WWU Münster
Faculty of Mathematics and Computer Science

Masterthesis

M. Sc. Mathematics

Efficient Computation of Transfer Matrices using the Block Conjugate Gradient Method

Author
Malte Bernhard Höltershinken
Matr. 427603

Supervisors
Prof. Dr. Carsten Wolters
Prof. Dr. Christian Engwer

27. September 2021

Contents

1 Introduction **1**

2 The Forward Problem **1**
2.1 The Forward Problem . . . 1
2.2 The Finite Element Approach . . . 3
2.3 Solving the linear system . . . 7
2.4 The transfer matrix approach . . . 14

3 Vectorization and its Limitations **19**
3.1 Introduction to Vectorization . . . 19
3.2 Vectorization in a Memory Bound Context . . . 22
3.3 Designing Algorithms to Utilize Vectorization . . . 25

4 The Block Conjugate Gradient Method **30**
4.1 A framework to generalize Krylov Methods . . . 31
4.1.1 The conjugate gradient method . . . 31
4.1.2 Generalizing inner product spaces . . . 35
4.1.3 The block conjugate gradient method . . . 41
4.2 Optimality results . . . 44
4.3 Convergence analysis . . . 46
4.4 Deflation Methods . . . 57
4.5 Preconditioning . . . 62

5 Implementation and Numerical Experiments **66**
5.1 Investigating the Choice of Parameters . . . 66
5.2 Comparision of CG and BCG . . . 69

6 Summary and Outlook **71**

A Chebyshev polynomials **73**

1 Introduction

The simulation of sensor measurements is one of the most important problems in bioelectromagnetism. Such simulations, for example, form the basis of source reconstruction in the brain, which is of great practical interest both in scientific research and in clinical applications, e.g. in the form of presurgical epilepsy diagnosis. In these applications, we often need to simulate sensor measurements for many different neural activities in the brain, and it is thus important to perform these simulations as efficiently as possible. A crucial part in the efficient implementation of such simulations is the *transfer matrix approach*. The aim of this thesis is to improve the efficiency of this approach.

In section 2, we will discuss how to associate a mathematical formulation to the problem of simulating sensor measurements. We will then study this mathematical formulation and talk about how to solve the corresponding problem numerically. Finally, we will show how the transfer matrix approach can be used to efficiently simulate sensor measurements for a large number of neural activities.

Our goal is to adapt the transfer matrix approach so that it is able to utilize a feature called *vectorization*. We will thus give an introduction to vectorization in section 3. We will then discuss why the transfer matrix approach in its previous form cannot utilize vectorization, and discuss how an adaption that can utilize vectorization might look.

In section 4 we then derive how the conjugate gradient algorithm, which lies at the heart of the transfer matrix approach in its previous form, can be adapted to incorporate the results of section 3. To this end, we derive a generalization of the conjugate gradient method, called the *block conjugate gradient method*. We then study the properties of this method and discuss its implementation. Finally, we will show how preconditioning can be incorporated into the block conjugate gradient method.

In section 5 we will then discuss the implementation of the block conjugate gradient method into the DUNEuro toolbox, and evaluate its performance in the context of the transfer matrix approach.

Finally, appendix A assembles facts about Chebyshev polynomials that are needed at various parts of this thesis.

2 The Forward Problem in Bioelectromagnetism and the Transfer Matrix Approach

2.1 Introduction to the Forward Problem

One of the most central problems in bioelectromagnetism is the *forward problem*. It can, in general terms, be stated as follows.

Problem 1. Given some head geometry, some neural activity inside the head geometry and a number of sensors, the *forward problem in bioelectromagnetism* is the problem of computing the sensor measurements the neural activity would generate.

This definition is quite vague and leaves open a number of points corresponding to different modeling approaches and measurement modalities.

For instance, it is not yet specified what kind of sensors are used. Most commonly the sensors are either electrodes measuring the electric potential at some point on the scalp or magnetometers measuring magnetic flux. The process of using electrodes to measure potentials on the scalp is called *Electroencephalography (EEG)* [1], and we thus call the problem of simulating electrode measurements corresponding to some neural activity the *EEG forward problem*. Similarly, the process of measuring magnetic fluxes generated by the brain is called *Magnetoencephalography (MEG)*, and we call the problem of simulating the measured magnetic fluxes the *MEG forward problem*. If we want to simultaneously simulate electric potentials and magnetic fluxes generated by the brain we speak of the *MEEG forward problem*.

But even once we have specified the measurement modalities there are still numerous approaches to associate a mathematical formulation to the above problem. A recent example is the *BEM-FMM* method proposed by Makarov et al. in [1], which is based on an integral equation in terms of the surface charge density.

Another approach is to start from Maxwell's equations. A careful consideration then shows that a quasistatic approximation only introduces a small error. A detailed discussion on this topic can be found in [2], pp. 425-427.

[1] The name derives from the ancient Greek word enképhalos („within the head“)

As [2] then shows, the quasistatic approximation can be used to derive the following partial differential equation for the electric potential. For the rest of this Master's thesis, whenever we talk about the EEG forward problem, we are talking about *this* partial differential equation with boundary conditions.

Definition 2 (EEG forward problem)**.** Let Ω be the head domain and $\Gamma = \partial\Omega$ its boundary. Let u be the electric potential, σ the conductivity and j^P the neural activity. Let n be the unit outer normal. We then have the following partial differential equation for the electric potential u.

$$\begin{aligned} \operatorname{div}(\sigma \nabla u) &= \operatorname{div}\left(j^P\right) && \text{in } \Omega \\ \langle \sigma \nabla u, n \rangle &= 0 && \text{on } \Gamma \end{aligned}$$

We now have a mathematical formulation of the EEG forward problem. It remains to derive a mathematical formulation of the MEG forward problem. We give a derivation that leads to the formulation as it is implemented in the forward modeling software toolbox *duneuro*, see e.g. [3].

As described in [4], pp. 1103-1105, a MEG magnetometer consists of a conduction loop Υ that encloses some surface F. The magnetometer then measures the magnetic flux through this surface, or more concretely if B is the magnetic field and n the unit outer normal of the surface F, the magnetometer measures the value

$$\int_F \langle B, n \rangle \, dS.$$

To compute this integral numerically, one chooses a suitable quadrature rule, which means a set of point-weight pairs $\{(s_1, w_1), ..., (s_n, w_n)\}$, and makes the approximation

$$\int_F \langle B, n \rangle \, dS \approx \sum_{i=1}^{n} w_i \cdot \langle B(s_i), n(s_i) \rangle.$$

We thus see that after choosing quadrature rules for the different magnetometers the central part in the computation of the magnetic fluxes is given by the computation of the values

$$\langle B(s_i), v_i \rangle$$

for a number of pairs (s_i, v_i). Hence our goal is to derive some way to compute approximations to these values.

Our starting point is *Biot-Savarts Law*, which can be stated as

$$B(x) = \frac{\mu_0}{4\pi} \int_\Omega j(y) \times \frac{x - y}{\|x - y\|^3} \, d\lambda(y),$$

where μ_0 is the vacuum permeability and j is the current density in the head domain, see e.g. [5]. The standard approach in bioelectromagnetism, as it is for example described in [2], is now to split the current density j into two parts. First the neural activity j^P, and secondly the passive volume currents generated by the electric field σE, so that in total we have

$$j = j^P + \sigma E = j^P - \sigma \nabla u,$$

where u is the electric potential. This decomposition was also tacitly used in our derivation of the EEG forward problem above. Again following [2], it is further common to model the neural activity j^P as a *current dipole*. Concretely this means that we make the approximation

$$j^P = M \cdot \delta_{x_0},$$

where M is some vector called the *Moment* that models the strength and direction of current flow, x_0 is the position of the dipole and δ is the Dirac delta distribution. Applying this approach to Biot-Savart's Law gives

$$\begin{aligned}
B(x) &= \frac{\mu_0}{4\pi}\int_\Omega j(y)\times\frac{x-y}{\|x-y\|^3}\,d\lambda(y)\\
&= \frac{\mu_0}{4\pi}\int_\Omega \left(j^P(y)-\sigma\nabla u\right)\times\frac{x-y}{\|x-y\|^3}\,d\lambda(y)\\
&= \frac{\mu_0}{4\pi}\int_\Omega M\delta_{x_0}\times\frac{x-y}{\|x-y\|^3}\,d\lambda(y) - \frac{\mu_0}{4\pi}\int_\Omega \sigma\nabla u\times\frac{x-y}{\|x-y\|^3}\,d\lambda(y)\\
&= \underbrace{\frac{\mu_0}{4\pi}M\times\frac{x-x_0}{\|x-x_0\|}}_{=:B^P(x)} \underbrace{-\frac{\mu_0}{4\pi}\int_\Omega \sigma\nabla u\times\frac{x-y}{\|x-y\|^3}\,d\lambda(y)}_{=:B^S(x)}\\
&= B^P(x)+B^S(x).
\end{aligned}$$

We call $B^P(x)$ the *primary magnetic field* and $B^S(x)$ the *secondary magnetic field*. We thus get

$$\langle B(x),v\rangle = \langle B^P(x),v\rangle + \langle B^S(x),v\rangle,$$

and to compute the magnetic flux we need to compute values of the form

$$\langle B^P(x),v\rangle = \frac{\mu_0}{4\pi}\langle M\times\frac{x-x_0}{\|x-x_0\|},v\rangle$$

and

$$\langle B^S(x),v\rangle = -\frac{\mu_0}{4\pi}\int_\Omega \langle\sigma\nabla u\times\frac{x-y}{\|x-y\|^3},v\rangle\,d\lambda(y).$$

The formula for the term $\langle B^P(x),v\rangle$ is easy to implement and fast to compute and thus causes no problems.

The term $\langle B^S(x),v\rangle$ is more complicated, as it involves the potential u and an integral over the head domain.

We have now derived the MEG forward problem.

Definition 3 (MEG forward problem)**.** Let Ω be the head domain and $\Gamma=\partial\Omega$ its boundary. Let u be the electric potential inside the head, σ the conductivity tensor, $j^P=M\delta_{x_0}$ the neural activity and B the magnetic field. Then the magnetic field at position x in direction v is given by

$$\langle B(x),v\rangle = \langle B^P(x),v\rangle + \langle B^S(x),v\rangle,$$

where

$$\langle B^P(x),v\rangle = \frac{\mu_0}{4\pi}\langle M\times\frac{x-x_0}{\|x-x_0\|},v\rangle$$

and

$$\langle B^S(x),v\rangle = -\frac{\mu_0}{4\pi}\int_\Omega \langle\sigma\nabla u\times\frac{x-y}{\|x-y\|^3},v\rangle\,d\lambda(y).$$

When actually using the above formulas to compute the magnetic field we of course use some numerical approximation of the electric potential.

2.2 The finite element method for the EEG forward problem

We have seen that the computation of the electric potential is the central part in solving the forward problem, as it is needed for simulating the electrode measurements as well as for simulating the magnetometer measurements. But we have yet to discuss how we can actually compute the potential. As we have seen in section 1.2 the electric potential is given by the differential equation

$$\begin{aligned}
\operatorname{div}(\sigma\nabla u) &= \operatorname{div}\left(j^P\right) && \text{in } \Omega\\
\langle\sigma\nabla u,n\rangle &= 0 && \text{on } \Gamma
\end{aligned}$$

In this master's thesis, we follow the example of duneuro and focus on computing numerical solutions to this problem using the *finite element method*. The first step in this approach is to replace the differential formulation of the EEG forward problem with a so-called *weak formulation*. There are numerous ways to associate a weak formulation to the differential equation, as can be seen by looking at [6], [7] or [8]. All of these approaches lead to a problem of the following form.

Definition 4 (General form of the weak formulation)**.** One defines a function space H, a bilinear form a on H and a linear functional l on H. Then a function $u \in H$ is called a *weak solution*, if it fulfills

$$a(u, v) = l(v) \qquad \text{for all } v \in H$$

For this definition to make sense the choice of H, a and l has to be related to the differential equation in some sensible way. Typically, one uses some version of the fundamental lemma of the calculus of variations and then applies partial integration. Some approaches additionally add penalty terms to the form a to enforce certain constraints.

The space H is typically infinite dimensional. It turns out that in many cases one can show that there is exactly one weak solution, and there is a natural way to compute approximations to this weak solution in finite dimensional subspaces. This is based on a variational formulation of what it means to be a "weak solution".

Theorem 5 (Characterization Theorem)**.** Let V be a linear space, a a bilinear form on V and l a linear functional on V. We further assume a to be symmetric positive definite. Let $u \in V$. Then the following are equivalent.

1. We have $a(u, v) = l(v)$ for all $v \in V$
2. The functional
$$v \mapsto \frac{1}{2}a(v, v) - l(v)$$
attains its minimum over V at u

Furthermore there is at most one $u \in V$ with $a(u, v) = l(v)$ for all $v \in V$.

Proof. A proof of this theorem can be found in [9], 2.2. □

This theorem is the basis for proving existence of weak solutions and for constructing approximate solutions via the finite element method. We first talk about the existence of weak solutions. For this, we need some mathematical notion.

Definition 6. Let E be a normed vector space and let a be a bilinear form on E. We then call a *continuous* if there is a constant C, so that for all $x, y \in E$ we have

$$|a(x, y)| \leq C\|x\|\|y\|.$$

Now let a be symmetric. We say that a is *coercive* if there exists some $\epsilon > 0$ so that for all $x \in E$ we have

$$a(x, x) \geq \epsilon\|x\|^2.$$

We see that every coercive form is also symmetric positive definite.

In practice, the bilinear form in a weak formulation is often continuous and coercive, or becomes so once restricted to a suitable subspace. This gives importance to the following theorem.

Theorem 7 (Lax-Milgram)**.** Let H be a Hilbert space, a a continuous, coercive bilinear form on H and l a continuous linear functional on H. Then there exists a unique $u \in H$ with

$$u = \arg\min_{v \in H} \frac{1}{2}a(v, v) - l(v).$$

By the characterization theorem this is equivalent to

$$a(u, v) = l(v) \qquad \text{for all } v \in H$$

Proof. A proof can be found in [9], 2.5. □

As illustrated above this theorem shows that for many weak formulations there exists a unique weak solution. But the combination of the characterization theorem and the Lax-Milgram theorem also naturally leads to a method for computing approximations to the weak solution. As the weak solution u is given by

$$u = \arg\min_{v \in H} \frac{1}{2} a(v,v) - l(v),$$

we can try to compute an approximation to this $u \in H$ by choosing some suitable finite dimensional subspace $U \subset H$ and try to find a solution $u_h \in U$ to the problem

$$u_h = \arg\min_{v \in U} \frac{1}{2} a(v,v) - l(v),$$

which by the characterization theorem is equivalent to

$$a(u_h, v) = l(v) \qquad \text{for all } v \in U.$$

This approach to computing approximations to the weak solution is called the *Ritz-Galerkin Method*. The natural question is then of course how to choose the finite dimensional subspace in a sensible way. The usual approach is to split the underlying domain Ω into simple subdomains, e.g. tetrahedra or hexahedra, and define the finite dimensional subspace to be the space of functions that have a simple structure on every subdomain and fulfill some global condition like continuity. A Ritz-Galerkin method using such an approach is typically called a *Finite Element Method*, or *FEM*.

Now every finite dimensional subspace $U \subset H$ has a basis $\{\varphi_1, ..., \varphi_n\}$, and since a is bilinear and l is linear we see that

$$a(u_h, v) = l(v) \qquad \text{for all } v \in U.$$

is equivalent to

$$a(u_h, \varphi_i) = l(\varphi_i) \qquad \text{for } 1 \leq i \leq n.$$

We search for an $u_h \in U$, so if such an u_h exists there must be some representation $u_h = \sum_{j=1}^n x_j \varphi_j$, and we see that the condition above is equivalent to

$$\sum_{j=1}^n a(\varphi_j, \varphi_i) x_j = l(\varphi_i) \qquad \text{for } 1 \leq i \leq n.$$

Lemma 8. If we set $A = (a(\varphi_j, \varphi_i))_{1 \leq i,j \leq n}$, $b = (l(\varphi_i))_{1 \leq i \leq n}$ and

$$\Phi : \mathbb{R}^n \xrightarrow{\cong} U; x \mapsto \sum_{j=1}^n x_j \varphi_j,$$

we see that for $x \in \mathbb{R}^n$ the following statements are equivalent.

