbol{A}_\sigma^T, \mathbf{r}^{(0)})$.

We use the QMRS method for the indefinite systems with a possibly indefinite preconditioner arising from SAUG and EIGSOLV methods (cf. Section 4.2).

6.4 CGS

The Conjugate Gradient Squared method (CGS) by Sonneveld [70] is a variant of Biconjugate Gradient method (BiCG). The BiCG method generates two CG-like sequences of vectors, one

[21] Let $\boldsymbol{J}$ be a nonsingular matrix. The matrix $\boldsymbol{B}$ is called $\boldsymbol{J}$-symmetric, if $\boldsymbol{B}^T\boldsymbol{J} = \boldsymbol{J}\boldsymbol{B}$.

based on a system with the original coefficient matrix $\boldsymbol{A}$, and one on $\boldsymbol{A}^T$. Instead of orthogonalising each sequence, they are made mutually orthogonal, or "bi-orthogonal". Like QMR, also BiCG follows the *Petrov-Galerkin approach.*

CGS applies the updating operations for the $\boldsymbol{A}$-sequence and the $\boldsymbol{A}^T$-sequences both to the same vectors. Ideally, this would double the convergence rate, but in practice convergence may be more irregular than for BiCG, which may sometimes lead to unreliable results. A practical advantage is, that CGS does not need the multiplications with the transpose of the coefficient matrix. CGS solves linear systems with a *general non-symmetric* coefficient matrix.

We use the CGS method for the non-symmetric form of the correction Equation (5.31) of JDSYM. For our symmetric problems CGS proved to somewhat less efficient than the methods discussed before.

While the choice of methods mentioned in the sections above is more or less evident due to the limited number of efficient methods available, this is not really the case with CGS. CGS is one of many methods suited for general non-symmetric linear systems. Numerical experiments with Matlab have shown that for our non-symmetric problems CGS was superior to BiCG, Bi-CGSTAB, QMR and GMRES. Fokkema et al. made similar observations on the usefulness of CGS in the context of Newton processes [24].

7 Matrix-vector products

The sparse matrix-vector product is an important computational kernel that runs ineffectively on many computers with super-scalar RISC processors. In this chapter we analyse the performance of the sparse matrix-vector product with symmetric matrices originating from the FEM and describe three techniques that lead to a fast implementation. We also show how these optimisations can be incorporated into an efficient parallel implementation using message-passing. The benefit of these optimisations is documented by experimental results on six different machines. The work presented in this chapter is published in [34].

Tab. 7.1 shows the percentage of time spent in the sparse matrix-vector multiplication, depending on the preconditioner and the method for solving the eigenvalue problem (cf. Section 4). The numbers shown in Tab. 7.1 are averages, that were obtained from experiments with several different grids.

Method	*None*	*Diag*	*SSOR*	*2LevJACjac*	*2LevSGSssor*
SI	77.3%	78.4%	43.6%	49.9%	29.8%
SAUG	67.4%	62.6%	43.2%	32.0%	20.5%
AD	41.6%	40.4%	29.5%	26.0%	20.8%
EIGSOLV	74.6%	74.8%	44.8%	39.5%	22.9%

Table 7.1: *Percentage of time spent in the sparse matrix-vector multiplication*
Percentage of time spent in the sparse matrix-vector multiplication with respect to the total time spent for solving the matrix eigenvalue problem. The percentage is given for various different preconditioners and various different methods for solving the positive-definite and the indefinite matrix eigenvalue problem. The values shown were obtained using a straight-forward implementation of the matrix-vector multiplication (cf. Alg. 7.1). The numbers were obtained using JDSYM and represent averages over several grids.

The data in Tab. 7.1 shows that a substantial amount of time is spent in the sparse matrix-vector multiplication routine. Using no preconditioner or the diagonal preconditioner leads to a lot of inner iterations. Thus, the ratios in Tab. 7.1 are highest for these two preconditioners. Even if the more efficient SSOR preconditioner is used (cf. Section 8.4) usually about 45% of the computation time is consumed by multiplying sparse matrices with dense vectors. So, from this point of view, it is worthwhile to spend some effort into the optimisation of this computational kernel.

7.1 Performance analysis of the sparse matrix-vector product

In this paper we focus on large symmetric sparse matrices, that do not fit into the memory cache. While our ideas can be applied to general sparse matrices, we present algorithms and results for matrices stored in *symmetric sparse skyline format* (SSS). We use the SSS format, that is described in detail in Appendix A.3 on page 150, for symmetric global finite element matrices, like e.g. $\boldsymbol{A}$ or $\boldsymbol{M}$. For such matrices, the SSS format is much better suited than the CSR or the CSC format, since it requires only about half the memory and the matrix-vector product is also much faster.

Algorithm 7.1 shows a straight-forward matrix-vector multiplication code $\mathbf{y} := \boldsymbol{A}\mathbf{x}$ for a matrix stored in SSS format. The accesses to the matrix data structure are in a stride-1 loop, the access pattern on $\mathbf{x}$ and $\mathbf{y}$ is irregular and depends on the sparsity structure of $\boldsymbol{A}$.

```
for (i = 0; i < n; i ++) {          /* loop over rows of lower triangle */
  xi = x[i];                        /* load x[i] */
  s = 0.0;
  k2 = ia[i+1];
  for (k = ia[i]; k < k2; k ++) {   /* loop over nonzero elems of row */
    j = ja[k];                      /* load column index j */
    v = va[k];                      /* load matrix element A[i,j] */
    s = s + v*x[j];                 /* s = s + A[i,j] * x[j] */
    y[j] = y[j] + v*xi;             /* y[j] = y[j] + A[j,i] * x[i] */
  }
  y[i] = da[i]*xi + s;              /* y[i] = A[i,i] * x[i] + s */
}
```

Algorithm 7.1: *matrix-vector multiplication code for matrices stored in SSS format*
This C code is a straight-forward implementation of the matrix-vector multiplication for sparse symmetric matrices stores in SSS format.

To compute an upper bound for the performance of the sparse matrix-vector product for a given architecture, profound knowledge of the design of the processor and the memory subsystem is required.

In [76] Wadleigh and Potler show how to compute the optimal out-of-cache performance of the BLAS-1 `daxpy()` routine for the HP PA-8500 architecture. For this computation they need to know the cache line size, the cache miss[22] penalty and how many outstanding memory requests the processor supports. Computing the optimal out-of-cache performance of the sparse matrix-vector product in this manner is much more difficult as its performance also depends on the sparsity pattern of the matrix.

Often the details, that need to be known for such a computation, are not published or difficult to access. Our straight-forward approach is more portable and gives comparable results:

We compute the ratio η of the number of floating point operations to the number of bytes of memory traffic. For the best case scenario where $\mathbf{x}$ and $\mathbf{y}$ are read only once from memory and then kept in-cache, the number of floating-point operations (flops) is $4n_{\text{nz}} + 2n$ and the amount of data transferred from and to memory is $12n_{\text{nz}} + 28n$ bytes[23]. For this approximation we assume a very large write-back cache[24] and double precision arithmetic. This yields

$$\eta = \frac{4n_{\text{nz}} + 2n}{12n_{\text{nz}} + 28n}.$$

For the test matrices *cav1* and *cav2* (see Appendix E), that are used in this chapter for numerical experiments, we get $\eta \approx 0.32$.

If we multiply η by the memory bandwidth of the system we get an upper bound of the performance (Mflop rate) of the sparse matrix-vector product.

[22] A cache miss is a failure to find requested data in the cache memory; this means the slower memory must be searched.

[23] $12n_{\text{nz}} + 12n$ bytes for reading the matrix data, $8n$ bytes for reading $\mathbf{x}$ and $8n$ bytes for writing $\mathbf{y}$

[24] The write-back cache is a caching method in which modifications to data in the cache aren't copied to the cache source until absolutely necessary. Data modifications (e.g., write operations) to data stored in the L1 cache aren't copied to main memory until absolutely necessary. In contrast, a write-through cache performs all write operations in parallel – data is written to main memory and the L1 cache simultaneously. Write-back caching yields somewhat better performance than write-through caching because it reduces the number of write operations to main memory.

System	Measured Bandwidth Mbytes/s	Max. Perf. Mflops/s	Measured Perf. Mflops/s
Intel Linux PC	261.44	81.10	32.50
Sun Enterprise 3500	248.71	76.52	28.59
DEC Workstation	245.74	75.61	42.66
HP X-Class	558.14	171.74	48.22
HP V-Class	552.14	169.89	75.02
IBM SP2	1165.05	358.49	56.33
Intel Paragon	186.67	57.43	8.04

Table 7.2: *Machines used for the numerical experiments, their measured memory bandwidth β, the predicted maximal performance r_{opt} and the measured performance r_{act} of Alg. 7.1.* $\beta\eta =: r_{\text{opt}} \geq r_{\text{act}}$. The numbers in the last column are measured using the matrix *cav2*. Detailed specifications of the computers used in this experiment are given in Appendix B.

We determine the memory bandwidth by benchmarking highly optimised computational kernels tuned for the given hardware. For this benchmark we use the vendor supplied BLAS `daxpy` and other routines that have a similar ratio of read to write operations as the sparse matrix-vector product. This gives more realistic results than using the peak memory bandwidth reported by the vendor.

From Tab. 7.2 one can observe that the measured performance of the sparse matrix-vector multiplication code is far below the computed optimal performance, which is limited only by the memory bandwidth. The compiler is unable to generate efficient code, mainly because of data dependencies and irregular loops in Alg. 7.1. On the other hand, the assumptions on which this calculation is based, are not wholly realistic, because the accesses on the vectors $\mathbf{x}$ and $\mathbf{y}$ generate additional cache-misses. Especially on the IBM SP2, the cache is too small (128 Kbytes) to keep the vectors $\mathbf{x}$ and $\mathbf{y}$ in the cache.

For all experimental results in this chapter, we used the standard vendor supplied C compilers with all (safe) optimisations turned on. On the Intel Linux PC we used the GNU C compiler for our benchmarks.

7.2 Design of a fast sparse matrix vector product for one processor

We applied three techniques to improve the implementation of Alg. 7.1:

7.2.1 Software pipelining

We reorganise the source code in such a way, that the processor pipelines are better filled and thus the instruction level parallelism is increased. We achieve this by reducing the data-dependencies in the innermost loop iteration of Alg. 7.1 and by loading the data into registers earlier (data prefetching). We refer to this technique by the term *software pipelining*.

Most compilers are unable to perform these optimisations by themselves in a satisfactory way. Often compilers do not have the information necessary to reorder the instructions safely in such a way, that the load instructions are issued earlier. They end up generating conservative code.

Another reason for the bad measured Mflop rates in Tab. 7.2 is the compiler's inability to unroll more complex loops. E.g., revisiting Alg. 7.1, none of the compilers we used were able

to unroll the inner loop over k.

```
k = ia[0];
for (i = 0; i < n; i ++) {
  /* initialisation and mult with diagonal element */
  xi = x[i]; di = da[i];
  s = 0.0; k2 = ia[i+1]; yi = di*xi;
  if (k < k2) {
    /* first iteration: prefetch data */
    j = ja[k]; v = va[k]; yj = y[j];
    k ++;
    while (k < k2) {
      /* prefetch j and v for next iteration */
      j_ = ja[k]; v_ = va[k];
      /* calc using prefetched data */
      s += v*x[j];
      y[j] = yj + v*xi;
      /* prefetch y for next iteration */
      yj = y[j_];
      /* "rename" prefetched data */
      j = j_; v = v_;
      k ++;
    }
    /* last iteration: no prefetch */
    s += v*x[j];
    y[j] = yj + v*xi;
  }
  y[i] = yi + s;
}
```

Algorithm 7.2: *Optimised matrix-vector multiplication routine for sparse matrices stored in SSS format*
Variant of Alg.7.1, optimised by hand.

Alg. 7.2 is an optimised version of Alg. 7.1. Here all data is loaded from memory one loop iteration before it is actually needed. Thus the processor can better overlap computation and memory transfers. The time the processor is waiting for outstanding data is reduced.

7.2.2 Register blocking

The performance of the sparse matrix-vector multiplication routine is achieved by *register blocking*:

We split up the matrix $\boldsymbol{A}$ into a sum of m matrices

$$\boldsymbol{A} = \boldsymbol{A}_1 + \cdots + \boldsymbol{A}_m.$$

Each matrix $\boldsymbol{A}_i$ consists of small dense blocks of a *fixed size* [71]. Fig. 7.1 shows a simple example of such a matrix splitting.

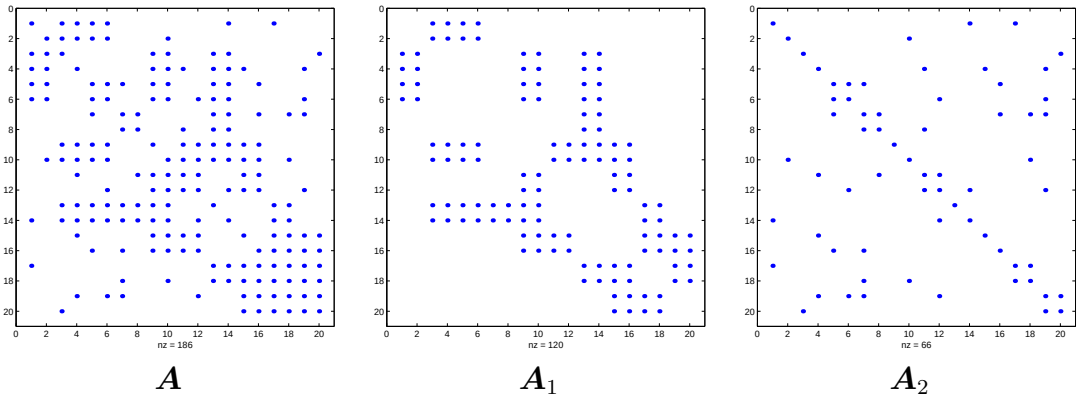


Figure 7.1: *Example for register blocking*
Splitting of the matrix $\boldsymbol{A}$ into a sum of two matrices $\boldsymbol{A} = \boldsymbol{A}_1 + \boldsymbol{A}_2$. $\boldsymbol{A}_1$ consist of dense 2×2-blocks, while $\boldsymbol{A}_2$ holds the remaining non-zero entries.

The storage format of the blocked matrices is equivalent to the SSS format, with the exception, that for each column index a whole dense block is stored instead of just one non-zero entry. This block oriented storage format has the following advantages:

Reduction of memory accesses When a matrix-vector multiplication with such a blocked matrix $\boldsymbol{A}_i$ is performed, fewer indices j have to be loaded from memory, because only one is needed per block.

Reuse of data When multiplying with a dense block, the needed (consecutive) vector elements of $\mathbf{x}$ and $\mathbf{y}$ can be loaded into the registers once and then be reused several times; thus, the name "register blocking".

Choice of block sizes In our approach we store at least two matrices: one contains the small dense blocks of equal size and the other contains the remaining nonzero elements[25].

We ran our experiments with 3×3, 3×1, 1×3, 2×2, 2×1, and 1×2 blocks. The remaining entries are stored unblocked.

On the one hand, reasonable block-sizes depend on the number of floating-point registers. A complete dense block must fit in the registers. If register blocking is used together with software pipelining (cf. Section 7.2.1), then two complete blocks must fit into the registers.

To choose a reasonable block size, also the matrix has to be considered: It only makes sense to use a certain block size, if the original matrix, contains many dense blocks of this size. When using the finite element method, this is often the case: When more than one degree of freedom is associated with the grid points, edges or faces of the mesh, the dense blocks emerge naturally, if an appropriate numbering of the DOFs is used.

If e.g. Problem II is discretised using node elements (cf. Section 3.2), the global FEM matrices consist of dense 3×3 blocks only, since the three components of the electrical field are stored per grid point. Due to the elimination of certain boundary DOFs, some of the 3×3 blocks are destroyed.

[25] In [41] another approach is presented: the authors store the *whole* matrix in small dense blocks, at the expense of having to store some zero entries explicitly.

Implementation As mentioned above, to store the matrix $\boldsymbol{A}_i$ of small dense blocks we use the same data structure as for the original matrix (SSS format, see Appendix A.3), with the exception that we store a whole block for each coordinate pair (i, j) instead of just one value. We build this data structure using a linear-time greedy algorithm that scans the matrix row by row.

The actual matrix-vector multiplication can be implemented in two ways:

Multiplying matrix-after-matrix Multiply each matrix $\boldsymbol{A}_i$ with vector $\mathbf{x}$ and sum the results.

Multiplying row-after-row Multiply with all $\boldsymbol{A}_i$ at the same time, row-by-row. When using this variant one can optionally store the nonzero elements in the same sequence as they are accessed, i. e. store dense blocks of *different* size in each row instead of storing the matrices $\boldsymbol{A}_i$ separately.

We implemented all above variants. None of them was clearly superior. The optimal routine has to be chosen depending on the matrix and the machine.

7.2.3 Matrix Reordering

As a third optimisation, we permute the matrix $\boldsymbol{A}$ in various different ways, to speed up the matrix-vector multiplication,

Cuthill-McKee reordering [33] is a heuristic algorithm to symmetrically permute the rows and columns of a matrix, in order to reduce its bandwidth:

$$\boldsymbol{A}_{\text{rcm}} = \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^T. \tag{7.1}$$

In addition, we investigate the so-called *reverse Cuthill-McKee reordering*, a slightly altered variant of Cuthill-McKee reordering.

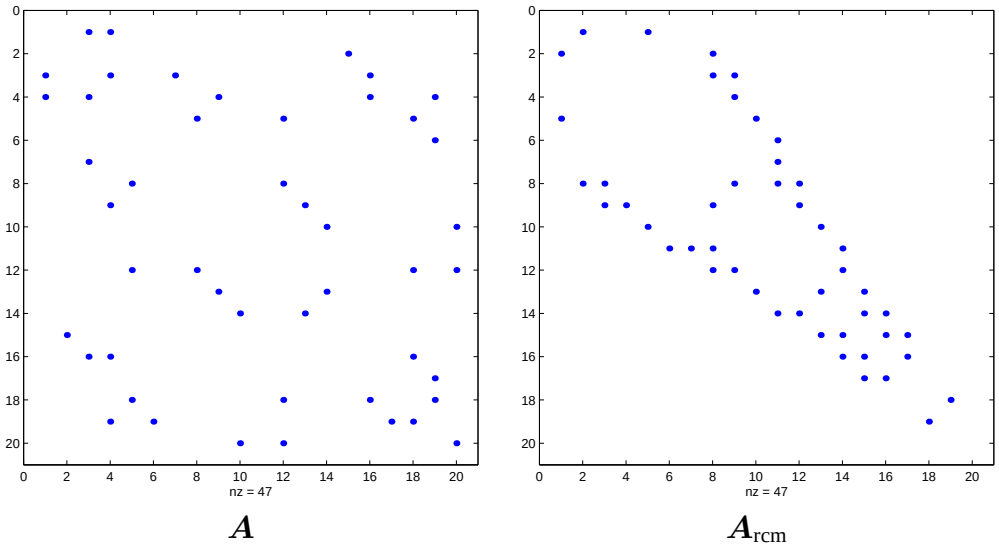


Figure 7.2: *Example for Cuthill-McKee reordering*
Original matrix $\boldsymbol{A}$ and Cuthill-McKee reordered matrix $\boldsymbol{A}_{\text{rcm}}$ with noticeably reduced bandwidth.

We expect the following consequences:

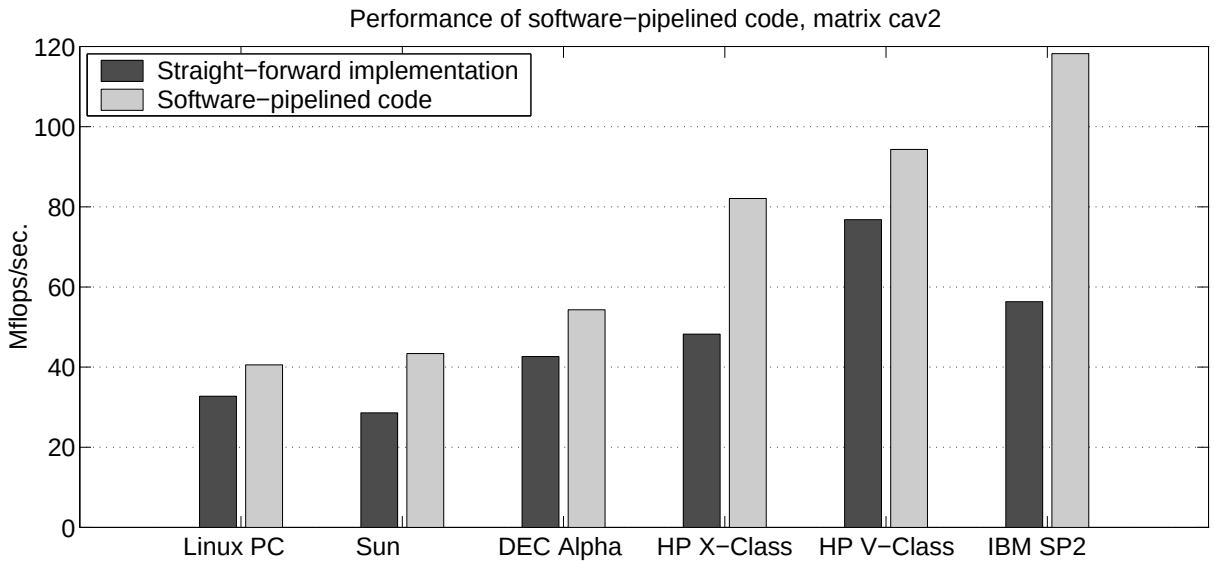


Figure 7.3: *Mflop rate of the software-pipelined and the unoptimised code*

Reduction of cache misses Because of the smaller bandwidth it is more likely that during matrix-vector multiplication vector elements that are accessed in a particular matrix row will be accessed again in the following row. Thus matrix reordering can reduce cache misses that accesses to $\mathbf{x}$ and $\mathbf{y}$ generate [71].

Construction of dense blocks The reordering alters the non-zero structure of the matrix. If the original matrix only has few dense blocks, the reordering may actually increase the number of dense blocks, and make register blocking (see Section 7.2.2) more efficient.

Parallel efficiency The matrix bandwidth has a direct influence on the number of messages, that have to be sent and received in a message-passing parallel implementation of the sparse matrix vector product. Thus, the reduced matrix bandwidth increases parallel efficiency of the sparse matrix vector multiplication.

7.2.4 Experimental results

For the numerical experiments we use the matrices *cav1* and *cav2* stemming from the grids *boxcav20x13x3* and *boxcav30x20x4* (see Appendices D and E). These matrices originate from Problem II, discretised using quadratic node elements. The matrices have a large amount of small dense blocks, because three degrees of freedom are located at each grid point of the FEM mesh. The exact amount of dense blocks are listed in the legend of Fig. 7.4.

For the experiments presented in this section (Figs. 7.3-7.6) we only show the results for matrix *cav2*. The results for the smaller matrix *cav1* are similar throughout.

The experiments are carried out on seven different machines as listed in Tab. 7.2. More detailed information on these machines is found in Appendix B. Even though some of these are parallel machines, the experiments of this subsection were all carried out on *one* processor. The experiments are organised as follows: First we benchmark the three optimisations separately (Figs. 7.3-7.5), then we measure the best performance by applying all optimisations at once (Fig. 7.6).

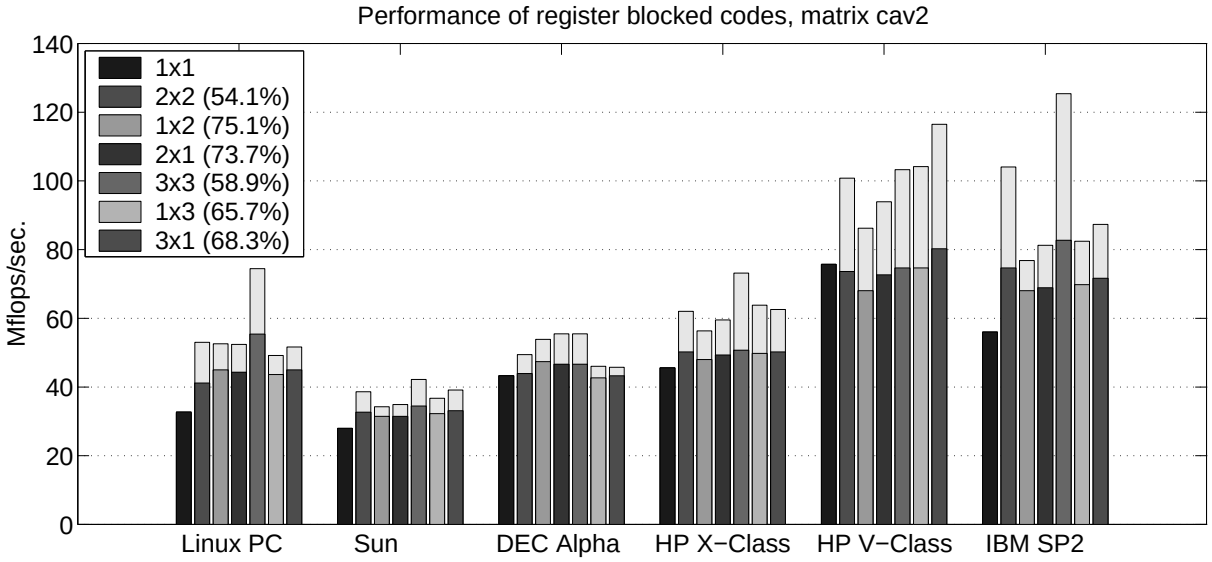


Figure 7.4: *Performance of the codes that use register blocking.*
The darker bars (bottom) show the overall performance of the code, including the blocked portion and the unblocked portion. The whole bars (including top parts in light gray) represent the performance of the code portion that multiplies the blocked part of the matrix only. The number in brackets is the percentage of nonzero elements that are stored in dense blocks of the given size

Fig. 7.3 shows the performance of software-pipelined code in comparison with the original code from Alg. 7.1. The benefit is substantial on all platforms. The improvement ranges from 24% on the Intel Linux PC to 110% on the IBM SP2.

Fig. 7.4 shows the impact of the block size on the performance of the register-blocked code. The maximal improvement is 69% on the Intel Linux PC. On the other platforms the improvement lies between 6% and 48%. As can be seen by the lighter coloured bars in Fig. 7.4 the performance of the code can be substantially higher for matrices consisting solely of small dense blocks.

Fig. 7.5 shows the performance of the unoptimised code when multiplying matrices with different orderings. Compared with the original ordering the performance cannot be increased substantially with Cuthill-McKee-type reorderings. However, the experiments with random ordering indicate that the performance depends heavily on the matrix ordering. In cases, where the original ordering is not so well suited for matrix-vector multiplication as in our case, the improvement of Cuthill-McKee-type reorderings is more substantial [71].

For each platform we choose the fastest code that takes all discussed optimisations into account and compare it with the corresponding unoptimised version. The results are shown in Fig. 7.6 and Tab. 7.3. On the SP2 we achieve an overall improvement of 151%, on the Intel Linux PC we still get an improvement of 97%, while on the HP X-Class, Sun Enterprise Server, HP V-Class and DEC Alpha Workstation we get performance increases of 80%, 73%, 51% and 43%. The results in Tab. 7.3 show that the performance of the best code is quite close to the predicted maximal performance (cf. Section 7.1) on some machines. On the Intel Linux PC and the DEC Workstation we reach 80% of the predicted maximal performance.

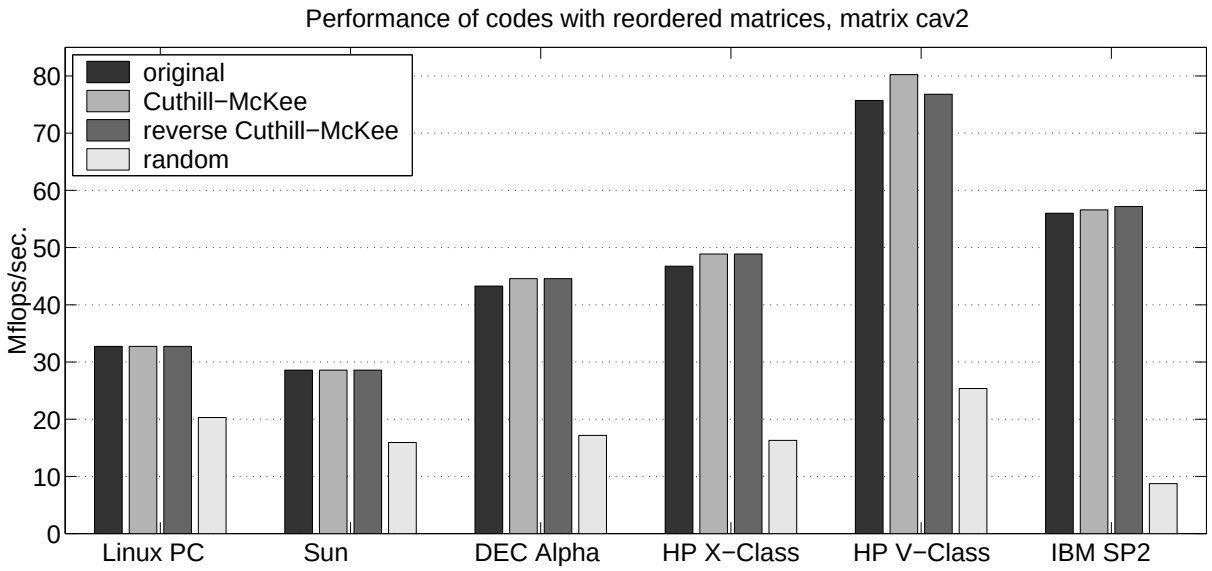


Figure 7.5: *Performance of the codes when working on matrices with different orderings.*

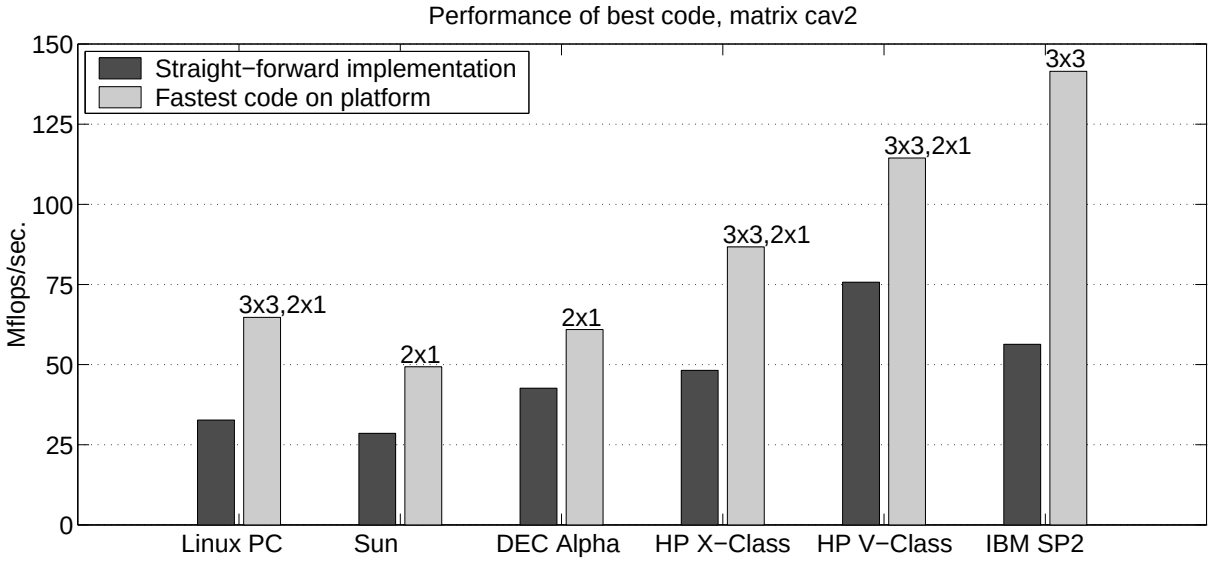


Figure 7.6: *Performance of the best code and the unoptimised code.*
For each architecture we use the code that performs best and compare it against the unoptimised code. The labels on top of the bars show the block sizes that yield the best performance for register blocking. (E.g. "3x3, 2x1" means that the matrix is split into three matrices, one containing all 3x3-blocks, the next containing the remaining 2x1-blocks and the third containing the still remaining elements.)

System	**Maximal Perf.** Mbytes/s	**Unoptimised Perf.** Mflops/s	**Optimised Perf.** Mflops/s
Intel Linux PC	81.10	32.50	64.73
Sun Enterprise 3500	76.52	28.59	49.33
DEC Workstation	75.61	42.66	60.95
HP X-Class	171.74	48.22	86.71
HP V-Class	169.89	75.02	114.40
IBM SP2	358.49	56.33	141.48

Table 7.3: *The predicted maximal performance and the performance of the unoptimised and the best code.*
The predicted maximal performance and the performance of the unoptimised code are explained in Tab. 7.2.

7.3 Parallel matrix-vector multiplication

This section deals with the parallelisation of the matrix-vector multiplication using the message-passing programming paradigm. We chose the MPI (Message Passing Interface) library to implement the communication between the processors.

7.3.1 Parallel implementation

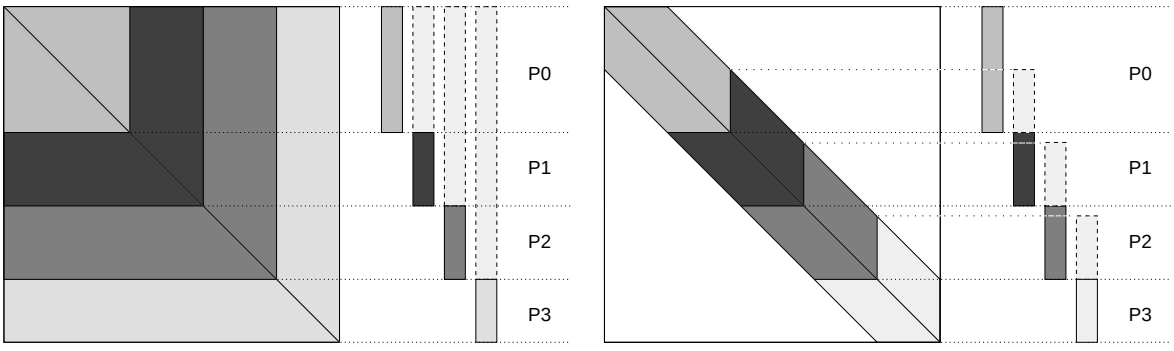
For the parallel implementation we distribute the lower triangular part of the matrix $\boldsymbol{A}$ stored in SSS format by block-rows (see Fig. 7.7). To balance the load we assign the same number of nonzeros to each processor. The distribution of the vectors $\mathbf{x}$ and $\mathbf{y}$ corresponds to the distribution of the matrix rows.

In a preprocessing step each processor collects the necessary information for the actual matrix-vector multiplication. This is done in the following way:

- ▷ The SSS storage format of the matrix implies that processor i needs only elements of the local portions of the $\mathbf{x}$-vector from processor j where $j < i$. For this purpose, the smallest block containing all the needed elements is determined in a preprocessing step. This information is exchanged.
- ▷ During the actual matrix-vector multiplication processor i receives the corresponding portion of the $\mathbf{y}$-vector from processor j. The same portion of the $\mathbf{x}$-vector is sent to processor j. This is due to the symmetry of the matrix.

For the actual parallel matrix-vector code we implemented three slightly different routines:

1. *Without latency-hiding*: Communicate parts of $\mathbf{x}$-vector, then multiply with local part of matrix, then communicate parts of $\mathbf{y}$-vector and form resulting vector.
2. *With latency hiding (upper before lower triangle)*: Communicate parts of $\mathbf{x}$-vector and at the same time multiply with local block-column in the upper triangle. Send the $\mathbf{y}$-vector to the other processors. Upon arrival of the remote parts of the $\mathbf{x}$-vector the local block-row in the lower triangle can be multiplied. Upon arrival of the $\mathbf{y}$-vectors form resulting vector.



(a) (b)

Figure 7.7: *Data distribution for parallel implementation*
The figure shows how the matrix is distributed across the processors for (a) a non-banded and (b) a banded matrix. Because the matrix is symmetric, only its lower triangle is stored. The vectors depicted on the right show the local portions of the $\mathbf{x}$-vector and which portions of the $\mathbf{x}$-vector must be known to each processor for the multiplication of the local portions of the matrix.

3. *With latency hiding (diagonal block first)*: Communicate parts of $\mathbf{x}$-vector and at the same time multiply with local diagonal block of the matrix [61]. Upon arrival of the remote parts of the $\mathbf{x}$-vector, multiply with the remaining local part of the matrix, then communicate parts of $\mathbf{y}$-vector and form resulting vector.

Routine 1 is a reasonable choice for machines that do not support latency-hiding. Routine 2 has the disadvantage that it does not exploit the symmetry of the matrix well, since the matrix has to be read from memory twice. Routine 3 has the disadvantage that the diagonal block of the matrix has to be stored separately to make the implementation efficient. This has to be done in a preprocessing step.

The parallel algorithm benefits from the matrix reordering done for the optimisation of the serial code (cf. Section 7.2.3). As can be seen from Fig. 7.7 the number of messages is reduced because of the smaller matrix bandwidth. Fig. 7.7a shows the worst case: for the multiplication the last processor (P3) needs parts of the vector $\mathbf{x}$ stored on all other processors. Whereas in Fig. 7.7b the same processor needs only the local parts of the neighbour processor. The results in Fig. 7.13 show how crucial the matrix reordering is.

7.3.2 Parallel Numerical Experiments

We carried out the parallel experiments on five platforms: the HP Exemplar X-Class and the HP Exemplar V-Class systems, the Intel Paragon, a Linux workstation cluster and the IBM SP2. The HP Exemplar X-Class and the HP Exemplar V-Class systems are both shared memory machines with 32 processors and a crossbar-switch interconnection network. The Intel Paragon has 150 compute nodes with distributed memory arranged in a 2D-grid. The Intel Beowulf Cluster consists of 251 dual CPU Pentium III processors. These 251 computing nodes are grouped into frames of 24 nodes. An Ethernet network connects the compute nodes. The IBM SP2 is a distributed memory machine with 64 processors connected through a multistage network. More detailed specifications of these machines are given in Appendix B.

The software-pipelining optimisation described in Section 7.2.1 was incorporated into the parallel version. Although it would be entirely possible to implement register blocking also for the parallel version, we did not do that. Unless otherwise mentioned the matrices are reordered using the reverse Cuthill-McKee algorithm.

In Figs. 7.8-7.13 the speedups and Mflop-rates are reported for the five platforms described above. The relation between the speedup and the Mflop-rate is linear and is given through the formula $\text{MflopRate}(p) = \text{Speedup}(p) \times \text{MflopRate}(1)$. The speedup curves given in Figs. 7.8-7.13 are calculated relative to the sequential version, optimised using software-pipelining. This explains, why the speedup shown for one processor may be less than one. Fig. 7.8 shows the measured performance for the HP X-Class. For the smaller matrix *cav1* we get super-linear speedup due to cache-effects. The results for the HP V-Class are shown in Fig. 7.9. On this platform we even get a higher super-linear speedup for matrix *cav1*. For this matrix the speedup is 21 with 8 processors. The speedups for matrix *cav2* are higher compared to the HP X-Class. The speedup for 8 processors is 6.7. Fig. 7.10 shows the measured performance for the Intel Paragon. The code scales well, especially for the matrix *cav2*. Apart from the super-linear speedups on the HP X-Class and the HP V-Class, this machine gives the best speedups, because of its fast network compared to the performance of its processors. The results for the Intel Beowulf Cluster are depicted in Fig. 7.11. For the numerical experiments we used 1 CPU per node. Up to 8 CPU's, the measured speedups for the matrix *cav2* are comparable to the speedups on the HP V-Class. The irregularities above 16 processors show up because in this case some processors have to communicate with processors on another frame. Fig. 7.12 shows

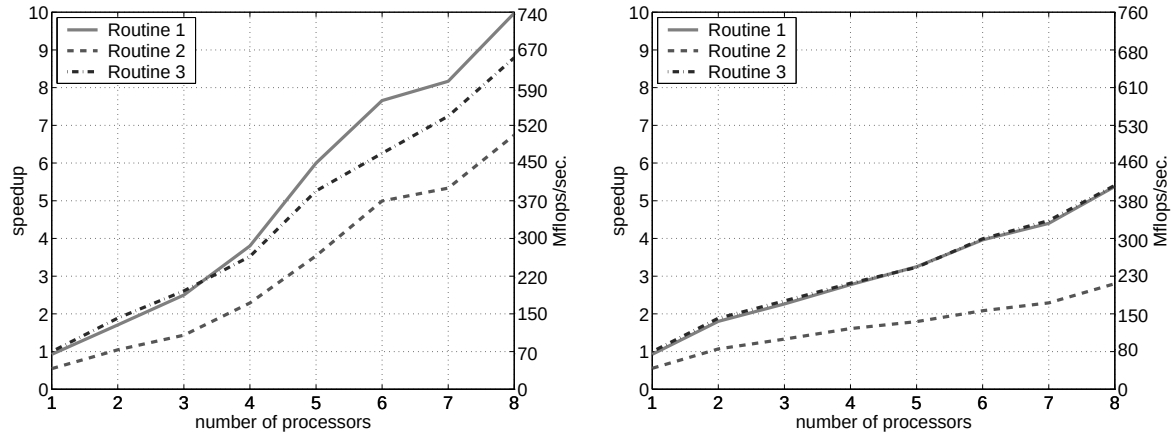


Figure 7.8: *Performance of the parallel matrix-vector multiplication code on the HP X-Class* Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* both using Cuthill-McKee ordering.

the measured performance for the IBM SP2. The code does not scale as well as on the Intel Paragon, because the SP2 has much faster processors and a relatively slow interconnection network.

The results in Figs 7.8-7.12 show that apart from cache-effects the speedups for *cav2* are always better than the speedups for *cav1*. This is due to the higher ratio of computational work compared to the number of messages.

Fig. 7.13 shows the influence of the reordering on the performance. When the matrices are left in their original ordering, the performance is unacceptably low. Even for small numbers of processors, where the number of messages is low, the performance is worse. These results are conducted on the IBM SP2, but this behaviour can also be observed on the other platforms.

7.4 Summary

In this chapter an upper bound for the performance of the sparse matrix-vector product was derived. It was shown, that straight-forward implementations perform poorly. The three techniques presented in this chapter improved the sequential performance by up to 151%. The message-passing implementation benefits from these optimisations and scales reasonably.

It would be worthwhile to extend the work presented in this chapter in order to allow the *automatic generation* of sparse matrix-vector multiplication codes, which are optimised to a given matrix and a given target architecture. The idea is to compute the set of optimal parameters (block size for register blocking, matrix reordering type, degree of loop-unrolling, etc.) which leads to the fastest matrix-vector multiplication code for a given type of sparse matrix on a specific architecture. This can be done by repeated benchmarking of the code for different sets of parameters.

While this approach has been successfully applied to other applications, such as the FFT [31] and the dense BLAS [77], no practical steps in this direction where done in the course of this dissertation. Im [42] presents a toolkit called “Sparsity” for the automatic generation of optimised matrix-vector multiplication codes. “Sparsity” focuses on general sparse matrices and hence does not exploit the symmetry of the matrices. Bik’s thesis [15] describes a “sparse compiler” that takes a dense matrix code as input, along with a sparse matrix, and generates a

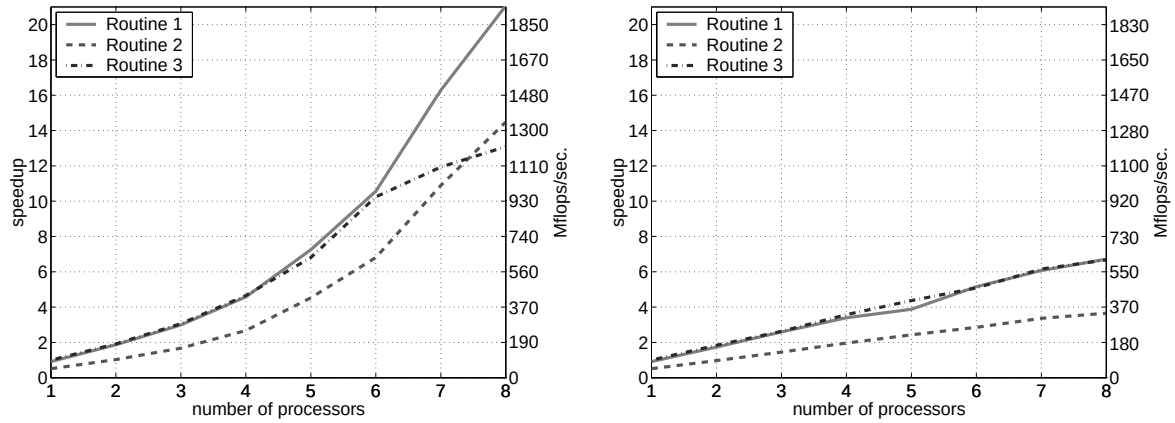
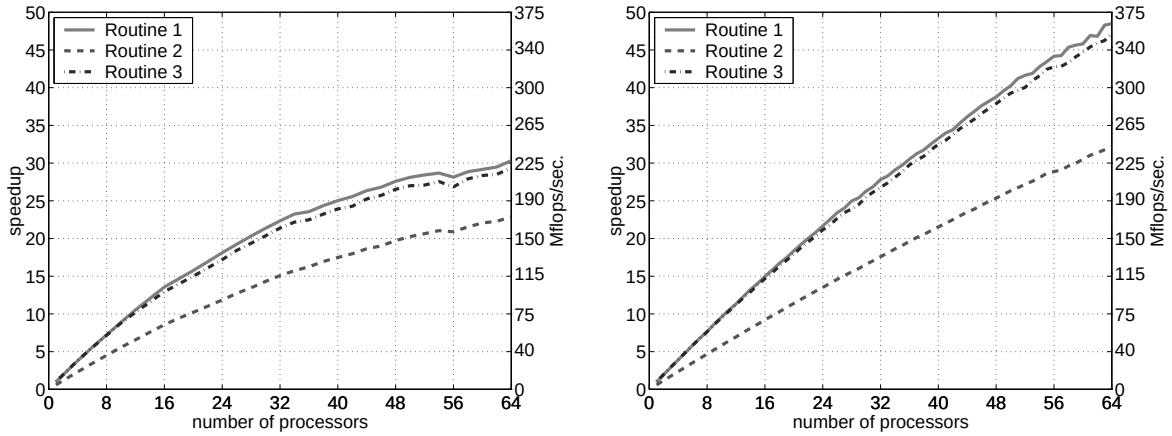


Figure 7.9: *Performance of the parallel matrix-vector multiplication code on the HP V-Class* Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* both using Cuthill-McKee ordering. Note the scaling of the diagrams and the high speedups reached with *cav1*.

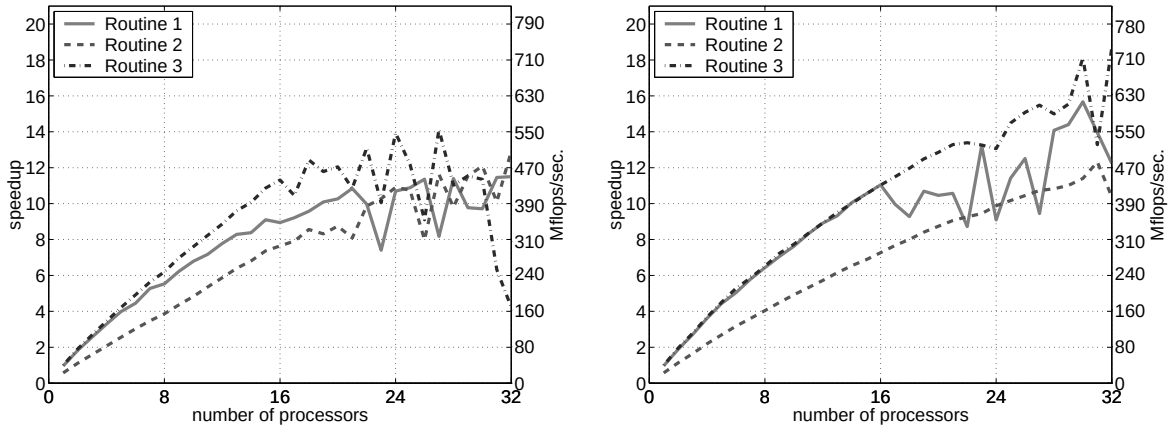
sparse version of the code. The compiler analyses the sparse input matrix, and then transforms the dense matrix code using dependence analysis of data accesses.

Another logical step would be to apply our ideas to preconditioners, that operate on sparse matrices. E.g., the forward- and backward substitution steps performed in the SSOR and the ILUS preconditioners are similar to the matrix-vector multiplication with respect to the structure of the innermost loop. The three optimisation techniques discussed in this chapter could be applied with only little modification.



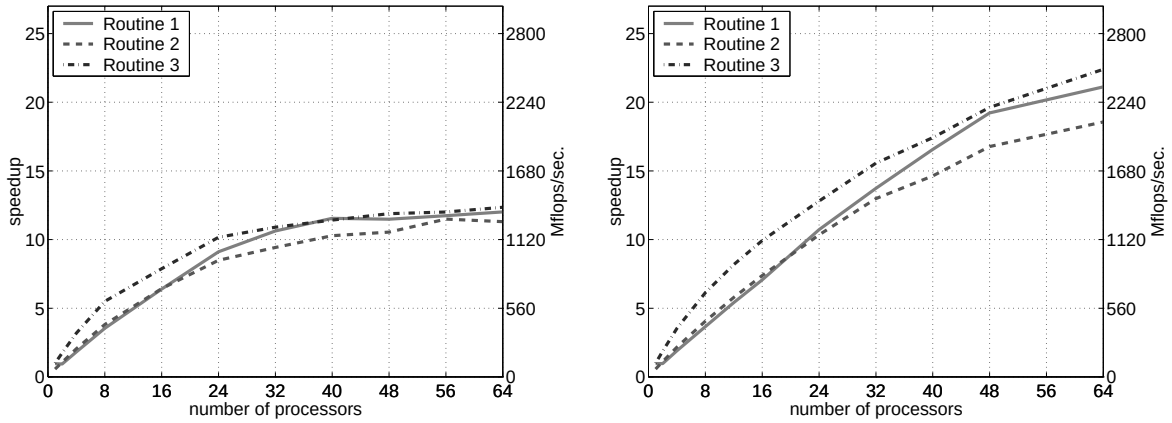
(a) (b)

Figure 7.10: *Performance of the parallel matrix-vector multiplication code on the Intel Paragon.*
Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* both using Cuthill-McKee ordering.



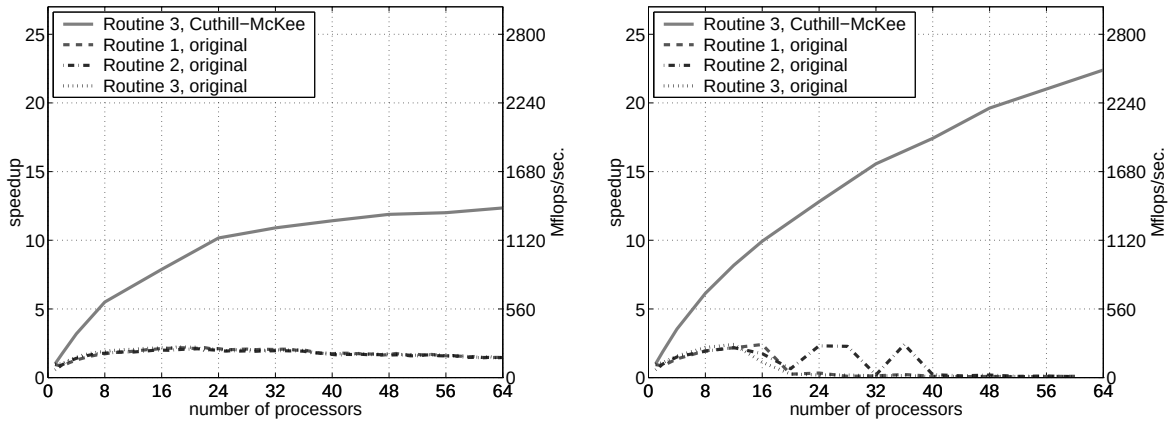
(a) (b)

Figure 7.11: *Performance of the parallel matrix-vector multiplication code on Intel Beowulf Cluster*
Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* both using Cuthill-McKee ordering.



(a) (b)

Figure 7.12: *Performance of the parallel matrix-vector multiplication code on the IBM SP2.* Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* both using Cuthill-McKee ordering.



(a) (b)

Figure 7.13: *Performance of the parallel matrix-vector multiplication code on the IBM SP2 with and without matrix reordering.*
Speedups and Mflop-rates are reported for matrices (a) *cav1* and (b) *cav2* using their original ordering and Cuthill-McKee ordering. For the reordered matrices only Routine 3 is shown, which gives the best results in this case.

8 Preconditioners

Both eigensolvers presented in Chapter 5 use an inner-outer iteration scheme. In the inner iteration a linear system of equations is solved using one of the Krylov subspace methods presented in Chapter 6.

With both JDSYM and ARPACK the cost for solving these linear systems is substantial. If *no* preconditioner is used, the time spent in the inner iteration is 65% – 90% of the time spent for solving the eigenvalue problem. The percentage depends on the grid and on the method for solving the eigenproblem. These numbers clearly show, that the inner iteration is the most time-consuming part of the eigensolver. In the following we discuss the preconditioners we used to speed up the inner iteration.

For both eigensolvers the matrix to precondition has the form $\boldsymbol{A} - \sigma \boldsymbol{M}$. When using ARPACK, the shift σ is constant, whereas with JDSYM the shift can change in each outer iteration. Since it would be too costly to construct a new preconditioner in every outer iteration, we construct a preconditioner for $\boldsymbol{A} - \tau \boldsymbol{M}$ in the case of JDSYM. τ is the target parameter used in the Jacobi-Davidson algorithm.

The goal of preconditioning is to find a matrix $\boldsymbol{K}$ with the following properties:

1. $\boldsymbol{K}$ is a good approximation of the matrix $\boldsymbol{A} - \sigma \boldsymbol{M}$ in some sense.
2. The system $\boldsymbol{K}\mathbf{x} = \mathbf{b}$ can be solved much faster than the original system.
3. The cost for constructing $\boldsymbol{K}$ is smaller than the benefit gained by using the preconditioner.

If $\boldsymbol{K}$ is a good approximation of the system matrix $\boldsymbol{A} - \sigma \boldsymbol{M}$ then the preconditioned matrix will have better spectral properties, and thus the number of steps required in the iterative method will be reduced. Note, that the introduction of a preconditioner introduces additional costs: firstly through its construction and secondly through its application. The use of a preconditioner only pays off if it reduces the number of iteration steps significantly.

The preconditioner presented in this section can be classified by two categories:

So called *black box* preconditioners operate on the system matrix only, without requiring information about the underlying problem. The Jacobi iteration, the Gauss-Seidel iteration and the ILUS preconditioner described below all fall into this category. The Jacobi and the Gauss-Seidel iterations belong to the family of *stationary methods*, which can also be used for preconditioning. Performing one step of the Jacobi iteration is equivalent to diagonal preconditioning.

The ILUS preconditioner [20] constructs an incomplete LDL^T-factorisation. ILUS is tailored to sparse matrices stored in SSS format (cf. Section A.3).

The second class of preconditioners are the *problem specific preconditioners*. They make use of the specific properties of the underlying problem.

The so called *two level hierarchical preconditioner* belongs to this category. This preconditioner exploits the special structure of the global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$, which results from the underlying hierarchical finite element basis.

8.1 Stationary iterative methods

Preconditioners of this type are based on a stationary iterative method, i.e. a method with a constant iteration matrix. Applying such a preconditioner means performing a few steps of the iterative method.

A stationary method can be built by splitting the matrix $\boldsymbol{A} - \sigma\boldsymbol{M}$ according to

$$\boldsymbol{A} - \sigma\boldsymbol{M} = \boldsymbol{S}_1 - \boldsymbol{S}_2, \tag{8.1}$$

where $\boldsymbol{S}_1$ must be non-singular. From $\boldsymbol{S}_1\mathbf{x} = \boldsymbol{S}_2\mathbf{x} + \mathbf{b}$ we get the iteration step

$$\boldsymbol{S}_1\mathbf{x}^{(k+1)} = \boldsymbol{S}_2\mathbf{x}^{(k)} + \mathbf{b}, \tag{8.2}$$

or equivalently

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{S}_1^{-1}\mathbf{r}^{(k)} \tag{8.3}$$

with $\mathbf{r}^{(k)} = \mathbf{b} - (\boldsymbol{A} - \sigma\boldsymbol{M})\mathbf{x}^{(k)}$.

8.1.1 m-step Jacobi preconditioning

The weighted Jacobi iteration is obtained from the matrix splitting (8.1) with $\boldsymbol{S}_1 = 1/\omega\boldsymbol{D}$, where $\boldsymbol{D}$ is the diagonal of $\boldsymbol{A} - \sigma\boldsymbol{M}$. This yields the iteration

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega\boldsymbol{D}^{-1}\mathbf{r}^{(k)}. \tag{8.4}$$

The preconditioning step is made up of applying the iteration step m times. The initial guess $\mathbf{x}^{(0)}$ is set to the zero vector.

The m-step Jacobi preconditioner is parametrised by the weight ω and the number of iteration steps m.

For $\omega = 1$ and $m = 1$ we obtain the simple and cheap *Jacobi* or *diagonal preconditioner*

$$\mathbf{x} = \boldsymbol{D}^{-1}\mathbf{b}. \tag{8.5}$$

8.1.2 m-step SSOR preconditioning

The *Symmetric Successive Over Relaxation* (SSOR) method is obtained from the matrix splitting (8.1) with $\boldsymbol{S}_1 = 1/\omega\boldsymbol{D} + \boldsymbol{L}$. $\boldsymbol{L}$ is the strictly lower triangle of the matrix $\boldsymbol{A} - \sigma\boldsymbol{M}$ and $\boldsymbol{D}$ represents its diagonal.

The first half-step of the SSOR iteration is [60, page 97]

$$\mathbf{x}^{(k+1/2)} = (\boldsymbol{D} + \omega\boldsymbol{L})^{-1}\left([-\omega\boldsymbol{L}^T + (1-\omega)\boldsymbol{D}]\mathbf{x}^{(k)} + \omega\mathbf{b}\right). \tag{8.6}$$

The iteration matrix in (8.6) is not symmetric. However, we can obtain a symmetric method by executing a second half-step (with $\boldsymbol{L}$ and $\boldsymbol{L}^T$ interchanged):

$$\mathbf{x}^{(k+1)} = (\boldsymbol{D} + \omega\boldsymbol{L}^T)^{-1}\left([-\omega\boldsymbol{L} + (1-\omega)\boldsymbol{D}]\mathbf{x}^{(k+1/2)} + \omega\mathbf{b}\right). \tag{8.7}$$

Thus a SSOR iteration consists of a half-step (8.6), followed by a half-step (8.7).

The preconditioning step is made up of applying the SSOR iteration step m times. The initial guess $\mathbf{x}^{(0)}$ is set to the zero vector.

We reduce the computational cost of the m-step SSOR preconditioner by applying the *Conrad-Wallach trick* [55, page 175]. This trick saves the computation of $\boldsymbol{L}^T\mathbf{x}^{(k)}$ in (8.6) and the computation of $\boldsymbol{L}\mathbf{x}^{(k+1/2)}$ in (8.7).

Like the m-step Jacobi preconditioner, also the m-step SSOR preconditioner is parametrised by the weight ω and the number of iteration steps m.

For $m = 1$, the preconditioner becomes [60, page 267]

$$\boldsymbol{K}_{\text{SSOR}} = \omega(2-\omega)(\boldsymbol{D} + \omega\boldsymbol{L})\boldsymbol{D}^{-1}(\boldsymbol{D} + \omega\boldsymbol{L}^T). \tag{8.8}$$

For $m = 1$ and $\omega = 1$, we obtain the *symmetric Gauss-Seidel preconditioner*:

$$\boldsymbol{K}_{\text{SGS}} = (\boldsymbol{D} + \boldsymbol{L})\boldsymbol{D}^{-1}(\boldsymbol{D} + \boldsymbol{L}^T). \tag{8.9}$$

8.2 ILUS preconditioning

The incomplete LU-factorisation belongs to the most popular "black box" preconditioners. The standard incomplete LU methods (ILU(0), ILUT and ILUTP) [60] do not exploit the symmetry of the matrix:

- ▷ These methods traverse the entire system matrix row-wise. Thus the whole system matrix (both the upper and the lower triangle) has to be stored in memory.
- ▷ An incomplete LU-factorisation is being computed. For symmetric matrices, a LDL^T-factorisation would be much more efficient in terms of memory consumption.
- ▷ The preconditioner constructed by these methods is generally non-symmetric, even if the system matrix is symmetric.

The ILUS preconditioner is designed to compute an incomplete factorisation of sparse matrices stored in *Sparse Skyline* (SSK) format [59]. The method can be slightly modified to also support (symmetric) sparse matrices stored in *Symmetric Sparse Skyline* (SSS) format (cf. Appendix A). The ILUS preconditioner presented below employs a two-stage scheme for reducing the fill-in. It also eliminates the disadvantages listed above.

Let $\boldsymbol{A}^\sigma = \boldsymbol{A} - \sigma\boldsymbol{M}$ be the matrix, for which the symmetric ILUS factorisation shall be computed. We define a sequence of matrices

$$\boldsymbol{A}^\sigma_{k+1} = \begin{pmatrix} \boldsymbol{A}^\sigma_k & \mathbf{v}_k \\ \mathbf{v}_k^T & \alpha_k \end{pmatrix}, \tag{8.10}$$

with $\boldsymbol{A}^\sigma_1 = a^\sigma_{1,1}$ and $\boldsymbol{A}^\sigma_n = \boldsymbol{A}^\sigma$. Let the incomplete LDL^T-decomposition of $\boldsymbol{A}^\sigma_k$ computed by ILUS be given by

$$\boldsymbol{A}^\sigma_k = \boldsymbol{L}_k\boldsymbol{D}_k\boldsymbol{L}_k^T. \tag{8.11}$$

The incomplete LDL^T-decomposition of $\boldsymbol{A}^\sigma_{k+1}$ is then computed by

$$\boldsymbol{A}^\sigma_{k+1} = \begin{pmatrix} \boldsymbol{L}_k & \mathbf{0} \\ \mathbf{y}_k^T & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{D}_k & \mathbf{0} \\ \mathbf{0}^T & d_{k+1} \end{pmatrix} \begin{pmatrix} \boldsymbol{L}_k^T & \mathbf{y}_k \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{8.12}$$

with

$$\mathbf{y}_k = \boldsymbol{D}_k^{-1}\boldsymbol{L}_k^{-1}\mathbf{v}_k \tag{8.13}$$

$$d_{k+1} = \alpha_k - \mathbf{y}_k^T\boldsymbol{D}_k^{-1}\mathbf{y}_k. \tag{8.14}$$

Thus the LDL^T-decomposition is computed row-by-row. To compute a new row of $\boldsymbol{L}$, a triangular system has to be solved and an inner product weighted by $\boldsymbol{D}_k$ has to be computed.

The difficulty lies in solving the triangular system $\boldsymbol{L}_k\mathbf{y}_k = \mathbf{v}_k$ efficiently. The straightforward approach is to solve the system exactly by forward substitution and then eliminate elements with small magnitude. With this method, the whole matrix $\boldsymbol{L}_k$ has to be accessed, even if $\mathbf{v}_k$ is sparse. The computational cost for the factorisation is at least $O(n\,\mathrm{nnz}(\boldsymbol{A}^\sigma))$[26], if the non-zero elements are more or less evenly distributed. For large n, the computational cost is unacceptably high.