1. $a(\Phi(x), v) = l(v)$ for all $v \in U$
2. $Ax = b$

We thus see that computing approximations to the weak solution using a Ritz-Galerkin method comes down to solving a linear system. The matrix A in this linear system is typically called *stiffness matrix*.

There are a few caveats when trying to apply the finite element method to the EEG forward problem. We are illustrating these problems and their solution in the context of a *Continuous Galerkin FEM* approach, but similar considerations are also valid for other FEM approaches. Details on this approach can e.g. be found in [10]. In this setting, we search for weak solutions in the Sobolev space $H^1(\Omega)$, and the bilinear form is given by

$$a : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}; (u,v) \mapsto \int_\Omega \langle \sigma \nabla u, \nabla v \rangle \, d\lambda.$$

Furthermore, if $\{\varphi_1, ..., \varphi_n\}$ is the basis of the corresponding finite element space as it is constructed in [10], we have

$$1 = \sum_{j=1}^n \varphi_j,$$

and the linear functional[2] is continuous and fulfills $l(1) = 0$. This last condition is actually necessary for a weak solution to exist, since for a weak solution u we have

$$l(1) = a(u, 1) = \int_\Omega \langle \sigma \nabla u, \nabla 1 \rangle \, d\lambda = 0.$$

When performing an EEG measurement one has to make the (mathematically) arbitrary choice of designating a reference electrode. Different choices for the reference electrode lead to potentials that are shifted by some constant. As the weak solution is an approximation[3] to the electric potential in the head, and the result of a finite element method is an approximation to the weak solution, we thus cannot expect the weak solution or the finite element solution to be uniquely determined. We instead expect them to reflect the arbitrary choice of a reference electrode in some way. We now want to investigate the structure of the sets of weak solutions and finite element solutions.

The central tool in proving the existence of weak solutions is the Lax-Milgram theorem. To apply this theorem, one needs a Hilbert space and a continuous coercive form on this Hilbert space. And while the Sobolev space $H^1(\Omega)$ is a Hilbert space, the form a is not coercive, as we have

$$a(1, v) = \int_\Omega \underbrace{\langle \sigma \nabla 1, \nabla v \rangle}_{=0} \, d\lambda = 0 = a(v, 1),$$

and thus in particular $a(1, 1) = 0$. But it turns out that the constant functions are the only obstacle to coercivity.

Lemma 9. We define

$$H^1_*(\Omega) = \{v \in H^1(\Omega) \mid \int_\Omega v \, d\lambda = 0\}$$

to be the space of all Sovolev functions with mean 0. Then a is continuous on $H^1(\Omega)$, and the restricted form $a\big|_{H^1_*(\Omega) \times H^1_*(\Omega)}$ is coercive.

Proof. [12], Lemmata 3.3 and 3.5 □

We can now prove that the arbitrary choice of a reference electrode is reflected in the weak solution.

Theorem 10. Let $\mathcal{L}$ be the set of solutions u to the problem

$$a(u, v) = l(v) \qquad \text{for all } v \in H^1(\Omega),$$

where a and l are defined as above. Then $\mathcal{L}$ is a 1-dimensional affine space. More concretely, there is some $u_0 \in H^1(\Omega)$ so that

$$\mathcal{L} = u_0 + \mathbb{R} \cdot 1.$$

We thus see that the weak solution is unique up to a constant, which corresponds to the free choice of a reference electrode.

Proof. Since Ω is bounded, integration is continuous with respect to the Sovolev norm $\| \cdot \|_{H^1(\Omega)}$. Thus $H^1_*(\Omega)$ is closed, and since $H^1_*(\Omega)$ is a linear subspace of $H^1(\Omega)$ we see that $H^1_*(\Omega)$ is again a Hilbert space. Since a restricted to $H^1_*(\Omega)$ is continuous and coercive, and the restriction of l to $H^1_*(\Omega)$ is continuous as well, the Lax-Milgram theorem implies that there is a unique $u_0 \in H^1_*(\Omega)$ with

$$a(u_0, v) = l(v) \qquad \text{for all } v \in H^1_*(\Omega).$$

Now let $v \in H^1(\Omega)$ be arbitrary. By subtracting the mean $c \in \mathbb{R}$ of v we can achieve $v - c \cdot 1 \in H^1_*(\Omega)$. Since a and l vanish on constants we get

$$a(u_0, v) = a(u_0, v - c \cdot 1) = l(v - c \cdot 1) = l(v).$$

We thus see that u_0 is a weak solution.

Let u be an arbitrary weak solution. Since a vanishes on constant functions one immediately sees that $u + c \cdot 1$ is also a weak solution for every $c \in \mathbb{R}$. In particular there is a c so that $u + c \cdot 1 \in H^1_*(\Omega)$. But then we have

[2] Only specifying that we are using a CG-FEM approach does not determine the linear functional l. This functional is derived by modeling the neural activity. For this, there are various different approaches called *source models*. Details can be found in [11]

[3] as a solution of the quasistatic approximation of the Maxwell equations

$$a(u + c \cdot 1, v) = l(v) \qquad \text{for all } v \in H^1_*(\Omega).$$

But u_0 is the unique element in $H^1_*(\Omega)$ with this property, and we thus get $u_0 = u + c \cdot 1$, meaning $u \in u_0 + \mathbb{R} \cdot 1$. □

It is now natural to ask if the set of finite element solutions reflects the arbitrary choice of a reference electrode in a similar way. The next theorem shows that this is indeed the case.

Theorem 11. Let $U \subset H^1(\Omega)$ be a finite dimensional linear subspace with $1 \in U$. Let

$$\mathcal{L}_{\mathrm{FE}} = \{u \in U \mid a(u, v) = l(v) \text{ for all } v \in U\}$$

be the set of all finite element solutions with respect to the subspace U. Then there exists some $u_0 \in U$ so that we have

$$\mathcal{L}_{\mathrm{FE}} = u_0 + \mathbb{R} \cdot 1.$$

Remark. This theorem is true for all linear functionals l, we do not need to assume continuity.

Proof. Every finite dimensional linear subspace of a topological vector space is closed, and every linear map from this subspace to another topological vector space is continuous, look e.g. at [13], Theorem 1.21. Thus U and $U \cap H^1_*(\Omega)$ are Hilbert subspaces of $H^1(\Omega)$, and the restriction of l to U is continuous, regardless of whether the functional was continuous on $H^1(\Omega)$. Since the form a is coercive of $H^1_*(\Omega)$, it is also coercive on $U \cap H^1_*(\Omega)$. By the Lax-Milgram theorem we see that there is a unique $u_0 \in U \cap H^1_*(\Omega)$ with

$$a(u_0, v) = l(v) \qquad \text{for all } v \in U \cap H^1_*(\Omega).$$

Since a and l vanish on constant functions it follows that $u_0 \in \mathcal{L}_{\mathrm{FE}}$, which can be shown in the same way as the corresponding statement in Theorem 10. The same argument as in Theorem 10 also shows that the finite element solution is unique up to a constant. □

We thus see that the arbitrary choice of a reference electrode is reflected in the set of weak solutions as well as in the set of finite element solutions. And while this is desirable for a faithful modeling, it is somewhat problematic for the computation of finite element solutions.

We typically don't search for a finite element solution u directly, but instead perform the computations in coordinates, meaning we solve the linear system

$$Ax = b,$$

where A, b are defined as in Lemma 8. We can then evaluate the finite element solution via the isomorphism $\Phi(x)$. We want to discuss this process in a little more detail.

2.3 Solving the linear system

We begin by investigating the structure of the solution set of the linear system $Ax = b$ and the properties of the matrix A.

Lemma 12. Let $\mathcal{M}$ be the set of solutions of the linear system

$$Ax = b,$$

where A, b are defined as in Lemma 8. Then there exists some $x_0 \in \mathbb{R}^n$ so that we have

$$\mathcal{M} = x_0 + \mathbb{R} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Proof. Let $u \in U$. By Lemma 8 u is a finite element solution if and only if $A\Phi^{-1}(u) = b$. If $\mathcal{L}_{\mathrm{FE}}$ is the set of finite element solutions we thus get that $\mathcal{M} = \Phi^{-1}(\mathcal{L}_{\mathrm{FE}})$. With Theorem 11 we now get

$$\mathcal{M} = \Phi^{-1}(u_0 + \mathbb{R} \cdot 1) = \Phi^{-1}(u_0) + \mathbb{R} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix},$$

where the last equality follows from $1 = \sum_{j=1}^{n} \varphi_j = \Phi\left(\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}\right)$. □

We thus see that the matrix A is not invertible, as the solution of the linear system of interest is not unique. But we can say the following.

Lemma 13. We use the notation of Lemma 8.

1. The matrix A is symmetric positive semidefinite
2. $\ker(A) = \mathbb{R} \cdot \begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}$
3. Let $1 \leq i \leq n$. Let A_i be the matrix obtained by deleting the i-th row of A. Let b_i be defined analogously. Then for $x \in \mathbb{R}^n$ the following are equivalent.
   (a) $Ax = b$
   (b) $A_i x = b_i$

Proof. 1): We have

$$a(u,v) = \int_\Omega \langle \sigma \nabla u, \nabla v \rangle \, d\lambda = \int_\Omega \langle \sigma \nabla v, \nabla u \rangle \, d\lambda = a(v,u),$$

where we have used the fact that the conductivity σ is symmetric positive definite. Furthermore, we have

$$a(u,u) = \int_\Omega \langle \sigma \nabla u, \nabla u \rangle \, d\lambda \geq 0.$$

This implies

$$a_{i,j} = a(\varphi_j, \varphi_i) = a(\varphi_i, \varphi_j) = a_{j,i},$$

meaning A is symmetric, and

$$\langle Ax, x \rangle = \sum_{i,j=1}^{n} a_{i,j} x_j x_i = \sum_{i,j=1}^{n} a(\varphi_j, \varphi_i) x_j x_i = a(\Phi(x), \Phi(x)) \geq 0.$$

We have now shown that A is symmetric positive semidefinite.

2): Since the solution set of the system $Ax = b$ is given by

$$\text{special solution} + \ker(A),$$

and affine subspaces uniquely determine the corresponding linear subspace, this statement follows from lemma 12.

3): We first show $(a) \iff (b)$. Obviously $(a) \implies (b)$. So let $1 \leq i \leq n$ and let x be a solution of $A_i x = b_i$. To show that x fulfills $Ax = b$, we need to show that the condition given by the i-th row of this system is fulfilled. We can compute

$$\begin{aligned}
\sum_{j=1}^{n} a_{i,j} x_j &= a(\Phi(x), \varphi_i) = a(\Phi(x), 1 - \sum_{\substack{k=1 \\ k \neq i}}^{n} \varphi_k) \\
&= -\sum_{\substack{k=1 \\ k \neq i}}^{n} a(\Phi(x), \varphi_k) = -\sum_{\substack{k=1 \\ k \neq i}}^{n} l(\varphi_k) \\
&= l(1 - \sum_{\substack{k=1 \\ k \neq i}}^{n} \varphi_k) = l(\varphi_i).
\end{aligned}$$

We thus see that $Ax = b$. □

To compute finite element approximations we need to solve the system $Ax = b$. As for finite element approaches the matrix A is typically large and sparse, the system lends itself to be solved by Krylov subspace methods. Since the matrix A is symmetric and positive semidefinite, one should use solvers that can exploit these features in some way. The *conjugate gradient* method is quite popular in this regard, as one can see e.g. by looking at [9]. The problem is that in most lectures and textbooks the conjugate gradient method is derived and studied for *positive definite* matrices, and it is not immediately clear how the methods behave in the context of positive semidefinite matrices. We first state the algorithm[4] and then discuss how to apply it to positive semidefinite matrices. The algorithm is a slightly adapted version of the one given in [14], Algorithm 6.18.

Algorithm 1 Conjugate Gradient Method

Require: - Finite dimensional Hilbert space H
- SPD operator $T : H \to H$
- Right hand side $y \in H$
- Initial guess $x_0 \in H$

$r_0 \leftarrow y - T(x_0)$
$p_0 \leftarrow r_0$
for $k = 0, 1, 2, ...$ until convergence **do**
$\alpha_k \leftarrow \frac{\langle r_k, r_k \rangle}{\langle T(p_k), p_k \rangle}$
$x_{k+1} \leftarrow x_k + \alpha_k p_k$
$r_{k+1} \leftarrow r_k - \alpha_k T(p_k)$
$\beta_k \leftarrow \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$
$p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
end for

In the above algorithm one typically checks for convergence by specifying a desired *residual reduction*. More concretely, one specifies some $\epsilon > 0$ and computes iterates up to the point where we have

$$\|r_k\| \leq \epsilon \cdot \|r_0\|.$$

A popular choice for ϵ is 10^{-10}. One then uses x_k as an approximate solution to the problem $T(x) = y$. A lot can (and will) be said about the convergence properties of this algorithm, but for now, it shall suffice to say that the iterates $x_0, x_1, ...$ of the conjugate gradient algorithm converge to the exact solution $T^{-1}(y)$, at least in exact arithmetic.

One natural approach to use this algorithm in the case of a system $Ax = b$ with A symmetric positive semidefinite would be to use the fact that

$$A|_{\mathrm{Im}(A)} : \mathrm{Im}(A) \to \mathrm{Im}(A); x \mapsto Ax$$

defines a symmetric positive definite operator. This follows from the fact that for symmetric matrices we have $\mathbb{R}^n = \ker(A) \bigoplus \mathrm{Im}(A)$, thus the restriction of A to its image is injective and hence the operator is bijective. Since A is symmetric and positive semidefinite, the same is true for the restriction of A to $\mathrm{Im}(A)$. In total A is a bijective symmetric positive semidefinite operator and thus symmetric positive definite.

Now the conjugate gradient method can be applied to the equation

$$A|_{\mathrm{Im}(A)}(x) = b$$

in the Hilbert space $\mathrm{Im}(A)$ and by standard theory converges to its unique solution. Since applying the operator is multiplication by the matrix A and the Hilbert space structure on $\mathrm{Im}(A)$ is the same as on $\mathbb{R}^n$, applying the conjugate gradient algorithm on the subspace $\mathrm{Im}(A)$ produces the same iterates as directly applying the conjugate gradient algorithm on the whole space H with an initial guess inside the subspace $\mathrm{Im}(A)$.

Now let's look at what happens when we apply the conjugate gradient algorithm with some arbitrary initial guess $x_0 \in H$. As we have $\mathbb{R}^n = \ker(A) \bigoplus \mathrm{Im}(A)$, we can split $x_0 = x_0^{\ker} + x_0^{\mathrm{Im}}$. Now let $x_k, k = 0, 1, ...$ be the iterates obtained by applying the conjugate gradient algorithm to the equation $Ax = b$ with initial guess x_0, and let $x_k^{\mathrm{Im}}, k = 0, 1, ...$ be the iterates obtained by applying the conjugate gradient algorithm with initial guess x_0^{Im}. By looking at the definition of the conjugate gradient algorithm one can easily convince oneself that we then have

$$x_k = x_0^{\ker} + x_k^{\mathrm{Im}}.$$

Since x_k^{Im} converges to a solution of $Ax = b$ and $x_0^{\ker}$ is inside the kernel of A, we see that x_k also converges to a solution of $Ax = b$. We have thus proven the following.

[4] In later chapters we will derive and investigate this algorithm and its generalization, the *Block Conjugate Gradient method*, in more detail, but for the present discussion giving the algorithm shall suffice. We will also ignore preconditioning for now.

Theorem 14. Let A be a symmetric positive semidefinite matrix, and let $b \in \mathrm{Im}(A)$. Then, for an arbitrary initial guess $x_0 \in \mathbb{R}^n$, the conjugate gradient algorithm applied to the problem $Ax = b$ converges to a solution. If we have $x_0 \in \mathrm{Im}(A)$, the iterates converge to the unique solution of $Ax = b$ inside $\mathrm{Im}(A)$.

By standard theory and since A is symmetric, the unique solution inside $\mathrm{Im}(A)$ is the minimum norm solution of $Ax = b$. So if we e.g. use the conjugate gradient algorithm with initial guess $0 \in \mathbb{R}^n$, the iterates converge to the minimum norm solution of the underlying linear system for every symmetric positive semidefinite matrix.

In the positive definite case, the speed of convergence is linked to the condition number of the matrix of the underlying system, which is given by the quotient of the largest and the smallest eigenvalue. Since the convergence in the positive semidefinite case can be traced back to applying the conjugate gradient method to the positive definite restriction of A to $\mathrm{Im}(A)$, the convergence theory in the positive semidefinite case can be derived from the convergence theory in the positive definite case. Hence the convergence depends on the condition number of $A|_{\mathrm{Im}(A)}$, which is simply the quotient of the largest and smallest non-zero eigenvalues of A.

We thus see that it is a valid approach to solve the system $Ax = b$ arising from a finite element discretization of the EEG forward problem by directly applying the conjugate gradient method.

But if we look at the implementation inside duneuro[5] we see that this is not the default method for solving the linear system, at least at the time of writing this thesis. The approach inside duneuro is instead based upon Lemma 13.3). There it was shown that the solutions of the system $Ax = b$ are the same as the solutions of the system $A_1 x = b_1$, where A_1 and b_1 are obtained from A and b by deleting the first row. Now the set of solutions of the equation $Ax = b$, or of the equation $A_1 x = b_1$ respectively, is a 1-dimensional affine family. If we now substitute the first row in A, which defines a redundant condition, with another linear condition that determines some specified member of the 1-dimensional affine family, we can arrive at a linear system with a unique solution.

One natural approach might for example be to specify the minimum norm solution. As this is the unique solution inside the image of A, which is the orthogonal space of the vector $(1, ..., 1)$, a linear condition that pinpoints this solution is

$$\sum_{i=1}^{n} x_i = 0.$$

If we thus substitute every entry in the first row with 1 and the first entry in the right hand side b with 0, the corresponding system $\widetilde{A}x = \widetilde{b}$ has as its unique solution the minimum norm solution of the equation $Ax = b$. But this is actually a very poor idea, mainly because it destroys some of the nice properties of A. The new matrix $\widetilde{A}$ has a fully occupied first row, and even more importantly the matrix is no longer symmetric. We can thus no longer use the conjugate gradient method to solve the linear system.

We thus want our new linear condition to lead to a matrix that is symmetric positive definite, so we can apply the conjugate gradient method. The trouble with this approach is that for our matrix to be symmetric, the first row has to match the first columns, but we cannot simply change the first column without changing the sets of solutions of the linear system. Now the central idea is that if the first entry of the solution is 0, the validity of the statement $Ax = b$ does not depend on the first column of A. In this setting, we can thus change the column of A without any problem, and we can thus change it in such a way that it fits the linear condition in the first row that fixes the first entry of x to be zero. We now make this more concrete.

Theorem 15. We use the notation of Lemma 8. Then there is exactly one solution $x \in \mathbb{R}^n$ of the linear system $Ax = b$ where the first component of x is 0.

If we then set

$$\hat{A} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n,2} & \dots & a_{n,n} \end{pmatrix}$$

and

$$\hat{b} = \begin{pmatrix} 0 \\ b_2 \\ \vdots \\ b_n \end{pmatrix},$$

[5] https://gitlab.dune-project.org/duneuro/duneuro/-/blob/master/duneuro/common/linear_problem_solver.hh

the matrix $\hat{A}$ is symmetric positive definite and x is given as the unique solution of the system

$$\hat{A}x = \hat{b}.$$

Remark. The theorem remains true if we substitute the i-th row and column instead of the first row and column, with the same proof.

Proof. In Lemma 12 we have shown that the solution set of the equation $Ax = b$ has the form

$$\mathcal{M} = x_{\text{special}} + \mathbb{R} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

We thus see that there is exactly one solution x of the equation whose first component is 0.

Now let $y \in \mathbb{R}^n$. We now show the following equivalence.

1. y fulfills $Ay = b$ and the first component of y is zero.
2. y fulfills $\hat{A}y = \hat{b}$.

First let y be a solution of $\hat{A}y = \hat{b}$. Looking at the first row of $\hat{A}$ and $\hat{b}$ we see that the first component of y is zero. Let $y = (y_i)_{1 \leq i \leq n}$ and $\hat{A} = (\hat{a}_{i,j})_{1 \leq i,j \leq n}$. We can now compute for $2 \leq i \leq n$

$$\sum_{j=1}^{n} a_{i,j} y_j \overset{y_1=0}{=} \sum_{j=2}^{n} a_{i,j} y_j = \sum_{j=2}^{n} \hat{a}_{i,j} y_j \overset{y_1=0}{=} \sum_{j=1}^{n} \hat{a}_{i,j} y_j = b_i.$$

Let A_1 be the matrix obtained by deleting the first row of A, and let $\widetilde{b}$ be the vector obtained by deleting the first entry of b. The calculation above shows that y fulfills $A_1 y = \widetilde{b}$. By theorem 13.3) we thus see that y fulfills $Ay = b$. We have thus shown 2) $\implies$ 1). If we have 1), the fact that the first component of y is zero implies that the linear condition defined by the first row of $\hat{A}y = \hat{b}$ is fulfilled. The same calculation as above then shows that the conditions corresponding to the other rows are also fulfilled, and we thus have $\hat{A}y = \hat{b}$.

As x is the unique solution of 1), we see that x is also given as the unique solution of $\hat{A}x = \hat{b}$. As this linear system has a unique solution, we see that $\hat{A}$ is invertible. One can also directly see from the definition that $\hat{A}$ is symmetric, since A is symmetric.

Now let $z = (z_i)_{1 \leq i \leq n} \in \mathbb{R}^n$. We then have

$$\begin{aligned} \langle \hat{A}z, z \rangle &= \sum_{i,j=1}^{n} \widetilde{a}_{i,j} z_j z_i = z_1^2 + \sum_{i,j=2}^{n} a_{i,j} z_j z_i \\ &= z_1^2 + a(\sum_{j=2}^{n} z_j \varphi_j, \sum_{j=2}^{n} z_i \varphi_i) \geq 0. \end{aligned}$$

This shows that $\hat{A}$ is positive semidefinite. Hence $\hat{A}$ is invertible and positive semidefinite and thus positive definite. □

We thus get another approach to solve the EEG forward problem. We first compute the stiffness matrix A and the right hand side b, then we compute $\hat{A}$ and $\hat{b}$ and then we finally solve $\hat{A}x = \hat{b}$ using a conjugate gradient method.

This is the approach duneuro uses per default. This of course raises the question of whether the two approaches we discussed differ. We want to investigate this briefly.

For this, we take a look at the EEG forward problem for a realistic head model. More concretely we use the tetrahedral head model from [15] and use a CG FEM approach with piecewise linear ansatz functions to compute a stiffness matrix A. We now want to solve some linear systems $Ax = b$, where we take b to be a right hand side so that the solution of this system defines a row in the EEG transfer matrix. We have not yet defined the transfer matrix, but for now it suffices to know that in this concrete instance there are two indices i, j so that $b = (\delta_i(k) - \delta_j(k))_{1 \leq k \leq n}$. Here i is the index of the node closest to the electrode whose row we want to compute, and j is the index of the node closest to the reference electrode. Note that, since the sum of the entries of the vector is zero, this vector is indeed contained inside the image of A, at least in exact arithmetic. As mentioned above, the first approach is to directly apply the CG method on the system $Ax = b$,

in the following called *freeDOF* (for „free degree of freedom“). The second approach is to adjust A and b to the matrices $\hat{A}$ and $\hat{b}$ according to Theorem 15 and solve the system $\hat{A}x = \hat{b}$ using a conjugate gradient method, in the following called *fixedDOF*. In both cases we use an AMG preconditioner using the default parameters from the ISTLBackend_SEQ_AMG class from dune-PDELab[6]. For the AMG preconditioner we use a BiCGSTAB solver as the coarse solver on the coarsest grid. Furthermore, we aim for a residual reduction of 10^{-10}.

We use the electrode at position $(119.02, 19.10, 58.21)$ as reference electrode. When solving for the right hand side corresponding to the electrode position $(111.23, -43.73, 57.68)$ the evolution of the residual reduction over the iterations is illustrated in figure 1.

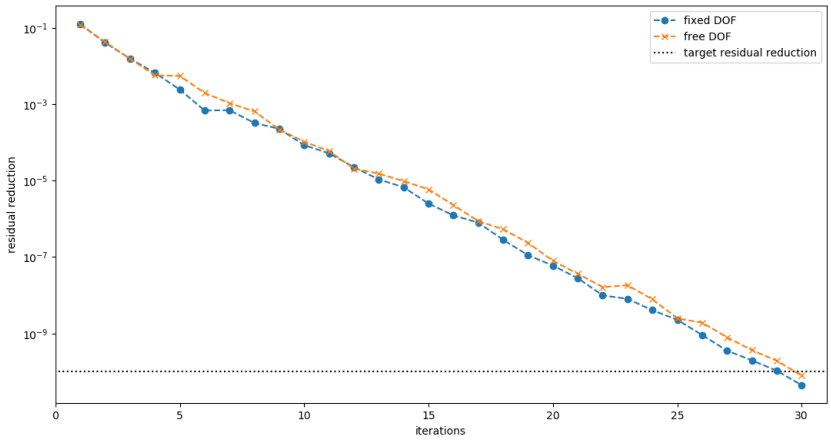

Figure 1: Development of residual reduction for the two different approaches for solving the linear system $Ax = b$ using an AMG preconditioner.

We see that both methods converge in 30 iterations and with a similar speed. Moreover, after projecting the solution of $\hat{A}x = \hat{b}$ onto the image of the matrix A, the relative error between the approximate solutions computed by the two approaches is of the order 10^{-7}. Furthermore, in every run on the author's machine, the times needed to solve the linear systems were within a 5% range of one another.

This behavior is typical. There are 73 electrode positions associated with the given head model. There are thus 72 different non-reference electrodes defining right hand sides. When running the experiment we see that for 71 of the non-reference electrodes both algorithms converge and the evolution of the residual reductions is essentially the same as the one seen in figure 1. But for one electrode, namely the one at position $(-22.32, 33.92, 127.57)$, the direct application of the conjugate gradient method to the system $Ax = b$ crashes. The reason for this crash is that the direct solver on the coarse grid of the AMG preconditioner fails to converge. This is not an inherent problem with the corresponding right hand side b, since if one tries to solve the system $Ax = b$ directly using a conjugate gradient method with an SSOR preconditioner the algorithm converges. The evolution of the residual reductions for both approaches using an SSOR preconditioner for the right hand side given by this electrode is illustrated in figure 2.

The author also investigated the performance of the two different approaches with right hand sides coming from the computation of the MEG transfer matrix, with random right hand sides in the image of the matrix A, and in spherical head models. The results were parallel to the ones seen above. In most cases, both approaches converge and are virtually identical performance-wise. But sometimes the application of the direct method for $Ax = b$ crashes because the coarse grid solver fails to converge. The $\hat{A}x = \hat{b}$ approach converged in every investigated case.

There are different ways to proceed. One way would be to investigate if different AMG parameters prevent a crash. Another way would be to fall back to a SSOR[7] preconditioned conjugate gradient method if the AMG preconditioned version fails. Yet another approach would be to test different solvers on the coarse grid. Perhaps the simplest way is to follow in the footsteps of duneuro and to default to solving the EEG forward problem by solving the linear system $\hat{A}x = \hat{b}$. This is the approach we use in the rest of this thesis. In the following, we call $\hat{A}$ the *modified stiffness matrix*.

[6] https://gitlab.dune-project.org/pdelab/dune-pdelab/-/blob/master/dune/pdelab/backend/istl/seqistlsolverbackend.hh
[7] or whatever preconditioner you prefer

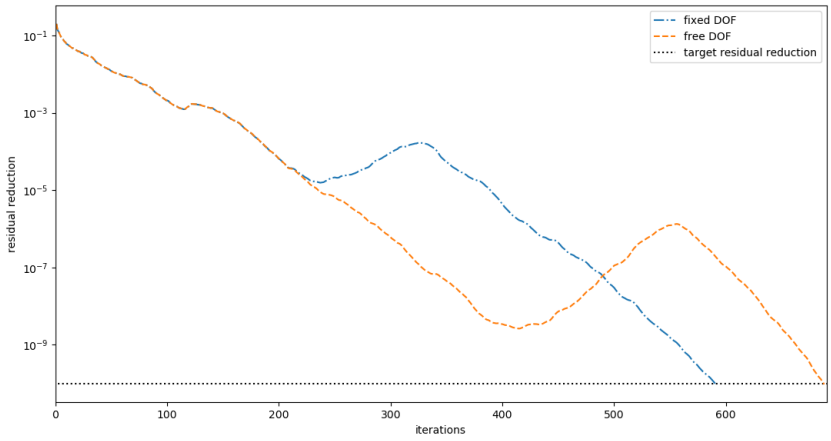

Figure 2: Development of residual reduction for the two different approaches for solving the linear system $Ax = b$ using an SSOR preconditioner.

There is still another common approach for dealing with the non-invertability of the stiffness matrix A. The basic idea is for example described in the diploma thesis [16]. The central fact used in the existence proofs above is that the underlying bilinear form a is coercive on the space $H^1_*(\Omega)$. If now $U \subset H^1(\Omega)$ is some finite dimensional FEM ansatz space, it was shown in the proof of Theorem 11 that there is exactly one finite element solution u_* inside the space $U \cap H^1_*(\Omega)$, and this solution is uniquely determined by

$$a(u, v) = l(v) \qquad \text{for all } v \in U \cap H^1_*(\Omega).$$

It is thus a natural approach to compute the unique finite element solution with mean 0. If $\{\psi_1, ..., \psi_k\}$ is a basis of $U \cap H^1_*(\Omega)$, and we set $B = (a(\psi_j, \psi_i))_{1 \le i,j \le k}$ and $w = (l(\psi_i))_{1 \le i \le k})$, by standard theory the coordinate representation z of u_* with respect to the basis $\{\psi_1, ..., \psi_k\}$ is given by the solution of the system $Bz = w$, and B is a symmetric positive definite matrix.

We claim that this is essentially the same as the $\hat{A}x = \hat{b}$ approach, in the following way. We investigate the basis $\{\varphi_1, ..., \varphi_n\}$ of the CG FEM ansatz space U as described above. The central property is again

$$\sum_{j=1}^{n} \varphi_i = 1.$$

Now for every $1 \le i \le n$ there exists some constant $c_i \in \mathbb{R}$ for which we have $\varphi_i - c_i \cdot 1 \in U \cap H^1_*(\Omega)$. Now we can see that $\{\varphi_2 - c_2 \cdot 1, ..., \varphi_n - c_n \cdot 1\}$ is a basis for $U \cap H^1_*(\Omega)$. This follows from the observation that

$$U = \text{span}\{\varphi_1, ..., \varphi_n\} = \text{span}\{1, \varphi_2 - c_2 \cdot 1, ..., \varphi_n - c_n \cdot 1\},$$

which means that $\{1, \varphi_2 - c_2 \cdot 1, ..., \varphi_n - c_n \cdot 1\}$ is a basis of U. Thus the set $\{\varphi_2 - c_2 \cdot 1, ..., \varphi_n - c_n \cdot 1\}$ is a linearly independent set in $U \cap H^1_*(\Omega)$ and hence $\dim(U \cap H^1_*(\Omega)) >= n - 1$. Since $1 \notin U \cap H^1_*(\Omega)$ we thus see that $\dim(U \cap H^1_*(\Omega)) = n - 1$ and $\{\varphi_2 - c_2 \cdot 1, ..., \varphi_n - c_n \cdot 1\}$ is a basis.