Saad suggests in [20] to solve the triangular system $\boldsymbol{L}_k\mathbf{y}_k = \mathbf{v}_k$ only approximately by computing a Neumann series

$$\mathbf{y}_k = \boldsymbol{L}_k^{-1}\mathbf{v}_k = (\boldsymbol{I} + \boldsymbol{E}_k + \boldsymbol{E}_k^2 + \cdots + \boldsymbol{E}_k^p)\mathbf{v}_k, \tag{8.15}$$

[26] at least n^2 times the average number of non-zeros entries per row of $\boldsymbol{A}$

with $\boldsymbol{E}_k = \boldsymbol{I} - \boldsymbol{L}_k$. The matrix $\boldsymbol{E}_k$ and its powers do not have to be computed explicitly, since (8.15) can be calculated using a Horner scheme. The advantage of this approach, is that the operations of the form $\mathbf{y} \leftarrow \boldsymbol{E}\mathbf{x}$ are in sparse-sparse mode. When performing such an operation, only parts of the matrix $\boldsymbol{E}$ have to be accessed, i.e. only the columns that are multiplied by the non-zero entries of the sparse vector. Since the entries of $\boldsymbol{E}$, resp. $\boldsymbol{L}$ are stored row-by-row (cf. Appendix A), an additional data structure, that allows to traverse the matrix column-by-column efficiently, is needed. We use n linked lists, where each list links the non-zero entries of one column. Three additional integer arrays are needed to implement this. They almost double the memory consumption for the matrix. The computational cost of this approach depends on the structure of the matrix and is at least $O(n^2\,p)$ and the most $O(n^2\,p^2)$. Since the p is typically small ($p < 10$), this approach is usually faster than the straight-forward approach.

Alternatively, George and Liu's algorithm for building the sparse Cholesky factorisation [33], which is based on the elimination tree of the input matrix, could be adapted to the incomplete case. We did not investigate the idea further.

The number of non-zero elements in the factor $\boldsymbol{L}$ is controlled using the parameters $\varepsilon_{\text{drop}}$ and n_{fill}: For each newly calculated row $\mathbf{y}_k$, only the $\text{nz}_k + n_{\text{fill}}$ entries with the largest magnitude are kept. nz_k is the number of non-zero entries of $\boldsymbol{A}^\sigma(k, 1{:}k-1)$. From the nz_k + n_{fill} entries only those that satisfy the drop tolerance condition

$$|y_i| \geq \varepsilon_{\text{drop}} \|\boldsymbol{A}^\sigma(k, 1{:}k-1)\|_2, \quad 0 < i < k, \tag{8.16}$$

are stored in the incomplete factor.

In [20] Saad suggests, to monitor the stability of the factorisation by checking the value $\|\boldsymbol{L}_k^{-1}\|$. As a cheap to compute approximation he uses $\|\boldsymbol{L}_k^{-1}\mathbf{1}\|_\infty$, where $\mathbf{1}$ is a vector with all entries equal to 1. As soon as this number exceeds a certain given bound, the factorisation is aborted.

In addition, we abort the factorisation, if a diagonal element becomes smaller than the machine precision $\varepsilon_{\text{mach}}$.

The ILUS algorithm for symmetric matrices is given in Matlab notation in Alg. 8.1.

The existence of incomplete factorisation preconditioners like ILUS has been proved for M-matrices. For indefinite matrices their use is potentially dangerous, since small diagonal elements may lead to ill-conditioned factors. It might be more stable to construct a sparse LDL^T-factorisation using Bunch-Kaufmann-Parlett pivoting[19], where D is a diagonal matrix, or possibly block diagonal matrix (with blocks of order 1 and 2). We did not investigate this idea.

Summary The ILUS preconditioner is an incomplete factorisation of the form $\boldsymbol{A}^\sigma \approx \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ which is obviously symmetric. Even if the original matrix $\boldsymbol{A}^\sigma$ is positive-definite, this is not necessarily the case for the preconditioner. The cost of computing the incomplete factorisation for the ILUS preconditioner as suggested in [20] is more than $O(n^2)$. The construction of ILUS preconditioner is much more expensive than the construction of its non-symmetric cousins ILU(0), ILUT and ILUTP, because the latter do not require the solution of a sparse linear system for each row. So, the ILUS preconditioner is certainly not well suited for large matrices.

function [$\boldsymbol{L}$, $\boldsymbol{D}$, r] = ILUS($\boldsymbol{A}$, $\varepsilon_{\text{drop}}$, n_{fill}, *type*, p, r_{lim})
  $n = \text{size}(\boldsymbol{A}, 1)$; $\boldsymbol{L} = 1.0$; $\mathbf{d} = \boldsymbol{A}(1,1)$
  if $1.0 + d_1 = 1.0$, error('diagonal element close to zero'); end
  $\mathbf{u} = 1.0$; $r = 1.0$
  for $i = 2 : n$
    $\mathbf{v} = \boldsymbol{A}(i, 1{:}i-1)$; $s_i = \varepsilon_{\text{drop}} \|\mathbf{v}\|$
    if *type* $= 0$
      $\mathbf{y} = \mathbf{v}/\boldsymbol{L}^T$
    else
      $\mathbf{y} = \mathbf{v}$
      for $k = 1 : p$
        $\mathbf{y} = \mathbf{y} - \mathbf{y}\boldsymbol{L}^T + \mathbf{v}$
      end
    end
    $\mathbf{y} = \mathbf{y}./\mathbf{d}$
    $\mathbf{y} = \mathbf{y}. * (|\mathbf{y}| >= s_i)$
    $[\text{dum}, \text{idx}] = \text{sort}(-|\mathbf{y}|)$
    for $k = \text{nnz}(\mathbf{v}) + n_{\text{fill}} + 1 : \text{length}(\mathbf{y})$
      $y_{\text{idx}(k)} = 0.0$
    end
    $\mathbf{y} = \text{sparse}(\mathbf{y})$
    $\mathbf{d}(i) = a_{i,i} - (\mathbf{y}. * \mathbf{d})\mathbf{y}^T$
    if $1.0 + d_i = 1.0$, error('diagonal element close to zero'); end
    $\boldsymbol{L} = [\boldsymbol{L}, \text{sparse}(i-1, 1); \mathbf{y}, 1.0]$
    $\mathbf{u} = [\mathbf{u}; 1.0 - \mathbf{y}\mathbf{u}]$
    if $|u_i| > r$, $r = |u_i|$; end
    if $r > r_{\text{lim}}$, error('r exceeds limit: factorisation unstable'); end
  end
  $\boldsymbol{D} = \text{spdiags}(\mathbf{d}^T, 0, n, n)$
end

Algorithm 8.1: *ILUS preconditioner*
This algorithm computes the ILUS preconditioner for a given symmetric matrix $\boldsymbol{A}$. The fill-in in the triangular matrix $\boldsymbol{L}$ is controlled by the parameters $\varepsilon_{\text{drop}}$ and n_{fill}. Depending on the parameter *type*, the triangular system is solved either by forward substitution or using a Neumann series of length p. The stability of the factorisation is evaluated using the values $r = \|\boldsymbol{L}^{-1}\mathbf{1}\|_\infty$ and $|d_i|$. The upper bound for r is r_{lim}.

8.3 2-level hierarchical basis preconditioner

The two-level hierarchical basis preconditioner was suggested in [10] by Bank. The preconditioner exploits the hierarchical organisation of the finite element basis functions, as described in Chapters 2 and 3. In our case this simply means, that the basis functions for the quadratic finite elements contain the basis functions of the linear elements.

Using quadratic finite elements and an appropriate numbering of the global DOFs , global matrices with a 2×2-block structure emerge,

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{pmatrix}, \quad \boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}_{11} & \boldsymbol{M}_{12} \\ \boldsymbol{M}_{21} & \boldsymbol{M}_{22} \end{pmatrix}, \tag{8.17}$$

where the matrices $\boldsymbol{A}_{11}$ and $\boldsymbol{M}_{11}$ correspond to the global matrices constructed using linear elements.

The two-level hierarchical basis preconditioner is based on the approximate solution of a linear system of the form

$$\begin{pmatrix} \boldsymbol{A}_{11}^{\sigma} & \boldsymbol{A}_{12}^{\sigma} \\ \boldsymbol{A}_{21}^{\sigma} & \boldsymbol{A}_{22}^{\sigma} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}. \tag{8.18}$$

The idea is the following: The (1,1)-block is solved exactly, which should produce already a good “approximation”. The remaining parts of the matrix, which are considerably larger, are taken into account only approximately.

The various variants of the two-level hierarchical basis preconditioner differ in the underlying block method, and also in the approximation method for the (2,2)-block.

We consider two block methods: The *block-Jacobi method*

$$\mathbf{x}_1 \leftarrow \boldsymbol{A}_{11}^{\sigma\,-1} \mathbf{b}_1 \tag{8.19}$$

$$\mathbf{x}_2 \leftarrow \tilde{\boldsymbol{A}}_{22}^{\sigma\,-1} \mathbf{b}_2 \tag{8.20}$$

and the *symmetric block-Gauss-Seidel method*

$$\mathbf{t}_1 \leftarrow \boldsymbol{A}_{11}^{\sigma\,-1} \mathbf{b}_1 \tag{8.21}$$

$$\mathbf{x}_2 \leftarrow \tilde{\boldsymbol{A}}_{22}^{\sigma\,-1} (\mathbf{b}_2 - \boldsymbol{A}_{21}^{\sigma} \mathbf{t}_1) \tag{8.22}$$

$$\mathbf{x}_1 \leftarrow \boldsymbol{A}_{11}^{\sigma\,-1} (\mathbf{b}_1 - \boldsymbol{A}_{12}^{\sigma} \mathbf{x}_2). \tag{8.23}$$

For the approximation of the (2,2)-block we consider two inner iterations: The *weighted Jacobi iteration* (8.4) and the *SSOR iteration* (8.6) and (8.7). Both iterations are started with an initial guess $\mathbf{x}_0 = \mathbf{0}$, and both iterations also are parameterised by the weight ω and the number of inner iterations m.

For weighted Jacobi iteration and $m = 1$ we get

$$\tilde{\boldsymbol{A}}_{22}^{\sigma} = 1/\omega \; \boldsymbol{D}_{22} \tag{8.24}$$

and SSOR iteration and $m = 1$ we get

$$\tilde{\boldsymbol{A}}_{22}^{\sigma} = (\boldsymbol{D}_{22} - \omega \boldsymbol{L}_{22}) \boldsymbol{D}_{22}^{-1} (\boldsymbol{D}_{22} - \omega \boldsymbol{L}_{22}^{T}). \tag{8.25}$$

Here $\boldsymbol{D}_{22}$ is the diagonal and $\boldsymbol{L}_{22}$ is the strictly lower triangle of $\boldsymbol{A}_{22}$.

The block methods and the inner iterations can be combined arbitrarily. Thus we get four different variants for the two-level hierarchical basis preconditioner:

1. 2LevJACjac(ω, m) is a block-Jacobi method, which uses the weighted Jacobi iteration for approximately solving the (2,2)-block.

2. 2LevJACssor(ω, m) is a block-Jacobi method, which uses the SSOR iteration for approximately solving the (2,2)-block.

3. 2LevSGSjac(ω, m) is a symmetric block-Gauss-Seidel method, which uses the Jacobi iteration for approximately solving the (2,2)-block.

4. 2LevSGSssor(ω, m) is a symmetric block-Gauss-Seidel method, which uses the SSOR iteration for approximately solving the (2,2)-block.

To exactly solve the linear system with the (1,1)-block, we use the SuperLU software package [23]. SuperLU is able to compute an LU-factorisation of a general non-symmetric, sparse matrix using partial pivoting. Thanks to an elimination algorithm which is based on "super nodes" a big portion of the factorisation can be implemented using fast routines for dense matrices (BLAS-2 and BLAS-3 routines). In addition SuperLU is optimised for modern cache-based computers.

As stated above, the SuperLU package computes LU-factorisations of general non-symmetric, sparse matrices. Optimised support for symmetric matrices is documented, but not yet implemented in the current 2.0 release. Therefore we have to use the standard algorithm for non-symmetric matrices for our symmetric matrices, thereby doubling the memory consumption for the factors and slowing down the forward- and backward substitution. Fortunately SuperLU can be configured to switch off the partial pivoting scheme which would be disadvantageous for positive-definite matrices.

8.4 Experimental results

A number of numerical experiments have been carried out to evaluate the performance of the various preconditioners. We compared the preconditioners introduced in this chapter using both the JDSYM and the ARPACK eigensolver.

Four different grids were used for the experiments: *boxcav16x10x3*, *copcav18*, *cop20k* and *cop40k*. The number of tetrahedra in these grids range from 5760 to 38597. Detailed information on these grids is given in Appendix D. The actual size of the resulting matrices depends on the type of finite elements used (nodal or vector). The properties of the resulting matrices are listed in Appendix E.

For all experiments quadratic elements were used, since they turned out to be far superior to the linear ones (cf. Section 3.4).

For the two-level and the SSOR preconditioner we chose the parameters $m = 1$ and $\omega = 1$ (these settings appear to be close to optimal in all cases).

The results for our implementation of the ILUS preconditioner were very disappointing, no matter how we chose the parameters $\varepsilon_{\text{drop}}$, n_{fill}, *type*, p and r_{lim}. Even for the smallest grid, the eigensolver failed to converge within one hour. With the other preconditioners, the eigensolver always converged in a few seconds for this grid. For this reason, no results for the ILUS preconditioner are shown in the following paragraphs.

8.4.1 Positive-definite eigenvalue problem

Tabs. 8.1 and 8.2 show the results of the various preconditioners for the symmetric positive-definite eigenvalue problem (cf. Section 4.1). it_{in} is the average number of inner iterations

per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem.

Grid	*Precon*	it_{in}	it_{out}	t_{tot}	t_{eig}
boxcav16x10x3	None	20.1	38	110.0	101.8
	Diag	8.9	34	59.0	50.7
	SSOR	3.6	31	55.0	46.5
	2LevJACjac	5.1	35	55.0	46.6
	2LevJACssor	4.5	34	64.0	54.8
	2LevSGSjac	3.6	31	61.0	51.6
	2LevSGSssor	**2.5**	25	**53.0**	**44.2**
copcav18	None	28.0	38	179.0	168.8
	Diag	14.2	37	107.0	95.3
	SSOR	4.2	33	**84.0**	**72.9**
	2LevJACjac	5.9	35	92.0	78.8
	2LevJACssor	4.8	35	109.0	95.9
	2LevSGSjac	4.3	33	110.0	96.5
	2LevSGSssor	**2.8**	**31**	98.0	84.8
cop20k	None	18.0	38	452.0	415.0
	Diag	12.5	37	347.0	308.2
	SSOR	4.3	32	279.0	240.2
	2LevJACjac	4.2	35	**265.0**	**219.6**
	2LevJACssor	3.8	34	302.0	256.9
	2LevSGSjac	2.9	32	318.0	270.4
	2LevSGSssor	**2.1**	29	296.0	247.4
cop40k	None	19.2	36	889.0	818.3
	Diag	15.0	36	752.0	677.3
	SSOR	5.2	32	624.0	547.2
	2LevJACjac	4.2	35	**623.0**	**508.6**
	2LevJACssor	4.4	38	813.0	702.3
	2LevSGSjac	3.0	31	709.0	595.7
	2LevSGSssor	**2.3**	30	694.0	583.2

Table 8.1: *Performance of preconditioners for the positive-definite eigenproblem and the JDSYM eigensolver*

By comparing the total execution times in Tabs. 8.1 and 8.2, it becomes evident, that it always pays off to use a preconditioner. The SSOR and two-level preconditioners work very well. Out of the four two-level type preconditioners, 2LevSGSssor reduces the average number of inner iteration steps the most. With JDSYM the cheaper 2LevJACjac variant is best in terms of total execution time (at least for the three larger grids). With ARPACK 2LevSGSssor is always the best choice.

It is remarkable, how well the standard SSOR preconditioner works, compared to the two-level preconditioner. Their performance is comparable in almost all cases (large grids together with ARPACK seem to be the exception).

Grid	*Precon*	it_{in}	it_{out}	t_{tot}	t_{eig}
boxcav16x10x3	None	171.9	26	402.0	393.3
	Diag	87.2	26	213.0	204.1
	SSOR	34.1	26	175.0	166.4
	2LevJACjac	41.0	26	148.0	139.4
	2LevJACssor	33.2	26	177.0	168.5
	2LevSGSjac	24.3	26	158.0	148.9
	2LevSGSssor	**15.7**	26	**138.0**	**128.8**
copcav18	None	269.3	26	785.0	774.2
	Diag	153.1	26	461.0	450.0
	SSOR	41.3	26	304.0	292.9
	2LevJACjac	54.4	26	307.0	294.5
	2LevJACssor	42.6	26	359.0	346.1
	2LevSGSjac	37.2	26	360.0	346.6
	2LevSGSssor	**21.3**	26	**267.0**	**254.2**
cop20k	None	159.4	26	1626.0	1588.3
	Diag	135.4	26	1390.0	1351.6
	SSOR	41.4	26	1021.0	982.8
	2LevJACjac	34.2	26	703.0	656.5
	2LevJACssor	29.0	26	802.0	755.8
	2LevSGSjac	22.3	26	830.0	781.8
	2LevSGSssor	**14.0**	26	**657.0**	**608.4**
cop40k	None	175.8	26	3619.0	3545.1
	Diag	165.0	26	3436.0	3361.8
	SSOR	50.0	26	2413.0	2338.7
	2LevJACjac	34.1	26	1672.0	1560.7
	2LevJACssor	30.2	26	2032.0	1917.0
	2LevSGSjac	22.0	26	2012.0	1894.3
	2LevSGSssor	**14.2**	26	**1512.0**	**1399.1**

Table 8.2: *Performance of preconditioners for the positive-definite eigenproblem and the ARPACK eigensolver*

8.4.2 Indefinite eigenvalue problem

Tabs. 8.4 to 8.3 show the results of the various preconditioners for the indefinite eigenvalue problem. As described in Section 4.2, several methods for solving the indefinite eigenvalue problem were implemented. For the following experiments we chose the three most successful methods: SAUG, AD and EIGSOLV.

Tabs. 8.4 and 8.5 show the results for JDSYM. For ARPACK we restricted our experiments to grid *cop20k*. The results are given in Tab. 8.3. it_{in} is the average number of inner iterations per outer iteration, it_{out} is the number of outer iterations, t_{tot} is the total computation time (including the construction of the global matrices and the preconditioner, but not including mesh generation) and t_{eig} is the time spent for solving the matrix eigenvalue problem.

Grid	*Method*	*Precon*	it_{in}	it_{out}	t_{tot}	t_{eig}
cop20k	SAUG	None	336.4	26	2278.0	2203.6
		Diag	262.3	26	1842.0	1768.2
		SSOR	72.0	26	1220.0	1146.8
		2LevJACjac	47.9	26	1352.0	1258.4
		2LevJACssor	28.7	26	**1131.0**	**1037.9**
		2LevSGSjac	35.9	26	1900.0	1805.5
		2LevSGSssor	**20.0**	26	1324.0	1228.9
cop20k	AD	None	1001.0	26	5750.0	5676.1
		Diag	1001.0	26	5861.0	5787.9
		SSOR	635.3	26	8040.0	7967.1
		2LevJACjac	49.3	26	1068.0	979.2
		2LevJACssor	28.1	26	799.0	711.0
		2LevSGSjac	35.5	26	1319.0	1233.2
		2LevSGSssor	**16.0**	26	**778.0**	**689.8**
cop20k	EIGSOLV	None	†			
		Diag	†			
		SSOR	270.1	26	6262.0	6243.7
		2LevJACjac	51.7	26	1451.0	1411.8
		2LevJACssor	32.7	26	1223.0	1183.8
		2LevSGSjac	37.8	26	1939.0	1898.8
		2LevSGSssor	**17.7**	26	**1087.0**	**1046.9**

Table 8.3: *Performance of preconditioners for the indefinite eigenproblem with grid* cop20k *and the ARPACK eigensolver*
† no convergence

For the SAUG and the EIGSOLV method, the Diag preconditioner already reduces the number of inner iterations and the execution times substantially. Oddly, this is not the case for the AD method. Here the Diag preconditioner seems to have a detrimental effect for some grids.

The two-level preconditioners always reduce the number of inner iteration steps most effectively. In almost all cases two-level preconditioners lead to the fastest execution times. Of the four two-level variants, 2LevJACssor and 2LevSGSssor are faster than the other two.

The SSOR preconditioner yields significantly better results than the Diag preconditioner. For the largest grid (*cop40k*) it even beats the two-level preconditioner in some cases. This indicates, that the two-level preconditioners do not scale very well with respect to execution time, due to the exact factorisation of the (1,1)-block.

When the Diag preconditioner or no preconditioner at all is used, it often happens, that the linear solver reaches the iteration step limit, without satisfying the convergence criterion. If this happens with the ARPACK eigensolver, the calculation is aborted, since the IRL method relies on an accurate Krylov subspace. In order to avoid such a breakdown, the maximum number of iteration steps is usually set very high (1000 for these experiments) when using ARPACK.

JDSYM on the other hand has no such requirements and can continue safely even if the computed correction does not meet the convergence criterion. Usually the outer iteration (JDSYM iteration) will then converge more slowly.

When comparing it_{in} for SAUG and AD, it is surprising to observe, that the two-level preconditioners are more effective with AD, but the other preconditioners (None, Diag and SSOR) are more effective with SAUG. We cannot explain this observation.

Grid	*Method*	*Precon*	it_{in}	it_{out}	t_{tot}	t_{eig}
boxcav16x10x3	SAUG	None	71.2	42	506.0	496.5
		Diag	18.6	39	117.0	108.2
		SSOR	8.0	35	91.0	82.2
		2LevJACjac	7.5	34	79.0	69.3
		2LevJACssor	4.9	33	**78.0**	**68.5**
		2LevSGSjac	4.9	34	97.0	87.0
		2LevSGSssor	**3.3**	32	89.0	79.3
boxcav16x10x3	AD	None	38.9	42	306.0	296.5
		Diag	37.6	45	258.0	248.6
		SSOR	10.3	42	134.0	125.0
		2LevJACjac	6.2	36	94.0	84.2
		2LevJACssor	3.7	35	**83.0**	**72.8**
		2LevSGSjac	4.9	36	122.0	112.2
		2LevSGSssor	**2.4**	31	87.0	76.8
boxcav16x10x3	EIGSOLV	None	442.2	59	4181.0	4177.7
		Diag	95.3	58	988.0	984.2
		SSOR	42.4	50	645.0	640.7
		2LevJACjac	11.4	47	151.0	146.9
		2LevJACssor	7.6	45	138.0	133.4
		2LevSGSjac	8.2	46	186.0	180.9
		2LevSGSssor	**4.6**	41	**134.0**	**129.5**
copcav18	SAUG	None	114.9	44	886.0	872.2
		Diag	28.8	41	233.0	218.9
		SSOR	10.4	38	**157.0**	**142.3**
		2LevJACjac	13.2	31	196.0	178.9
		2LevJACssor	6.5	35	174.0	157.5
		2LevSGSjac	7.9	37	250.0	232.8
		2LevSGSssor	**4.1**	33	183.0	165.4
copcav18	AD	None	120.9	51	1095.0	1080.8
		Diag	130.4	53	1355.0	1341.0
		SSOR	32.3	50	507.0	492.9
		2LevJACjac	8.6	40	228.0	211.8
		2LevJACssor	5.0	37	183.0	165.8
		2LevSGSjac	7.3	36	262.0	244.9
		2LevSGSssor	**3.2**	32	**169.0**	**151.7**
copcav18	EIGSOLV	None	627.6	66	8392.0	8387.4
		Diag	182.6	61	2346.0	2340.3
		SSOR	54.9	56	1178.0	1172.8
		2LevJACjac	20.3	50	492.0	484.0
		2LevJACssor	11.5	51	385.0	377.2
		2LevSGSjac	13.0	49	507.0	499.4
		2LevSGSssor	**6.8**	39	**288.0**	**280.4**

Table 8.4: *Performance of preconditioners for the indefinite eigenproblem with the smaller grids* boxcav *and* copcav18 *and the JDSYM eigensolver*

Grid	*Method*	*Precon*	it_{in}	it_{out}	t_{tot}	t_{eig}
cop20k	SAUG	None	81.7	45	2297.0	2224.7
		Diag	32.0	45	879.0	806.0
		SSOR	10.2	42	575.0	501.8
		2LevJACjac	8.1	36	585.0	492.2
		2LevJACssor	5.1	36	**538.0**	**446.1**
		2LevSGSjac	5.6	38	723.0	633.2
		2LevSGSssor	**3.3**	33	572.0	478.2
cop20k	AD	None	170.8	54	6118.0	6045.0
		Diag	212.6	56	8304.0	8231.2
		SSOR	57.2	54	3054.0	2980.5
		2LevJACjac	6.5	42	699.0	611.7
		2LevJACssor	4.0	39	589.0	502.2
		2LevSGSjac	4.8	38	729.0	644.6
		2LevSGSssor	**2.4**	34	**569.0**	**481.1**
cop20k	EIGSOLV	None	520.6	63	22741.0	22724.8
		Diag	176.3	57	7191.0	7172.2
		SSOR	67.3	59	4839.0	4820.3
		2LevJACjac	11.9	54	1105.0	1067.5
		2LevJACssor	9.6	48	978.0	941.3
		2LevSGSjac	9.7	48	1277.0	1241.0
		2LevSGSssor	**5.8**	45	**945.0**	**906.6**
cop40k	SAUG	None	75.6	45	4678.0	4412.5
		Diag	38.9	44	2451.0	2181.2
		SSOR	11.3	43	**1472.0**	**1205.1**
		2LevJACjac	8.6	38	1732.0	1383.3
		2LevJACssor	5.3	36	1725.0	1337.7
		2LevSGSjac	5.7	39	2137.0	1781.7
		2LevSGSssor	**3.4**	34	1594.0	1251.3
cop40k	AD	None	288.9	57	31010.0	30739.4
		Diag	331.7	62	28439.0	28169.5
		SSOR	95.2	58	14476.0	14204.1
		2LevJACjac	6.3	41	1856.0	1533.4
		2LevJACssor	3.9	39	1645.0	1320.2
		2LevSGSjac	4.8	37	1959.0	1632.1
		2LevSGSssor	**2.6**	35	**1561.0**	**1242.1**
cop40k	EIGSOLV	None	528.3	61	49548.0	49513.9
		Diag	231.6	60	21927.0	21889.2
		SSOR	77.2	57	11823.0	11784.7
		2LevJACjac	12.9	53	2830.0	2711.7
		2LevJACssor	9.1	51	2457.0	2338.8
		2LevSGSjac	8.9	50	3292.0	3171.9
		2LevSGSssor	**5.7**	42	**2180.0**	**2060.1**

Table 8.5: *Performance of preconditioners for the indefinite eigenproblem with the larger grids* cop20k *and* cop40k *and the JDSYM eigensolver*

Summary The two-level preconditioners are very fast for both the positive-definite and the indefinite eigenvalue problem. For the positive-definite eigenvalue problem, the 2LevJACjac variant is usually fastest when using JDSYM. For ARPACK 2LevSGSssor is always fastest. For the indefinite problem either 2LevJACssor and 2LevSGSssor are very effective. The downside of the two-level preconditioners is their memory consumption: In our implementation they all rely on an exact factorisation of the (1,1)-block of $\boldsymbol{A}^{\sigma}$. Such a factor consumes e.g. twice the amount of memory as the matrix $\boldsymbol{A}^{\sigma}$ for grid *cop40k*.

The SSOR preconditioner turns out to be surprisingly effective if the SAUG method is used. For large problems SSOR seems to be even faster than the two-level preconditioners in some cases. The SSOR preconditioner is the method of choice for large grids, since it has modest memory requirements compared to the two-level preconditioners (no factorisation needed).

For very large indefinite eigenproblems, the EIGSOLV method together with the SSOR preconditioner is to be preferred. This combination requires no factorisations at all and still converges reliably if JDSYM is used.

The Diag preconditioner should not be used, since it has no real advantages over the SSOR preconditioner.

9 Python implementation

This chapter describes the Python implementation of the finite element application for solving Maxwell's equations. The chapter starts with a section, that describes the development history which ultimately led to the Python implementation. In the following sections, the Python language itself, and the main python packages *Numerical Python* [7], *PySparse* and *PyFemax* are introduced (PySparse and PyFemax are written by ourselves). The chapter concludes with numerical results of large problems solved using the Python implementation.

9.1 Development history

Matlab implementation We started five years ago with the development of a code to solve the Maxwell eigenvalue problem. The first implementation was written in Matlab. While Matlab's vector syntax and the built-in matrix data types greatly simplified the programming task, it soon turned out, that a project of this scale cannot be done using Matlab.

Because Matlab's sparse matrix operations are extremely slow (cf. examples in Sections 9.4.1 and 9.4.3), only very small toy problems could be solved. Another handicap was Matlab's inflexible language: There may be only one (visible) function in a file; the name of the function is bound to the file name. This makes it very hard to modularise the code. Also namespace collisions hinder development of large applications using Matlab. Many other desirable language features, such as exception handling, are also missing in Matlab.

Fortran/C implementation It was clear, that the application had to be rewritten. Because of its vector operations, the inter-operability with Fortran 77 and the possibility to allocate memory dynamically, we chose Fortran 90. Using mainly Fortran 90, we rewrote and extended the whole application. Parts of the application were written in C or Fortran 77.

The native Fortran/C implementation includes all algorithms and methods described in this thesis. The application contains the following modules:

- support for two different kinds of finite elements: nodal and vector elements with both linear and quadratic basis functions
  - calculation of local finite element matrices
  - implementation of boundary conditions
  - assembly of the global finite element matrices
- Jacobi-Davidson eigensolver implementation (JDSYM)
- driver code for the ARPACK eigensolver
- five methods for solving the indefinite eigenvalue program
- code for handling sparse matrices
- optimised code for sparse matrix-vector multiplication
- implementation of PCG, CGS and MINRES iterative solvers, interfaces to SYMMLQ and QMRS iterative solvers
- implementation of Jacobi, SSOR, ILUS and 2-level hierarchical basis preconditioners

- ▷ reading mesh geometry data from input files in 3 different formats
- ▷ generation of mesh data for simple box shaped cavities
- ▷ interfaces to SuperLU and HP MLIB direct solvers
- ▷ reading Windows style INI files, used for parameter input

The Fortran 90 implementation showed good performance. However, it turned out that Fortran 90 was not very well suited for some tasks. Some parts had to be written in C or Fortran 77. E.g. the low-level optimisations of the sparse matrix-vector multiplication had to be implemented in C. Most other numerical kernels, like e.g. the SSOR preconditioner, were implemented in Fortran 77, since code written in Fortran 77 usually ran a little faster than Fortran 90 code[27].

From today's point of view Fortran 90 was not a good choice. The lack of general purpose pointers and especially function pointers troubled us. Fortran 90 assumed-shape arrays were another problem. They are not supported by any other language than Fortran 90 and their internal structure is not standardised.

The development time of this code spanned four years. The code, written in Fortran 77, Fortran 90 and C, is now approximately 33'000 lines long. The program is cluttered and no longer well modularised, since a lot of features were added, that were not planned in the beginning. This makes the code difficult to understand, maintain and extend.

Python implementation These reasons motivated us, to redesign and rewrite the application a second time. We had the following goals in mind:

Modularity The code should be organised in reusable modules. These modules should have no side effects.

Brevity The code should be short and concise, as e.g. Matlab code.

Object based Sparse and dense matrices and vectors should be implemented as objects, having attributes and methods. This improves the readability and extensibility of the application.

High performance The code's performance should be comparable to the Fortran/C implementation.

We felt Python to be the language of choice for the redesign. Python is an *interpreted, interactive, object-oriented* programming language [74]. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. New modules are easily written in C or C++. The Python implementation is copyrighted but freely usable and distributable, even for commercial use.

Since Python is an interpreted programming language, it is not well suited for high-performance numerical applications, out of the box. However, Python can easily be extended using modules written in C for performance critical tasks.

The *Numerical Python* package adds a fast, compact, multidimensional array language facility to Python. We use the Numerical Python package, which is written mainly in C, to implement operations with dense matrices and vectors.

[27] In the mid-nineties Fortran 90 compilers were still quite fresh on the market and so Fortran 77 compilers still seemed to optimise better at that time. This is however no longer the case now.

For manipulating sparse matrices we designed and implemented the *PySparse* package. *PySparse* is an extension to Python, that introduces new sparse matrix object types, some iterative solvers and preconditioners, the JDSYM eigensolver and an interface to the SuperLU direct solver [23]. *PySparse* is implemented using highly optimised C code.

Using the *PySparse* and *Numerical Python* packages as a foundation, the *PyFemax* package (written almost completely in Python) was developed. *PyFemax* (together with *PySparse*) is the rewrite of the native C/Fortran code. Some features implemented in the C/Fortran code were not ported to *PyFemax* (e.g. nodal finite elements, linear Nédélec finite elements, the ILUS preconditioner and the BESPALOV method). Still, *PyFemax* has everything necessary to solve large problems using vector finite elements.

What we are calling "the Python implementation" is actually a set of modules which are implemented using a mixed-language programming approach: The application logic, the input/output routines and the finite element code are implemented in Python. On the other hand, the time-critical parts, like the sparse and dense linear algebra routines, including iterative solvers, preconditioners, sparse matrix factorisations and the eigensolver are implemented in C and are tightly integrated into the Python framework.

Other people have successfully used Python for their scientific computing tasks. Quite a number a papers have been published in this area since 1997. For example, in [40] Hinsen demonstrates, how he developed parallel applications with the BSP (Bulk Synchronous Parallel) approach using Python. In [13] and [12] Beazley and Lomdahl describe how they transformed a monolithic molecular dynamics code for massively parallel processing systems into a flexible, highly modular, and powerful system for performing simulation, data analysis, and visualisation using Python.

9.2 The Python language

Python is an *interpreted*, *interactive*, *object-oriented* programming language. It is often compared to Tcl, Perl, Scheme or Java.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New extension modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

Python has a full set of string operations (including regular expression matching), and frees the user from most hassles of memory management. These and other features make it an ideal language for prototype development and other ad-hoc programming tasks.

Python also has some features that make it possible to write large programs, even though it lacks most forms of compile-time checking: a program can be constructed out of a number of modules, each of which defines its own name space, and modules can define classes which provide further encapsulation. Exception handling makes it possible to catch errors where required without cluttering all code with error checking.

A large number of extension modules have been developed for Python. Some are part of the standard library of tools, usable in any Python program (e.g. the math library and regular expressions). Others are specific to a particular platform or environment (e.g. UNIX, IP networking or X11) or provide application-specific functionality (e.g. image or sound processing).

Python also provides facilities for introspection[28], so that e.g. a debugger or profiler for

[28] introspection is code looking at other modules and functions in memory as objects, getting information about

Python programs can be written in Python itself. There is also a generic way to convert an object into a stream of bytes and back, which can be used to implement object persistence as well as various distributed object models.

The Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Macintosh, Amiga, NeXT and BeOS.

To give an introduction to the Python language would be out of the scope of this thesis. Instead we refer to the documentation available at the official Python web site http://www.python.org: To get started beginners should read the *Python Tutorial* [74]. The tutorial introduces many of Python's most noteworthy features, and gives a good idea of the language's flavour and style. The *Python Library Reference* [73] documents Python's standard objects and modules, the standard types of the language and its built-in functions and exceptions. The *Extending and Embedding the Python Interpreter* and *Python/C API Reference* manuals [72][75] describe how to write modules in C or C++ to extend the Python interpreter with new modules.

9.3 The Numerical Python package (NumPy)

The Numerical Python extensions (NumPy henceforth) is a set of extensions to the Python programming language which allows Python programmers to efficiently manipulate large sets of objects organised in grid-like fashion. These sets of objects are called arrays, and they can have any number of dimensions: one dimensional arrays are similar to standard Python sequences, two-dimensional arrays are similar to matrices from linear algebra.

Why are these extensions needed? The main reason is that manipulating a set of a million numbers in Python with the standard data structures such as lists, tuples or classes is much too slow and uses too much space. A more subtle reason for these extensions however is that the kinds of operations that programmers typically want to do on arrays, while sometimes very complex, can often be decomposed into a set of fairly standard operations. A program using such standard building blocks is typically shorter and easier to read. This decomposition has been developed similarly in many array languages like e.g. Matlab, Fortran 90, and others.

Apart from the new multidimensional array type, together with basic array operations and manipulations, NumPy comes with a set of libraries that support fast Fourier transforms, basic linear algebra, random number generation and masked arrays. In addition, NumPy is accompanied by a Matlab compatibility library, which provides many functions, that compatible with the Matlab function of the same name.

A thorough manual [7] for the Numeric package is available at http://www.pfdubois.com/numpy/. The manual describes the usage of NumPy and also explains how to access the NumPy features from within other C extensions. The Numeric package, not yet being included in the standard Python distribution, can be downloaded from the SourceForge project web page at http://sourceforge.net/projects/numpy.

9.4 The PySparse package

The PySparse package is our own implementation. It extends the Python interpreter by a set of sparse matrix types. PySparse also includes modules that implement

- ▷ iterative methods for solving linear systems of equations,
- ▷ a set of standard preconditioners,

them, and manipulating them.

- an interface to a direct solver for sparse linear systems of equations,
- and the JDSYM eigensolver.

All these modules are implemented as C extension modules for maximum performance. In the following sections all modules of PySparse are described in detail.

9.4.1 The spmatrix module

The *spmatrix* module is the foundation of the PySparse package. It extends the Python interpreter by three new types named *ll_mat*, *csr_mat* and *sss_mat*. These types represent sparse matrices in the LL-, the CSR- and SSS-formats respectively (cf. Appendix A). For all three formats, double precision values (C type `double`) are used to represent the non-zero entries.

The common way to use the *spmatrix* module is to first build a matrix in the LL-format. The LL-matrix is manipulated until it has its final shape and content. Afterwards it may be converted to either the CSR- or SSS-format, which need less memory and allow for faster matrix-vector multiplications.

A *ll_mat* object can be created from scratch, by reading data from a file (in MatrixMarket format) or as a result of a matrix operation (as e.g. matrix-matrix multiplication). The *ll_mat* object supports manipulating (reading, writing, add-updating) single entries or sub-matrices.

csr_mat and *sss_mat* are not constructed directly, instead they are created by converting *ll_mat* objects. Once created, *csr_mat* and *sss_mat* objects cannot be manipulated. Their main purpose is to support efficient matrix-vector multiplications.

spmatrix module functions

ll_mat(n, m, sizeHint=1000) creates a *ll_mat* object, that represents a general, all zero $m \times n$ matrix. The optional *sizeHint* parameter specifies the number of non-zero entries for which space is allocated initially.

If the total number of non-zero elements of the final matrix is known (approximately), this number can be passed as *sizeHint*. This will avoid costly memory reallocations.

ll_mat_sym(n, sizeHint=1000) creates a *ll_mat* object, that represents a *symmetric*, all zero $n \times n$ matrix. The optional *sizeHint* parameter specifies, how much space is initially allocated for the matrix.

ll_mat_from_mtx(fileName) creates a *ll_mat* object from a file named *fileName*, which must be stored in MatrixMarket Coordinate format as described at `http://math.nist.gov/MatrixMarket/formats.html`. Depending on the file content, either a symmetric or a general sparse matrix is generated.

matrixmultiply(A, B) computes the matrix-matrix multiplication

$$\boldsymbol{C} := \boldsymbol{A}\boldsymbol{B}$$

and returns the result $\boldsymbol{C}$ as a new *ll_mat* object representing a general sparse matrix. The parameters $\boldsymbol{A}$ and $\boldsymbol{B}$ are expected to be objects of type *ll_mat*.

dot(A, B) computes the "dot-product"

$$C := A^T B$$

and returns the result C as a new *ll_mat* object representing a general sparse matrix. The parameters A and B are expected to be objects of type *ll_mat*.

ll_mat objects *ll_mat* objects represent matrices stored in the LL format, which is described in Appendix A.4. *ll_mat* objects come in two flavours: *general* matrices and *symmetric* matrices. For symmetric matrices only the non-zero entries in the lower triangle are stored. Write operations to the strictly upper triangle are prohibited for the symmetric format. The `issym` attribute of an *ll_mat* object can be queried to find out whether or not the symmetric storage format is used.

The entries of a matrix can be accessed conveniently using two-dimensional array indices[29]. Following Python conventions, indices start with 0 and wrap around (so -1 is equivalent to the last index).

The following code creates an empty 5×5 matrix `A`, sets all diagonal elements to their respective row/column index and then copies the value of `A[0,0]` to `A[2,1]`.

```
>>> import spmatrix
>>> A = spmatrix.ll_mat(5, 5)
>>> for i in range(5):
...     A[i,i] = i+1
>>> A[2,1] = A[0,0]
>>> print A
ll_mat(general, [5,5], [(0,0): 1, (1,1): 2, (2,1): 1,
(2,2): 3, (3,3): 4, (4,4): 5])
```

The Python slice notation can be used to conveniently access sub-matrices.

```
>>> print A[:2,:]     # the first two rows
ll_mat(general, [2,5], [(0,0): 1, (1,1): 2])
>>> print A[:,2:5]    # columns 2 to 4
ll_mat(general, [5,3], [(2,0): 3, (3,1): 4, (4,2): 5])
>>> print A[1:3,2:5]  # submatrix
                      # starting at row 1 col 2,
                      # ending at row 2 col 4
ll_mat(general, [2,3], [(1,0): 3])
```

The slice operator always returns a new *ll_mat* object, containing a copy of the selected submatrix.

Write operations to slices are also possible:

[29] The standard Python language does not know multidimensional indices. However, thanks to Python's clever design, its easy to provide multidimensional indices for extension types, without any dirty hacks.

In the Python language, subscripts can be of any type (as it is customary for dictionaries). A two-dimensional index can be regarded as a 2-tuple (the brackets do not have to be written, so `A[1,2]` is the same as `A[(1,2)]`). If both tuple elements are integers, then a single matrix element is referenced. If at least one of the tuple elements is a slice (which is also a Python object), then a submatrix is referenced.

Subscripts have to be decoded at runtime. This task includes type checks, extraction of indices from the 2-tuple, parsing of slice objects and index bound checks.

```
>>> B = ll_mat(2, 2)           # create 2-by-2
>>> B[0,0] = -1; B[1,1] = -1   # diagonal matrix
>>> A[:2,:2] = B               # assign it to upper
>>>                            # diagonal block of A
>>> print A
ll_mat(general, [5,5], [(0,0): -1, (1,1): -1, (2,1): 1,
(2,2): 3, (3,3): 4, (4,4): 5])
```

ll_mat object attributes

A.shape returns a 2-tuple containing the shape of the matrix $\boldsymbol{A}$, i.e. the number of rows and columns.

A.nnz returns the number of non-zero entries stored in matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is stored in the symmetric format, only the number of non-zero entries in the lower triangle (including the diagonal) are returned.

A.issym returns true (a non-zero integer) if matrix $\boldsymbol{A}$ is stored in the symmetric LL format, i.e. only the non-zero entries in the lower triangle are stored. Returns false (zero) if matrix $\boldsymbol{A}$ is stored in the general LL format.

ll_mat object methods

A.to_csr() returns a newly allocated *csr_mat* object, which results from converting matrix $\boldsymbol{A}$.

A.to_sss() returns a newly allocated *sss_mat* object, which results from converting matrix $\boldsymbol{A}$. This function works for *ll_mat* objects in both the symmetric and the general format. If $\boldsymbol{A}$ is stored in the general format, only the entries in the lower triangle are used for the conversion. No check, whether $\boldsymbol{A}$ is symmetric, is performed.

A.matvec(x, y) computes the sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size.

A.matvec_transp(x, y) computes the transposed sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}^T\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size.

A.export_mtx(fileName, precision=16) exports matrix $\boldsymbol{A}$ to file named *fileName*. The matrix is stored in MatrixMarket Coordinate format as described at http://math.nist.gov/MatrixMarket/formats.html. Depending on the properties of *ll_mat* object $\boldsymbol{A}$ the generated file either uses the symmetric or a general MatrixMarket Coordinate format. The optional parameter *precision* specifies the number of decimal digits that are used to express the non-zero entries in the output file.

A.shift(sigma, M) performs a daxpy-like operation on matrix $\boldsymbol{A}$,

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \sigma \boldsymbol{M}.$$

The parameter σ is expected to be a Python Float object. The parameter $\boldsymbol{M}$ is expected to an object of type *ll_mat*.

A.copy() returns a new *ll_mat* object, that represents a copy of the *ll_mat* object $\boldsymbol{A}$. So,

```
>>> B = A.copy()
```

is equivalent to

```
>>> B = A[:,:]
```

which is *not* the same as

```
>>> B = A
```

The latter version only returns a reference to the same object and assigns it to `B`. Subsequent changes to `A` will therefore also be visible in `B`.

A.update_add_mask(B, ind0, ind1, mask0, mask1) This method is provided for efficiently assembling global finite element matrices. The method adds the matrix $\boldsymbol{B}$ to entries of matrix $\boldsymbol{A}$. The indices of the entries to be updated are specified by *ind0* and *ind1*. The individual updates are enabled or disabled using the *mask0* and *mask1* arrays.

The operation is equivalent to the following Python code:

```
for i in range(len(ind0)):
    for j in range(len(ind1)):
        if mask0[i] and mask1[j]:
            A[ind0[i],ind1[j]] += B[i,j]
```

All five parameters are NumPy arrays. $\boldsymbol{B}$ is an array of rank two. The four remaining parameters are rank-1 arrays. Their length corresponds to either the number of rows or the number of columns of $\boldsymbol{B}$.

This method is not supported for *ll_mat* objects of symmetric type, since it would generally result in an non-symmetric matrix. *update_add_mask_sym* must be used in that case. Attempting to call this method using a *ll_mat* object of symmetric type will raise an exception.

A.update_add_mask_sym(B, ind, mask) This method is provided for efficiently assembling symmetric global finite element matrices. The method adds the matrix $\boldsymbol{B}$ to entries of matrix $\boldsymbol{A}$. The indices of the entries to be updated are specified by *ind*. The individual updates are enabled or disabled using the *mask* array.

The operation is equivalent to the following Python code:

```
for i in range(len(ind)):
    for j in range(len(ind)):
        if mask[i]:
            A[ind[i],ind[j]] += B[i,j]
```

The three parameters are all NumPy arrays. $\boldsymbol{B}$ is an array of rank two representing a square matrix. The four remaining parameters are rank-1 arrays. Their length corresponds to the order of matrix $\boldsymbol{B}$.

A.delete_rows(mask) deletes rows from matrix $\boldsymbol{A}$. The rows to be deleted are specified by the mask parameter, which is a 1D integer NumPy array. If *mask[i] = 0*, then row *i* is deleted, otherwise row *i* is kept. This method may not be applied to a matrix in symmetric format.

A.delete_cols(mask) deletes columns from matrix $\boldsymbol{A}$. The columns to be deleted are specified by the mask parameter, which is a 1D integer NumPy array. If *mask[i] = 0*, then column *i* is deleted, otherwise column *i* is kept. This method may not be applied to a matrix in symmetric format.

A.delete_rowcols(mask) deletes rows and columns from matrix $\boldsymbol{A}$. The rows and columns to be deleted are specified by the mask parameter, which is a 1D integer NumPy array. If *mask[i] = 0*, then row and column *i* are deleted, otherwise row and column *i* are kept.

A.norm(p) returns the *p*-norm of matrix $\boldsymbol{A}$. The parameter *p* is a string identifying the type of norm to be computed. If *p = '1'*, then the 1-norm of $\boldsymbol{A}$ is returned. If *p = 'inf'*, then the infinity-norm of $\boldsymbol{A}$ is returned. If *p = 'fro'*, then the Frobenius norm of $\boldsymbol{A}$ is returned.

csr_mat and sss_mat objects *csr_mat* objects represent matrices stored in the CSR format, which is described in Appendix A.1. *sss_mat* objects represent matrices stored in the SSS format (c.f. Appendix A.3). The only way to create a *csr_mat* or a *sss_mat* object is by conversion of a *ll_mat* object using the *to_csr()* or the *to_sss()* method respectively. The purpose of the *csr_mat* and the *to_sss()* objects is to provide fast matrix-vector multiplications for sparse matrices. In addition, a matrix stored in the CSR or SSS format uses less memory than the same matrix stored in the LL format, since the *link* array is not needed.

csr_mat and *sss_mat* objects only provide limited capabilities to access matrix entries or sub-matrices using two-dimensional indices.

csr_mat and sss_mat object attributes

A.shape returns a 2-tuple containing the shape of the matrix $\boldsymbol{A}$, i.e. the number of rows and columns.

A.nnz returns the number of non-zero entries stored in matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is an *sss_mat* object, the non-zero entries in the strictly upper triangle are not counted.

csr_mat and sss_mat object methods

A.matvec(x, y) computes the sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size.

A.matvec_transp(x, y) computes the transposed sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}^T\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size. For *sss_mat* objects *matvec_transp* is equivalent to *matvec*.

Example: 2D-Poisson matrix This section illustrates the use of the *spmatrix* module to build the well known 2D-Poisson matrix resulting from a $n \times n$ square grid.

```
def poisson2d(n):
    L = spmatrix.ll_mat(n*n, n*n)
    for i in range(n):
        for j in range(n):
            k = i + n*j
            L[k,k] = 4
            if i > 0:
                L[k,k-1] = -1
            if i < n-1:
                L[k,k+1] = -1
            if j > 0:
                L[k,k-n] = -1
            if j < n-1:
                L[k,k+n] = -1
    return L
```

Using the symmetric variant of the *ll_mat* object, this gets even shorter.

```
def poisson2d_sym(n):
    L = spmatrix.ll_mat_sym(n*n)
    for i in range(n):
        for j in range(n):
            k = i + n*j
            L[k,k] = 4
            if i > 0:
                L[k,k-1] = -1
            if j > 0:
                L[k,k-n] = -1
    return L
```

To illustrate the use of the slice notation to address sub-matrices, let's build the 2D Poisson matrix using the diagonal and off-diagonal blocks.

```
def poisson2d_sym_blk(n):
    L = spmatrix.ll_mat_sym(n*n)
    I = spmatrix.ll_mat_sym(n)
    P = spmatrix.ll_mat_sym(n)
    for i in range(n):
        I[i,i] = -1
    for i in range(n):
        P[i,i] = 4
        if i > 0: P[i,i-1] = -1
    for i in range(0, n*n, n):
        L[i:i+n,i:i+n] = P
        if i > 0: L[i:i+n,i-n:i] = I
    return L
```

Performance comparison with Matlab Let's compare the performance of three Python codes above with the following Matlab functions:

The Matlab function `poisson2d` is equivalent to the Python function with the same name

Function	$n = 100$	$n = 300$	$n = 500$	$n = 1000$
Python `poisson2d`	0.44	4.11	11.34	45.50
Python `poisson2d_sym`	0.26	2.34	6.55	26.33
Python `poisson2d_sym_blk`	0.03	0.21	0.62	2.22
Matlab `poisson2d`	28.19	3464.9	38859	∞
Matlab `poisson2d_blk`	6.85	309.20	1912.1	∞
Matlab `poisson2d_kron`	0.21	2.05	6.23	29.96

Table 9.1: *Performance comparison of Python and Matlab functions to generate the 2D Poisson matrix*
The execution times are given in seconds. Matlab version 6.0 Release 12 was used for these timings.

```
function L = poisson2d(n)
  L = sparse(n*n);
  for i = 1:n
    for j = 1:n
      k = i + n*(j-1);
      L(k,k) = 4;
      if i > 1, L(k,k-1) = -1; end
      if i < n, L(k,k+1) = -1; end
      if j > 1, L(k,k-n) = -1; end
      if j < n, L(k,k+n) = -1; end
    end
  end
```

The function `poisson2d_blk` is an adaption of the Python function `poisson2d_sym_blk` (except for exploiting the symmetry, which is not directly supported in Matlab).

```
function L = poisson2d_blk(n)
  e = ones(n,1);
  P = spdiags([-e 4*e -e], [-1 0 1], n, n);
  I = -speye(n);
  L = sparse(n*n);
  for i = 1:n:n*n
    L(i:i+n-1,i:i+n-1) = P;
    if i > 1, L(i:i+n-1,i-n:i-1) = I; end
    if i < n*n - n, L(i:i+n-1,i+n:i+2*n-1) = I; end
  end
```

The function `poisson2d_kron` demonstrates one of the most efficient ways to generate the 2D Poisson matrix in Matlab.

```
function L = poisson2d_kron(n)
  e = ones(n,1);
  P = spdiags([-e 2*e -e], [-1 0 1], n, n);
  L = kron(P, speye(n)) + kron(speye(n), P);
```

The execution times reported in Tab. 9.1 clearly show, that the Python implementation is superior to the Matlab implementation. If the fastest versions are compared for both languages, Python is approximately 10 times faster. Comparing the straight forward `poisson2d` versions, one is struck by the result that, the Matlab function is incredibly slow. The Python version is more then three orders of magnitude faster! This result really raises the doubt, whether Matlab's sparse matrix format is appropriately chosen.

The performance difference between Python's `poisson2d_sym` and `poisson2d_sym_blk` indicates, that a lot of time is spent parsing indices.

9.4.2 The precon module

The *precon* module provides preconditioners, which can be used e.g. for the iterative methods implemented in the in the *itsolvers* module or the JDSYM eigensolver (in the *jdsym* module).

In the PySparse framework, any Python object that has the following properties can be used as a preconditioner:

- a *shape* attribute, which returns a 2-tuple describing the dimension of the preconditioner,
- and a *precon* method, that accepts two vectors $\mathbf{x}$ and $\mathbf{y}$, and applies the preconditioner to $\mathbf{x}$ and stores the result in $\mathbf{y}$. Both $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size.

The *precon* module implements two new object types *jacobi* and *ssor*, representing Jacobi and the SSOR preconditioners as described in Sections 8.1.1 and 8.1.2.

precon module functions

jacobi(A, omega=1.0, steps=1) creates a *jacobi* object, representing the Jacobi preconditioner. The parameter $\boldsymbol{A}$ is the system matrix used for the Jacobi iteration. The matrix needs to be subscriptable using two-dimensional indices, so e.g. an *ll_mat* object would work. The optional parameter ω, which defaults to 1.0, is the weight parameter as described in Section 8.1.1. The optional *steps* parameter (defaults to 1) specifies the number of iteration steps.

ssor(A, omega=1.0, steps=1) creates a *ssor* object that represents the SSOR preconditioner. The parameter $\boldsymbol{A}$ is the system matrix used for the SSOR iteration. The matrix $\boldsymbol{A}$ has to be an object of type *sss_mat*. The optional parameter ω, which defaults to 1.0, is the relaxation parameter as described in Section 8.1.1. The optional *steps* parameter (defaults to 1) specifies the number of iteration steps.

jacobi and ssor objects Both *jacobi* and *ssor* objects provide the *shape* attribute and the *precon* method, that every preconditioner object in the PySparse framework must implement. Apart from that, there is nothing noteworthy to say about these objects.

Example: diagonal preconditioner The diagonal preconditioner is just a special case of the Jacobi preconditioner, with $\omega = 1.0$ and steps $= 1$, which happen to be the default values of these parameters.

It is however easy to implement the diagonal preconditioner using a Python class:

```
class diag_prec:
    def __init__(self, A):
        self.shape = A.shape
        n = self.shape[0]
        self.dinv = Numeric.zeros(n, 'd')
        for i in xrange(n):
            self.dinv[i] = 1.0 / A[i,i]
    def precon(self, x, y):
        Numeric.multiply(x, self.dinv, y)
```

So,

```
>>> D1 = precon.jacobi(A, 1.0, 1)
```

and

```
>>> D2 = diag_prec(A)
```

yield functionally equivalent preconditioners. `D1` is probably faster than `D2`, because it is fully implemented in C.

9.4.3 The itsolvers module

The *itsolvers* module provides a set of iterative methods for solving linear systems of equations.

The iterative methods are callable like ordinary Python functions. All these functions expect the same parameter list, and all function return values also follow a common standard.

Any user-defined iterative solvers should also follow these conventions, since other PySparse modules rely on them (e.g. the *jdsym* module)

Parameter list Let's illustrate the calling conventions, using the PCG iterative method defined as *info, iter, relres = pcg*($\boldsymbol{A}$, $\mathbf{b}$, $\mathbf{x}$, *tol, maxit,* $\boldsymbol{K}$).

- $\boldsymbol{A}$ The parameter $\boldsymbol{A}$ represents the coefficient matrix of the linear system of equations. $\boldsymbol{A}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods for multiplying with a vector.
- $\mathbf{b}$ The parameter $\mathbf{b}$, representing the right-hand-side of the linear system, is a rank-1 NumPy array.
- $\mathbf{x}$ The parameter $\mathbf{x}$ is also a rank-1 NumPy array. On input, $\mathbf{x}$ holds the initial guess. On output, $\mathbf{x}$ holds the approximate solution of the linear system.
- *tol* The *tol* parameter is a float value representing the requested error tolerance. The exact meaning of this parameter depends on the actual iterative solver.
- *maxit* The *maxit* parameter is an integer that specifies the maximum number of iterations to be executed.
- $\boldsymbol{K}$ The *optional* $\boldsymbol{K}$ parameter represents a preconditioner object that supplies the *shape* attribute and the *precon* method.

The iterative solvers may accept additional parameters, which are passed as keyword arguments.

Return value All iterative solvers return a tuple with three elements (*info, iter, relres*):

info is an integer that contains the exit status of the iterative solver. *info* has one of the following values

- 2 iteration converged, residual is as small as seems reasonable on this machine.
- 1 iteration converged, $\mathbf{b} = \mathbf{0}$, so the exact solution is $\mathbf{x} = \mathbf{0}$.
- 0 iteration converged, relative error appears to be less than *tol*.
- -1 iteration not converged, maximum number of iterations was reached.

-2 iteration not converged, the system involving the preconditioner was ill-conditioned.

-3 iteration not converged, an inner product of the form $\mathbf{x}^T \boldsymbol{K}^{-1}\mathbf{x}$ was not positive, so the preconditioning matrix $\boldsymbol{K}$ does not appear to be positive definite.

-4 iteration not converged, the matrix $\boldsymbol{A}$ appears to be very ill-conditioned

-5 iteration not converged, the method stagnated

-6 iteration not converged, a scalar quantity became too small or too large to continue computing

So, $info >= 0$ indicates, that $\mathbf{x}$ holds an acceptable solution, and $info < 0$ indicates an error condition.

Note that not all iterative solvers check for all above error conditions.

iter holds the number of iterations performed.

relres holds relative error of the solution or the relative residual as computed by the iterative method. What this actually is, depends on the specific iterative method used.

itsolvers module functions The module functions defined in the *itsolvers* module implement various iterative methods (PCG, MINRES, QMRS and CGS, cf. Section 6). The parameters and return values conform to the conventions described above.

pcg(A, b, x, tol, maxit, K) The *pcg* function implements the Preconditioned Conjugate Gradients method.

minres(A, b, x, tol, maxit, K) The *minres* function implements the MINRES method.

qmrs(A, b, x, tol, maxit, K) The *qmrs* function implements the QMRS method.

cgs(A, b, x, tol, maxit, K) The *min* function implements the CGS method.

Example: Solving the poisson system Let's solve the Poisson system

$$\boldsymbol{L}\mathbf{x} = \mathbf{1}, \tag{9.1}$$

using the PCG method. $\boldsymbol{L}$ is the 2D Poisson matrix, introduced in Section 9.4.1 and $\mathbf{1}$ is a vector with all entries equal to one.
The Python solution for this task looks as follows:

```
import Numeric, spmatrix, precon, itsolvers
n = 300
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
info, iter, relres = itsolvers.pcg(L.to_sss(),
                                   b, x, 1e-12, 2000)
```

The code makes use of the Python function `poisson2d_sym_blk`, which was defined in Section 9.4.1.
Incorporating e.g. a SSOR preconditioner is straight-forward:

```
import Numeric, spmatrix, precon, itsolvers
n = 300
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
S = L.to_sss()
Kssor = precon.ssor(S)
info, iter, relres = itsolvers.pcg(S, b, x, 1e-12, 2000, Kssor)
```

The Matlab solution (without preconditioner) may look as follows:

```
n = 300;
L = poisson2d_kron(n);
[x,flag,relres,iter] = pcg(L, ones(n*n,1), 1e-12, 2000,
                           [], [], zeros(n*n,1));
```

Performance comparison with Matlab and native C To evaluate the performance of the Python implementation we solve the 2D Poisson system (9.1) using the PCG method. The Python timings are compared with results of a Matlab and a native C implementation.

The native C and the Python implementation use the same core algorithms for PCG method and the matrix-vector multiplication. On the other hand, C reads the matrix from an external file instead of building it on the fly. In contrast to the Python implementation, the native C version does not suffer from the overhead generated by the runtime argument parsing and calling overhead.

Function	Size	t_{constr}	t_{solv}	t_{tot}
Python	$n = 100$	0.03	1.12	1.15
	$n = 300$	0.21	49.65	49.86
	$n = 500$	0.62	299.39	300.01
native C	$n = 100$	0.30	0.96	1.26
	$n = 300$	3.14	48.38	51.52
	$n = 500$	10.86	288.67	299.53
Matlab	$n = 100$	0.21	8.85	9.06
	$n = 300$	2.05	387.26	389.31
	$n = 500$	6.23	1905.67	1911.8

Table 9.2: *Performance comparison of Python, Matlab and native C implementations to solve the linear system* (9.1) *without preconditioning*
The execution times are given in seconds. t_{constr} is the time for constructing the matrix (or reading it from a file in the case of native C). t_{solv} is the time spent in the PCG solver. t_{tot} is the sum of t_{constr} and t_{solv}. Matlab version 6.0 Release 12 was used for these timings.

Tab. 9.2 shows the execution times for the Python, the Matlab and the native C implementation for solving the linear system (9.1). Matlab is not only slower when building the matrix, also the matrix-vector multiplication seems to be implemented inefficiently. Considering t_{solv}, the performance of Python and native C is comparable. The Python overhead is under 4% in this case.

9.4.4 The jdsym module

The *jdsym* module provides an implementation of the JDSYM algorithm (cf. Algorithm 5.7), that is conveniently callable from Python. The module exports a single function called *jdsym*.

jdsym(A, M, K, kmax, tau, jdtol, itmax, linsolver, **keywords) invokes the JDSYM eigenvalue solver (cf. Section 5.1.2). JDSYM computes eigenpairs of a generalised matrix eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x} \tag{9.2}$$

or a standard eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \mathbf{x}, \tag{9.3}$$

where $\boldsymbol{A}$ is symmetric and $\boldsymbol{M}$ is symmetric positive-definite.

Arguments The *jdsym* function has seven mandatory arguments

$\boldsymbol{A}$ This parameter represents the matrix $\boldsymbol{A}$ in (9.2) or (9.3). $\boldsymbol{A}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods.

$\boldsymbol{M}$ This parameter represents the matrix $\boldsymbol{M}$ in (9.2). $\boldsymbol{M}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods. If the standard eigenvalue problem (9.3) is to be solved, the `None` value can be passed for this parameter.

$\boldsymbol{K}$ The $\boldsymbol{K}$ parameter represents a preconditioner object that supplies the *shape* attribute and the *precon* method. If no preconditioner is to used, then the `None` value can be passed for this parameter.

kmax is an integer that specifies the number of eigenpairs to be computed.

tau is a float value that specifies the target value τ. Eigenvalues in the vicinity of τ will be computed.

jdtol is a float value that specifies the convergence tolerance for eigenpairs $(\lambda, \mathbf{x})$. The converged eigenpairs satisfy $\|\boldsymbol{A}\mathbf{x} - \lambda\boldsymbol{M}\mathbf{x}\|_2 < \mathit{jdtol}$.

itmax is an integer that specifies the maximum number of Jacobi-Davidson iterations to undertake.

linsolver is a function that implements an iterative method for solving linear systems of equations. The function *linsolver* is required to conform to the standards mentioned in Section 9.4.3.