With this choice of basis, we see that

$$\begin{aligned} B &= (a(\varphi_{j+1} - c_{j+1} \cdot 1, \varphi_{i+1} - c_{i+1} \cdot 1))_{1 \le i,j \le n-1} \\ &= (a(\varphi_{j+1}, \varphi_{i+1}))_{1 \le i,j \le n-1} = (a_{i+1,j+1})_{1 \le i,j \le n-1}. \end{aligned}$$

and $w_i = l(\varphi_{i+1} - c_{i+1} \cdot 1) = l(\varphi_{i+1}) = \hat{b}_{i+1}$. We thus see that for $x = (x_0, y)$ we have

$$\hat{A}x = \hat{b} \iff \begin{pmatrix} 1 & 0 \\ 0 & B \end{pmatrix} x = \begin{pmatrix} 0 \\ w \end{pmatrix} \iff x_0 = 0 \text{ and } By = w.$$

The author finds it quite remarkable that these two different approaches lead to the same linear system, since in the first derivation we derive a linear equation for the coordinates of the unique finite element solution with coefficient 0 for the first basis function and adjust the linear system so that it becomes positive definite, while

in the second derivation we derive a linear equation for the coordinates of the unique finite element solution with mean 0 in a different coordinate system.

We have now discussed at great lengths how to solve the EEG forward problem. We do this by using a FEM approach, which in the end comes down to solving some linear system $\hat{A}x = \hat{b}$. In this modeling, the neural activity whose measurements we want to simulate is woven into the right hand side $\hat{b}$. But in practice, we often face the problem of solving the forward problem for many different neural activities, and solving every single forward problem by applying the whole FEM approach quickly becomes infeasible. We thus need a smarter way to solve the forward problem for a great number of different neural activities. Such a „smart way“ is given by the *transfer matrix approach*, which we will study in the next subsection.

2.4 The transfer matrix approach

Up to this point we have talked about the *forward problem*, which consists of simulating the measurements some given neural activity would generate. Another central problem in bioelectromagnetism is the so-called *inverse problem*. Here we are given measurements and the goal is to estimate the neural activity causing these measurements. Modern approaches to solving the inverse problem rely heavily on the feasibility of performing a great number of forward simulations in a small amount of time. For example in the RAMUS method one might want to solve the forward problem for over 10,000 different source positions, as it is e.g. illustrated in [17].

In the preceding subsection, we saw that solving the EEG forward problem using a FEM approach leads to a linear system

$$\hat{A}x = \hat{b}.$$

A realistic head model leads to large linear systems, and even though the matrix $\hat{A}$ is sparse and symmetric positive definite and can thus be solved quite efficiently using a preconditioned conjugate gradient method, solving such a linear system for every source position quickly becomes computationally infeasible. We thus need an idea to speed up the solution process for the forward problem. The central insight is the following.

One does not need the full finite element solution $x = \hat{A}^{-1}b$, but only the corresponding sensor measurements.

We first focus on the EEG measurements. In this case, we are interested in the potentials at the electrode positions. Let $\xi_1, ..., \xi_k \in \Omega$ be the electrode positions. If u is the electric potential, we then want to compute the vector

$$V = \begin{pmatrix} u(\xi_1) \\ \vdots \\ u(\xi_k) \end{pmatrix}$$

of measured electric potentials. We can in general not compute the exact potential, and instead use the finite element solution as an approximation. Let $\{\varphi_1, ..., \varphi_n\}$ be a basis of the finite element ansatz space U and let

$$\Phi : \mathbb{R}^n \to U; y \mapsto \sum_{j=1}^{n} y_j \varphi_j$$

be the *Galerkin isomorphism*. If $x = \hat{A}^{-1}b$ are the coordinates of the chosen finite element solution, then $\Phi(x)$ is one finite element approximation for the electric potential.

In an actual measurement one of the electrodes is chosen as the reference electrode. Without loss of generality we can assume the electrode at position ξ_1 to be this reference electrode. Thus the electric potential at the position ξ_1 is fixed to be 0. To simulate the EEG measurements we thus ought to compute the unique finite

element solution whose potential at ξ_1 is zero. This finite element solution is given by $\Phi(x) - \Phi(x)(\xi_1) \cdot 1$, and we can thus compute

$$\begin{aligned} V &= \begin{pmatrix} \Phi(x)(\xi_1) - \Phi(x)(\xi_1) \\ \vdots \\ \Phi(x)(\xi_k) - \Phi(x)(\xi_1) \end{pmatrix} = \sum_{j=1}^{n} x_j \begin{pmatrix} \varphi_j(\xi_1) - \varphi_j(\xi_1) \\ \vdots \\ \varphi_j(\xi_k) - \varphi_j(\xi_1) \end{pmatrix} \\ &= \begin{pmatrix} \varphi_1(\xi_1) - \varphi_1(\xi_1) & \dots & \varphi_n(\xi_1) - \varphi_n(\xi_1) \\ \vdots & & \vdots \\ \varphi_1(\xi_k) - \varphi_1(\xi_1) & \dots & \varphi_n(\xi_k) - \varphi_n(\xi_1) \end{pmatrix} \cdot x \\ &= \begin{pmatrix} 0 & \dots & 0 \\ \varphi_1(\xi_2) - \varphi_1(\xi_1) & \dots & \varphi_n(\xi_2) - \varphi_n(\xi_1) \\ \vdots & & \vdots \\ \varphi_1(\xi_k) - \varphi_1(\xi_1) & \dots & \varphi_n(\xi_k) - \varphi_n(\xi_1) \end{pmatrix} \cdot x. \end{aligned}$$

If we thus set $T = (\varphi_j(\xi_l) - \varphi_j(\xi_1))_{\substack{1 \le l \le k \\ 1 \le j \le n}}$, we see that we can compute the vector V of simulated measurements via $V = T \cdot x$, and V is a linear map in the finite element solution.

We further note that the matrix T can typically be computed quite easily. For example in the case of continuous Galerkin FEM with piecewise polynomial ansatz spaces, we see that the expression $\varphi_j(\xi_i)$ can only be nonzero if φ_j is a degree of freedom corresponding to the mesh element containing ξ_i.

We have now derived how we can compute the simulated electrode measurements starting from the finite element solution. Taking this one step further we get

$$V = Tx = T\hat{A}^{-1}\hat{b}.$$

This leads us to the definition of the *EEG transfer matrix*

$$T^{\mathrm{EEG}} := T\hat{A}^{-1}.$$

Once we have computed the EEG transfer matrix we thus see that solving the EEG forward problem reduces to the following.

1. Assemble the right hand side vector $\hat{b}$.
2. Perform the matrix-vector multiplication $T^{\mathrm{EEG}} \cdot \hat{b}$.

This is in general much faster than solving the linear system $\hat{A}x = \hat{b}$ and then using x to compute the simulated potentials. We call this approach for solving the EEG forward problem the *EEG transfer matrix approach*, or just *transfer matrix approach*.

We see that solving the EEG forward problem can be massively accelerated once the transfer matrix has been computed. This of course raises the following question.

How do we compute the EEG transfer matrix?

Directly computing the transfer matrix using its definition is undesirable, since it involves computing the matrix $\hat{A}^{-1}$. This is a bad idea for a number of reasons, not least of which is that $\hat{A}^{-1}$ is in general a dense matrix, while the matrix $\hat{A}$ is in general sparse[8]. A better approach can be derived by the computation

$$T^{\mathrm{EEG}} \cdot \hat{A} = T \cdot \hat{A}^{-1} \cdot \hat{A} = T \iff \hat{A}^{\intercal} \cdot (T^{\mathrm{EEG}})^{\intercal} = T^{\intercal} \iff \hat{A} \cdot (T^{\mathrm{EEG}})^{\intercal} = T^{\intercal}.$$

In the last equivalence, we have used the fact that $\hat{A}$ is symmetric. Looking at the last equivalence

$$\hat{A} \cdot (T^{\mathrm{EEG}})^{\intercal} = T^{\intercal},$$

we see that the columns of $(T^{\mathrm{EEG}})^{\intercal}$, which are the rows of T^{EEG}, are given as solutions of linear systems of the form $\hat{A}x_i = b_i$. Here the right hand sides b_i are the (transposed) rows of the matrix T, and as discussed above it is relatively easy to assemble this matrix.

This leads to the following scheme for computing the transfer matrix.

[8] This can be seen by looking at the Cayley-Hamilton Theorem, which implies that the inverse B^{-1} of an invertible matrix B is a linear combination of powers of B. It is easily seen that powers of B in general become less sparse the higher the exponent.

1. Assemble the matrices $\hat{A}$ and $T^\intercal = (b_1, ..., b_k)$.
2. Iterate over all electrodes and compute the i-th row $x_i^\intercal$ of the EEG transfer matrix as the solution of the linear system
$$\hat{A}x_i = b_i.$$

We want to remind the reader that the first electrode was chosen as a reference electrode, and correspondingly the first row of T was the zero-vector. Thus $b_1 = 0$ and we directly see that $x_1 = 0$, meaning that the EEG transfer matrix is of the form
$$T^{\mathrm{EEG}} = \begin{pmatrix} 0 & \dots & 0 \\ * & \dots & * \\ \vdots & & \vdots \\ * & \dots & * \end{pmatrix}.$$

We thus see solving the linear system $\hat{A}x_i = b_i$ is only non-trivial for the non-reference electrodes.

In total, we see that the transfer matrix can be computed by setting the first row to zero and then iterating over all non-reference electrodes and computing the corresponding row of the transfer matrix as the solution of some linear system $\hat{A}x_i = b_i$. This is the way the computation of the EEG transfer matrix was implemented in duneuro prior to this thesis.

To compute the EEG transfer matrix for k electrode positions we thus need to solve $k-1$ linear systems with the matrix $\hat{A}$. The alternative approach, consisting of solving the EEG forward problem using a complete finite element approach, needs to solve one linear system per source we want to simulate. We thus see that it is beneficial to use the transfer matrix approach as soon as the number of sources to simulate is at least as great as the number of electrodes.

We have now seen how we can significantly speed up the solution of the EEG forward problem for many source positions using the EEG transfer matrix approach. Something similar is possible for the MEG forward problem. This will be the topic of the following paragraphs.

In Definition 3 we recorded the results of the derivation of the MEG forward problem. Given some position $x \in \mathbb{R}^3$ and some direction $v \in \mathbb{R}^3$, the goal of the MEG forward problem for the dipole $M \cdot \delta_{x_0}$ is to compute the value
$$\langle B(x), v\rangle = \langle B^P(x), v\rangle + \langle B^S(x), v\rangle,$$
where
$$\langle B^P(x), v\rangle = \frac{\mu_0}{4\pi}\langle M \times \frac{x - x_0}{\|x - x_0\|}, v\rangle$$
and
$$\langle B^S(x), v\rangle = -\frac{\mu_0}{4\pi}\int_\Omega \langle \sigma \nabla u \times \frac{x - y}{\|x - y\|^3}, v\rangle \, d\lambda(y).$$

Here the computation of the term $\langle B^P(x), v\rangle$ does not pose any problems. For example, the MEG system at the Institute for Biomagnetism and Biosignalanalysis consists of 275 sensors and, even for more accurate quadrature rules, computing this value for a few thousand position-value pairs is inexpensive.

The situation is more complicated for the second term $\langle B^S(x), v\rangle$. The formula given above relies on the electric potential u inside the head domain. For numerically solving the MEG forward problem we use the finite element approximation of the electric potential. Additionally, we need to integrate over the head domain Ω. We typically do this by iterating over the grid elements in the mesh discretization of the head domain Ω and then computing the corresponding integrals over the elements by choosing some suitable quadrature rule on the reference element and using the transformation formula to pull back the integral.

Performing this process for a great number of source positions is computationally infeasible, if only because it needs the whole finite element solution u, which means solving a linear system $\hat{A}x = \hat{b}$ for every source, as discussed above. When we encountered similar problems in the EEG case the central insight was that we only needed to simulate the potential at the electrodes. Similarly, the process of solving the MEG forward problem can be accelerated using that we are only interested in the magnetic fluxes measured at the MEG sensors. We now focus on a single sensor. After choosing a quadrature rule $\{(s_1, w_1), ..., (s_m, w_m)\}$ for this sensor we are interested in computing the approximation of the magnetic flux
$$\sum_{i=1}^m w_i \langle B(s_i), n(s_i)\rangle = \underbrace{\sum_{i=1}^m w_i \langle B^P(s_i), n(s_i)\rangle}_{=:\Psi^P} + \underbrace{\sum_{j=1}^m w_i \langle B^S(s_i), n(s_i)\rangle}_{=:\Psi^S}.$$

We call Ψ^P the *primary magnetic flux* and Ψ^S the *secondary magnetic flux*.

As discussed, the computation of the primary magnetic flux is unproblematic. We now take a closer look at the secondary magnetic flux. Let $\{\varphi_1, ..., \varphi_n\}$ be the chosen basis of the finite element ansatz space and let $x \in \mathbb{R}^n$ be the coordinates of the finte element solution given by $\hat{A}x = \hat{b}$. We now set $v_i := n(s_i)$ and compute

$$\begin{aligned}
\Psi^S &= \sum_{i=1}^{m} w_i \langle B^S(s_i), v_i \rangle = \sum_{i=1}^{m} -w_i \frac{\mu_0}{4\pi} \int_\Omega \langle \sigma \nabla u \times \frac{s_i - y}{\|s_i - y\|^3}, v_i \rangle \, d\lambda(y) \\
&= \sum_{i=1}^{m} -w_i \frac{\mu_0}{4\pi} \int_\Omega \langle \sigma \nabla \left(\sum_{j=1}^{n} x_j \varphi_j \right) \times \frac{s_i - y}{\|s_i - y\|^3}, v_i \rangle \, d\lambda(y) \\
&= -\frac{\mu_0}{4\pi} \sum_{j=1}^{n} \left(\sum_{i=1}^{m} w_i \int_\Omega \langle \sigma \nabla \varphi_j \times \frac{s_i - y}{\|s_i - y\|^3}, v_i \rangle \, d\lambda(y) \right) x_j \\
&= -\frac{\mu_0}{4\pi} \sum_{j=1}^{n} \left(\int_\Omega \sum_{i=1}^{m} w_i \langle \sigma \nabla \varphi_j \times \frac{s_i - y}{\|s_i - y\|^3}, v_i \rangle \, d\lambda(y) \right) x_j.
\end{aligned}$$

If we thus define for $1 \leq j \leq n$

$$b_j := \int_\Omega \sum_{i=1}^{m} w_i \langle \sigma \nabla \varphi_j \times \frac{s_i - y}{\|s_i - y\|^3}, v_i \rangle \, d\lambda(y)$$

and we set $b = (b_j)_{1 \leq j \leq n}$, we see that we have

$$\Psi^S = -\frac{\mu_0}{4\pi} \langle b, x \rangle.$$

We thus see that the simulated magnetic flux is a linear functional in the coordinates of the finite element solution.

Now assume there are p MEG sensors. For $1 \leq i \leq p$ let Ψ_i^S be the simulated secondary magnetic flux for the sensor i, and let $b(i)$ be the vector defined above so that

$$\Psi_i^S = -\frac{\mu_0}{4\pi} \langle b(i), x \rangle.$$

If we now set $R = (b(1), ..., b(p))$ we see that we can compute the vector of simulated magnetic fluxes via

$$\begin{pmatrix} \Psi_1^S \\ \vdots \\ \Psi_p^S \end{pmatrix} = R^\intercal \cdot x = R^\intercal \cdot \hat{A}^{-1} \cdot \hat{b}.$$

If we now define the *MEG transfer matrix* as $T^{\mathrm{MEG}} := R^\intercal \cdot \hat{A}^{-1}$, we see that once we are in possession of this matrix computing the secondary magnetic fluxes for all sensor positions reduces to assembling the *modified EEG FEM right hand side* $\hat{b}$ and performing a matrix-vector multiplication with T^{MEG}.