The remaining (optional) arguments are specified using keyword arguments:

jmax is an integer that specifies the maximum dimension of the search subspace. (default: 25)

jmin is an integer that specifies dimension of the search subspace after a restart. (default: 10)

blksize is an integer that specifies the block size used in the JDSYM algorithm. (default: 1)

blkwise is an integer that affects the convergence criterion if $\mathit{blksize} > 1$ (cf. Section 5.1.4). (default: 0)

V0 is NumPy array of rank one or two. It specifies the initial search subspace. (default: a randomly generated initial search subspace)

optype is an integer specifying the operator type used in the correction equation. If *optype* $= 1$, the non-symmetric version is used. If *optype* $= 2$, the symmetric version is used. See Section 5.1.2 for more information. (default: 2)

linitmax is an integer specifying the maximum number steps taken in the inner iteration (iterative linear solver). (default: 200)

eps_tr is a float value setting the tracking parameter ε_{tr} described in Section 5.1.2. (default: 10^{-3})

toldecay is a float value, that influences the dynamic adaption of the stopping criterion of the inner iteration. *toldecay* corresponds to the value γ in Section 5.1.2. (default: 1.5)

clvl is an integer specifying the "verbosity" of the *jdsym* function. The higher the *clvl* parameter, the more output is sent to the standard output. *clvl* $= 0$ produces no output. (default: 0)

strategy is an integer specifying shifting and sorting strategy of JDSYM. *strategy* $= 0$ enables the default JDSYM algorithm. *strategy* $= 1$ enables JDSYM to avoid convergence to eigenvalues smaller than τ (cf. the EIGSOLV method described in Section 5.1.5). (default: 0)

projector is used to keep the search subspace and the eigenvectors in a certain subspace. The parameter *projector* can actually be any Python object, that provides a *shape* attribute and a *project* method. The *project* method takes a vector (a rank-1 NumPy array) as its sole argument and projects that vector in-place. This parameter can be used to implement the DIRPROJ and SAUG methods described in Sections 4.2.1 and 4.2.2. (Default: no projection)

Return value The *jdsym* module function returns a tuple with four elements (*kconv*, *lmbd*, *Q*, *it*):

kconv is an integer that indicates the number of converged eigenpairs.

lmbd is a rank-1 NumPy array containing the converged eigenvalues.

$\boldsymbol{Q}$ is a rank-2 NumPy array containing the converged eigenvectors. The *i*-th eigenvector is accessed by `Q[:,i]`.

it is an integer indicating the number of Jacobi-Davidson steps (outer iteration steps) performed.

Example: Maxwell problem The following code illustrates the use of the *jdsym* module. Two matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are read from files. A Jacobi preconditioner from $\boldsymbol{A} - \tau\boldsymbol{M}$ is built. Then the JDSYM eigensolver is called, calculating 5 eigenvalues near 25.0 and the associated eigenvalues to an accuracy of 10^{-10}. We set *strategy* $= 1$ to avoid convergence to the high-dimensional null space of ($\boldsymbol{A}$, $\boldsymbol{M}$).

```
import spmatrix, itsolvers, jdsym, precon

A = spmatrix.ll_mat_from_mtx('edge6x3x5_A.mtx')
M = spmatrix.ll_mat_from_mtx('edge6x3x5_B.mtx')
tau = 25.0

Atau = A.copy()
Atau.shift(-tau, M)
K = precon.jacobi(Atau)

A = A.to_sss(); M = M.to_sss()
k_conv, lmbd, Q, it  = \
      jdsym.jdsym(A, M, K, 5, tau, 1e-10, 150, itsolvers.qmrs,
                  jmin=5, jmax=10, clvl=1, strategy=1)
```

This code takes 33.71 seconds to compute the five wanted eigenpairs. A native C version, using the same computational kernels, takes 35.64 for the same task. We expected the Python version to be slower due to the overhead generated when calling the matrix-vector multiplication and the preconditioner, but surprisingly the Python code was even a bit faster.

9.4.5 The superlu module

The *superlu* module interfaces the SuperLU library to make it usable by Python code. SuperLU is a software package written in C, that is able to compute a LU-factorisation of a general non-symmetric, sparse matrix with partial pivoting [23].

The *superlu* module exports a single function, called *factorize*.

factorize(A, diag_pivot_thresh, drop_tol, relax, panel_size, permc_spec)
The *factorize* module function computes a LU-factorisation of the matrix $\boldsymbol{A}$. All but the first parameter are optional and can be specified using keyword arguments.

$\boldsymbol{A}$ is a *csr_mat* object that represents the matrix to be factorised.

diag_pivot_thresh is a float value in the interval $[0, 1]$ representing the threshold for partial pivoting. *diag_pivot_thresh* $= 0$ corresponds to no pivoting. *diag_pivot_thresh* $= 1$ corresponds to partial pivoting. (default: 1.0)

drop_tol is a float value in the interval $[0, 1]$ representing the drop tolerance parameter. *drop_tol* $= 0$ corresponds to the exact factorisation.
CAUTION: the *drop_tol* has no effect in the current and all older SuperLU releases (versions 2.0 and below). (default: 0.0)

relax is an integer that controls the degree of relaxing supernodes. (default: 1)

panel_size is an integer specifying the maximum number of columns that form a panel. (default: 10)

permc_spec is an integer specifying the matrix ordering used for the factorisation:

0 natural ordering

1 MMD applied to the structure of $\boldsymbol{A}^T\boldsymbol{A}$

2 MMD applied to the structure of $\boldsymbol{A}^T + \boldsymbol{A}$

3 COLAMD, approximate minimum degree column ordering

(default: 2)

The *factorize* function returns an object of type *superlu_context*. This object encapsulates the LU-factors computed during the factorisation.

superlu_context object attributes

shape The *shape* attribute, returns a 2-tuple describing the dimension of the factorised matrix $\boldsymbol{A}$.

nnz The *nnz* attribute returns an integer holding the total number of non-zero entries stored in both the $\boldsymbol{L}$ and the $\boldsymbol{U}$ factors.

superlu_context object methods

solve(b, x, trans='N') The *solve* method accepts two rank-1 NumPy arrays $\mathbf{b}$ and $\mathbf{x}$ of appropriate size and assigns the solution of the linear system

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

to $\mathbf{x}$. If the optional parameter *trans* is set to `'T'`, then the transposed system

$$\boldsymbol{A}^T\mathbf{x} = \mathbf{b}$$

is solved instead.

Example: 2D Poisson matrix Let's now solve the 2D Poisson system

$$\boldsymbol{L}\mathbf{x} = \mathbf{1},$$

using a factorisation. $\boldsymbol{L}$ is the 2D Poisson matrix, introduced in Section 9.4.1 and $\mathbf{1}$ is a vector with all one entries.
The Python solution for this task looks as follows:

```
import Numeric, spmatrix, superlu
n = 100
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
LU = superlu.factorize(L.to_csr(), diag_pivot_thresh=0.0)
LU.solve(b, x)
```

The code makes use of the Python function `poisson2d_sym_blk`, which was defined in Section 9.4.1.

9.5 The PyFemax package

PyFemax is a set of Python modules which allow to compute eigensolutions of the time-harmonic Maxwell equations (Problem II) using vector finite elements. These modules have been designed to be efficient both in terms of memory and execution time.

PyFemax builds on top of the *Numeric* and the *PySparse* packages (cf. Sections 9.3 and 9.4).

PyFemax consist of several modules which will be briefly introduced in the following sections. We will not discuss every single function defined in these modules. Instead we will only give a short description of the module and mention the functions needed for building a Maxwell eigensolver.

Tab. 9.3 shows size of the source code of the Python implementation (consisting of PySparse and PyFemax) and the native C implementation. The numbers show, that the Python implementation is considerably leaner, despite the fact, that it is designed to be reusable and that it contains a fair amount of boiler plate code to implement Python interfaces.

Language	*PyFemax and PySparse*	*Native implementation*
Fortran 90		6300
Fortran 77		2800
C	7500	2700
Python	1200	
Total	8700	11800

Table 9.3: *Code size of Python and native implementation*
The numbers given are lines of code, including comment. For the native implementation, only the parts that are also implemented in PyFemax are counted. The whole native implementation is approximately 33'000 lines long.

9.5.1 The ansysmesh and psimesh modules

The purpose of the *ansysmesh* and *psimesh* modules is to read mesh data from a file stored in the ANSYS or the PSI[30] format. Both modules define a function *read*, which takes a file name as its only argument. The *read* functions return a MeshData object, that contains the read mesh information. The MeshData class is defined in module *mesh*.

9.5.2 The boxmesh module

The *boxmesh* module is used to generate tetrahedral meshes of rectangular box-shaped domains. The domain is first subdivided nel_x times on the x-axis, nel_y times on the y-axis and nel_z times on the z-axis. Thus, the domain is subdivided into $\text{nel}_x\text{nel}_z\text{nel}_z$ equally sized bricks. Each brick is then cut to form 6 tetrahedra of equal volume.

The module *boxmesh* defines a function *generate*, that creates a tetrahedral mesh as stated above. It takes the dimensions of the domain and the subdivisions in each direction as arguments. A third parameter allows the user to treat some of the surfaces as symmetry planes. All arguments are lists of length 3. The *generate* function returns a MeshData object, that contains the generated mesh information. The MeshData class is defined in module *mesh*.

[30] The PSI format is generated using a Fortran77 program written by Dr. Stefan Adam.

The module function *analyticEigvals* computes analytic eigenvalues of the rectangular box-shaped cavity, according to the process outlined in Appendix C.2. The arguments for this function are the same as the ones for *generate*. An additional parameter specifies the number of analytic eigenvalues to compute.

9.5.3 The mesh module

The *mesh* module is usually not directly imported by user-written modules, instead it is imported by all modules that read or generate mesh data. The *mesh* module defines a class MeshData whose purpose is to hold all mesh information, such as node coordinates, information on the connectivity of nodes, edges, faces and tetrahedra, mappings from local to global DOFs, boundary and symmetry-plane data, etc. This data is supposed to be independent of a particular finite element type.

9.5.4 The nedelec module

The purpose of the *nedelec* module is to create the global matrices necessary to solve the matrix eigenvalue problem for the Nédélec finite element discretisation or the mixed Nédélec-Lagrange finite element discretisation.

The creation of the global matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{Y}$, $\boldsymbol{C}$ and $\boldsymbol{H}$ proceeds in several stages.

1. The preprocessing step, implemented by the *preprocess* function
2. The matrix assembly step, implemented by the *assemble* function
3. Construction of the $\boldsymbol{Y}$ matrix, implemented by the *constructY* function
4. Calculation of the $\boldsymbol{C}$ and $\boldsymbol{H}$ matrices.

Steps 3 and 4 are not necessary, if the EIGSOLV method (cf. Section 4.2.5) is to be used.

nedelec module functions

preprocess(meshData, symBCInfo) incorporates the boundary conditions and prepares the data structures necessary for assembling the global finite element matrices. The parameter *meshData* is a MeshData object resulting from either reading a mesh file or generating a mesh. The parameter *symBCInfo* determines what kind of boundary conditions are implemented on the symmetry planes. If bit i is set, then $\mathbf{e} \cdot \mathbf{n} = 0$ is implemented on symmetry plane i, otherwise $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ is implemented.

The *preprocess* function returns a PreprocessData object, which contains information such as the order of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$, the number of columns of matrix $\boldsymbol{C}$ and index arrays that map old DOF numbers to new DOF numbers (the DOF numbers change because some DOFs are eliminated due to the boundary conditions).

assemble(meshData, preprocessData) assembles the global FEM matrices $\boldsymbol{A}$ and $\boldsymbol{M}$. The first argument *meshData* is a MeshData object resulting from either reading a mesh file or generating a mesh. The second argument *preprocessData* is a PreprocessData object generated by *preprocess()*. The function *assemble* returns two *ll_mat* objects representing the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$.

constructY(meshData, preprocessData) constructs the global FEM matrix $\boldsymbol{Y}$ from the edge-node connectivity information. The first argument *meshData* is a MeshData object resulting from either reading a mesh file or generating a mesh. The second argument *preprocessData* is a PreprocessData object generated by *preprocess()*. The function *constructY* returns an *ll_mat* objects representing the matrix $\boldsymbol{Y}$.

Example code The following code shows how to read the mesh data from an ANSYS mesh file and how to generate the matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{Y}$, $\boldsymbol{C}$ and $\boldsymbol{H}$ from it.

```
import ansysmesh, nedelec, spmatrix

meshData = ansysmesh.read('boxcav16x10x3.ans')
preprocessData = nedelec.preprocess(meshData, 0)
A, M = nedelec.assemble(meshData, preprocessData)
Y = nedelec.constructY(meshData, preprocessData)
C = spmatrix.matrixmultiply(M, Y)
H = spmatrix.dot(Y, C)
```

9.5.5 The nedelec_projection module

The *nedelec_projection* module implements the DIRPROJ and SAUG methods for solving the (indefinite) matrix eigenvalue problem. As described in Sections 4.2.1 and 4.2.2 these methods are realised by introducing projections at several places in the JDSYM eigensolver.

The projections are introduced in two ways:

- ▷ by adding the projection to the *precon* method of the preconditioner object.
- ▷ by calling *jdsym* with the optional *projector* parameter and passing an object that invokes the projection, when its *project* method is called.

Both ways of invoking the preconditioner can be incorporated into the same object. The object of type *ProjectedPrecon* is then passed twice to *jdsym*: once as preconditioner and once as projector.

Since the code of the *nedelec_projection* module is very short and because it illustrates, how complex operations can be built from simple building blocks in Python, some portions are given here:

```
class ProjectedPrecon:
    def __init__(self, K, Y, Hsolve, C):
        m, n = C.shape
        self.shape = (m, m)
        self.K = K
        self.Y = Y
        self.Hsolve = Hsolve
        self.C = C
        self.temp0 = Numeric.zeros(m, 'd')
        self.temp1 = Numeric.zeros(n, 'd')
        self.temp2 = Numeric.zeros(n, 'd')
    def precon(self, x, y):
        "Computes y <- (I - Y*inv(H)*C') * inv(K)*x"
        self.K.precon(x, y)
        self.C.matvec_transp(y, self.temp1)
        self.Hsolve.solve(self.temp1, self.temp2)
        self.Y.matvec(self.temp2, self.temp0)
        y -= self.temp0
    def project(self, x):
        "Computes x <- (I - Y*inv(H)*C')*x"
        self.C.matvec_transp(x, self.temp1)
        self.Hsolve.solve(self.temp1, self.temp2)
        self.Y.matvec(self.temp2, self.temp0)
        x -= self.temp0
```

The *dirproj* and *saug* functions, which implement the methods of the respective name, now become really simple.

```
def dirproj(A, M, K, Y, Hsolve, C, kmax, tau, tol, itmax,
            itsolver, **keywords):
    "DIRPROJ method"
    # construct preconditioner that incorporates projection
    Kproj = ProjectedPrecon(K, Y.to_csr(), Hsolve, C.to_csr())
    # add projector object to the keyword arguments for jdsym
    keywords['projector'] = Kproj
    # call jdsym and return results
    args = (A.to_sss(), M.to_sss(), Kproj, kmax, tau, tol,
            itmax, itsolver)
    return jdsym.jdsym(*args, **keywords)
```

The only difference between *dirproj* and *saug* is, that *saug* passes the original preconditioner $\boldsymbol{K}$.

```
def saug(A, M, K, Y, Hsolve, C, kmax, tau, tol, itmax,
         itsolver, **keywords):
    "SAUG method"
    # construct preconditioner that incorporates projection
    Kproj = ProjectedPrecon(K, Y.to_csr(), Hsolve, C.to_csr())
    # add projector object to the keyword arguments for jdsym
    keywords['projector'] = Kproj
    # call jdsym with orig preconditioner and return results
    args = (A.to_sss(), M.to_sss(), K, kmax, tau, tol,
            itmax, itsolver)
    return jdsym.jdsym(*args, **keywords)
```

Section 9.6.1 contains a sample code, which illustrates the use of the *nedelec_projection* module.

9.5.6 The nedelec_ad and nedelec_ad_opt modules

The *nedelec_ad* module implements the AD method for solving the (indefinite) matrix eigenvalue problem. For efficiency reasons some functionality is implemented in C (imported from module *nedelec_ad_opt*).

As described in Section 4.2.3 the AD method transforms the original matrix eigenvalue problem into a positive-definite matrix eigenvalue problem with reduced dimension.

nedelec_ad module functions

ad(preprocessData, A, M, C, Y) prepares the matrices $\boldsymbol{A}$, $\boldsymbol{M}$ and $\boldsymbol{C}$ for use with the AD method. The function computes the indices of the Nedelec DOFs to be eliminated. The corresponding rows and columns of matrices $\boldsymbol{A}$, $\boldsymbol{M}$ and $\boldsymbol{C}$ are then removed. Matrix $\boldsymbol{Y}$ is unaltered. The resulting matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are positive definite.

The *ad* method returns a object holding information about the AD elimination process.

ad_backtransform(adData, Y, luH, C, Qad) transforms the eigenvectors obtained from the AD eigenvalue problem back to eigenvectors of the original problem. *adData* is the object returned by the *ad* method. *luH* is an object for solving linear systems with matrix $\boldsymbol{H}$. *Qad* is the matrix of converged eigenvectors obtained from the AD transformed matrix eigenvalue problem. $\boldsymbol{Y}$ and $\boldsymbol{C}$ are objects representing the corresponding matrices.

The *ad_backtransform* method returns a matrix containing the corresponding eigenvectors of the original indefinite eigenvalue problem.

9.5.7 precon2level module

The *precon2level* module implements the four variants of the two-level preconditioner described in Section 8.3. The preconditioner is implemented as Python class named *prec2lev*.
The constructor of class *prec2lev* is called as follows:

prec2lev(type, A, n11, A11solve=None)

type selects the variant of the two-level preconditioner to be built. *type* can be one of *prec2lev.JACjac*, *prec2lev.JACssor*, *prec2lev.SGSjac*, *prec2lev.SGSssor*.

$\boldsymbol{A}$ is an *ll_mat* object representing the matrix to precondition.

n11 is an integer representing the order of linear diagonal block, located in the upper left corner of the matrix $\boldsymbol{A}$.

A11solve is an object representing a solver for the (1,1)-block of matrix $\boldsymbol{A}$. This object must provide a method *solve(b,x)* which computes $\mathbf{x} \leftarrow \boldsymbol{A_{11}}^{-1}\mathbf{b}$. If this parameter is not specified, then a direct method is used, implemented by the *superlu* module.

The usage of the *precon2lev* module is illustrated in Section 9.6.

9.5.8 postprocess module

The *postprocess* module provides a set of functions for evaluating and processing the computed eigensolutions.

printModes(lmbd, Q, A, M) lists computed eigenvalues, the associated frequencies measured in MHz and the eigenpair residuals.

lmbd and $\boldsymbol{Q}$ are the objects containing the converged eigenvalues and eigenvectors. $\boldsymbol{A}$ and $\boldsymbol{M}$ are the matrices forming the generalised eigenvalue problem.

The frequencies listed by *printModes* are only correct, if the node coordinates in the mesh data are given in metres.

exportVtk(meshData, preprocessData, lmbd, Q, selection, fileName) exports selected eigensolutions to a VTK data file.

meshData is an object holding the mesh data as defined in the *mesh* module, *preprocessData* is the object returned by the *nedelec.preprocess* function. *lmbd* and $\boldsymbol{Q}$ are objects containing the converged eigenvalues and eigenvectors and *selection* is an integer array holding the indices of the eigensolutions to be exported. If *selection = range(len(lmbd))*, then all solutions are exported. *fileName* is the name of the file the VTK data is written to.

The exported VTK data can be viewed using the free MayaVi scientific data visualiser [58] available at `http://mayavi.sourceforge.net`. The VTK data format is described in [63].

9.5.9 The nedelec_elmat and nedelec_elmat_opt modules

The *nedelec_elmat* and *nedelec_elmat_* modules are responsible for computing the element matrices $\boldsymbol{A}^{(e)}$ and $\boldsymbol{M}^{(e)}$. The most time consuming calculations were moved to the *nedelec_elmat_* module, which is written in C. Both modules are usually not directly imported by the user.

9.5.10 The nedelec_eval and nedelec_eval_opt module

The *nedelec_eval* module provides functions for evaluating the electric field and its curl in a tetrahedral element. For efficiency reasons some functionality is implemented in C (imported from module *nedelec_eval_opt*). Both modules are usually not directly imported by the user.

9.6 Experimental results

We illustrate the use of the PySparse and PyFemax packages by solving two Maxwell problems called *SwiftTest* and *MonsterTest*. The aim of these experiments is to show the clarity and easy of use of the Python implementation and to estimate the efficiency of the Python implementation by comparing timings with the native Fortran/C implementation and also to show the reliability of the Python implementation by solving a really huge problem.

9.6.1 SwiftTest

The SwiftTest solves a Maxwell problem of considerable size (mesh *cop20k*) using the SAUG method and the two level preconditioner. This is the fastest approach we know of to compute eigensolutions of the above mesh (cf. Section 4). The same problem is also solved with the native Fortran/C application using the same settings. The performance of both implementations is then compared.

The Python code for SwiftTest is the following:

```
import spmatrix, itsolvers, precon, jdsym
import ansysmesh, mesh, nedelec, nedelec_projection, superlu
from precon2level import prec2lev

meshData = ansysmesh.read('cop20k.ans')
sigma = 5.0
preprocessData = nedelec.preprocess(meshData, 0x0)

A, M = nedelec.assemble(meshData, preprocessData)
Y = nedelec.constructY(meshData, preprocessData)
C = spmatrix.matrixmultiply(M, Y)
H = spmatrix.dot(Y, C)

Asigma = A[:,:]
Asigma.shift(-sigma, M)
K = prec2lev(prec2lev.JACssor, Asigma,
             preprocessData.nMatNedLin)

Hlu = superlu.factorize(H.to_csr(), diag_pivot_thresh=0)

k_conv, lmbd, Q, it = \
     nedelec_projection.saug(A, M, K, Y, Hlu, C, 10, 0.0, 1e-6,
                             300, itsolvers.qmrs,
                             jmin=10, jmax=20, eps_tr=1e-3,
                             toldecay=1.5, linitmax=500)
```

Function	*Time for PyFemax*	*Time for native implementation*
Reading mesh data	59.5	1.4
Constr. of global matrices	89.1	66.3
Constr. of preconditioner	30.9	26.8
Solution time	1837.9	1590.5
Total time	2017.4	1685.0

Table 9.4: *Performance comparison of PyFemax and native implementation*
All execution times are given in seconds.

The results in Tab. 9.4 show that the native implementation is faster than the Python implementation.

Especially reading and processing the mesh data is slow using PyFemax. The reason is, that the *ansysmesh* module is implemented entirely in Python, whose file I/O operations are quite slow. *ansysmesh* could be implemented as a C extension module to improve this. We feel that this is not worth the effort, since compared to the solution time, the benefit would almost be negligible.

The construction of the global matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{Y}$, $\boldsymbol{C}$, $\boldsymbol{H}$ and $\boldsymbol{H}^{-1}$ is quite fast using PyFemax, when taking into account that the majority of the code is interpreted.

The construction of the preconditioner is dominated by the factorisation of the matrix $\boldsymbol{A}_{11}$. So the performance of PyFemax and the native implementation is expected to be roughly the same.

The execution time of the eigensolver is 15% slower in PyFemax. This has several reasons:

- ▷ In PyFemax, each time a matrix-vector multiplication, a preconditioner or a projection is invoked, the corresponding method has to be called via Python. The arguments of the method have to be checked for the correct type at runtime. The C arrays of type `double *` have to be converted to NumPy arrays. In the native implementation, a simple function call is necessary, which is of course much faster.

- ▷ The SAUG method is realised using the *ProjectedPrecon* class in the *nedelec_projection* module which is written in Python. So, in each inner iteration, a (small) piece of Python code has to be interpreted.

- ▷ The matrix-vector multiplication and the SSOR preconditioner of the native implementation are written in Fortran 77, which is usually still a bit faster than the corresponding C version, that was used for implementing the *spmatrix* module.

So, comparing the total execution times, PyFemax is 20% slower than the native implementation. That is the price we paid for having a clean, concise, well modularised code, that can also be used interactively.

9.6.2 MonsterTest

The MonsterTest solves a really huge Maxwell problem (it's actually the largest problem we solved so far with our software). First, a tetrahedral mesh is generated for the rectangular box cavity. The cuboid is divided into $88 \times 56 \times 13$ bricks, which in turn are divided into 6 tetrahedra each, resulting in a total of 384384 tetrahedra.

The resulting global FEM matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ are of order 2366746. The number of stored non-zero values is 24.6 and 46.8 millions (only the lower triangle is stored).

To save memory the EIGSOLV method was used for avoiding convergence to zero eigenvalues. Also for memory reasons, the SSOR preconditioner was employed. The correction equation was solved using QMRS. This combination of methods represents the best choice in consideration of the given memory constraints. As can be seen in the code below, we freed the memory for the objects `A`, `M`, `meshData` and `preprocessData` as soon as they were no longer needed, to save storage space.

The code uses a total of 2.4 Gbytes memory for this problem. The matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ use together 823.1 Mbytes, the preconditioner uses another 536 Mbytes and the rest is mostly taken up by the eigenvectors and temporary matrices in JDSYM.

The code shown below took 4 days and 13 hours of CPU time to compute the requested 10 eigensolutions to an accuracy of 10^{-6} on the HP Superdome machine[31]. The calculation was done in 116 Jacobi-Davidson iteration steps with an average of 321 inner iteration steps per outer iteration step.

The MonsterTest demonstrates, that our methods and software (including the Python implementation) are well suited for large scale computations.

[31] The HP Superdome machine (stardust) was chosen for its large memory (cf. Appendix B). Only one processor was used for this experiment.

```
import boxmesh, nedelec, itsolvers, precon, jdsym

meshData = boxmesh.generate([5.2, 3.3, 0.77], [88, 56, 13],
                            [0, 0, 0])
sigma = 1.2

preprocessData = nedelec.preprocess(meshData, 0x0)
A, M = nedelec.assemble(meshData, preprocessData)

del meshData, preprocessData

Asigma = A[:,:]
Asigma.shift(-sigma, M)
Asigma_sss = Asigma.to_sss(); del Asigma
K = precon.ssor(Asigma_sss, 1.2, 1)

A_sss = A.to_sss(); del A
M_sss = M.to_sss(); del M

k_conv, lmbd, Q, it = \
    jdsym.jdsym(A_sss, M_sss, K, 10, sigma, 1e-6, 150,
                itsolvers.qmrs, jmin=10, jmax=20, eps_tr=1e-3,
                toldecay=2.0, linitmax=300, strategy=1)
```

10 Conclusions and future work

In this final chapter we restate our aims and then summarise the most important results from the earlier chapters. We also mention some drawbacks of our methods and speculate how they might be improved.

Aims

The aim of this project was to develop software for the efficient and accurate computation of electromagnetic oscillations in cavities of particle accelerators. We tried to find answers to the following questions:

- ▷ Which type of finite element discretisation is best suited to solve the Maxwell eigenvalue problem?
- ▷ How can the indefinite eigenvalue problem stemming from the discretisation with Nédélec vector finite elements be solved efficiently?
- ▷ How does the performance of JDSYM compare to the performance of the well-established IRL method for our symmetric generalised eigenvalue problems?
- ▷ How big is the performance penalty that has to be paid if the mixed-language programming approach with Python is used for solving the Maxwell eigenvalue problem?

Results

In Chapter 3 we studied both nodal finite elements and Nédélec vector finite elements to discretise the Maxwell eigenvalue problem. We came to the conclusion that Nédélec vector finite elements are better suited for this kind of calculations, because they prevent spurious modes and permit an easy incorporation of the boundary conditions. In addition, Nédélec vector finite elements yield accurate solutions for cavities of arbitrary shape. Node elements can only be used safely for a very limited set of cavities.

In Chapter 4 we studied ways to solve the matrix eigenvalue problems arising from the finite elements discretisation. The symmetric positive-definite eigenvalue problem constructed using node finite elements can be solved efficiently using a shift-and-invert based approach. For the symmetric indefinite eigenvalue problem we developed a number of methods for computing the desired smallest *positive* eigenvalues and the associated eigenvectors. The SAUG and AD methods, which both exploit the knowledge of a sparse basis of the null space of matrix $\boldsymbol{A}$, turned out to be very efficient. The EIGSOLV method, which is a modification of the ARPACK and JDSYM eigensolvers, is less efficient but requires less memory than the other two methods. The numerical experiments presented in this chapter clearly show, that JDSYM is considerably faster than ARPACK in all cases.

In Chapter 5 we showed, how the Jacobi-Davidson algorithm can be adapted to solve the symmetric generalised eigenvalue problem (JDSYM algorithm) efficiently. In addition we modified JDSYM to avoid convergence to zero eigenvalues. We pointed out, that it is hard to choose JDSYM's parameters optimally. The best choice depends on the problem and the mesh size.

In Chapter 7, we derived an upper bound for the performance of the sparse matrix-vector product and showed that straight-forward implementations perform poorly. The three optimisation techniques we presented try improve the instruction-level parallelism and the cache

hit-rate. They increase the performance by up to 151%. Our message-passing implementation also benefits from these optimisations and scales reasonably.

In Chapter 8 we analysed a number of preconditioners. The two-level hierarchical basis preconditioner turned out to be very effective. With this preconditioner the inner system can be solved in only a few iteration steps. Moreover, the number of inner iterations seems to be independent of the mesh size. The drawback of the two-level hierarchical basis preconditioner is, that it requires a factorisation of the upper left diagonal block of the matrix $\boldsymbol{A} - \sigma \boldsymbol{M}$ (the block corresponding to the basis functions of the linear finite element). For very large eigenproblems, where this factorisation is impractical, we use a standard SSOR preconditioner instead of the two-level preconditioner, which turns out to be surprisingly effective.

The redesign of the finite element application using the mixed-language programming approach (using Python and C) was successful. The sparse linear algebra package (PySparse) we developed is easy to use but also very fast. The experimental results in Chapter 9 illustrate that we get the performance of a native implementation combined with the ease-of-use of Matlab code.

With our finite element package PyFemax, it is possible to write the entire finite element application in a few lines of code, while providing full flexibility to steer the computation using Python code. According to our experiments, the Python implementation is about 10%–20% slower than the native implementation.

Conclusion

We developed efficient and stable methods for computing the desired eigensolutions. Our software is able to handle large eigenvalue problems with an order over two millions on a single-processor workstation. Using the Python implementation our methods can be used, combined and extended easily.

Suggestions for future work

During this project it turned out, that the most effective methods (the projection methods described in Chapter 4 and the two-level preconditioner discussed in Chapter 8) require an accurate solution of a linear system with a large sparse matrix. In our implementation these systems are solved directly. However, for very large grids the required matrix factorisations are impractical. The triangular factors become so large, that they occupy a multiple of the memory taken up by the global FEM matrices.

Future work should aim at replacing those factorisations by a method that requires less memory, but is efficient and still as accurate as a direct solution. Such a method will most likely be an iterative method, which must be very efficient and converge in only a few steps since it is called in each iteration of the linear solver. An algebraic multigrid approach should be investigated. This would turn the present two-level preconditioner into a multilevel preconditioner. The matrix factorisations required when using algebraic multigrid are much smaller than those necessary for a direct solution.

In the course of this project, other inverse-free eigensolvers gained some popularity. E.g., in [2] Arbenz compares the BRQMIN [52] and the LOBPCG [47] eigensolvers with the JDSYM eigensolver. The performance of BRQMIN and LOBPCG is very promising. The suitability of these eigensolvers for solving large scale Maxwell eigenvalue problems should be further examined.

A Storage formats for sparse matrices

This paragraph describes the various memory storage formats for the sparse matrices we came across during this project.

A.1 CSR Compressed Sparse Row Format

The Compressed Sparse Row format is the format we generally use for non-symmetric matrices. Its data structure consists of three arrays.

va The double precision array *va* of length n_{nz} contains the non-zero entries of the matrix, stored row by row

ja The integer array *ja* of length n_{nz} contains the column indices of the non-zero entries stored in *va*

ia The integer array *ia* of length $n + 1$ contains the pointers (indices) to the beginning of each row in the arrays *va* and *ja*. The last element of *ia* has the value $n_{\text{nz}} + 1$.

Here n is the number of rows of the matrix and n_{nz} is the number of its non-zero entries. Even though the order of the entries is not prescribed in this format, we sort the entries of each row by ascending column indices. This enables us to use more efficient algorithms for certain operations.

A.2 CSC Compressed Sparse Column Format

The Compressed Sparse Column Format is identical to the Compressed Sparse Row Format with the difference that the matrix is stored column-by-column instead of row-by-row. In other words, a matrix $\boldsymbol{A}$ in CSR format is identical to the matrix $\boldsymbol{A}^T$ in CSC format.

If a general matrix has more rows than columns (e.g. $\boldsymbol{C}$ and $\boldsymbol{Y}$), we prefer storing it in CSC format rather than in CSR format. In this way the performance of the matrix-vector multiplication tends to be slightly better, because of increased number of iterations in the inner loops.

A.3 SSS Sparse Symmetric Skyline Format

The SSS format is closely related to the CSR format. It is used for sparse *symmetric* matrices. The diagonal is stored in a separate (full) vector and the strict lower triangle is stored in CSR format:

va The double precision array *va* of length n_{nz} contains the non-zero entries of the strict lower triangle, stored row by row

ja The integer array *ja* of length n_{nz} contains the column indices of the non-zero entries stored in *va*

ia The integer array *ia* of length $n + 1$ contains the pointers (indices) to the beginning of each row in the arrays *va* and *ja*. The last element of *ia* has the value $n_{\text{nz}} + 1$.

da the double precision array *da* of length n stores all diagonal entries of the matrix.

Here n is the order of the matrix and n_{nz} is the number of non-zero entries in the strict lower triangle.

We sort the entries of each row by ascending column indices, like we do with the CSR format.

The SSS format has the advantage over the CSR format, that it requires roughly half of the storage space and that the matrix-vector multiplication can be implemented more efficiently (cf. Section 7.1).

We store the global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ in SSS format.

A.4 LL Linked List Format

The CSR, CSC and SSS storage formats described above are all memory-efficient and well suited for doing matrix-vector multiplication. However, they are ill-suited adding new non-zero entries or removing existing non-zero entries, because large amounts of memory would have to be moved. E.g. for assembling the global matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ all these storage formats are unfit.

For such operations we propose the *Linked List Format* matrix storage format. The non-zero entries of each matrix row are stored as a linked list, sorted by ascending column index.

val The double precision array *val* of length n_{alloc} contains the non-zero entries of matrix.

col The integer array *col* of length n_{alloc} contains the column indices of the non-zero entries stored in *val*

link the integer array *link* of length n_{alloc} stores the pointer (index) to the next non-zero entry of the same row. A value of -1 indicates that there is no next entry.

root The integer array *root* of length n contains the pointers to the first entry of each row. The other entries of the same row can be located by following the *link* array.

free The integer *free* points to the first entry of the *free list*, i.e. a linked list of unoccupied spots in the *val* and *col* arrays. This list is populated when non-zero entries are removed from the matrix.

Here n is the number of rows of the matrix and n_{alloc} is number of allocated elements in the arrays *val*, *col* and *link*. Note that the number of stored non-zero entries is less or equal to n_{alloc}.

Adding a new non-zero entry is rather cheap: It requires a linear search through the corresponding linked list. If an entry with matching column index is found that entry is updated. If the search was not successful, a new entry is created either by filling an empty spot from the free list or by adding a new element at the end of the *val*, *col* and *link* arrays.

Removing a non-zero entry also requires a linear search through the corresponding linked list followed by some cheap linked list manipulation.

The linear searches can be aborted early, since the entries of each row are stored (linked) with increasing column index.

The *val*, *col* and *link* arrays can be dynamically enlarged to make place for new entries. The total number of non-zero entries does not have to be known beforehand.

If a symmetric matrix is to be stored in the LL format, we only store its lower triangular part to conserve memory.

These properties make LL the ideal storage format for assembling global finite element matrices and other matrix operations, such as e.g. the matrix-matrix multiplication.

Once the construction of a matrix in LL format is complete, the matrix can be converted to other formats which support faster matrix-vector multiplications (like CSR, CSC or SSS).

B Machines

B.1 Sun Enterprise E3500 (zuse)

This shared-memory multiprocessor machine is the application server at the Institute of Scientific Computing at ETH Zürich. We used this machine for most *sequential* numerical experiments.

System information

Shared memory MIMD machine
6 CPUs
3 GB shared memory

CPUs	336 MHz SUN UltraSPARC-II
Architecture	Superscalar SPARC Version 9, UltraSPARC
Cache	1st level: 16 KB instruction, and 16 KB data, on chip
	2nd level: 4 MB, external

System interconnect

84 MHz Sun Gigaplane	
Bandwidth	2.68 GB/s

B.2 DEC Alpha Workstation (darwin)

This single processor workstation installed at the Institute of Scientific Computing at ETH Zürich was used for benchmarking the optimised sparse matrix-vector multiplication described in Chapter 7.

System information

Digital AlphaStation 500/500
512 MB memory

CPU	Alpha 21164 EV 5.6
On-chip cache	primary: 8 KB instruction, and 8 KB data
	secondary: 96 KB 3-way data and instruction
External cache	8 MB data and instruction
Operating system	Digital UNIX v.4.0d

B.3 HP Exemplar X-Class (sella)

The HP Exemplar X-Class machine was a shared memory super computer installed at ETH Zürich, used mainly as an application server. It was operational from May 1997 until April 2000. We used this machine to measure the performance of the parallel sparse matrix-vector multiplication as described in Section 7.3.

System information

HP Exemplar SPP2000/X-32 system
Shared memory ccNUMA MIMD machine
32 processors (2 hypernodes with 16 CPUs each)
8 GB shared memory (4 GB per hypernode)

CPUs	HP PA-8000
	PA-RISC 2.0 architecture (64-bit)
	180 MHz clock cycle
	720 Mflops peak performance
	1-level, direct mapped cache, 1 MB instruction, 1MB data
Interconnect	
Intranode	8×8 crossbar-switch (2 CPUs per port)
	960 MB/s per port in both directions
	15.3 GB/s total bandwidth
Internode	CTI (derived from SCI)
Operating system	SPP-UX

B.4 HP Exemplar V-Class (tornado)

The HP Exemplar V-Class machine is the successor of the HP Exemplar X-Class system installed at ETH Zürich since April 2000. We used sella for benchmarking the parallel sparse matrix-vector multiplication described in Section 7.3.

System information

HP Exemplar V2500 SCA system
Shared memory ccNUMA MIMD machine
32 processors (2 hypernodes with 16 CPUs each)
16 GB shared memory (8 GB per hypernode)
252 GB disk storage

CPUs	HP PA-8500
	PA-RISC 2.0 architecture (64-bit)
	440 MHz clock cycle
	1760 Mflops peak performance
	one level, direct mapped cache
	512 KB instruction, 1 MB data
Interconnect	
Intranode	8×8 crossbar-switch (2 CPUs per port)
	960 MB/s per port in both directions
	15.36 GB/s total bandwidth
Internode	CTI (derived from SCI)
Operating system	HP-UX 11.10 64 bit

B.5 HP-Superdome (stardust)

The HP-Superdome machine is the successor of the HP Exemplar V-Class system (tornado) installed at ETH Zürich since April 2001. We used stardust for benchmarking the parallel

sparse matrix-vector multiplication described in Section 7.3. The system information below reflects the configuration at the time these experiments where conducted. It is now outdated.

System information

Shared memory ccNUMA MIMD machine
System topology 2 cabinets
6 cell boards per cabinet
4 CPUs and 4 GB memory in each cell board
Totalling 48 processors and 48 GB memory
360 GB disk storage
CPUs HP PA-8600
PA-RISC 2.0 architecture (64-bit)
552 MHz clock rate
2.2 Gflops peak performance
on-chip 1.5 MB ECC cache
Interconnect between cell boards
via a mesh of 4×4 crossbar switches
6.4 GB/s per to crossbar mesh
38.4 GB/s total
Interconnect in cell board
via cell controller ASIC
6.4 GB/s cell controller to CPU bandwidth
3.2 GB/s cell controller to memory bandwidth
Operating system HP-UX 11.11 64 bit

B.6 Intel Paragon

The Intel Paragon at ETH Zürich is the oldest computer, that we used for our parallel experiments. Because the network capacity is very good compared to the single processor performance, parallel algorithms usually scale well on this machine. The Intel Paragon at ETHZ was operational from July 1994 until September 1999.

System information

Paragon XP/S 22 MP
Distributed memory MIMD machine
150 compute nodes, 6 service nodes, 1 boot node
64 MB memory per node, ∼10 GB total memory
2 compute CPUs and 1 communication CPU per node
nodes are capable of doing computation and communication in parallel
48 GB total disk storage
CPUs Intel i860XP
Clock rate 50 MHz
Peak performance 75 Mflops
Interconnect 2D mesh, wormhole routing
Startup time 65 μs
Peak bandwidth 167 MB/s
Operating system Paragon OSF/1

B.7 IBM SP/2

The IBM SP/2 at ETH Zürich is a distributed memory machine mainly used for molecular dynamics simulations. We used this machine for benchmarking the parallel sparse matrix-vector multiplication described in Section 7.3.

System information

IBM SP2 large scale server
64 node distributed memory MIMD machine
256 MB memory per node (total memory: 16 GB)
1 CPU per node
nodes are capable of doing computation and communication in parallel

CPUs	POWER2 SC
Clock rate	160 MHz
L1 cache	128 KB data, 32 KB instruction
L2 cache	n/a
performance	peak: 640 Mflops
	Linpack 100: 315 Mflops
Interconnect	multistage network using 4×4 crossbar switches as elements
Bandwidth	110 MB per link
Startup time	31 microseconds
Operating system	AIX

B.8 Linux Workstation Cluster (asgard)

Asgard is a large (Beowulf) workstation cluster consisting of Intel Linux nodes installed at ETH Zürich. We used asgard for benchmarking the parallel sparse matrix-vector multiplication described in Section 7.3. The system information below reflects the configuration at the time these experiments where conducted. It is now outdated.

System information

Workstation cluster overview
- 3 login nodes
- 1 file server node
- 251 compute nodes

Compute node specs
- Dual Pentium Intel N440BX mainboard
- 100 MHz system bus
- 2 Intel Pentium CPUs
- 1 GB main memory
- 6 GB local hard disk

Cluster topology	grouped into frames of 24 nodes
Interconnect	Ethernet technology
Intra-frame	100Mbit/s switched Ethernet
Inter-frame	1 Gbit/s switched Ethernet
CPU specs	Intel Pentium III

Clock rate	500 MHz
L1-cache	16 KB data, 16 KB instruction
L2-cache	512 KB
Operating system	Linux
Distribution	SuSE Linux 6.3
kernel	2.2.13-SMP

C Model problems

This appendix describes the analytic solutions of Problem I and Problem II described in Chapters 2 and 3 for a rectangular brick shaped domain.

We use the analytic solutions to validate our methods, algorithms and software. The analytic solutions also help us to quantify the discretisation error of the various finite elements.

C.1 Laplace problem in a cuboid

This section describes the analytic solutions of Problem I

$$\Delta u + \lambda u = 0 \quad u : \mathbb{R}^3 \to \mathbb{R} \qquad \text{on } \Omega \tag{C.1a}$$

$$u = 0 \qquad \text{on } \partial\Omega = \Gamma \tag{C.1b}$$

in a rectangular brick shaped domain (cuboid)

$$\Omega = [0, \pi a] \times [0, \pi b] \times [0, \pi c].$$

The analytic non-trivial eigensolutions of (C.1) are uniquely identified by their *wave mode indices* k_x, k_y and k_z, which are all positive integers. With given k_x, k_y, k_z the corresponding eigenfrequency λ is calculated by

$$\lambda = k_x^2/a^2 + k_y^2/b^2 + k_z^2/c^2$$

and the associated eigenfunctions have the form

$$u = \hat{u} \sin(k_x x/a) \sin(k_y y/b) \sin(k_z z/c).$$

The amplitude $\hat{u}$ can be chosen freely.

C.2 Maxwell's equations in a cuboid

This section describes the analytic solutions of Problem II

$$\mathbf{rot}\,\mathbf{rot}\,\mathbf{e}(\mathbf{x}) = \lambda\,\mathbf{e}(\mathbf{x}), \qquad \mathbf{x} \in \Omega, \tag{C.2a}$$

$$\operatorname{div}\mathbf{e}(\mathbf{x}) = 0, \qquad \mathbf{x} \in \Omega, \tag{C.2b}$$

$$\mathbf{n} \times \mathbf{e} = 0, \qquad \mathbf{x} \in \Gamma. \tag{C.2c}$$

in a rectangular brick shaped domain (cuboid)

$$\Omega = [0, \pi a] \times [0, \pi b] \times [0, \pi c].$$

The analytic solutions of (C.2) are uniquely identified by their *wave mode indices* k_x, k_y and k_z, which are all non-negative integers. With given k_x, k_y, k_z the corresponding eigenfrequency λ is calculated by

$$\lambda = k_x^2/a^2 + k_y^2/b^2 + k_z^2/c^2$$

and the associated eigenfunctions have the form

$$\mathbf{e} = \begin{pmatrix} e_x \cos(k_x x/a) \sin(k_y y/b) \sin(k_z z/c) \\ e_y \sin(k_x x/a) \cos(k_y y/b) \sin(k_z z/c) \\ e_z \sin(k_x x/a) \sin(k_y y/b) \cos(k_z z/c) \end{pmatrix}.$$

The analytic solutions of (C.2) can be divided into three classes depending on the values of the wave mode indices k_x, k_y, k_z:

at least two wave mode indices are equal to zero In this case the electric field $\mathbf{e}$ vanishes. This are trivial solutions, that we are not interested in.

exactly one wave mode index is equal to zero In this case the electric field $\mathbf{e}$ is parallel to a coordinate axis in the entire domain Ω.

Without loss of generality we assume $k_z = 0$. Then the resulting eigenfield has the form

$$\mathbf{e} = \begin{pmatrix} 0 \\ 0 \\ e_z \sin(k_x x/a) \sin(k_y y/b) \end{pmatrix}$$

with

$$\lambda = k_x^2/a^2 + k_y^2/b^2.$$

The amplitude e_z can be chosen freely.

all wave mode indices are greater than zero In this case the amplitude parameters are coupled with the wave mode indices by the divergence free condition $\operatorname{div} \mathbf{e} = 0$. This results in the equation

$$k_x e_x/a + k_y e_y/b + k_z e_z/c = 0.$$

With given k_x, k_y and k_z, *two degrees of freedom* remain to choose the amplitude parameters e_x, e_y and e_z. That means that two linearly independent eigenfunctions belong to the eigenfrequency $\lambda = k_x^2/a^2 + k_y^2/b^2 + k_z^2/c^2$. The eigenvalue λ has multiplicity 2 and the corresponding eigenspace has dimension 2.

For calculating the electro-magnetic fields in cavities of particle accelerators, eigensolutions of the second type are of particular interest. Eigenfields of this type have the smallest eigenfrequencies. Most of the time, eigenfields of this type are used for the acceleration of the particles.

D Grids

This appendix describes the tetrahedral meshes, we used for the numerical experiments. Tab. D.1 contains a list of all grids, together with their properties.

Identifier	# nodes	# edges	# faces	# tetrahedra
box8x4x6	315	1674	2512	1152
boxcav16x10x3	1228	7463	11996	5760
boxcav20x13x3	1176	6573	10078	4680
boxcav30x20x4	3255	19254	30400	14400
boxcav88x56x13	71022	469005	782368	384384
copcav2	125	588	832	320
copcav14	1005	6176	9848	4368
copcav18	1653	10176	16264	7344
cop10k	2023	11758	18370	8634
cop20k	4431	26855	42879	20454
cop40k	7985	49643	80256	38597
cop300k	55210	362945	603474	295738
slac	122	546	751	326
accel	39826	238045	375260	177048

Table D.1: *Properties of tetrahedral meshes*

More detailed information on the meshes and their origin are presented in the following sections.

D.1 Rectangular brick shaped cavity

We use rectangular brick shaped meshes for a number of reasons.

Firstly, the analytical eigensolutions of Problem I and II are known for this kind of domains (cf. Appendix C). Thus these meshes are well-suited for validating our methods and software.

Secondly, in other publications, experimental results are often found for rectangular domains. E.g. in [45] Jin presents results for a rectangular brick shaped cavity with the same aspect-ratio as grid *box8x4x6* shown in Fig. D.2.

Another reason is, that the cyclotron at the Paul Scherrer Institute has accelerating cavities (cf. Fig. D.1), which are approximately cuboid shaped. Grid *boxcav16x10x3* in Fig. D.3 is a model of such a accelerator cavity. Grids of this kind allow us to compare our calculated eigensolutions with real measured data.

For grid *box8x4x6* the cuboid shaped domain $\Omega = (0, 1.0) \times (0, 0.5) \times (0, 0.75)$ was subdivided into $8 \times 4 \times 6$ equally sized cubes. Each cube was then subdivided into 6 tetrahedra of equal volume. For grid *boxcav16x10x3* the domain $\Omega = (0, 5.2) \times (0, 3.3) \times (0, 0.77)$ is subdivided into $16 \times 10 \times 3$ equally sized cuboids. Each cuboid is then subdivided into 12 tetrahedra of equal volume.

Figure D.1: *PSI accelerator cavity, approximately cuboid shaped*

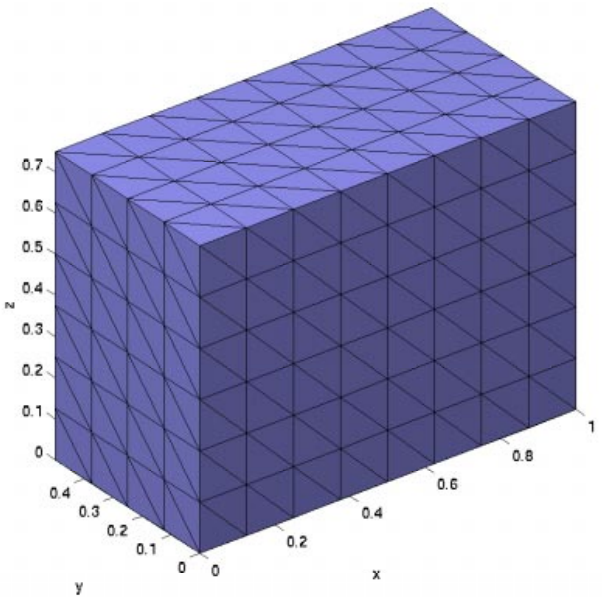


Figure D.2: *Grid for a cuboid shaped cavity* box8x4x6

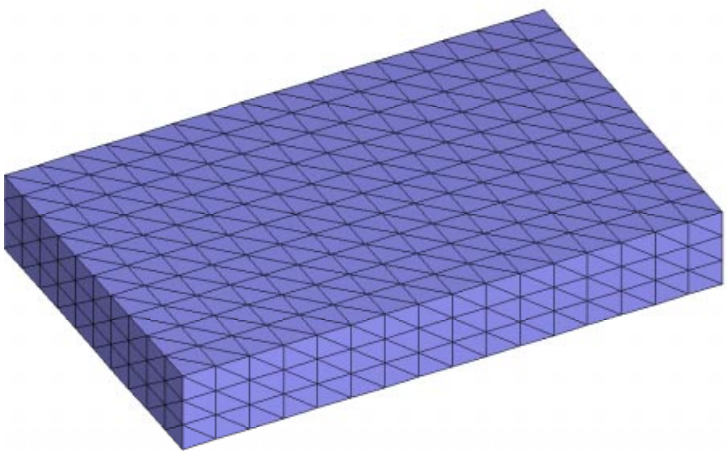


Figure D.3: *Grid for a cuboid shaped cavity* boxcav16x10x3

D.2 Eight-shaped cavity

The “eight-shaped” accelerator cavity is currently being developed at the Paul Scherrer Institute. A smaller prototype (cf. Fig. D.4) has already been built.

The grids *copcav2*, *copcav14* and *copcav18* represent discretisations of this accelerator cavity (cf. Fig. D.5). These grids all have three symmetry planes. The standing electromagnetic waves in such cavities are symmetric with respect to these symmetry planes as well. It is therefore sufficient to discretise only one eighth of the cavity (cf. Figs. D.6 and grids *cop10k*, *cop20k*, *cop40k* and *cop300k*).

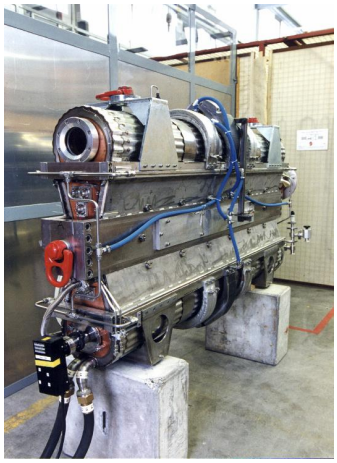


Figure D.4: *PSI model/prototype of the “eight-shaped” accelerator cavity*

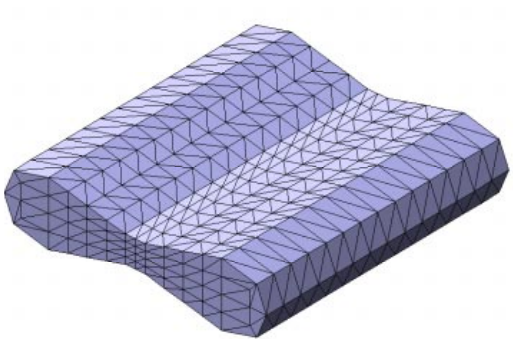


Figure D.5: *Grid* copcav18 *of the “eight-shaped” accelerator cavity*

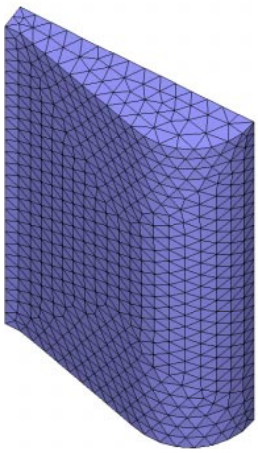


Figure D.6: *Grid* cop10k *of one eighth of the “eight-shaped” accelerator cavity*

D.3 SLAC accelerator cavity

The grid *slac* represents a very coarse discretisation of a cavity installed at the “Stanford Linear Accelerator Center”. This cavity is one element of a so-called linear accelerator. This grid also takes symmetries with respect to three symmetry planes into account.

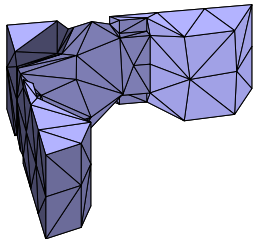


Figure D.7: *Grid* slac, *cavity element of a linear accelerator*

D.4 ACCEL cyclotron

The grid *accel* represents a discretisation of a cyclotron RF-structure to be delivered by ACCEL GmbH for the PSI PROSCAN project. It takes symmetry with respect to one plane into account. Unlike all other grids, *accel* represents a complete RF-structure, not just one isolated cavity.

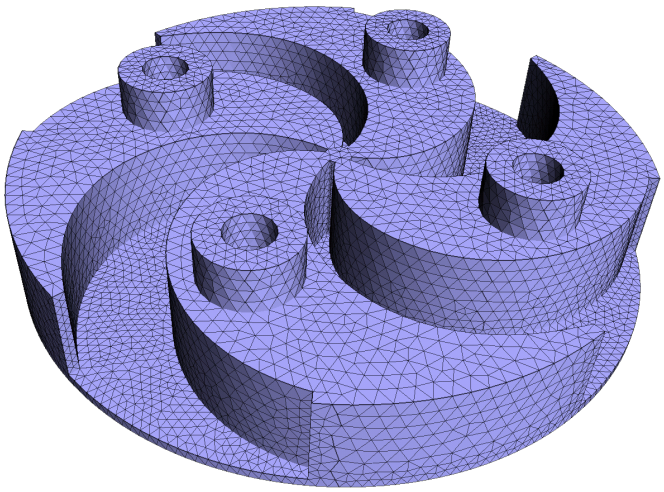


Figure D.8: *Grid* accel, *cyclotron RF-structure*

E Matrices

This appendix describes the global finite element matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{C}$, $\boldsymbol{Y}$ and $\boldsymbol{H}^{-1}$, which were used for the numerical experiments throughout this document.

Tab. E.1 lists the properties for the matrices stemming from rectangular box cavities and Tab. E.2 lists the properties for the matrices stemming from the remaining meshes. The three digit code in the second column specifies the type of finite element used to generate the matrices. The first digit, being *X* in all cases, says that 3D-Maxwell problem was solved. The second digit indicates the type of finite element: It is either *n* for node elements, or *v* for vector elements. The third digit indicates the degree of the finite elements (*1* for linear elements, or *2* for quadratic elements). n is the order of the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$. n_c is the number of columns of the matrices $\boldsymbol{C}$ and $\boldsymbol{Y}$. $\text{nnz}_{\boldsymbol{A},\boldsymbol{M}}$ denotes the number of non-zero entries stored in the matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ together. $\text{nnz}_{\boldsymbol{C},\boldsymbol{Y}}$ denotes the number of non-zero entries stored in the matrices $\boldsymbol{C}$ and $\boldsymbol{Y}$ together. $\text{nnz}_{\boldsymbol{H}^{-1}}$ is the number of non-zero entries stored in the L- and U-factors generated by factorising matrix $\boldsymbol{H}$.

Some of the grids include symmetry planes (cf. Section D). The precise properties of the corresponding matrices depend on the boundary conditions enforced at these symmetry planes. The numbers in Tabs. E.1 and E.2 reflect the case, where $\mathbf{e} \times \mathbf{n} = \mathbf{0}$ is enforced at all symmetry planes. For other combinations of boundary conditions the numbers in Tabs. E.1 and E.2 would differ slightly.

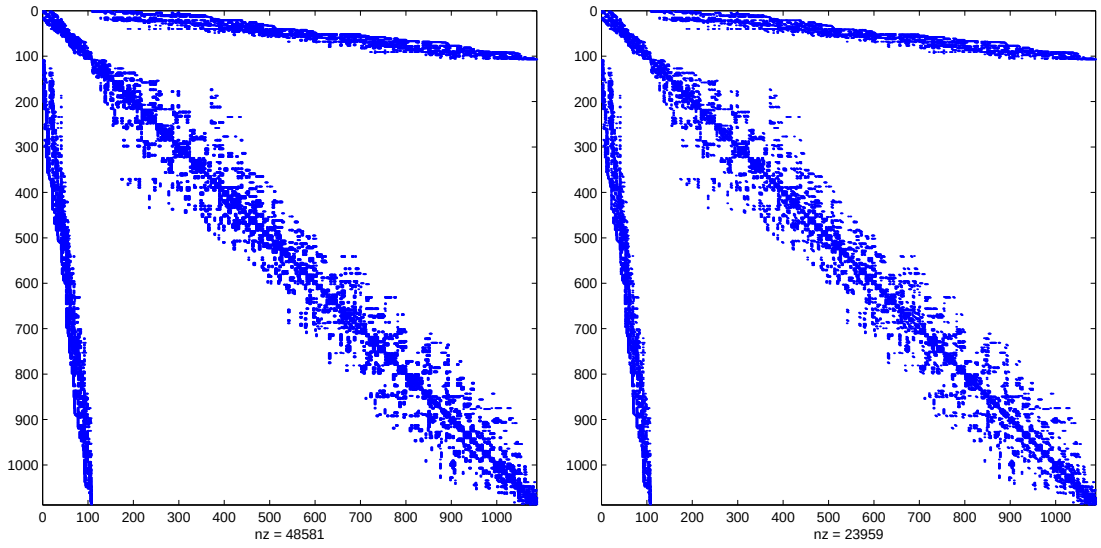
Grid	*Code*	n	n_c	$\text{nnz}_{\boldsymbol{A},\boldsymbol{M}}$	$\text{nnz}_{\boldsymbol{C},\boldsymbol{Y}}$	$\text{nnz}_{\boldsymbol{H}^{-1}}$
box8x4x6	Xn1	457		7.4K		
	Xn2	4159		141K		
	Xv1	1050	105	11K	5.0K	1.8K
	Xv2	6292	1.1K	143K	66K	144K
boxcav16x10x3	Xn1	2.6K		51.5K		
	Xn2	22K		939K		
	Xv1	6.0K	750	78.5K	38.5K	16.0K
	Xv2	34K	6785	1.0M	492K	759K
boxcav20x13x3	Xn1	1948		37K		
	Xn2	17K		676K		
	Xv1	4.4K	456	52K	26K	18K
	Xv2	26K	4.8K	674K	332K	896K
boxcav30x20x4	Xn1	6.3K		139K		
	Xn2	54K		2.3M		
	Xv1	14K	1.6K	183K	97K	129K
	Xv2	83K	16K	2.2M	1.1M	5.9M
boxcav88x56x13	Xn1	185K		4.9M		
	Xn2	1.5M		72M		
	Xv1	428K	57K	5.9M	3.5M	43M
	Xv2	2.3M	485K	71M	N/A	N/A

Table E.1: *Properties of global finite element matrices stemming from rectangular box cavities*

Grid	*Code*	n	n_c	$nnz_{\boldsymbol{A},\boldsymbol{M}}$	$nnz_{\boldsymbol{C},\boldsymbol{Y}}$	$nnz_{\boldsymbol{H}^{-1}}$
cop10k	Xn1	3.7K		81K		
	Xn2	32K		1.5M		
	Xv1	8.4K	919	111K	54K	48K
	Xv2	49K	9.3K	1.4M	697K	2.1M
cop20k	Xn1	9.1K		216K		
	Xn2	77K		3.8M		
	Xv1	20K	2.4K	287K	148K	247K
	Xv2	119K	23K	3.5M	1.8M	10M
cop40k	Xn1	17K		432K		
	Xn2	147K		7.4M		
	Xv1	40K	4921	568K	301K	781K
	Xv2	229K	45K	6.9M	3.5M	30M
cop300k	Xn1	141K		3.7M		
	Xn2	1.1M		61.3M		
	Xv1	326K	43K	4.8M	2.7M	25M
	Xv2	1.8M	370K	60M	N/A	N/A
copcav18	Xn1	3.3K		80K		
	Xn2	29K		1.3M		
	Xv1	7.8K	867	98K	53K	61K
	Xv2	45K	8.6K	1.1M	621K	2.0M
slac	Xn1	107		1145		
	Xn2	1,0K		35.4K		
	Xv1	249	21	2.3K	672	82
	Xv2	1.6K	270	37.5K	14.3K	6.1K
accel	Xn1	106K		2.8M		
	Xn2	778K		43.4M		
	Xv1	197K	26K	3.0M	1.4M	N/A
	Xv2	1.1M	277K	34M	21.3M	N/A

Table E.2: *Properties of global finite element matrices stemming from other than rectangular box meshes*

Fig. E.1 shows the non-zeros structure of the global FEM matrices $\boldsymbol{A}$ and $\boldsymbol{M}$ constructed using quadratic node elements from mesh *slac*. In the upper left corner the (1,1)-block of order 107 is visible. This diagonal block is equal to the entire global matrix obtained if linear node elements were used.