Similar to the EEG transfer matrix we can derive a linear condition defining T^{MEG} via the calculation

$$T^{\mathrm{MEG}} \cdot \hat{A} = R^\intercal \iff \hat{A} \cdot (T^{\mathrm{MEG}})^\intercal = R,$$

where we have again used the symmetry of A. We see that, in complete analogy with the EEG case, the rows of the MEG transfer matrix are given as solutions of linear systems $\hat{A}x = b(i)$. One approach to compute the MEG transfer matrix is thus given by iterating over all MEG sensors, assembling the corresponding right hand side b, and solving the system $\hat{A}x = b$ to compute the corresponding row of the transfer matrix. This is essentially the approach used in duneuro. There is only a slight caveat. As of the moment of writing these paragraphs duneuro only supports quadrature rules $\{(s_1, w_1), ..., (s_m, w_m)\}$ with $m = 1$ and $w_1 = 1$, or more concretely it is currently only possible to directly compute values

$$\langle B^S(x_i), v_i \rangle,$$

and each such value corresponds to a row in the transfer matrix. There is a discussion about extending the methods inside duneuro to be able to work on general quadrature rules, and when reading this text in the future this feature might already be implemented.

We now summarize all relevant definitions about the EEG and MEG transfer matrices in the following definition.

Definition 16 (EEG and MEG transfer matrices)**.** Let $\{\varphi_1, ..., \varphi_n\}$ be a basis of the finite element ansatz space. Let $\hat{A}$ be the modified stiffness matrix.

Assume there are k electrodes at positions $\xi_1, ..., \xi_k$, where the electrode at position ξ_1 is chosen as reference eletrode. Now define the matrix $S \in \mathbb{R}^{n\times k}$ to be

$$S = (\varphi_i(\xi_j) - \varphi_i(\xi_1))_{\substack{1\le i\le n\\ 1\le j\le k}}.$$

Then the EEG transfer matrix $T^{\mathrm{EEG}} \in \mathbb{R}^{k\times n}$ is given by

$$\hat{A} \cdot \left(T^{\mathrm{EEG}}\right)^{\intercal} = S.$$

Now assume there are p MEG sensors. For $1 \le j \le p$ let $\{(s_1^j, w_1^j), ..., (s_{m_j}^j, w_{m_j}^j)\}$ be our quadrature rule of choice for the surface enclosed by the j-th MEG coil, and let $v_l^j = n(s_l^j)$ be the corresponding normal vectors to this surface. Now define the matrix $R \in \mathbb{R}^{n\times p}$ to be

$$R = \left(\int_\Omega \sum_{l=1}^{m_j} w_l^j \langle \sigma \nabla \varphi_i \times \frac{s_l^j - y}{\|s_l^j - y\|^3}, v_l^j \rangle \, d\lambda(y) \right)_{\substack{1\le i\le n\\ 1\le j\le p}}.$$

Then the MEG transfer matrix $T^{\mathrm{MEG}} \in \mathbb{R}^{p\times n}$ is given by

$$\hat{A} \left(T^{\mathrm{MEG}}\right)^{\intercal} = R.$$

In many cases we want to solve the EEG forward problem *and* the MEG forward problem for a great number of sources. For this we define the *MEEG transfer matrix* $T^{\mathrm{MEEG}} \in \mathbb{R}^{(k+p)\times n}$ to be

$$T^{\mathrm{MEEG}} = \begin{pmatrix} T^{\mathrm{EEG}} \\ T^{\mathrm{MEG}} \end{pmatrix}.$$

Once the MEEG transfer matrix has been computed, simulating electrode measurements and magnetic fluxes reduces to assembling an EEG FEM right hand side and performing a matrix-vector multiplication. With the definitions above the MEEG transfer matrix is given by

$$\hat{A} \left(T^{\mathrm{MEEG}}\right)^{\intercal} = (S, R).$$

Remark.

1. As mentioned above, at the moment of writing this paragraph duneuro only supports quadrature rules of the form $\{(s_1, 1)\}$. Hence the matrix R in duneuro has the somewhat simpler form

$$R = \left(\int_\Omega \langle \sigma \nabla \varphi_i \times \frac{x_j - y}{\|x_j - y\|^3}, v_j \rangle \, d\lambda(y) \right)_{\substack{1\le i\le n\\ 1\le j\le p}}$$

   for a set $\{(x_1, v_1), ..., (x_p, v_p)\}$ of position-direction pairs. Since all our numerical experiments were run using duneuro, they always used this special form of the matrix R.

2. While the definition of the matrix R may look daunting at first, it can be computed quite inexpensively in many cases. Since in a FEM approach our domain Ω is triangulated into simple domains T_i, we have $\int_\Omega = \sum_i \int_{T_i}$, and typically in FEM approaches only few φ_j are non-vanishing on any given T_i. We can thus assemble the matrix R in one iteration over the grid elements, where for every grid element we can cheaply assemble the integrals over this element for all non-vanishing basis functions for all sensors. As discussed earlier, the assembling of the matrix S can also be performed cheaply.

To give the reader a rough idea of the time consumption of assembling the matrix right hand side (S, R) compared to the remaining part of the transfer matrix computation, we list the time taken by some tasks in the computation of the MEEG transfer matrix in an example run in the following table.

task	total	stiffness matrix	assembling RHS	solving linear system
time (sec)	3848	20,67	724,5	3008

Table 1: Example for the time consumption of different parts of the transfer matrix computation

Here we computed the MEEG transfer matrix using a block conjugate gradient method[9] for the realistic head model, electrode positions and MEG coil positions given in [15]. The computation was performed on a single core of an Intel Skylake-SP Xeon Gold 6148 processor. We see that in this example assembling the matrix right hand side (S, R) took about 19% of the total computation time.

[9] We will introduce this method for solving the matrix equation defining the transfer matrix in a later chapter

We have now introduced the transfer matrices and have described how they can massively speed up the process of solving the EEG and MEG forward problems. We have seen that the transfer matrices are given by a matrix equation

$$\hat{A}X = B,$$

where $\hat{A}$ is a large, sparse, symmetric positive definite matrix. The common approach to solve this matrix equation is to read it column-wise as a set of linear systems $\hat{A}x_i = b_i$, and solve the systems iteratively using a preconditioned conjugate gradient method. This is also the approach currently implemented in duneuro.

As illustrated above, computing the transfer matrix is a key step in applications such as solving the inverse problem. These applications are of great practical interest, for example in presurgical epilepsy diagnosis (e.g. [18]). Hence there is also a need for these applications to run as efficiently as possible. When measuring the time requirements one sees that the computation of the transfer matrices is oftentimes a bottleneck, and it is hence imperative to try to reduce this computation time. This is what we are going to investigate for the rest of this thesis.

We thus search for some approach to speed up the computation of the transfer matrices. The central observation, that will lead us to such an approach, is the following.

The current implementation of the transfer matrix computation does not properly utilize *vectorization*.

This immediately raises three questions.

1. What is vectorization?
2. Why can't the current implementation exploit vectorization?
3. How can we design a new approach that can utilize vectorization?

We are going to investigate these questions in the following chapters.

3 Vectorization and its Limitations

We are now going to tackle the first question, namely we want to give an introduction to what is meant by the term "vectorization".

3.1 Introduction to Vectorization

Modern processors are highly complex devices, capable of numerous techniques to optimize computation time. These include superscalar execution, out-of-order execution and speculative execution of instructions (look e.g. at [19]).

One such technique is the possibility to apply the same instruction to multiple elements of data at the same time. In his seminal paper [20] from 1966 Flynn established the name

***S**ingle **I**nstruction Stream - **M**ultiple **D**ata Stream*,

or **SIMD**, for processors possessing such capabilities. Over the years there have been a number of different realizations of this concept, but on modern (and many older) processors one typically realizes SIMD as follows. These SIMD-compatible processors possess so called *vector registers*. These come in different sizes, but typically these registers can store either 128 bit, 256 bit or 512 bit of data. The processor then supports certain instructions that work on these vector registers. We call these instructions *vector instructions*, and the usage of vector instructions instead of scalar instructions is called *vectorization*. Here *scalar instructions* means all instructions that are not vector instructions.

For example on a system where a double takes up 64 bits of space and the system possesses a 128 bit vector register, one could load two doubles into such a register and then operate on both of them simultaneously. Ideally, this can lead to a speedup of a factor of 2.

Somewhat frustratingly, there are even more sets for vector instructions than there are processor manufacturers. Since the experiments in this thesis were run on an Intel Core Prozessor i5-6200U and an Intel Skylake-SP Xeon Gold 6148 processor, we will focus on Intel processors in the following, but everything we talk about applies to other processors in a similar way.

Intel has gone through various stages of vector instructions, as can be seen by looking at Intel's architecture manual [21]. A simplified description of these instruction sets might be given as follows.

1. The **SSE** (**S**treaming **S**IMD **E**xtensions) instruction sets add instructions to work on 128 bit vector registers.
2. The **AVX1** and **AVX2** (**A**dvanced **V**ector **E**xtensions) instruction sets add support for 256 bit vector registers.
3. The **AVX-512** instruction set adds support for 512 bit vector registers.

Not all processors support all of these instruction sets. Especially the AVX-512 instruction set is only available on selected processor, in particular those geared towards **H**igh **P**erformance **C**omputing (HPC). For example, the author's laptop containing an i5-6200U processor only supports SSE and AVX instructions, but not AVX-512 instructions, while the Intel Skylake-SP Xeon Gold 6148 processor used for the numerical experiments supports SSE, AVX and AVX-512 instructions.

In the following we assume a double to have a size of 64 bit. SIMD instruction sets typically also allow to work on multiple integers, longs or floats at the same time. But since we are interested in applying SIMD instructions to the process of solving the forward problem, we are focussing on doubles. We thus ideally hope to get the following performance increase when applying SIMD instructions.

1. SSE $\rightsquigarrow$ 2 doubles at once, 2x speed up
2. AVX $\rightsquigarrow$ 4 doubles at once, 4x speed up
3. AVX-512 $\rightsquigarrow$ 8 doubles at once, 8x speed up

Sadly, this is not what happens. In practice, SIMD instructions are very power intensive, as in general working on larger registers requires more power than working on smaller registers. To handle this increased power consumption the processor clock speed is throttled for some vector instructions. According to [22] for example, when executing many 512 bit AVX-512 instructions on an Intel Xeon Silver 4116 CPU, the clock speed is throttled to 1.1 GHz, while the standard clock speed for non-vector instructions is 2.1 GHz. In this setting, the best possible speedup one could achieve using 512 bit vectorization is about a factor of 4x, even though we can work on 8 doubles at the same time. Again, it is not possible to give a throttling factor that applies to every processor, and even giving a factor for a single processor is a little misleading, as the amount of throttling does not only depend on the instruction sets used, but also on the number of active CPU cores and what kind of instructions they execute.

To give the reader a rough idea about what kind of speedup we can expect in the best case in our own numerical tests, we benchmarked the effect of vectorization when using an Intel Skylake-SP Xeon Gold 6148 processor, since this is the main processor we are going to run our experiments on. The experiment was set up in the following way. We set up three double arrays of size 2^8. We fill the first two with pseudorandom entries. We then measure the time it takes to perform the task

```
for(int i = 0; i < 2^20; ++i) {
        sum_vector = vector_1 + vector_2
}
```

Inside the loop, we compute the sum in four different ways.

1. Without vectorization (using instructions on single doubles)
2. With 128 bit vectorization (using SSE instructions)
3. With 256 bit vectorization (using AVX instructions)
4. With 512 bit vectorization (using AVX-512 instructions)

The code was written in C++ and compiled using the Intel C++ compiler. We used the compiler intrinsics given in [23] to fine-tune the vectorization.

Performing this experiment on a single core of an Intel Skylake-SP Xeon Gold 6148 processor leads to the results in figure 3.

We see that for 128 bit vectorization we achieve the ideal speedup of a factor of 2. For 256 bit vectorization we achieve a speedup of a factor of 3.7, which is slightly lower than the ideal factor of 4. For 512 bit vectorization we achieve a speedup of a factor of 6.4, which is about 80% of the ideal speedup factor of 8.

In practice however, the corresponding speedup factors might be lower than the ones given above. This can be for a variety of reasons.

1. Not the whole program might benefit from vectorization. In general, we are bound by *Amdahl's law*. Assume that p% of a program's execution time can be vectorized. Assume that our vector registers can

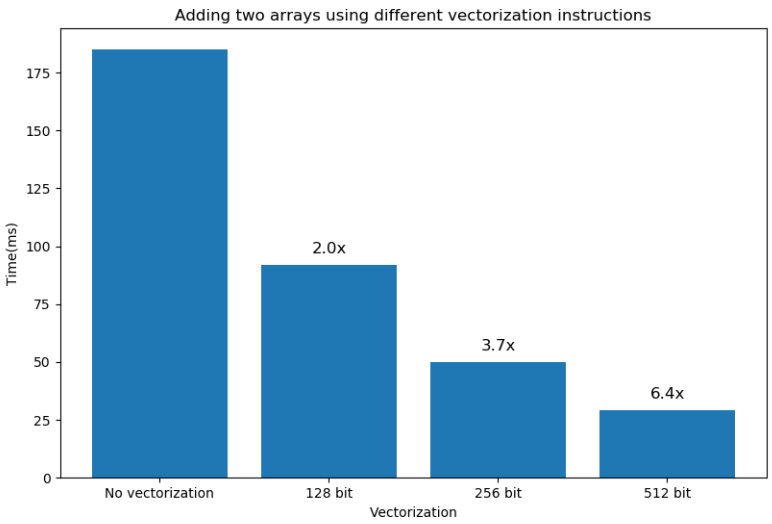

Figure 3: Performance benchmark for adding two arrays using different forms of vectorization

work on N data elements at the same time. Let t be the time the program execution takes when no vectorization is employed. Then in the ideal case using vectorization leads to an execution time

$$\frac{pt}{N} + (1-p)t.$$

Thus the best possible speedup factor is

$$\frac{t}{\frac{pt}{N} + (1-p)t} = \frac{1}{\frac{p}{N} + 1 - p} \in [1, N].$$

For example, if 50% of our scalar program can benefit from vectorization, and our vector registers can work on 2 values at the same time, the best possible speedup factor is $\frac{4}{3}$.

2. As mentioned above, when executing AVX-512 instructions the clock speed of the processor can get throttled. But once the execution of the 512 bit vector instructions has ended, the clock speed is not directly increased back to its former value, but only gets increased after a certain amount of time. Under unfavorable conditions, for example if scalar instructions and vector instructions alternate frequently, this effect might even lead to larger execution times when employing vectorization as compared to not employing vectorization.

3. In practice one rarely directly writes assembler code or uses compiler intrinsics to embed vectorization instructions into source code. Instead one uses some compiler and basically hopes that the compiler automatically vectorizes the code. The author firmly believes that this is the right approach.[10] But there are many factors that might prevent the compiler from producing optimal machine code. The data might for example not be properly aligned in memory, or the compiler cannot rule out pointer aliasing. In these cases, the compiler requires help from the programmer to overcome these obstacles, and giving the compiler the help it requires can be quite challenging. But even then, it might be possible that the auto-vectorization of the compiler is not mature enough to perform proper vectorization. But since duneuro is written in C++, we have access to compilers like gcc or the Intel C++ compiler, and can thus rely on pretty advanced auto-vectorization features.

4. When the number of the data elements we want to work on is not divisible by the number of data elements that fit into the vector register we might have to handle the tail end of the data separately, which can add a typically small overhead.

There is still another factor that might inhibit vectorization from having any noticeable effect whatsoever on the time needed to execute a program. This is the topic of the next subsection.

[10] https://stackoverflow.com/questions/2684364/why-arent-programs-written-in-assembly-more-often/2685541#2685541

3.2 Vectorization in a Memory Bound Context

Even though there are some pitfalls when trying to speed up computations using vectorization, looking at figure 3 shows us that properly employed vectorization can significantly accelerate computations. Looking at the claim made at the end of subsection 2.4, this raises one question.

Why does the computation of the transfer matrices in its current implementation not benefit from vectorization?