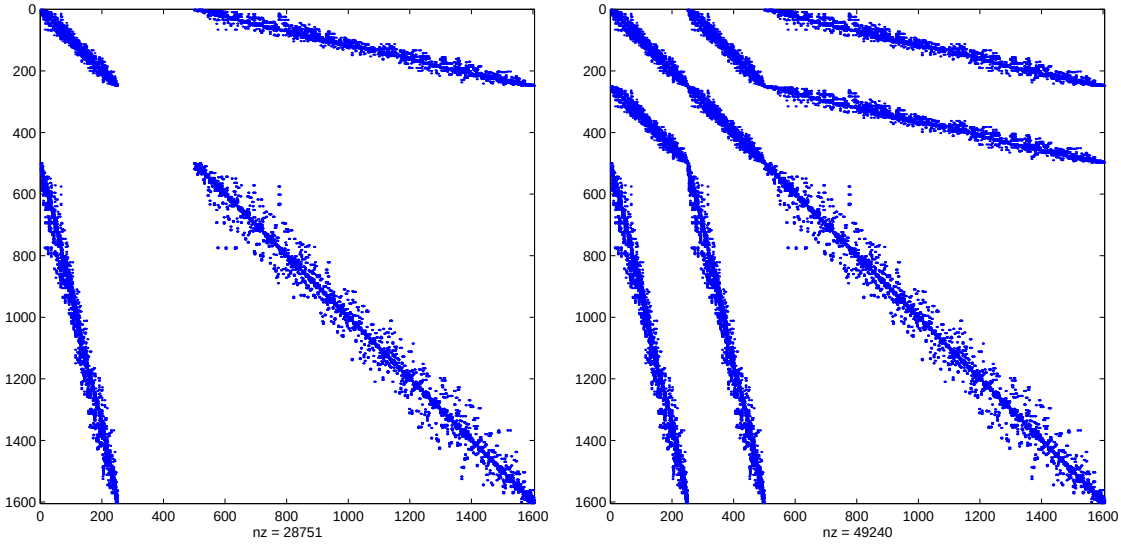


(a) matrix $\boldsymbol{A}$ (b) matrix $\boldsymbol{M}$

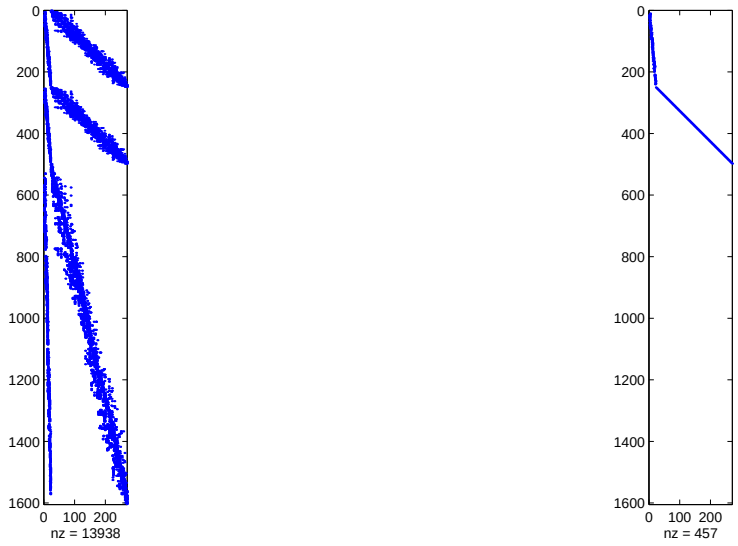
Figure E.1: *Non-zero structure of the global FEM matrices* $\boldsymbol{A}$ *and* $\boldsymbol{M}$ *constructed using quadratic node elements from mesh* slac.

Fig. E.2 shows the non-zero structure of the global FEM matrices $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{C}$ and $\boldsymbol{Y}$ constructed using quadratic edge elements from mesh *slac*. In the upper left corner of $\boldsymbol{A}$ and $\boldsymbol{M}$ the (1,1)-block of order 249 is visible. The (1,1)-block is associated with the basis functions of the linear vector element. The remaining (2,2)-diagonal block is associated with the additional basis functions of the quadratic vector element. The (2,2)-block itself has also a 2×2-block structure, where the upper left diagonal block corresponds to edge DOFs and the lower right diagonal block corresponds to face DOFs.

The columns of the matrices $\boldsymbol{C}$ and $\boldsymbol{Y}$ are associated with the Lagrange DOFs described in Section 3.3.5. The first 23 columns belong to node DOFs while the remaining columns belong to DOFs associated with edge mid-points. $\boldsymbol{Y}$ is extremely sparse. Its non-zero entries are either 1 or -1. The diagonal submatrix of $\boldsymbol{Y}$ as shown in Fig.2(d) is an identity matrix of order 249.



(a) matrix $\boldsymbol{A}$ (b) matrix $\boldsymbol{M}$



(c) matrix $\boldsymbol{C}$ (d) matrix $\boldsymbol{Y}$

Figure E.2: *Non-zero structure of the global FEM matrices* $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{C}$ *and* $\boldsymbol{Y}$ *constructed using quadratic edge elements from mesh* slac.

F Element matrices

This appendix contains detailed information on the element matrices mentioned in Chapters 2.5 and 3.2.2.

F.1 Element matrices for Problem I

The matrix $\boldsymbol{M}^{(\mathbf{0})}$ is used for calculating the mass element matrix $\boldsymbol{M}^{(e)}$. $\boldsymbol{M}^{(\mathbf{0})}$ is defined by

$$\{\boldsymbol{M}^{(\mathbf{0})}\}_{ij} = \iiint_{T_0} N_i^{(0)} N_j^{(0)} \, \mathrm{d}\boldsymbol{\xi}.$$

The matrix displayed below was computed using Maple V.

$$\boldsymbol{M}^{(\mathbf{0})} = \frac{1}{5040} \begin{bmatrix} 84 & 42 & 42 & 42 & 14 & 14 & 14 & 7 & 7 & 7 \\ 42 & 84 & 42 & 42 & 14 & 7 & 7 & 14 & 14 & 7 \\ 42 & 42 & 84 & 42 & 7 & 14 & 7 & 14 & 7 & 14 \\ 42 & 42 & 42 & 84 & 7 & 7 & 14 & 7 & 14 & 14 \\ 14 & 14 & 7 & 7 & 4 & 2 & 2 & 2 & 2 & 1 \\ 14 & 7 & 14 & 7 & 2 & 4 & 2 & 2 & 1 & 2 \\ 14 & 7 & 7 & 14 & 2 & 2 & 4 & 1 & 2 & 2 \\ 7 & 14 & 14 & 7 & 2 & 2 & 1 & 4 & 2 & 2 \\ 7 & 14 & 7 & 14 & 2 & 1 & 2 & 2 & 4 & 2 \\ 7 & 7 & 14 & 14 & 1 & 2 & 2 & 2 & 2 & 4 \end{bmatrix}$$

Due to the hierarchical construction of the basis functions, the matrix used for linear elements is the upper left 4×4 diagonal block.

To calculate the element stiffness matrix, the matrices $\boldsymbol{K}_{\boldsymbol{\xi}}^{11}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{22}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{33}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{12}$, $\boldsymbol{K}_{\boldsymbol{\xi}}^{13}$ and $\boldsymbol{K}_{\boldsymbol{\xi}}^{23}$ as described in Section 2.5 are needed. These matrices are defined by

$$\{\boldsymbol{K}_{\boldsymbol{\xi}}^{kl}\}_{ij} = \iiint_{T_0} \frac{\partial N_i^{(0)}}{\partial \xi_k} \frac{\partial N_j^{(0)}}{\partial \xi_l} \, \mathrm{d}\boldsymbol{\xi}.$$

The matrices displayed below were computed using Maple V.

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{11} = \frac{1}{120} \begin{bmatrix} 20 & -20 & 0 & 0 & 0 & 5 & 5 & -5 & -5 & 0 \\ -20 & 20 & 0 & 0 & 0 & -5 & -5 & 5 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 5 & -5 & 0 & 0 & 0 & 2 & 1 & -2 & -1 & 0 \\ 5 & -5 & 0 & 0 & 0 & 1 & 2 & -1 & -2 & 0 \\ -5 & 5 & 0 & 0 & 0 & -2 & -1 & 2 & 1 & 0 \\ -5 & 5 & 0 & 0 & 0 & -1 & -2 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{22} = \frac{1}{120}\begin{bmatrix} 20 & 0 & -20 & 0 & 5 & 0 & 5 & -5 & 0 & -5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -20 & 0 & 20 & 0 & -5 & 0 & -5 & 5 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & -5 & 0 & 2 & 0 & 1 & -2 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 5 & 0 & -5 & 0 & 1 & 0 & 2 & -1 & 0 & -2 \\ -5 & 0 & 5 & 0 & -2 & 0 & -1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -5 & 0 & 5 & 0 & -1 & 0 & -2 & 1 & 0 & 2 \end{bmatrix}$$

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{33} = \frac{1}{120}\begin{bmatrix} 20 & 0 & 0 & -20 & 5 & 5 & 0 & 0 & -5 & -5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -20 & 0 & 0 & 20 & -5 & -5 & 0 & 0 & 5 & 5 \\ 5 & 0 & 0 & -5 & 2 & 1 & 0 & 0 & -2 & -1 \\ 5 & 0 & 0 & -5 & 1 & 2 & 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -5 & 0 & 0 & 5 & -2 & -1 & 0 & 0 & 2 & 1 \\ -5 & 0 & 0 & 5 & -1 & -2 & 0 & 0 & 1 & 2 \end{bmatrix}$$

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{12} = \frac{1}{120}\begin{bmatrix} 20 & 0 & -20 & 0 & 5 & 0 & 5 & -5 & 0 & -5 \\ -20 & 0 & 20 & 0 & -5 & 0 & -5 & 5 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 \\ 5 & 0 & -5 & 0 & 1 & 1 & 1 & -1 & 0 & -1 \\ 5 & 0 & -5 & 0 & 1 & 0 & 2 & -1 & 0 & -2 \\ -5 & 0 & 5 & 0 & -1 & -1 & -1 & 1 & 0 & 1 \\ -5 & 0 & 5 & 0 & -1 & 0 & -2 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{13} = \frac{1}{120}\begin{bmatrix} 20 & 0 & 0 & -20 & 5 & 5 & 0 & 0 & -5 & -5 \\ -20 & 0 & 0 & 20 & -5 & -5 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 \\ 5 & 0 & 0 & -5 & 1 & 2 & 0 & 0 & -1 & -2 \\ 5 & 0 & 0 & -5 & 1 & 1 & 1 & 0 & -1 & -1 \\ -5 & 0 & 0 & 5 & -1 & -2 & 0 & 0 & 1 & 2 \\ -5 & 0 & 0 & 5 & -1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{K}_{\boldsymbol{\xi}}^{23} = \frac{1}{120}\begin{bmatrix} 20 & 0 & 0 & -20 & 5 & 5 & 0 & 0 & -5 & -5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -20 & 0 & 0 & 20 & -5 & -5 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & -5 & 2 & 1 & 0 & 0 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \\ 5 & 0 & 0 & -5 & 1 & 1 & 1 & 0 & -1 & -1 \\ -5 & 0 & 0 & 5 & -2 & -1 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -5 & 0 & 0 & 5 & -1 & -1 & -1 & 0 & 1 & 1 \end{bmatrix}$$

F.2 Element matrices for Problem II

F.2.1 Element matrices for Nédélec vector elements

For the calculation of the element matrices, the gradients of the simplex coordinates $L_1, \ldots, L_4$[32] are required. These gradients are stored in the vectors $\mathbf{g}_1, \ldots, \mathbf{g}_4$ according to (3.30).

$$\mathbf{g}_i = \nabla L_i.$$

The simplex coordinates are linear functions in x, y and z.

$$L_i = c_{0,i} + c_{1,i}x + c_{2,i}y + c_{3,i}z.$$

From the interpolation condition at the corners $P_j = (x_j, y_j, z_j)$ of the tetrahedron

$$L_i(x_j, y_j, z_j) = \delta_{i,j}$$

follows the matrix equation

$$\begin{pmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{pmatrix} \begin{pmatrix} c_{0,1} & c_{0,2} & c_{0,3} & c_{0,4} \\ c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

[32] $L_1, \ldots, L_4$ are function in x, y, z

The coefficients $c_{j,i}$ can be obtained by inverting the matrix on the left side. The gradients $\mathbf{g}_i$ are then computed from

$$\mathbf{g}_i = (c_{1,i}, c_{2,i}, c_{3,i})^T.$$

The inner products of the $\mathbf{g}_i$ are stored in the matrix $\boldsymbol{G} \in \mathbb{R}^{4\times 4}$ according to Equation 3.31 on page 31. The entries of $\boldsymbol{G}$ are needed for calculating the entries of the mass element matrix $\boldsymbol{M}^{(e)}$.

Calculating the entries of the matrix $\boldsymbol{C} \in \mathbb{R}^{6\times 6}$, which are needed for the curl element matrix $\boldsymbol{A}^{(e)}$ is trivial. It can be done according to Equation 3.35 on page 32.

Mass element matrix The entries of the mass element matrix $\boldsymbol{M}^{(e)}$, with

$$\{\boldsymbol{M}^{(e)}\}_{i,j} = \iiint_{T_e} \mathbf{N}_i^{(e)} \cdot \mathbf{N}_j^{(e)} \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z,$$

are obtained from the formulas below. Due to symmetry reasons only the entries in the upper triangle are given.

$$\begin{aligned}
m_{1,1}^{(e)} &= J/60\,(g_{11} + g_{22} - g_{12}) \\
m_{1,2}^{(e)} &= J/120\,(g_{11} - g_{12} - g_{13} + 2g_{23}) \\
m_{1,3}^{(e)} &= J/120\,(g_{11} - g_{12} - g_{14} + 2g_{24}) \\
m_{1,4}^{(e)} &= J/120\,(-g_{22} + g_{12} + g_{23} - 2g_{13}) \\
m_{1,5}^{(e)} &= J/120\,(-g_{22} + g_{12} + g_{24} - 2g_{14}) \\
m_{1,6}^{(e)} &= J/120\,(g_{13} + g_{24} - g_{14} - g_{23}) \\
m_{1,7}^{(e)} &= J/60\,(g_{22} - g_{11}) \\
m_{1,8}^{(e)} &= J/120\,(-g_{11} + g_{12} - g_{13} + 2g_{23}) \\
m_{1,9}^{(e)} &= J/120\,(-g_{11} + g_{12} - g_{14} + 2g_{24}) \\
m_{1,10}^{(e)} &= J/120\,(g_{22} - g_{12} + g_{23} - 2g_{13}) \\
m_{1,11}^{(e)} &= J/120\,(g_{22} - g_{12} + g_{24} - 2g_{14}) \\
m_{1,12}^{(e)} &= J/120\,(-g_{13} - g_{14} + g_{23} + g_{24}) \\
m_{1,13}^{(e)} &= J/720\,(g_{12} - 2g_{11}) \\
m_{1,14}^{(e)} &= J/720\,(g_{22} - g_{12}) \\
m_{1,15}^{(e)} &= J/720\,(-g_{13} + 2g_{23}) \\
m_{1,16}^{(e)} &= J/360\,(-g_{14} + g_{24}) \\
m_{1,17}^{(e)} &= J/720\,(2g_{22} - g_{12}) \\
m_{1,18}^{(e)} &= J/720\,(-2g_{13} + g_{23}) \\
m_{1,19}^{(e)} &= J/720\,(-g_{14} + 2g_{24}) \\
m_{1,20}^{(e)} &= J/720\,(-2g_{11} + g_{12}) \\
m_{1,21}^{(e)} &= J/720\,(2g_{23} - 2g_{13}) \\
m_{1,22}^{(e)} &= J/720\,(g_{24} - 2g_{14}) \\
m_{1,23}^{(e)} &= J/720\,(g_{12} - g_{11}) \\
m_{1,24}^{(e)} &= J/720\,(2g_{22} - g_{12}) \\
m_{2,2}^{(e)} &= J/60\,(g_{11} + g_{33} - g_{13}) \\
m_{2,3}^{(e)} &= J/120\,(g_{11} - g_{13} - g_{14} + 2g_{34}) \\
m_{2,4}^{(e)} &= J/120\,(g_{33} - g_{13} - g_{23} + 2g_{12}) \\
m_{2,5}^{(e)} &= J/120\,(g_{12} + g_{34} - g_{14} - g_{23}) \\
m_{2,6}^{(e)} &= J/120\,(-g_{33} + g_{13} + g_{34} - 2g_{14}) \\
m_{2,7}^{(e)} &= J/120\,(-g_{11} - g_{12} + g_{13} + 2g_{23}) \\
m_{2,8}^{(e)} &= J/60\,(g_{33} - g_{11}) \\
m_{2,9}^{(e)} &= J/120\,(-g_{11} + g_{13} - g_{14} + 2g_{34}) \\
m_{2,10}^{(e)} &= J/120\,(g_{33} - g_{13} + g_{23} - 2g_{12}) \\
m_{2,11}^{(e)} &= J/120\,(-g_{12} - g_{14} + g_{23} + g_{34}) \\
m_{2,12}^{(e)} &= J/120\,(g_{33} - g_{13} + g_{34} - 2g_{14}) \\
m_{2,13}^{(e)} &= J/720\,(-2g_{11} + g_{13}) \\
m_{2,14}^{(e)} &= J/720\,(-2g_{12} + g_{23}) \\
m_{2,15}^{(e)} &= J/720\,(2g_{33} - g_{13}) \\
m_{2,16}^{(e)} &= J/720\,(-g_{14} + 2g_{34}) \\
m_{2,17}^{(e)} &= J/360\,(-g_{12} + g_{23}) \\
m_{2,18}^{(e)} &= J/720\,(g_{33} - g_{13}) \\
m_{2,19}^{(e)} &= J/360\,(-g_{14} + g_{34}) \\
m_{2,20}^{(e)} &= J/720\,(-g_{11} + g_{13}) \\
m_{2,21}^{(e)} &= J/720\,(2g_{33} - g_{13}) \\
m_{2,22}^{(e)} &= J/720\,(g_{34} - 2g_{14}) \\
m_{2,23}^{(e)} &= J/720\,(g_{31} - 2g_{11}) \\
m_{2,24}^{(e)} &= J/720\,(2g_{23} - g_{12})
\end{aligned}$$

$$m^{(e)}_{3,3} = J/60\ (g_{11} + g_{44} - g_{14})$$
$$m^{(e)}_{3,4} = J/120\ (g_{12} + g_{34} - g_{13} - g_{24})$$
$$m^{(e)}_{3,5} = J/120\ (g_{44} - g_{14} - g_{24} + 2g_{12})$$
$$m^{(e)}_{3,6} = J/120\ (g_{44} - g_{14} - g_{34} + 2g_{13})$$
$$m^{(e)}_{3,7} = J/120\ (-g_{11} - g_{12} + g_{14} + 2g_{24})$$
$$m^{(e)}_{3,8} = J/120\ (-g_{11} - g_{13} + g_{14} + 2g_{34})$$
$$m^{(e)}_{3,9} = J/60\ (g_{44} - g_{11})$$
$$m^{(e)}_{3,10} = J/120\ (-g_{13} - g_{12} + g_{34} + g_{24})$$
$$m^{(e)}_{3,11} = J/120\ (g_{44} - g_{14} + g_{24} - 2g_{12})$$
$$m^{(e)}_{3,12} = J/120\ (g_{44} - g_{14} + g_{34} - 2g_{13})$$
$$m^{(e)}_{3,13} = J/720\ (-g_{11} + g_{14})$$
$$m^{(e)}_{3,14} = J/720\ (-2g_{12} + g_{24})$$
$$m^{(e)}_{3,15} = J/360\ (-g_{13} + g_{34})$$
$$m^{(e)}_{3,16} = J/720\ (2g_{44} - g_{14})$$
$$m^{(e)}_{3,17} = J/720\ (-g_{12} + 2g_{24})$$
$$m^{(e)}_{3,18} = J/720\ (-2g_{13} + g_{34})$$
$$m^{(e)}_{3,19} = J/720\ (2g_{44} - g_{14})$$
$$m^{(e)}_{3,20} = J/720\ (-2g_{11} + g_{14})$$
$$m^{(e)}_{3,21} = J/720\ (2g_{34} - g_{13})$$
$$m^{(e)}_{3,22} = J/720\ (g_{44} - g_{14})$$
$$m^{(e)}_{3,23} = J/720\ (g_{41} - 2g_{11})$$
$$m^{(e)}_{3,24} = J/720\ (2g_{24} - 2g_{12})$$
$$m^{(e)}_{4,4} = J/60\ (g_{22} + g_{33} - g_{23})$$
$$m^{(e)}_{4,5} = J/120\ (g_{22} - g_{23} - g_{24} + 2g_{34})$$
$$m^{(e)}_{4,6} = J/120\ (-g_{33} + g_{23} + g_{34} - 2g_{24})$$
$$m^{(e)}_{4,7} = J/120\ (-g_{22} - g_{12} + g_{23} + 2g_{13})$$
$$m^{(e)}_{4,8} = J/120\ (g_{33} + g_{13} - g_{23} - 2g_{12})$$
$$m^{(e)}_{4,9} = J/120\ (-g_{12} + g_{13} - g_{24} + g_{34})$$
$$m^{(e)}_{4,10} = J/60\ (g_{33} - g_{22})$$
$$m^{(e)}_{4,11} = J/120\ (-g_{22} + g_{23} - g_{24} + 2g_{34})$$
$$m^{(e)}_{4,12} = J/120\ (g_{33} - g_{23} + g_{34} - 2g_{24})$$
$$m^{(e)}_{4,13} = J/360\ (-g_{12} + g_{13})$$
$$m^{(e)}_{4,14} = J/720\ (-2g_{22} + g_{23})$$
$$m^{(e)}_{4,15} = J/720\ (g_{33} - g_{23})$$
$$m^{(e)}_{4,16} = J/720\ (-g_{24} + 2g_{34})$$
$$m^{(e)}_{4,17} = J/720\ (-2g_{22} + g_{23})$$
$$m^{(e)}_{4,18} = J/720\ (2g_{33} - g_{23})$$
$$m^{(e)}_{4,19} = J/720\ (-2g_{24} + g_{34})$$
$$m^{(e)}_{4,20} = J/720\ (-g_{12} + 2g_{13})$$
$$m^{(e)}_{4,21} = J/720\ (2g_{33} - g_{23})$$
$$m^{(e)}_{4,22} = J/720\ (2g_{34} - 2g_{24})$$
$$m^{(e)}_{4,23} = J/720\ (g_{31} - 2g_{12})$$
$$m^{(e)}_{4,24} = J/720\ (g_{23} - g_{22})$$
$$m^{(e)}_{5,5} = J/60\ (g_{22} + g_{44} - g_{24})$$
$$m^{(e)}_{5,6} = J/120\ (g_{44} - g_{24} - g_{34} + 2g_{23})$$
$$m^{(e)}_{5,7} = J/120\ (-g_{22} + g_{24} - g_{12} + 2g_{14})$$
$$m^{(e)}_{5,8} = J/120\ (-g_{12} + g_{14} - g_{23} + g_{34})$$
$$m^{(e)}_{5,9} = J/120\ (g_{44} + g_{14} - g_{24} - 2g_{12})$$
$$m^{(e)}_{5,10} = J/120\ (-g_{22} - g_{23} + g_{24} + 2g_{34})$$
$$m^{(e)}_{5,11} = J/60\ (g_{44} - g_{22})$$
$$m^{(e)}_{5,12} = J/120\ (g_{44} + g_{34} - g_{24} - 2g_{23})$$
$$m^{(e)}_{5,13} = J/720\ (-g_{12} + 2g_{14})$$
$$m^{(e)}_{5,14} = J/720\ (-2g_{22} + g_{24})$$
$$m^{(e)}_{5,15} = J/720\ (-2g_{23} + g_{34})$$
$$m^{(e)}_{5,16} = J/720\ (2g_{44} - g_{24})$$
$$m^{(e)}_{5,17} = J/720\ (-g_{22} + g_{24})$$
$$m^{(e)}_{5,18} = J/360\ (-g_{23} + g_{34})$$
$$m^{(e)}_{5,19} = J/720\ (g_{44} - g_{24})$$
$$m^{(e)}_{5,20} = J/360\ (-g_{12} + g_{14})$$
$$m^{(e)}_{5,21} = J/720\ (2g_{43} - g_{23})$$
$$m^{(e)}_{5,22} = J/720\ (2g_{44} - g_{24})$$
$$m^{(e)}_{5,23} = J/720\ (g_{41} - 2g_{12})$$
$$m^{(e)}_{5,24} = J/720\ (g_{42} - 2g_{22})$$
$$m^{(e)}_{6,6} = J/60\ (g_{33} + g_{44} - g_{34})$$
$$m^{(e)}_{6,7} = J/120\ (-g_{13} + g_{14} - g_{23} + g_{24})$$
$$m^{(e)}_{6,8} = J/120\ (-g_{33} + g_{34} - g_{13} + 2g_{14})$$
$$m^{(e)}_{6,9} = J/120\ (g_{44} + g_{14} - g_{34} - 2g_{13})$$
$$m^{(e)}_{6,10} = J/120\ (-g_{33} + g_{34} - g_{23} + 2g_{24})$$
$$m^{(e)}_{6,11} = J/120\ (g_{44} + g_{24} - g_{34} - 2g_{23})$$
$$m^{(e)}_{6,12} = J/60\ (g_{44} - g_{33})$$
$$m^{(e)}_{6,13} = J/720\ (-g_{13} + 2g_{14})$$
$$m^{(e)}_{6,14} = J/360\ (-g_{23} + g_{24})$$
$$m^{(e)}_{6,15} = J/720\ (-2g_{33} + g_{34})$$
$$m^{(e)}_{6,16} = J/720\ (g_{44} - g_{34})$$
$$m^{(e)}_{6,17} = J/720\ (2g_{24} - g_{23})$$

$$m^{(e)}_{6,18} = J/720\ (-2g_{33} + g_{34})$$
$$m^{(e)}_{6,19} = J/720\ (2g_{44} - g_{34})$$
$$m^{(e)}_{6,20} = J/720\ (-2g_{13} + g_{14})$$
$$m^{(e)}_{6,21} = J/720\ (g_{43} - g_{33})$$
$$m^{(e)}_{6,22} = J/720\ (2g_{44} - g_{34})$$
$$m^{(e)}_{6,23} = J/720\ (2g_{41} - 2g_{13})$$
$$m^{(e)}_{6,24} = J/720\ (g_{42} - 2g_{23})$$
$$m^{(e)}_{7,7} = J/60\ (g_{11} + g_{22} + g_{12})$$
$$m^{(e)}_{7,8} = J/120\ (g_{11} + g_{12} + g_{13} + 2g_{23})$$
$$m^{(e)}_{7,9} = J/120\ (g_{11} + g_{12} + g_{14} + 2g_{24})$$
$$m^{(e)}_{7,10} = J/120\ (g_{22} + g_{12} + g_{23} + 2g_{13})$$
$$m^{(e)}_{7,11} = J/120\ (g_{22} + g_{12} + g_{24} + 2g_{14})$$
$$m^{(e)}_{7,12} = J/120\ (g_{13} + g_{14} + g_{23} + g_{24})$$
$$m^{(e)}_{7,13} = J/720\ (g_{12} + 2g_{11})$$
$$m^{(e)}_{7,14} = J/720\ (g_{22} + g_{12})$$
$$m^{(e)}_{7,15} = J/720\ (g_{13} + 2g_{23})$$
$$m^{(e)}_{7,16} = J/360\ (g_{14} + g_{24})$$
$$m^{(e)}_{7,17} = J/720\ (2g_{22} + g_{12})$$
$$m^{(e)}_{7,18} = J/720\ (2g_{13} + g_{23})$$
$$m^{(e)}_{7,19} = J/720\ (g_{14} + 2g_{24})$$
$$m^{(e)}_{7,20} = J/720\ (2g_{11} + g_{12})$$
$$m^{(e)}_{7,21} = J/720\ (2g_{23} + 2g_{13})$$
$$m^{(e)}_{7,22} = J/720\ (g_{24} + 2g_{14})$$
$$m^{(e)}_{7,23} = J/720\ (g_{12} + g_{11})$$
$$m^{(e)}_{7,24} = J/720\ (2g_{22} + g_{12})$$
$$m^{(e)}_{8,8} = J/60\ (g_{11} + g_{33} + g_{13})$$
$$m^{(e)}_{8,9} = J/120\ (g_{11} + g_{13} + g_{14} + 2g_{34})$$
$$m^{(e)}_{8,10} = J/120\ (g_{33} + g_{13} + g_{23} + 2g_{12})$$
$$m^{(e)}_{8,11} = J/120\ (g_{12} + g_{14} + g_{23} + g_{34})$$
$$m^{(e)}_{8,12} = J/120\ (g_{33} + g_{13} + g_{34} + 2g_{14})$$
$$m^{(e)}_{8,13} = J/720\ (2g_{11} + g_{13})$$
$$m^{(e)}_{8,14} = J/720\ (2g_{12} + g_{23})$$
$$m^{(e)}_{8,15} = J/720\ (2g_{33} + g_{13})$$
$$m^{(e)}_{8,16} = J/720\ (g_{14} + 2g_{34})$$
$$m^{(e)}_{8,17} = J/360\ (g_{12} + g_{23})$$
$$m^{(e)}_{8,18} = J/720\ (g_{33} + g_{13})$$
$$m^{(e)}_{8,19} = J/360\ (g_{14} + g_{34})$$
$$m^{(e)}_{8,20} = J/720\ (g_{11} + g_{13})$$
$$m^{(e)}_{8,21} = J/720\ (2g_{33} + g_{13})$$
$$m^{(e)}_{8,22} = J/720\ (g_{34} + 2g_{14})$$
$$m^{(e)}_{8,23} = J/720\ (g_{31} + 2g_{11})$$
$$m^{(e)}_{8,24} = J/720\ (2g_{23} + g_{12})$$
$$m^{(e)}_{9,9} = J/60\ (g_{11} + g_{44} + g_{14})$$
$$m^{(e)}_{9,10} = J/120\ (g_{12} + g_{13} + g_{24} + g_{34})$$
$$m^{(e)}_{9,11} = J/120\ (g_{44} + g_{14} + g_{24} + 2g_{12})$$
$$m^{(e)}_{9,12} = J/120\ (g_{44} + g_{14} + g_{34} + 2g_{13})$$
$$m^{(e)}_{9,13} = J/720\ (g_{11} + g_{14})$$
$$m^{(e)}_{9,14} = J/720\ (2g_{12} + g_{24})$$
$$m^{(e)}_{9,15} = J/360\ (g_{13} + g_{34})$$
$$m^{(e)}_{9,16} = J/720\ (2g_{44} + g_{14})$$
$$m^{(e)}_{9,17} = J/720\ (g_{12} + 2g_{24})$$
$$m^{(e)}_{9,18} = J/720\ (2g_{13} + g_{34})$$
$$m^{(e)}_{9,19} = J/720\ (2g_{44} + g_{14})$$
$$m^{(e)}_{9,20} = J/720\ (2g_{11} + g_{14})$$
$$m^{(e)}_{9,21} = J/720\ (2g_{34} + g_{13})$$
$$m^{(e)}_{9,22} = J/720\ (g_{44} + g_{14})$$
$$m^{(e)}_{9,23} = J/720\ (g_{41} + 2g_{11})$$
$$m^{(e)}_{9,24} = J/720\ (2g_{24} + 2g_{12})$$
$$m^{(e)}_{10,10} = J/60\ (g_{22} + g_{33} + g_{23})$$
$$m^{(e)}_{10,11} = J/120\ (g_{22} + g_{23} + g_{24} + 2g_{34})$$
$$m^{(e)}_{10,12} = J/120\ (g_{33} + g_{23} + g_{34} + 2g_{24})$$
$$m^{(e)}_{10,13} = J/360\ (g_{12} + g_{13})$$
$$m^{(e)}_{10,14} = J/720\ (2g_{22} + g_{23})$$
$$m^{(e)}_{10,15} = J/720\ (g_{33} + g_{23})$$
$$m^{(e)}_{10,16} = J/720\ (g_{24} + 2g_{34})$$
$$m^{(e)}_{10,17} = J/720\ (2g_{22} + g_{23})$$
$$m^{(e)}_{10,18} = J/720\ (2g_{33} + g_{23})$$
$$m^{(e)}_{10,19} = J/720\ (2g_{24} + g_{34})$$
$$m^{(e)}_{10,20} = J/720\ (g_{12} + 2g_{13})$$
$$m^{(e)}_{10,21} = J/720\ (2g_{33} + g_{23})$$
$$m^{(e)}_{10,22} = J/720\ (2g_{34} + 2g_{24})$$
$$m^{(e)}_{10,23} = J/720\ (g_{31} + 2g_{12})$$
$$m^{(e)}_{10,24} = J/720\ (g_{23} + g_{22})$$
$$m^{(e)}_{11,11} = J/60\ (g_{22} + g_{44} + g_{24})$$
$$m^{(e)}_{11,12} = J/120\ (g_{44} + g_{24} + g_{34} + 2g_{23})$$