To see why this is the case, we begin by looking at an example. Similar to the example underlying figure 3, we initialize three double arrays, fill two of them with random entries and then measure the time it takes to compute their sum into the third array. We do this once without using vectorization, and once with 128 bit vectorization. In contrast to the previous experiment, we now measure the time for various array sizes, where the size is given in the number of doubles the array contains. We use sizes of the form 2^n, for $n \in \{12, 13, ..., 19\}$. The experiment was run on a single core of an Intel Skylake-SP Xeon Gold 6148 processor. This leads to the results illustrated in figure 4.

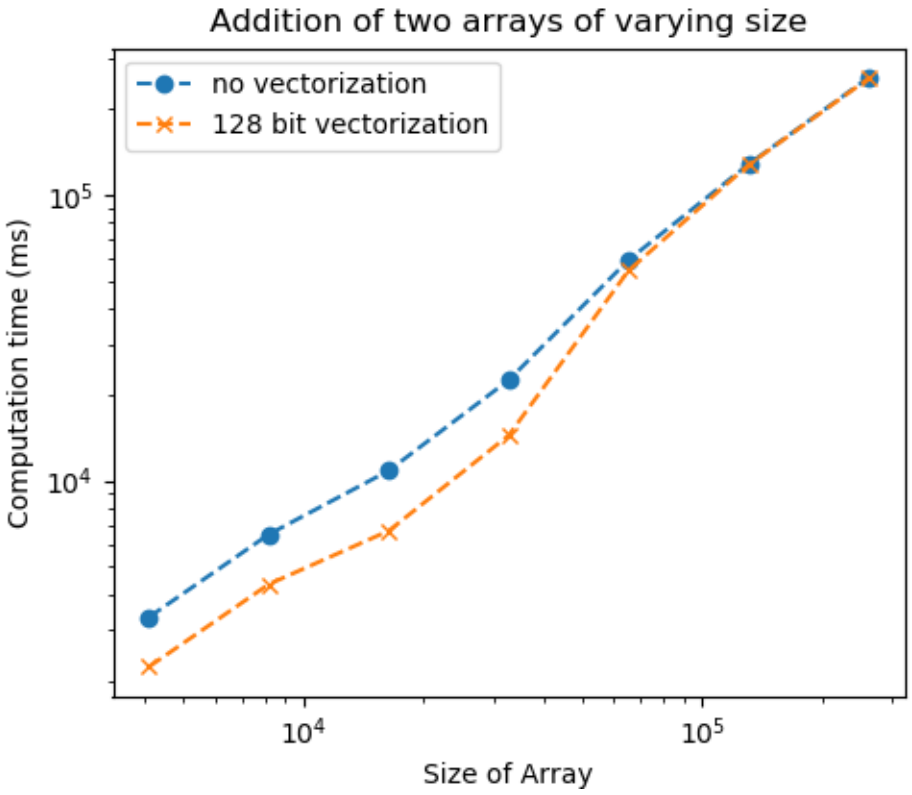

Figure 4: Time to compute the sum of two arrays of varying size

For small array sizes we get the expected results. Since the y-axis is scaled logarithmically, we see that there is a constant factor of almost 2 between the computation times with and without vectorization, which corresponds to the fact that 128 bit vectorization enables us to work on two doubles at the same time. But, perhaps surprisingly, once our arrays pass a certain size we see virtually no difference in computation times with or without vectorization. To explain this behavior, and why it also applies to the process of solving linear systems using a preconditioned conjugate gradient method, we want to talk about what determines the runtime of simple loops. „Simple“ in this context means loops consisting of loading values, storing values, and performing computations on values. There are various models that were developed to analytically derive the runtime of such loops. We orient ourselves on the relatively recent model described in [24]. For this, we first want to take a look at the following statement. Imagine an assembly line while reading the following theorem.

Theorem 17. Assume that some task consists of n subtasks $p_1, ..., p_n$. Assume that there are n workers $w_1, ..., w_n$. Assume that the worker w_i is specialized on task p_i and always takes the time t_i to perform this task. Assume that task p_i has to be completed before work on task p_{i+1} can begin, and that the workers $w_1, ..., w_n$ can work in parallel. Assume that task p_{i_0} takes the most time to perform.

Then the times it takes to perform the task k times is

$$k \cdot t_{i_0} + \sum_{i \neq i_0} t_i.$$

Proof. We prove the theorem via induction on the number of subtasks n. For $n = 1$ the statement is clear. Now assume the statement is true for n, and we want to show that it is also true for $n+1$. We show this via induction on the number of times k we want to perform the task. For $k = 1$ the total time is obviously given by

$$t_1 + ... + t_{n+1},$$

and hence in this case the statement of the theorem is valid. Now assume the statement is true for k, and we want to show it is also valid for $k + 1$. Now there are two different cases.

Case 1: We have $t_{n+1} \geq t_i$ for $1 \leq i \leq n$.

By our induction hypotheses, the total time required to solve the first k instances of the task is then

$$k \cdot t_{n+1} + \sum_{i=1}^{n} t_i,$$

and by our induction hypotheses on the number of subtasks, the time needed to perform the first n subtasks $k + 1$ times is

$$(k+1) \cdot t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} t_i,$$

where $t_{i_0} = \max\{t_1, ..., t_n\}$. But now we have

$$(k+1) \cdot t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} t_i = k \cdot t_{i_0} + \sum_{i=1}^{n} t_i \leq k \cdot t_{n+1} + \sum_{i=1}^{n} t_i.$$

But this means that once the first k instances of the task have been finished, the first n subtasks of the last instance of the task have already been finished, and the last worker can begin working on the last subtask of the last instance of the task immediately. Thus the total time requirement is

$$t_{n+1} + k \cdot t_{n+1} + \sum_{i=1}^{n} t_i = (k+1) \cdot t_{n+1} + \sum_{i=1}^{n} t_i.$$

Case 2: There is some $1 \leq i_0 \leq n$ with $t_{i_0} = \max\{t_1, ..., t_{n+1}\}$. By our induction hypotheses, the time required to solve the first k instances of the task is then

$$k \cdot t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n+1} t_i,$$

and by our induction hypotheses on the number of subtasks, the time needed to perform the first n subtasks $k + 1$ times is

$$(k+1) \cdot t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} t_i.$$

We then have

$$k \cdot t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n+1} t_i \leq (k+1) t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} t_i.$$

Hence, after the last worker has finished the last subtask of the k-th instance of the task, this worker has to wait for the first n subtasks of the last instance of the task to be finished before he can begin working on the final subtask of the final instance of the task. We thus get a total time of

$$(k+1) t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} t_i + t_{n+1} = (k+1) t_{i_0} + \sum_{\substack{i=1 \\ i \neq i_0}}^{n+1} t_i.$$

This proves the theorem. □

We thus see that if we have an assembly line situation with n subtasks taking $t_1, ..., t_n$ amount of time to be performed respectively, and if the number of times k we need to perform this task is large, the total time needed to perform all tasks is approximately given by

$$t_{\text{total}} \approx k \cdot \max\{t_1, ..., t_n\},$$

meaning that the total runtime is approximately the number of tasks to perform multiplied by the duration of the longest subtask. Thus the total time needed to perform all tasks is essentially determined by the subtask that takes the longest to complete.

This model can be used to estimate the time it takes to execute simple loops. To perform one iteration of such a loop, the necessary data has to be loaded from the highest level in the memory hierarchy they reside in, then the corresponding computations have to be performed on the loaded data, and finally the results of these computations have to be written back to memory. Modern processors are capable of performing these tasks in parallel. This means that while the results of the first iteration are written to memory, the computations of the second iteration can be performed, and the data for the third iteration can be loaded from memory at the same time.[11] For details we refer to [24], or for a simplified version to [25].

We thus see that the computation time is dominated by the subtask that takes the longest to complete. Since the subtasks are either data transfers or computations inside the CPU core, we see that the computation time is either dominated by the data transfers between the different levels of the memory hierarchy or by the actual execution of computations inside the processor.

Definition 18. If the memory transfers dominate the runtime of a loop, we call the loop *memory bound*. If the actual computations dominate the runtime, we call the loop *CPU bound*.

We also use the terms memory bound and CPU bound to more generally describe programs where either the data transfers or the actual computations dominate the total runtime. We can now make the following observation.

Remark. Vectorization can only speed up the actual computations inside the processor. The time needed to load values from memory is unaffected by vectorization. We thus see that **memory bound programs cannot benefit from vectorization**, since the loading of the values completely dominates the runtime.

Over the past decades, the clock speed of processors has increased significantly faster than the rate at which data can be transferred to or from main memory, typically called the *memory bandwidth*. Today we are in the situation that performing an arithmetic operation inside the processor is typically much faster than loading the value from main memory. To combat this, a more complex memory hierarchy has been developed. Modern processors for example typically possess so-called *caches*. Caches are small units of memory that are located close to the processor core and can be accessed faster than main memory. On modern architectures there are typically three levels of caches, called L1, L2 and L3 cache. Loading data from the L1 cache is faster than loading data from the L2 cache, loading from L2 is faster than loading from L3 and loading from L3 is faster than loading from main memory. For a concise review of the development of CPU clock speeds, memory access speeds and the development of the memory hierarchy in its current form we refer to [26].

To give the reader a rough idea of what is possible on modern processors, we look at a single core of an Intel Skylake-SP Xeon Gold 6148 processor. According to [25], this processor is able to perform $76.8 \cdot 10^9$ floating point operations per second under optimal conditions, and can load $13.345 \cdot 10^9$ bytes per second from main memory to the L1 cache. Considering that a double precision value typically consists of 8 byte, the time it takes to load two double values from memory is thus significantly greater than the time needed to compute their sum in this example. This also explains the behavior we see in figure 4. When the arrays are small, they fit inside a level of the memory hierarchy that is located close to the processor and can be accessed fast enough, so that the longest subtask in the program is the actual computation of the sum. Thus the loop is CPU bound and can benefit from vectorization. Once the arrays pass a certain size, they need to be stored in another level of the memory hierarchy with larger capacity and slower access times. Once this happens, the loading of the values dominates the runtime and the loop becomes memory bound. In this case, vectorization can no longer speed up the computation, as the runtime is given by the time it takes to perform memory transfers.

In the concrete example in figure 4, the loop becomes memory bound when the array size changes from 2^{15} to 2^{16}. The computation needs three double precision arrays, and thus for the size 2^n takes up

$$2^n \cdot 3 \cdot 8 = 3 \cdot 2^{n+3}$$

[11] It is highly dependent on the hardware what operations can be performed in parallel. Some processors can overlap data transfers between different levels of the memory hierarchy, others can't. Some processors have two uni-directional links between different levels of the memory hierarchy and can thus perform loading and storing operations simultaneously, other processors have only a single bi-directional link and can't overlap loading and storing operations.

bytes of memory. For $n = 15$ the computation thus takes up approximately 0.79 megabyte, and for $n = 16$ the computation takes up approximately 1.57 megabyte. The computation was performed on an Intel Skylake-SP Xeon Gold 6148 processor. On this machine the L2 cache has a size of 1024 kilobyte, and we see that the behavior in figure 4 can be explained by the fact that the arrays no longer fit into the L2 cache.

A similar consideration shows why our naive implementation of the transfer matrix computation cannot utilize vectorization. We compute the transfer matrices by iteratively solving linear systems using the preconditioned conjugate gradient method. This method is based on vector-vector operations, like the inner product, and matrix-vector operations, like matrix-vector multiplication. If we now want to compute an inner product, each entry of one of the vectors is only needed for a single fused multiply add operation. Thus similar to the example discussed above, for large vectors the loading of the values dominates the runtime and the computation of the dot product cannot benefit from vectorization. Looking at other vector-vector operations and matrix-vector operations, we see that the ratio of computations performed inside the CPU to bytes loaded is comparably small. We thus see that for large matrices and vectors, the runtime of vector-vector and matrix-vector operations is dominated by the loading of the values operated on. We arrive at the following conclusion.

Remark. Vector-vector operations and matrix-vector operations on high dimensional vectors are in general **memory bound** and can thus not benefit from vectorization.

We want to briefly discuss what „high dimensional“ in the previous remark means. We take a look at a finite element approach for the relatively small spherical mesh from [3]. We use piecewise linear ansatz functions. The corresponding linear system contains about $55,000$ degrees of freedom, and the stiffness matrix contains about $782,000$ nonzero entries. If we use a sparse format like the compressed row storage (CRS) format as described in [27] and assume that indices are stored as integers, and integers use 32 bytes of memory, the stiffness matrix alone takes up about 12.5 megabyte of storage. This is larger than the L3 cache on most personal computers, and larger than the L2 cache on almost all processors, even those geared towards high performance computing. Thus the data required for the matrix-vector operations needed for the preconditioned conjugate gradient method used in the computation of the rows of the transfer matrices can in general not be stored inside the L2 cache, even for relatively small head models. Hence we see that, even for simple spherical head models, the preconditioned conjugate gradient method is in general memory bound and can thus not fully benefit from vectorization. This is even worse for realistic head models, as in this case even the largest common L3 caches can no longer store the stiffness matrix, even if stored in the CRS format.

We have thus illustrated why the computation of the transfer matrices cannot benefit from vectorization.

3.3 Designing Algorithms to Utilize Vectorization

We have seen in the previous section that once the matrices and vectors we work on get too large, they have to be placed in levels of the memory hierarchy that can only be accessed slowly, and thus the time needed for loading the values dominates the runtime of algorithms performing vector-vector and matrix-vector operations. As vectorization only speeds up the computations inside the processor, these methods cannot utilize vectorization. More generally, if we want to optimize a memory bound algorithm, we don't even need to think about making the computations themselves more efficient, as the limiting factor is the loading of the data itself. To make such an algorithm more efficient, we thus need to reduce the amount of data that is loaded from slow levels of the memory hierarchy to faster levels of the memory hierarchy. If we are able to reduce this amount enough, we might even be able to get to the point where the computations inside the processor determine the runtime, and we can then utilize vectorization or other strategies to optimize the computation itself.

As discussed above, in our applications we can always assume that a significant portion of the data we need to operate on resides in the main memory or in the L3 cache. We will thus discuss how algorithms need to be designed if they want to use the memory efficiently, and then try to apply the results to improve our currently memory bound approach of computing the transfer matrices.

We first briefly talk about caches. A more detailed discussion about the inner workings of cache memory can be found in [28]. Central to the idea of a cache is the so-called *locality principle*, as described in [29]. This principle states that computation oftentimes happens in a local manner. The common idea is that computation on the one hand is *temporally local*, which means that data that is used in one computation is likely to be used in another computation in the near future, meaning the temporal distance between two uses of the same data is oftentimes small. On the other hand the idea is that computation is also *spatially local*, meaning that if we operate on some data at a certain position in memory, we are also likely to operate on data that is stored closely to this data in memory.

These principles are reflected in the way caches operate. As discussed above, a cache is typically a small unit of memory that can be accessed fast, and there are typically different levels of cache, that get progressively bigger

and slower. If the processor wants to load data into its registers, it looks for it first in the L1 cache, then if it is not found there it looks in the L2 cache, then in the L3 cache and finally in main memory. If the data is not found on a certain level of cache we speak of a *cache miss* on that level. If on the other hand the data is present we speak of a *cache hit*. In general, we want to minimize cache misses, as each cache miss necessitates loading the desired data from a slower level of the memory hierarchy. In the case of a cache miss, the loaded data is then stored in the lower level caches. If this data is then needed in the near future, it is probable that it is still contained inside the cache and can then be loaded fast. Thus the cache performs well if the underlying program is temporarily local. Furthermore, when a cache miss occurs, in general not only the desired data is loaded, but also the data with a close memory address to the desired data. For example, if the desired data is stored at address x is not contained in the cache, not only the value at address x will be loaded into the cache, but also the values at addresses $x-m, x-(m-1), ..., x-1, x+1, x+2, ..., x+n$. The concrete amount of data loaded can vary between the different levels of cache, but this feature enables the processor to exploit spatial locality in a program, as a spatially local program oftentimes needs to work on data close to the data it works on at the moment. With this prefetching strategy, this spatially close data is probably already contained inside some level of cache and can be accessed quickly. Such a block of data, that gets loaded into the cache because the value at address x got accessed, is called a *cache line*.

We thus arrive at the following two design principles for writing code that effectively uses the memory hierarchy, and thus minimizes expensive loading operations.

1. If we work on data at position x and also need to perform work on data at position $x+i$ for i small, we should perform the work on the data at $x+i$ as soon as possible, as it is probably contained in the cache line loaded together with the data at x and can thus be accessed quickly. If we wait too long the data at $x+i$ might already have been evicted from the cache. Alternatively, we could store our data in such a way that data that needs to be worked on at a similar time is stored at similar memory addresses. To summarize, this means that we should design algorithms to be **spatially local**.
2. If we need to perform multiple tasks on some data element x, we should do these tasks with minimal time in between, so that we can perform as much of these tasks as possible while the data is still contained inside the cache and can be accessed quickly. More succinctly, this means that we should design algorithms to be **temporally local**.

To boil this down into a single punchline we could say the following.

Design the algorithm so that it uses all data in a cache line as often as possible!

Now let us take a look at the transfer matrix problem. This is given by a matrix equation

$$A \cdot X = B.$$

The previous implementation in duneuro was to solve this equation by reading it as a series of linear systems $Ax_i = b_i$, and solving these systems separately using the PCG method. But this artificially destroys the inherent temporal locality of the matrix system $A \cdot X = B$. For example, in every iteration of the PCG method we need to compute a matrix-vector product $A \cdot p_i$. When performing a matrix-vector product, we need every entry of the matrix A only for a single computation, and once we need to perform the next matrix-vector product the relevant entries from the matrix A have probably already been evicted from the cache. This approach thus has a terrible temporal locality, and since we have to reload the entries of the matrix for every matrix-vector multiplication we get a memory bound algorithm, as discussed in the last subsection.

Now imagine that we perform the PCG algorithms in parallel. Then, instead of computing the matrix-vector product $A \cdot p_i$ separately for every instance of the algorithm, we could compute a single matrix-matrix product $A \cdot (p_1, ..., p_k)$. In this product, the entries of A are needed for more than a single computation and thus have the potential to be reused before they get evicted from the cache. This can lead to a higher temporal locality of the algorithm and can thus speed up the algorithm.

This naturally leads to the following approach.

Idea. If we can derive an algorithm for solving $A \cdot X = B$ that, instead of splitting this equation into a set of linear systems $A \cdot x_i = b_i$ to be solved separately, directly solves the system by working on **blocks** of vectors $(p_1, ..., p_k)$, this might lead to significant speedup.

This idea then naturally leads to the **block conjugate gradient** method we will derive in the next section.

But before we go about deriving this generalization of the conjugate gradient method, we will first discuss the potential of the above idea. We have illustrated why switching from operating on vectors to operating on matrices can improve the locality of the program and thus decrease the time spent loading values from memory.

But to utilize vectorization this time has to be reduced to such an extent that the actual computations inside the CPU dominate the computation time, meaning the computation has to become **CPU bound**. At first sight, it is not obvious if this is the case. We want to spend the rest of this section convincing the reader that this is indeed what happens.

Remark. On contemporary hardware properly implemented matrix multiplication is CPU-bound and can thus benefit from vectorization. With matrix multiplication, we mean the multiplication of two dense matrices. To a lesser extent, the same is also true for sparse-matrix-dense-matrix multiplication.

But why is this the case? The authors in [30] ascribe this to a *surface-to-volume* effect, since if we want to multiply two $n \times n$ matrices, the naive[12] algorithm for matrix multiplication performs $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ amount of data. Because of this, we expect the time needed to perform the actual computations to dominate the time needed for loading the values, and hence we expect a CPU-bound algorithm.

While the above statement is useful for quickly stifling a discussion, it does not do justice to the complexity of the problem. We want to briefly discuss how an efficient implementation might look, with the aim of arriving at the current implementation in the DUNE blockkrylov framework.[13] We orient our derivation of an optimized algorithm on [32].

Suppose we are given two matrices A, B. A is an $m \times n$ matric, B is an $n \times l$ matrix. Let the $m \times l$ matrix C be their product. If one were to implement an algorithm for computing C, it might look as follows.

```
for(int i = 0; i < m; ++i) {
        for(int j = 0; j < l; ++j) {
                C[i, j] = 0;
                for(int k = 0; k < n; ++k) {
                        C[i, j] += A[i, k] * B[k, j];
                }
        }
}
```

We call this algorithm the *naive approach*. This essentially computes the product matrix by computing the (i, j)-th entry as the dot product of the i-th row of A with the j-th column of B. We already know from section 3.2 that computing the dot product is memory bound, as the data loaded into the cache is not reused, meaning we have no temporal locality. But the situation in the algorithm above is even worse, as it has also terrible spatial locality. We want to elaborate on this point further. Duneuro is a C++ toolbox. In C++, we typically store matrices in a row-wise fashion. Meaning that for example the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is stored in memory as an array $[1, 2, 3, 4, 5, 6, 7, 8, 9]$. This means that the (i, j)-th entry of the matrix is stored at position $l \cdot (i - 1) + j$ inside the array. This means that the entries $B[k, j]$ and $B[k + 1, j]$ are separated by l matrix entries in memory. Thus, even for relatively small matrices, the cache line loaded by $B[k, j]$, does not contain the value $B[k + 1, j]$. And when the algorithm gets to the point where the data inside the cache line of $B[k, j]$ is needed for further computations, e.g. for the computation of $C[i, j + 1]$, this cache line has probably already been evicted by other cache lines that go mostly unused.

We thus see that almost all memory accesses to the matrix B in the loop above result in a cache miss, resulting in many expensive loading operations. Hence we see that the inner loop of the algorithm above is even worse than computing an inner product, as the computation of the inner product can usually exploit spatial locality, since vectors are typically stored contiguously in memory.

To make the computation of the product more efficient, we need to improve temporal locality and spatial locality.

We first focus on spatial locality. The problem with the naive approach is that the matrix B is accessed column-wise, which leads to many cache misses, as described above. To improve spatial locality, we need to access B in a row-wise manner. This can be achieved by computing the product row-wise, where we compute whole rows of the column at the same time.

```
// Initialise C to zero
for(int i = 0; i < m; ++i) {
        for(int j = 0; j < l; ++l) {
```

[12] There are algorithms for matrix multiplication with lower asymptotic complexity. For example, Strassen's famous algorithm shows that matrices can be multiplied in $\mathcal{O}(n^{\log_2 7})$ arithmetic operations. Details can for example be found in [31].

[13] `https://gitlab.dune-project.org/core/dune-istl/-/blob/blockkrylov/dune/istl/blockkrylov/matrixalgebra.hh`

```
                C[i, j] = 0;
        }
}
// Compute product row-wise
for(int i = 0; i < m; ++i) {
        for(int k = 0; k < n; ++k) {
                for(int j = 0; j < l; ++j) {
                        C[i, j] += A[i, k] * B[k, j];
                }
        }
}
```

As we can derive this algorithm by essentially changing the order of the two innermost loops of the naive approach, we call it the *loop interchange approach*. Looking at the memory access pattern of this algorithm, we see that the access pattern for all matrices is row-wise. We thus see that this algorithm is spatially local, and uses all data in the loaded cache lines.

But while the loop interchange approach has a good spatial locality, its temporal locality is still not optimal. If we for example take a look at the above algorithm, we see that the algorithms runs over the whole matrix B before one of its values is used again. Furthermore, if the number of rows l gets to large, it might happen that the entries in the rows of C get evicted from the cache before they are used again in the next iteration in the innermost loop. We thus see that the temporal locality of the loop interchange approach is somewhat lacking. An approach to combat this is *loop blocking*. To simplify notation, assume that we can decompose $n = n_1 \cdot n_2$, $l = l_1 \cdot l_2$ and $m = m_1 \cdot m_2$. Then we can write A as a block matrix $A = (A_{i,j})_{\substack{1 \le i \le m_1 \\ 1 \le j \le n_1}}$, where each submatrix $A_{i,j}$ is a $m_2 \times n_2$ matrix. Similarly we can write $B = (B_{i,j})$ with $n_2 \times l_2$ submatrices and $C = (C_{i,j})$ with $m_2 \times l_2$ submatrices. Since we have $C = A \cdot B$, we immediately see that the submatrices $C_{i,j}$ fulfill the relation

$$C_{i,j} = \sum_{k=1}^{n_1} A_{i,k} \cdot B_{k,i}.$$

But then we can compute the product $C = A \cdot B$ in the following way.

```
// initialize C to zero
for(int i = 0; i < n; ++i) {
        for(int j = 0; j < l; ++j) {
                C[i, j] = 0;
        }
}
// compute C by computing the submatrices C_{i,j}
for(int i = 0; i < m_1; ++i) {
        for(int k = 0; k < n_1; ++k) {
                for(int j = 0; j < l_1; ++j) {
                        // compute matrix matrix product and sum
                        // in some way, e.g. using the loop
                        // interchange algorithm
                        C_{i,j} += A_{i,k} * B_{k,j};
                }
        }
}
```

Now assume that the values m_2, n_2 and l_2 were chosen in such a way that all submatrices needed in the innermost loop fit into the cache. Then the computation inside the innermost loop can be performed without any unnecessary loads. Furthermore, since we use the whole submatrices that were loaded, the algorithm is spatially local, and since we compute matrix products inside the innermost loop, the loaded values of all matrices are used in multiple computations, and the algorithm is thus also temporally local. Furthermore, the submatrix $A_{i,k}$ can be kept inside the cache while iterating over the innermost loop, and thus the temporal locality of the matrix A is as good as in the loop interchange algorithm.

We see that the above approach can increase the temporal locality of the loop interchange algorithm. To achieve good performance for this algorithm, the sizes of m_2, n_2 and l_2 have to be chosen with some care. They should be chosen small enough for the submatrices to fit inside a low level cache, but under this constraint as large as possible to maximize the temporal locality of the algorithm, since multiplying larger matrices leads to higher data reuse.

Since this approach was derived by subdividing the matrices into submatrices, we call it *tiled matrix multiplication*. We note that in general, this approach is a little more difficult than described here, as the matrix dimensions n, m, l might not be nicely divisible. In a general implementation, we thus choose suitable sizes for the submatrices $A_{i,j}, B_{i,j}$ and $C_{i,j}$ and cover the matrices A, B and C with them as good as possible. This

leaves some parts of the matrices uncovered, and the influences of these parts on the product $C = A \cdot B$ then have to be computed in a clean up loop.

We are now going to run a numerical experiment to compare the runtimes of the different algorithms for matrix multiplications we derived. We implemented the algorithms as stated above using C++ and compiled them using the Intel C++ compiler. We generated two executables, one with vectorization and one without vectorization. We used a single core of an Intel Skylake-SP Xeon Gold 6148 processor. In these experiments, we measure the time it takes to multiply two 2048×2048 matrices. For the tiled approach we took $m_2 = n_2 = l_2 = 16$, as our tests showed that this tile size leads to the best performance on this processor. The results are shown in table 2.

	naive	loop interchange	tiled
no vectorization	24817 ms	6304 ms	1873 ms
with vectorization	24250 ms	3567 ms	600 ms
speedup	1×	1.8×	3.1×

Table 2: Comparision of dense matrix multiplication algorithms, with and without vectorization

This table gives us two insights. First, looking at the rows, we see that simply optimizing for the memory access pattern leads to a significant speed up in the computation. Secondly, and for us more important, looking at the columns and comparing them we see that properly implemented matrix multiplication can exploit vectorization, as for example for the tiled approach we see that using vectorization leads to a significant speedup. Compare this with the naive approach, where using vectorization does not lead to any noticeable speed up, as this algorithm is entirely memory bound. We further note that our speed up of 3.1× in the tiled approach is still considerably slower than the theoretical limit of vectorization, as can be seen by looking at figure 3. This is mainly due to the fact that the author's implementation of matrix multiplication is still not optimal. Indeed, there is a great deal of literature on the efficient implementation of matrix multiplication, and developers of high performance libraries for linear algebra go to great lengths to make their algorithms as efficient as possible. To get a picture of how such a developer might design an algorithm for matrix multiplication, the reader can for example take a look at [33].

Up to this point we have talked about dense-matrix-dense-matrix multiplication. And while the eventually derived block conjugate gradient algorithm will incorporate such operations, our motivating example was that instead of performing a number of matrix-vector multiplications $A \cdot p_i$ in series, we could instead perform a single matrix-matrix multiplication $A \cdot (p_1, ..., p_k)$. But here A is the (modified) stiffness matrix from a finite element approach. This matrix is typically sparse, and typically stored using some sparse matrix format. These formats make optimizing matrix multiplication somewhat more difficult, although similar ideas as in the dense case can be applied. Since this section is already quite long, we will not go into detail and simply refer the interested reader to the literature, for example [34].

Instead, we want to show that going from iteratively computing $A \cdot p_i$ to computing $A \cdot (p_1, ..., p_k)$ is beneficial by performing a small experiment. We take A to be the modified stiffness matrix derived from a continuous Galerkin FEM approach for the realistic head model given in [15]. This is a tetrahedral head model, and we used piecewise linear ansatz functions. The resulting matrix has 885214 rows. The matrix is stored in the CRS format as described in [27]. We then fill 64 vectors $p_i \in \mathbb{R}^{885214}$ with random entries and measure the time it takes to compute $A \cdot p_i$ iteratively for all vectors, and the time it takes to compute $A \cdot (p_1, ..., p_{64})$.[14] We use the matrix multiplication algorithms implemented in the BCRS matrix class in dune-istl[15] for both computations. We note that this sparse-matrix-dense-matrix multiplication algorithm is essentially a version of the loop interchange approach described above. The results are shown in table 3.

	iterative sparse-matrix-vector	sparse-matrix-dense-matrix
no vectorization	133 sec	87 sec
with vectorization	133 sec	33 sec
speed up	1×	2.6×

Table 3: Comparision of iterative sparse-matrix-vector multiplication and sparse-matrix-dense-matrix multiplication, once without vectorization and once with vectorization

[14] We actually measure the times it takes to perform these tasks 64 times to reduce uncertainty.
[15] `https://gitlab.dune-project.org/core/dune-istl/-/blob/master/dune/istl/bcrsmatrix.hh`

Looking at this table, we see that when iteratively computing matrix-vector products, we get no noticeable speedup when using vectorization. This is what we expect, as we concluded in our discussion in section 3.2 that matrix-vector operations are memory bound and can thus not benefit from vectorization. If we use sparse-matrix-dense-matrix multiplication on the other hand, we see that using vectorization leads to a speedup of a factor of 2.6×. If we now compare the naive approach of iteratively applying matrix-vector multiplication and the approach using matrix-matrix multiplication and vectorization, we see that the execution time gets reduced from 133 seconds to 33 seconds, meaning we get a total speedup of 4×.

While this is quite good, the usage of vectorization only speeds up the computation by a factor of 2.6×, which is relatively far from the theoretical best case in figure 3. the author believes that a more fine tuned implementation of sparse-matrix-dense-matrix multiplication could lead to an even greater speed up. But since this master's thesis has to be finished at some point, we will use the implementation in dune-istl as is.

We hope that at this point the reader is thoroughly convinced that going from a matrix-vector based formulation for the computation of the transfer matrix to a matrix-matrix based formulation is a good idea. Because this is exactly what we are going to do in the next section.

4 The Block Conjugate Gradient Method

Our goal is to improve the computation of the transfer matrices. These are given by a matrix equation $A \cdot X = B$. We saw in the last section that an algorithm that solves this equation by directly working on matrices can lead to a substantial speed up, as such an algorithm has the possibility to reduce expensive memory accesses and can thus lead to a CPU bound algorithm that can benefit from vectorization. The natural question is then of course how such an algorithm can be derived.

The conjugate gradient algorithm in its preconditioned form has proven itself to be extremely effective at solving linear systems $A \cdot x = b$ for A large, sparse and symmetric positive definite, and is typically the method of choice for such systems. It thus seems natural to try to generalize the conjugate gradient method to somehow work directly on matrices.