$$m^{(e)}_{11,13} = J/720\ (g_{12} + 2g_{14})$$
$$m^{(e)}_{11,14} = J/720\ (2g_{22} + g_{24})$$
$$m^{(e)}_{11,15} = J/720\ (2g_{23} + g_{34})$$
$$m^{(e)}_{11,16} = J/720\ (2g_{44} + g_{24})$$
$$m^{(e)}_{11,17} = J/720\ (g_{22} + g_{24})$$
$$m^{(e)}_{11,18} = J/360\ (g_{23} + g_{34})$$
$$m^{(e)}_{11,19} = J/720\ (g_{44} + g_{24})$$
$$m^{(e)}_{11,20} = J/360\ (g_{12} + g_{14})$$
$$m^{(e)}_{11,21} = J/720\ (2g_{43} + g_{23})$$
$$m^{(e)}_{11,22} = J/720\ (2g_{44} + g_{24})$$
$$m^{(e)}_{11,23} = J/720\ (g_{41} + 2g_{12})$$
$$m^{(e)}_{11,24} = J/720\ (g_{42} + 2g_{22})$$
$$m^{(e)}_{12,12} = J/60\ (g_{33} + g_{44} + g_{34})$$
$$m^{(e)}_{12,13} = J/720\ (g_{13} + 2g_{14})$$
$$m^{(e)}_{12,14} = J/360\ (g_{23} + g_{24})$$
$$m^{(e)}_{12,15} = J/720\ (2g_{33} + g_{34})$$
$$m^{(e)}_{12,16} = J/720\ (g_{44} + g_{34})$$
$$m^{(e)}_{12,17} = J/720\ (2g_{24} + g_{23})$$
$$m^{(e)}_{12,18} = J/720\ (2g_{33} + g_{34})$$
$$m^{(e)}_{12,19} = J/720\ (2g_{44} + g_{34})$$
$$m^{(e)}_{12,20} = J/720\ (2g_{13} + g_{14})$$
$$m^{(e)}_{12,21} = J/720\ (g_{43} + g_{33})$$
$$m^{(e)}_{12,22} = J/720\ (2g_{44} + g_{34})$$
$$m^{(e)}_{12,23} = J/720\ (2g_{41} + 2g_{13})$$
$$m^{(e)}_{12,24} = J/720\ (g_{42} + 2g_{23})$$
$$m^{(e)}_{13,13} = J/1260\ g_{11}$$
$$m^{(e)}_{13,14} = J/2520\ g_{12}$$
$$m^{(e)}_{13,15} = J/5040\ g_{13}$$
$$m^{(e)}_{13,16} = J/2520\ g_{14}$$
$$m^{(e)}_{13,17} = J/2520\ g_{12}$$
$$m^{(e)}_{13,18} = J/2520\ g_{13}$$
$$m^{(e)}_{13,19} = J/2520\ g_{14}$$
$$m^{(e)}_{13,20} = J/2520\ g_{11}$$
$$m^{(e)}_{13,21} = J/2520\ g_{13}$$
$$m^{(e)}_{13,22} = J/1260\ g_{14}$$
$$m^{(e)}_{13,23} = J/2520\ g_{11}$$
$$m^{(e)}_{13,24} = J/5040\ g_{12}$$
$$m^{(e)}_{14,14} = J/1260\ g_{22}$$
$$m^{(e)}_{14,15} = J/2520\ g_{23}$$
$$m^{(e)}_{14,16} = J/5040\ g_{24}$$
$$m^{(e)}_{14,17} = J/2520\ g_{22}$$
$$m^{(e)}_{14,18} = J/2520\ g_{23}$$
$$m^{(e)}_{14,19} = J/2520\ g_{24}$$
$$m^{(e)}_{14,20} = J/2520\ g_{12}$$
$$m^{(e)}_{14,21} = J/5040\ g_{23}$$
$$m^{(e)}_{14,22} = J/2520\ g_{24}$$
$$m^{(e)}_{14,23} = J/1260\ g_{21}$$
$$m^{(e)}_{14,24} = J/2520\ g_{22}$$
$$m^{(e)}_{15,15} = J/1260\ g_{33}$$
$$m^{(e)}_{15,16} = J/2520\ g_{34}$$
$$m^{(e)}_{15,17} = J/2520\ g_{23}$$
$$m^{(e)}_{15,18} = J/2520\ g_{33}$$
$$m^{(e)}_{15,19} = J/2520\ g_{34}$$
$$m^{(e)}_{15,20} = J/2520\ g_{13}$$
$$m^{(e)}_{15,21} = J/2520\ g_{33}$$
$$m^{(e)}_{15,22} = J/5040\ g_{34}$$
$$m^{(e)}_{15,23} = J/2520\ g_{31}$$
$$m^{(e)}_{15,24} = J/1260\ g_{32}$$
$$m^{(e)}_{16,16} = J/1260\ g_{44}$$
$$m^{(e)}_{16,17} = J/2520\ g_{24}$$
$$m^{(e)}_{16,18} = J/2520\ g_{34}$$
$$m^{(e)}_{16,19} = J/2520\ g_{44}$$
$$m^{(e)}_{16,20} = J/2520\ g_{14}$$
$$m^{(e)}_{16,21} = J/1260\ g_{43}$$
$$m^{(e)}_{16,22} = J/2520\ g_{44}$$
$$m^{(e)}_{16,23} = J/5040\ g_{41}$$
$$m^{(e)}_{16,24} = J/2520\ g_{42}$$
$$m^{(e)}_{17,17} = J/1260\ g_{22}$$
$$m^{(e)}_{17,18} = J/5040\ g_{23}$$
$$m^{(e)}_{17,19} = J/1260\ g_{24}$$
$$m^{(e)}_{17,20} = J/5040\ g_{12}$$
$$m^{(e)}_{17,21} = J/2520\ g_{23}$$
$$m^{(e)}_{17,22} = J/2520\ g_{24}$$
$$m^{(e)}_{17,23} = J/2520\ g_{21}$$
$$m^{(e)}_{17,24} = J/2520\ g_{22}$$

$$m^{(e)}_{18,18} = J/1260\ g_{33}$$
$$m^{(e)}_{18,19} = J/5040\ g_{34}$$
$$m^{(e)}_{18,20} = J/1260\ g_{13}$$
$$m^{(e)}_{18,21} = J/2520\ g_{33}$$
$$m^{(e)}_{18,22} = J/2520\ g_{34}$$
$$m^{(e)}_{18,23} = J/2520\ g_{31}$$
$$m^{(e)}_{18,24} = J/2520\ g_{32}$$
$$m^{(e)}_{19,19} = J/1260\ g_{44}$$
$$m^{(e)}_{19,20} = J/5040\ g_{14}$$
$$m^{(e)}_{19,21} = J/2520\ g_{43}$$
$$m^{(e)}_{19,22} = J/2520\ g_{44}$$
$$m^{(e)}_{19,23} = J/2520\ g_{41}$$
$$m^{(e)}_{19,24} = J/2520\ g_{42}$$
$$m^{(e)}_{20,20} = J/1260\ g_{11}$$
$$m^{(e)}_{20,21} = J/2520\ g_{13}$$
$$m^{(e)}_{20,22} = J/2520\ g_{14}$$
$$m^{(e)}_{20,23} = J/2520\ g_{11}$$
$$m^{(e)}_{20,24} = J/2520\ g_{12}$$
$$m^{(e)}_{21,21} = J/1260\ g_{33}$$
$$m^{(e)}_{21,22} = J/2520\ g_{34}$$
$$m^{(e)}_{21,23} = J/5040\ g_{31}$$
$$m^{(e)}_{21,24} = J/2520\ g_{32}$$
$$m^{(e)}_{22,22} = J/1260\ g_{44}$$
$$m^{(e)}_{22,23} = J/2520\ g_{41}$$
$$m^{(e)}_{22,24} = J/5040\ g_{42}$$
$$m^{(e)}_{23,23} = J/1260\ g_{11}$$
$$m^{(e)}_{23,24} = J/2520\ g_{12}$$
$$m^{(e)}_{24,24} = J/1260\ g_{22}$$

Curl element matrix The upper left diagonal block of $\boldsymbol{A}^{(e)}$, with

$$\{\boldsymbol{A}^{(e)}\}_{i,j} = \iiint_{T_e} \mathbf{rot}\,\mathbf{N}^{(e)}_i \cdot \mathbf{rot}\,\mathbf{N}^{(e)}_j \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z,$$

is obtained from matrix $\boldsymbol{C}$ defined in (3.35) by

$$\boldsymbol{A}^{(e)}_{1:6,1:6} := 2J/3\,\boldsymbol{C}.$$

Because of (3.33b), we have

$$\boldsymbol{A}^{(e)}_{7:12,1:24} = \mathbf{0} \quad \text{and} \quad \boldsymbol{A}^{(e)}_{1:24,7:12} = \mathbf{0}.$$

The entries of $\boldsymbol{A}^{(e)}_{1:24,1:6}$ and $\boldsymbol{A}^{(e)}_{1:6,1:24}$ are calculated by the following formulas

$$\boldsymbol{A}^{(e)}_{1:6,13} = J/12\ (-\boldsymbol{C}_{1:6,1} - \boldsymbol{C}_{1:6,2})$$
$$\boldsymbol{A}^{(e)}_{1:6,14} = J/12\ (-\boldsymbol{C}_{1:6,4} - \boldsymbol{C}_{1:6,5})$$
$$\boldsymbol{A}^{(e)}_{1:6,15} = J/12\ (-\boldsymbol{C}_{1:6,6} + \boldsymbol{C}_{1:6,2})$$
$$\boldsymbol{A}^{(e)}_{1:6,16} = J/12\ (\boldsymbol{C}_{1:6,3} + \boldsymbol{C}_{1:6,5})$$
$$\boldsymbol{A}^{(e)}_{1:6,17} = J/12\ (-\boldsymbol{C}_{1:6,4} + \boldsymbol{C}_{1:6,1})$$
$$\boldsymbol{A}^{(e)}_{1:6,18} = J/12\ (-\boldsymbol{C}_{1:6,6} + \boldsymbol{C}_{1:6,4})$$
$$\boldsymbol{A}^{(e)}_{1:6,19} = J/12\ (\boldsymbol{C}_{1:6,3} + \boldsymbol{C}_{1:6,6})$$
$$\boldsymbol{A}^{(e)}_{1:6,20} = J/12\ (-\boldsymbol{C}_{1:6,1} - \boldsymbol{C}_{1:6,3})$$
$$\boldsymbol{A}^{(e)}_{1:6,21} = J/12\ (\boldsymbol{C}_{1:6,2} + \boldsymbol{C}_{1:6,4})$$
$$\boldsymbol{A}^{(e)}_{1:6,22} = J/12\ (\boldsymbol{C}_{1:6,5} + \boldsymbol{C}_{1:6,6})$$
$$\boldsymbol{A}^{(e)}_{1:6,23} = J/12\ (-\boldsymbol{C}_{1:6,2} - \boldsymbol{C}_{1:6,3})$$
$$\boldsymbol{A}^{(e)}_{1:6,24} = J/12\ (-\boldsymbol{C}_{1:6,5} + \boldsymbol{C}_{1:6,1})$$

and

$$\boldsymbol{A}^{(e)}_{1:6,1:24} = \boldsymbol{A}^{(e)^T}_{1:24,1:6}$$

For $\boldsymbol{A}^{(e)}_{13:24,13:24}$, formulas for the entries in the upper triangle are given. The remaining formulas are omitted for symmetry reasons.

$$
\begin{aligned}
a^{(e)}_{13,13} &= J/120\ (2c_{11} + 2c_{12} + 2c_{22}) \\
a^{(e)}_{13,14} &= J/120\ (c_{14} + 2c_{15} + c_{24} + c_{25}) \\
a^{(e)}_{13,15} &= J/120\ (c_{16} - c_{12} + c_{26} - c_{22}) \\
a^{(e)}_{13,16} &= J/120\ (-c_{13} - c_{15} - 2c_{23} - c_{25}) \\
a^{(e)}_{13,17} &= J/120\ (c_{14} - 2c_{11} + c_{24} - c_{21}) \\
a^{(e)}_{13,18} &= J/120\ (c_{16} - c_{14} + 2c_{26} - c_{24}) \\
a^{(e)}_{13,19} &= J/120\ (-2c_{13} - c_{16} - c_{23} - c_{26}) \\
a^{(e)}_{13,20} &= J/120\ (c_{11} + c_{13} + c_{21} + 2c_{23}) \\
a^{(e)}_{13,21} &= J/120\ (-c_{12} - c_{14} - 2c_{22} - c_{24}) \\
a^{(e)}_{13,22} &= J/120\ (-2c_{15} - c_{16} - c_{25} - 2c_{26}) \\
a^{(e)}_{13,23} &= J/120\ (c_{12} + 2c_{13} + c_{22} + c_{23}) \\
a^{(e)}_{13,24} &= J/120\ (c_{15} - c_{11} + c_{25} - c_{21}) \\
a^{(e)}_{14,14} &= J/120\ (2c_{44} + 2c_{45} + 2c_{55}) \\
a^{(e)}_{14,15} &= J/120\ (c_{46} - 2c_{42} + c_{56} - c_{52}) \\
a^{(e)}_{14,16} &= J/120\ (-c_{43} - c_{45} - c_{53} - c_{55}) \\
a^{(e)}_{14,17} &= J/120\ (c_{44} - c_{41} + c_{54} - 2c_{51}) \\
a^{(e)}_{14,18} &= J/120\ (c_{46} - 2c_{44} + c_{56} - c_{54}) \\
a^{(e)}_{14,19} &= J/120\ (-c_{43} - c_{46} - 2c_{53} - c_{56}) \\
a^{(e)}_{14,20} &= J/120\ (2c_{41} + c_{43} + c_{51} + c_{53}) \\
a^{(e)}_{14,21} &= J/120\ (-c_{42} - c_{44} - c_{52} - c_{54}) \\
a^{(e)}_{14,22} &= J/120\ (-c_{45} - c_{46} - 2c_{55} - c_{56}) \\
a^{(e)}_{14,23} &= J/120\ (2c_{42} + c_{43} + c_{52} + 2c_{53}) \\
a^{(e)}_{14,24} &= J/120\ (c_{45} - 2c_{41} + c_{55} - c_{51}) \\
a^{(e)}_{15,15} &= J/120\ (2c_{66} - 2c_{62} + 2c_{22}) \\
a^{(e)}_{15,16} &= J/120\ (-c_{63} - 2c_{65} + c_{23} + c_{25}) \\
a^{(e)}_{15,17} &= J/120\ (2c_{64} - c_{61} - c_{24} + c_{21}) \\
a^{(e)}_{15,18} &= J/120\ (c_{66} - c_{64} - c_{26} + 2c_{24}) \\
a^{(e)}_{15,19} &= J/120\ (-c_{63} - 2c_{66} + c_{23} + c_{26}) \\
a^{(e)}_{15,20} &= J/120\ (c_{61} + c_{63} - 2c_{21} - c_{23}) \\
a^{(e)}_{15,21} &= J/120\ (-c_{62} - 2c_{64} + c_{22} + c_{24}) \\
a^{(e)}_{15,22} &= J/120\ (-c_{65} - c_{66} + c_{25} + c_{26}) \\
a^{(e)}_{15,23} &= J/120\ (c_{62} + c_{63} - 2c_{22} - c_{23}) \\
a^{(e)}_{15,24} &= J/120\ (2c_{65} - c_{61} - c_{25} + 2c_{21}) \\
a^{(e)}_{16,16} &= J/120\ (2c_{33} + 2c_{35} + 2c_{55}) \\
a^{(e)}_{16,17} &= J/120\ (-c_{34} + c_{31} - 2c_{54} + c_{51}) \\
a^{(e)}_{16,18} &= J/120\ (-2c_{36} + c_{34} - c_{56} + c_{54}) \\
a^{(e)}_{16,19} &= J/120\ (c_{33} + c_{36} + c_{53} + 2c_{56}) \\
a^{(e)}_{16,20} &= J/120\ (-c_{31} - 2c_{33} - c_{51} - c_{53}) \\
a^{(e)}_{16,21} &= J/120\ (2c_{32} + c_{34} + c_{52} + 2c_{54}) \\
a^{(e)}_{16,22} &= J/120\ (c_{35} + 2c_{36} + c_{55} + c_{56}) \\
a^{(e)}_{16,23} &= J/120\ (-c_{32} - c_{33} - c_{52} - c_{53}) \\
a^{(e)}_{16,24} &= J/120\ (-c_{35} + c_{31} - 2c_{55} + c_{51}) \\
a^{(e)}_{17,17} &= J/120\ (2c_{44} - 2c_{14} + 2c_{11}) \\
a^{(e)}_{17,18} &= J/120\ (c_{46} - c_{44} - c_{16} + c_{14}) \\
a^{(e)}_{17,19} &= J/120\ (-c_{43} - 2c_{46} + 2c_{13} + c_{16}) \\
a^{(e)}_{17,20} &= J/120\ (c_{41} + c_{43} - c_{11} - c_{13}) \\
a^{(e)}_{17,21} &= J/120\ (-c_{42} - 2c_{44} + c_{12} + c_{14}) \\
a^{(e)}_{17,22} &= J/120\ (-c_{45} - c_{46} + 2c_{15} + c_{16}) \\
a^{(e)}_{17,23} &= J/120\ (c_{42} + c_{43} - c_{12} - 2c_{13}) \\
a^{(e)}_{17,24} &= J/120\ (2c_{45} - c_{41} - c_{15} + c_{11}) \\
a^{(e)}_{18,18} &= J/120\ (2c_{66} - 2c_{64} + 2c_{44}) \\
a^{(e)}_{18,19} &= J/120\ (-c_{63} - c_{66} + c_{43} + c_{46}) \\
a^{(e)}_{18,20} &= J/120\ (c_{61} + 2c_{63} - 2c_{41} - c_{43}) \\
a^{(e)}_{18,21} &= J/120\ (-2c_{62} - c_{64} + c_{42} + c_{44}) \\
a^{(e)}_{18,22} &= J/120\ (-c_{65} - 2c_{66} + c_{45} + c_{46}) \\
a^{(e)}_{18,23} &= J/120\ (c_{62} + c_{63} - 2c_{42} - c_{43}) \\
a^{(e)}_{18,24} &= J/120\ (c_{65} - c_{61} - c_{45} + 2c_{41}) \\
a^{(e)}_{19,19} &= J/120\ (2c_{33} + 2c_{36} + 2c_{66}) \\
a^{(e)}_{19,20} &= J/120\ (-c_{31} - c_{33} - c_{61} - c_{63}) \\
a^{(e)}_{19,21} &= J/120\ (c_{32} + c_{34} + c_{62} + 2c_{64}) \\
a^{(e)}_{19,22} &= J/120\ (2c_{35} + c_{36} + c_{65} + c_{66}) \\
a^{(e)}_{19,23} &= J/120\ (-c_{32} - 2c_{33} - c_{62} - c_{63}) \\
a^{(e)}_{19,24} &= J/120\ (-c_{35} + c_{31} - 2c_{65} + c_{61})
\end{aligned}
$$

$$a^{(e)}_{20,20} = J/120\ (2c_{11} + 2c_{13} + 2c_{33})$$
$$a^{(e)}_{20,21} = J/120\ (-c_{12} - c_{14} - 2c_{32} - c_{34})$$
$$a^{(e)}_{20,22} = J/120\ (-c_{15} - c_{16} - c_{35} - 2c_{36})$$
$$a^{(e)}_{20,23} = J/120\ (2c_{12} + c_{13} + c_{32} + c_{33})$$
$$a^{(e)}_{20,24} = J/120\ (c_{15} - 2c_{11} + c_{35} - c_{31})$$
$$a^{(e)}_{21,21} = J/120\ (2c_{22} + 2c_{24} + 2c_{44})$$
$$a^{(e)}_{21,22} = J/120\ (c_{25} + 2c_{26} + c_{45} + c_{46})$$
$$a^{(e)}_{21,23} = J/120\ (-c_{22} - c_{23} - c_{42} - c_{43})$$
$$a^{(e)}_{21,24} = J/120\ (-c_{25} + c_{21} - 2c_{45} + c_{41})$$
$$a^{(e)}_{22,22} = J/120\ (2c_{55} + 2c_{56} + 2c_{66})$$
$$a^{(e)}_{22,23} = J/120\ (-c_{52} - 2c_{53} - c_{62} - c_{63})$$
$$a^{(e)}_{22,24} = J/120\ (-c_{55} + c_{51} - c_{65} + c_{61})$$
$$a^{(e)}_{23,23} = J/120\ (2c_{22} + 2c_{23} + 2c_{33})$$
$$a^{(e)}_{23,24} = J/120\ (c_{25} - 2c_{21} + c_{35} - c_{31})$$
$$a^{(e)}_{24,24} = J/120\ (2c_{55} - 2c_{51} + 2c_{11})$$

References

[1] S. Adam, P. Arbenz, and R. Geus. Eigenvalue solvers for electromagnetic fields in cavities. Technical Report 275, ETH Zürich, Computer Science Department, October 1997. (Available at URL `http://www.inf.ethz.ch/publications/`).

[2] P. Arbenz. A comparison of factorization-free eigensolvers with application to cavity resonators. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2331 of *LNCS*, pages 295–304. Springer, 2002.

[3] P. Arbenz and Z. Drmač. On positive semidefinite matrices with known null space. Technical Report 352, ETH Zürich, Computer Science Department, November 2000.

[4] P. Arbenz and R. Geus. Parallel solvers for large eigenvalue problems originating from Maxwell's equations. In D. Pritchard and J. Reeve, editors, *Euro-Par '98 Parallel Processing*. Springer-Verlag, 1998. (Lecture Notes in Computer Science, 1470).

[5] P. Arbenz and R. Geus. Eigenvalue solvers for electromagnetic fields in cavities. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing*, pages 363–373. Springer-Verlag, 1999. (Lecture Notes in Computational Science and Engineering, 8).

[6] P. Arbenz, R. Geus, and S. Adam. Solving Maxwell eigenvalue problems for accelerating cavities. *Phys. Rev. ST Accel. Beams*, 4:022001, 2001. (Electronic journal available from `http://prst-ab.aps.org/`).

[7] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, September 2001.

[8] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the solution of algebraic eigenvalue problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. A practical guide.

[9] R. E. Bank. *PLTMG: a software package for solving elliptic partial differential equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1994. Users' guide 7.0.

[10] R. E. Bank. Hierarchical bases and the finite element method. *Acta Numerica*, 5:1–43, 1996.

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

[12] D. M. Beazley and P. S. Lomdahl. Controlling the data glut in large-scale molecular-dynamics simulations. *Computers in Physics*, 11(3):230–238, May/June 1997.

[13] D. M. Beazley and P. S. Lomdahl. Feeding a large-scale physics application to Python. In *Proceedings of the 6th International Python Conference*, October 1997.

[14] A. N. Bespalov. Finite element method for the eigenmode problem of a RF cavity resonator. *Soviet Journal of Numerical Analysis and Mathematical Modelling*, 3:163–178, 1988.

[15] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, May 1996.

[16] Å. Björck. Numerics of Gram-Schmidt orthogonalization. *Linear Algebra Appl.*, 197(198):297–316, 1994. Second Conference of the International Linear Algebra Society (ILAS) (Lisbon, 1992).

[17] D. Braess. *Finite elements*. Cambridge University Press, 1997.

[18] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 24th edition, 1989.

[19] J. R. Bunch, L. Kaufman, and B. N. Parlett. Decomposition of a symmetric matrix. *Numer. Math.*, (27):95–109, 1976.

[20] E. Chow and Y. Saad. $ILUS$: an incomplete LU preconditioner in sparse skyline format. *Internat. J. Numer. Methods Fluids*, 25(7):739–748, 1997.

[21] R. Dautray and J.-L. Lions. *Mathematical Analysis and Numerical Methods for Science and Technology*, volume 1: Physical Origins and Classical Methods. Springer-Verlag, Berlin, 1990.

[22] E. R. Davidson. Monster matrices: Their eigenvalues and eigenvectors. *Computers in Physics*, 7(5):519–522, Sep/Oct 1993.

[23] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755 (electronic), 1999.

[24] D. R. Fokkema, G. L. G. Sleijpen, and H. A. Van der Vorst. Generalized conjugate gradient squared. *J. Comput. Appl. Math.*, 71(1):125–146, 1996.

[25] D. R. Fokkema, G. L. G. Sleijpen, and H. A. Van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the reduction of matrix pencils. Technical report, Universiteit Utrecht, Department of Mathematics, January 1996.

[26] D. R. Fokkema, G. L. G. Sleijpen, and H. A. Van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1):94–125 (electronic), 1999.

[27] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, November 1992.

[28] R. Freund and N. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[29] R. W. Freund and F. Jarre. A QMR-based interior-point algorithm for solving linear programs. *Math. Programming*, 76(1, Ser. B):183–210, 1997. Interior point methods in theory and practice (Iowa City, IA, 1994).

[30] R. W. Freund and N. M. Nachtigal. Software for simplified Lanczos and QMR algorithms. *Appl. Numer. Math.*, 19(3):319–341, 1995. Special issue on iterative methods for linear equations (Atlanta, GA, 1994).

[31] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP*, page 1381, 1998.

[32] M. Genseberger and G. L. G. Sleijpen. Alternative correction equations in the Jacobi-Davidson method. *Numer. Linear Algebra Appl.*, 6(3):235–253, 1999.

[33] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[34] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. *Parallel Computing*, 27:883–896, 2001.

[35] V. Girault and P.-A. Raviart. *Finite Element Methods for the Navier-Stokes Equations*. Springer-Verlag, Berlin, 1986. (Springer Series in Computational Mathematics, 5).

[36] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[37] R. G. Grimes, J. G. Lewis, and H. D. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl.*, 15(1):228–272, 1994.

[38] R. Gruber and J. Rappaz. *Finite Element Methods in Linear Ideal Magnetohydrodynamics*. Springer-Verlag, Berlin, 1985.

[39] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[40] K. Hinsen. High level scientific programming with Python. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2331 of *LNCS*, pages 691–700. Springer, 2002.

[41] E. Im and K. Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[42] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California, May 2000.

[43] J. D. Jackson. *Classical Electrodynamics*. Wiley, New York, 2nd edition, 1975.

[44] C. G. J. Jacobi. Über eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden linearen Gleichungen. *Astronom. Nachr.*, pages 297–306, 1845.

[45] J. Jin. *The Finite Element Method in Electromagnetics*. Wiley, New York, 1993.

[46] F. Kikuchi. Mixed and penalty formulations for finite element analysis of an eigenvalue problem in electromagnetism. *Comput. Methods Appl. Mech. Eng.*, 64:509–521, 1987.

[47] A. V. Knyazev. Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541 (electronic), 2001. Copper Mountain Conference (2000).

[48] L. D. Landau and E. M. Lifshitz. *Electrodynamics of Continuous Media*. Pergamon Press, Oxford, 2nd edition, 1984.

[49] R. B. Lehoucq and D. C. Sorensen. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM J. Matrix Anal. Appl.*, 17(4):789–821, 1996.

[50] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK users' guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998. Solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods.

[51] R. Leis. Zur Theorie elektromagnetischer Schwingungen in anisotropen Medien. *Math. Z.*, 106:213–224, 1968.

[52] D. E. Longsine and S. F. McCormick. Simultaneous Rayleigh-quotient minimization methods for $Ax = \lambda Bx$. *Linear Algebra Appl.*, 34:195–234, 1980.

[53] R. B. Morgan and D. S. Scott. generalizations of Davidson's method for computing eigenvalues of sparse symmetric matrices. *SIAM J. Sci. Stat. Comput.*, 7(3):817–825, July 1986.

[54] J. C. Nédélec. Mixed finite elements in $\Re^3$. *Numer. Math.*, 35:315–341, 1980.

[55] J. M. Ortega. *Introduction to parallel and vector solution of linear systems*. Plenum Press, New York, 1989.

[56] C. Paige and M. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[57] B. N. Parlett. *The symmetric eigenvalue problem*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998. Corrected reprint of the 1980 original.

[58] P. Ramachandran. Mayavi: A free tool for CFD data visualization. In *4th Annual CFD Symposium*. Aeronautical Society of India, August 2001.

[59] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[60] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.

[61] Y. Saad and A. Malevsky. P-SPARSLIB: A portable library of distributed memory sparse iterative solvers. Technical Report UMSI 95-180, MSI, 1995.

[62] J. S. Savage. Comparing high order vector basis functions. Ansoft Corporation.

[63] W. J. Schroeder, editor. *The VTK User's Guide*. Kitware, Inc., 2001.

[64] H. R. Schwarz. *FORTRAN-Programme zur Methode der finiten Elemente*. Teubner, Stuttgart, 3rd edition, 1991.

[65] H. R. Schwarz. *Methode der finiten Elemente*. Teubner, Stuttgart, 3rd edition, 1991.

[66] P. P. Silvester and R. L. Ferrari. *Finite Elements for Electrical Engineers*. Cambridge University Press, Cambridge, 3rd edition, 1996.

[67] G. L. G. Sleijpen, A. G. L. Booten, D. R. Fokkema, and H. A. Van der Vorst. Jacobi-Davidson type methods for generalized eigenproblems and polynomial eigenproblems. *BIT*, 36(3):595–633, 1996. International Linear Algebra Year (Toulouse, 1995).

[68] G. L. G. Sleijpen and H. A. Van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, 1996.

[69] G. L. G. Sleijpen, H. A. Van der Vorst, and M. J. The main effects of rounding errors in krylov solvers for symmetric linear systems. Technical Report 1006, University Utrecht, Department of Mathematics, 1997.

[70] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[71] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[72] G. van Rossum and F. L. Drake. *Extending and Embedding the Python Interpreter*. PythonLabs, April 2002.

[73] G. van Rossum and F. L. Drake. *Python Library Reference*. PythonLabs, April 2002.

[74] G. van Rossum and F. L. Drake. *Python Tutorial*. PythonLabs, April 2002.

[75] G. van Rossum and F. L. Drake. *Python/C API Reference Manual*. PythonLabs, April 2002.

[76] K. Wadleigh and A. Potler. Advanced optimization for PA-8x00 processors. Presentation at the HiPer'98 conference in Zurich, 1998.

[77] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *SC '98 Proceedings*, 1998.

[78] T. V. Yioultsis and T. D. Tsiboukis. Development and implementation of second and third order vector finite elements in various 3-D electromagnetic field problems. *IEEE Transactions on Magnetics*, 33(2):1812–1815, March 1997.

Curriculum vitae

Person

Name	*Roman Geus*
Date of Birth	*6. September 1970*
Nationality	*Swiss*
Citizen of	*Arbon TG*

Education

1985–1990	Mittelschule in Romanshorn TG, Matura Typus C
1990–1996	Study in computer science (IIIc) at ETH Zürich
1996	graduated as Dipl. Informatik-Ing. ETH
1996–2002	Ph.D. student and teaching assistant at the Institute of Scientific Computing, ETH Zürich

Practical work

1992–2001	Design and implementation of software for calculating shapes of high-precision cam lobes at “Sulzer Rüti AG” and “hat engineering AG”, Arbon TG
1993	Development of database software for “Winterthur Versicherungen” during a 12 week internship
1995	Optimisation and parallelisation of software for the simulation of molecules during the “Summer Student Internship Programs (SSIP’95)” at Centro Svizzero di Calcolo Scientifico (CSCS), Manno TI

Interests and skills

- ◇ Programming in general (C, C++, Java, Fortran 77, Fortran 90, Python)
- ◇ Object-oriented programming with scripting languages
- ◇ Parallel programming, High performance computing
- ◇ Numerical algorithms, e.g. eigenvalue solvers and operations with sparse matrices
- ◇ Optimisation of algorithms (assembler, cache, . . .)
- ◇ Network programming (Web applications, Peer-to-peer networking, . . .)