A few approaches immediately come to mind. In the following, we assume X and B to be $n \times k$ matrices. One naive approach would be to identify the $n \times k$ matrices with $n \cdot k$-dimensional vectors. One could then read the equation $A \cdot X = B$ as a linear system in $\mathbb{R}^{n \cdot k}$ and can solve this system using the standard (preconditioned) conjugate gradient method. Then the application of the linear operator A to $n \cdot k$-dimensional vectors can be implemented as a matrix-matrix multiplication, leading to a more efficient algorithm. Yet another approach would be to read the equation $A \cdot X = B$ column-wise as a series of linear systems $A \cdot x_i = b_i$. But instead of solving these systems by iterating over every single one of them and solving them separately, a single iteration of our algorithm would consist of advancing the conjugate gradient algorithms of every system by a single iteration. If we store the vectors needed for the separate conjugate gradient algorithms in suitable matrix data structures, advancing all separate conjugate gradient algorithms by a single iteration could be done using matrix-matrix operations, which leads to a more efficient algorithm. A little less obvious would be the original *block conjugate gradient* algorithm as it was given in O'Leary's seminal paper [35]. This algorithm can essentially be derived by substituting all vectors and vector operations in the conjugate gradient algorithm by matrices and the corresponding matrix operations. One can then show that all the nice properties of the standard conjugate gradient algorithm are still true when replaced with their matrix equivalents, and one can then show that this algorithm has good convergence properties.

And it doesn't stop there. Once the above approaches are known, one can derive a number of hybrid forms of these algorithms. We thus see that there are various possible approaches to solve the equation $A \cdot X = B$ using a matrix operation based variation of the conjugate gradient algorithm.

But it turns out, thanks to the work of Frommer, Lund and Szyld in [36], [37] and [38], that all of the various approaches mentioned above **are just special cases of a single abstract *block conjugate gradient* algorithm!** In their work, they introduce an abstract mathematical framework and show how the different approaches discussed above can be seen as concrete realizations of the abstract conjugate gradient method in this framework. Once derived, we will call this abstract version of the conjugate gradient method the *block conjugate gradient method*.

This abstract framework can be derived by taking a close look at the conjugate gradient algorithm, or any other Krylov method for that matter, and extracting the mathematical concepts needed to formulate them. These concepts can then be generalized in a natural way, and the generalized concepts can then be used to derive generalized versions of the respective Krylov methods. Our aim in the next subsection is to derive this generalized framework and study its properties.

4.1 A framework to generalize Krylov Methods

4.1.1 The conjugate gradient method

We want to show why the abstract framework in the works of Frommer et al. is the natural language to generalize Krylov methods. To do this, we discuss a derivation of the conjugate gradient method and take a close look at what mathematical concepts are actually needed during this derivation. Our derivation is inspired by [14].

The conjugate gradient method can be seen as a specific instance of a so called *projection method*. Assume we want to solve a linear system $T(x) = y$, where H is a finite dimensional inner product space over the field $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$, $T : H \to H$ is a linear operator and $x, y \in H$. Assume an initial guess $x^0 \in H$ is given and we think that for some ansatz space $\mathcal{K}$ the set $x^0 + \mathcal{K}$ contains good approximations to the actual solution x. Given some test space $\mathcal{L}$, we call a vector $x_\mathrm{p} \in x^0 + \mathcal{K}$ *projection solution* if its residual is orthogonal to the test space, meaning

$$\langle y - T(x_\mathrm{p}), v\rangle = 0 \qquad \text{for all } v \in \mathcal{L}.$$

The natural question is then of course how one can go about choosing a sensible ansatz space $\mathcal{K}$. One such heuristic can be derived from the Cayley-Hamilton theorem. It states that for an endomorphism T with characteristic polynomial χ_T we have

$$\chi_T(T) = 0.$$

A proof can be found in almost any book about Linear Algebra, e.g. [39]. Let H be n-dimensional. By taking a close look at the Leibniz formula for determinants, or by looking at [39], one sees that the characteristic polynomial is of the form

$$\chi_T(t) = (-1)^n \cdot \det(T) + \sum_{k=1}^{n} a_k t^k.$$

Now let us assume that the operator T is invertible. Then $\det(T) \neq 0$, and if id is the identity operator we thus have

$$\mathrm{id} = \sum_{k=1}^{n} \frac{(-1)^{n+1} a_k}{\det(T)} T^k.$$

Multiplying both sides by T^{-1} then gives

$$T^{-1} = \sum_{k=0}^{n-1} \frac{(-1)^{n+1} a_{k+1}}{\det(T)} T^k.$$

We thus see that there is some polynomial $p = \sum_{k=0}^{n-1} b_k t^k$ of degree at most $n-1$ so that $T^{-1} = p(T)$. Now assume we have some initial guess x_0 and want to find a vector $v \in H$ so that $T(x_0 + v) = y$. This is equivalent to $T(v) = y - T(x_0) =: r^0$. This means we want to compute

$$v = T^{-1}(r^0) = p(T)(r^0) = \sum_{k=0}^{n-1} b_k T^k(r^0).$$

Then for $m < n$ the first few summands $\sum_{k=0}^{m} b_k T^k(r^0)$ might be a good approximation for v. Hence a natural ansatz space to search for approximation of v is $\{q(T)(r^0) \mid q \in P_m\}$, where P_m is the space of all polynomial of degree at most m. We thus see that it is a natural approach to choose the space

$$\mathcal{K}^{m+1}(T, r^0) := \{q(T)(r^0) \mid q \in P_m\} = \mathrm{span}\{r^0, T(r^0), ..., T^m(r^0)\}$$

as an ansatz space for the projection method. We call these spaces *Krylov subspaces*. Furthermore, we call a method that uses a projection method with a Krylov subspace as an ansatz space a *Krylov method*.

Now assume the operator T is positive definite[16] and we want to solve the linear equation $T(x) = y$. Then the *conjugate gradient method* is defined as a method that computes iterates $x^1, x^2, ... \in H$ so that x_k is the solution of a projection approach with ansatz space $\mathcal{K}^k(T, r^0)$ and test space $\mathcal{K}^k(T, r^0)$. We first show that this definition makes sense. Note that T defines a norm on H via the formula $\|x\|_T := \sqrt{\langle T(x), x\rangle}$.

[16] When saying positive definite we implicitly assume that T is symmetric if $\mathbb{K} = \mathbb{R}$ and hermitian if $\mathbb{K} = \mathbb{C}$.

Theorem 19. Let $k \geq 1$. Then there exists exactly one solution $x_k \in H$ of the problem

$$x^k \in x^0 + \mathcal{K}^k(T, r^0) \tag{1}$$
$$y - T(x^k) \perp \mathcal{K}^k(T, r^0) \tag{2}$$

This x^k can be characterized as the best possible approximation in the $\|\cdot\|_T$-norm to the exact solution x that can be formed using the ansatz space, meaning

$$x^k = \underset{z \in x^0 + \mathcal{K}^k(T, r^0)}{\arg\min} \|x - z\|_T. \tag{3}$$

Proof. It is a well known theorem that, given some point and a closed convex subset of a Hilbert space, there exists a unique best approximation to the given point inside the closed convex subset. A proof can e.g. be found in [40], 16.4. Since $\|\cdot\|_T$ is a norm coming from the inner product $\langle T(v), w\rangle$, and since every norm on a finite dimensional space is complete, we can use (3) to uniquely define x^k. It then remains to show that (3) is equivalent to (1) plus (2).

Lets first assume (1) plus (2). Let $v \in \mathcal{K}^k(T, r^0)$. Then

$$\langle T(x - x^k), v\rangle = \langle y - T(x^k), v\rangle = 0,$$

meaning $x - x^k \perp_T \mathcal{K}^k(T, r^0)$. Here $\perp_T$ means orthogonal with respect to the inner product induced by T. Now let $z \in x^0 + \mathcal{K}^k(T, r^0)$. Then there is some $v \in \mathcal{K}^k(T, r^0)$ so that $z = x^0 + v$, and using Pythagoras's theorem we get

$$\|x - z\|_T^2 = \|x - x^k + \underbrace{x^k - z}_{\in \mathcal{K}^k(T, r^0)}\|_T^2 = \|x - x^k\|_T^2 + \|x^k - z\|_T^2 \geq \|x - x^k\|_T^2.$$

This shows (3). Now assume (3) is true and we want to show (2). By the same argument as before, it suffices to show that $x - x^k \perp_T \mathcal{K}^k(T, r^0)$. So let $0 \neq v \in \mathcal{K}^k(T, r^0)$. Let $\langle T(u), w\rangle =: \langle u, w\rangle_T$. By (3) we then have for all $\lambda \in \mathbb{K}$

$$\begin{aligned}\|x - x^k\|_T^2 &\leq \|x - x^k - \lambda v\|_T^2 \\ &= \|x - x^k\|_T^2 + |\lambda|^2\|v\|_T^2 - \overline{\lambda}\langle x - x^k, v\rangle_T - \lambda\langle v, x - x^k\rangle_T,\end{aligned}$$

which gives

$$0 \leq |\lambda|^2\|v\|_T^2 - \overline{\lambda}\langle x - x^k, v\rangle_T - \lambda\langle v, x - x^k\rangle_T.$$

Setting $\lambda = \frac{\langle x - x^k, v\rangle_T}{\|v\|_T^2}$ then gives

$$0 \leq -\frac{|\langle x - x^k, v\rangle_T|^2}{\|v\|_T^2} \leq 0,$$

which implies $\langle x - x^k, v\rangle_T = 0$. We have thus shown $x - x^k \perp_T \mathcal{K}^k(T, r^0)$, which implies (2). □

We have now defined what vectors $x^1, x^2, ...$ the conjugate gradient method is supposed to compute. But to arrive at an actual algorithm, we need to specify how we can actually compute these iterates. To arrive at such an algorithm, assume that $p^0, p^1, ..., p^m$ are chosen in such a way that for $0 \leq k \leq m$ we have

$$\mathcal{K}^{k+1}(T, r^0) = \operatorname{span}\{p^0, ..., p^k\},$$

and $\langle p^i, p^j\rangle_T = \mu_i \delta_{i,j}$, where $\mu_i \neq 0$ for all $0 \leq i \leq m$. Inspired by the Gram-Schmidt process we can define for $k \geq 0$

$$z^k = x_0 + \sum_{j=0}^{k-1} \frac{\langle x - x^0, p^j\rangle_T}{\langle p^j, p^j\rangle_T} p^j.$$

Then we have $z^k \in x^0 + \mathcal{K}^k(T, r^0)$, and furthermore for every $1 \leq i \leq k - 1$

$$\begin{aligned}\langle y - T(z^k), p^i\rangle &= \langle x - z^k, p^i\rangle_T = \langle x - x_0, p^i\rangle_T - \sum_{j=1}^{k-1} \frac{\langle x - x^0, p^j\rangle_T}{\langle p^j, p^j\rangle_T} \cdot \langle p^j, p^i\rangle_T \\ &= \langle x - x^0, p^i\rangle_T - \langle x - x^0, p^i\rangle_T = 0.\end{aligned}$$

Since $\{p_0, ..., p_{k-1}\}$ span $\mathcal{K}^k(T, r^0)$ we see that $y - T(z^k) \perp \mathcal{K}^k(T, r^0)$. Hence z^k fulfills (1) and (2), and since this uniquely determins the conjugate gradient iterates we see that $z^k = x^k$ for $k \geq 1$. By definition, we also have $z^0 = x^0$. The central takeaway of this computation is that we thus have

$$\begin{aligned} x^{k+1} &= x^0 + \sum_{j=0}^{k} \frac{\langle x - x^0, p^j \rangle_T}{\langle p^j, p^j \rangle_T} p^j = x^0 + \sum_{j=0}^{k-1} \frac{\langle x - x^0, p^j \rangle_T}{\langle p^j, p^j \rangle_T} p^j + \frac{\langle x - x^0, p^k \rangle_T}{\langle p^k, p^k \rangle_T} p^k \\ &= x^k + \frac{\langle x - x^0, p^k \rangle_T}{\langle p^k, p^k \rangle_T} p^k. \end{aligned} \tag{4}$$

Thus, if we have computed x^k and have some way to compute a vector

$$p^k \in \mathcal{K}^{k+1}(T, r^0) \cap \mathcal{K}^k(T, r^0)^{\perp_T},$$

we can compute the next conjugate gradient iterate x^{k+1} by a simple update $x^{k+1} = x^k + \lambda^k p^k$. It turns out that computing such a p^k can be done quite easily.[17] Define for $k \geq 0$ the *residual* of x^k to be $r^k := y - T(x^k)$. Then (2) is equivalent to $r^k \perp \mathcal{K}^k(T, r^0)$. We then have the following.

Lemma 20. For all $k \geq 1$ we have

$$\operatorname{span}\{r^0, r^1, ..., r^{k-1}\} \subset \mathcal{K}^k(T, r^0).$$

Proof. We proof the lemma by induction on k. For $k = 1$ the statement follows directly from the definition of the Krylov space $\mathcal{K}^1(T, r^0)$. So assume that the statement is true for some $k \geq 1$. Then

$$\begin{aligned} r^k &= y - T(x^k) = y - T(x^{k-1}) - T(x^k - x^{k-1}) \\ &= r^{k-1} - T(\underbrace{x^k - x^{k-1}}_{\in \mathcal{K}^k(T, r^0)}) \in \mathcal{K}^{k+1}(T, r^0), \end{aligned}$$

where we have used $r^{k-1} \in \mathcal{K}^k(T, r^0)$ and $T(\mathcal{K}^k(T, r^0)) \subset \mathcal{K}^{k+1}(T, r^0)$. Hence we have

$$\operatorname{span}\{r^0, ..., r^k\} \subset \mathcal{K}^{k+1}(T, r^0).$$

□

Now assume we already know some vector

$$p^{k-1} \in \mathcal{K}^k(T, r^0) \cap \mathcal{K}^{k-1}(T, r^0)^{\perp_T}.$$

If $r^k = 0$, then x^k is already the exact solution and we are finished. So assume $r^j \neq 0$ for all $1 \leq j \leq k$. Furthermore, let $w_1, ..., w_{k-1}$ be an orthonormal basis of $\mathcal{K}^{k-1}(T, r^0)$.[18]. By lemma 20 we know that $r^k \in \mathcal{K}^{k+1}(T, r^0)$. Thus, by the Gram Schmidt process, we can define a vector $0 \neq p^k \in \mathcal{K}^{k+1}(T, r^0) \cap \mathcal{K}^k(T, r^0)^{\perp_T}$ by setting

$$p^k := r^k - \frac{\langle r^k, p^{k-1} \rangle_T}{\langle p^{k-1}, p^{k-1} \rangle_T} p^{k-1} - \sum_{j=1}^{k-1} \langle r^k, w_j \rangle_T w_j.$$

But now for $1 \leq j \leq k-1$ we have $T(w_j) \in T(\mathcal{K}^{k-1}(T, r^0)) \subset \mathcal{K}^k(T, r^0)$, and since by definition we have $r^k \perp \mathcal{K}^k(T, r^0)$ we have for $1 \leq j \leq k-1$

$$\langle r^k, w_j \rangle_T = \langle r^k, T(w_j) \rangle = 0.$$

Hence we can define p^k by a simple update

$$p^k = r^k - \frac{\langle r^k, p^{k-1} \rangle_T}{\langle p^{k-1}, p^{k-1} \rangle_T} p^{k-1}. \tag{5}$$

These considerations set us up for a simple iteration to compute the iterates of the conjugate gradient method.

1. If we know x^k and some vector $p^k \in \mathcal{K}^{k+1}(T, r^0) \cap \mathcal{K}^k(T, r^0)^{\perp_T}$, we can compute the next iterate x^{k+1} by the simply updating according to equation (4). We can then also directly compute the next residual r^{k+1}.

[17] The fact that this is so simple is actually the heart of the conjugate gradient algorithm, and differentiates it from other Krylov methods that e.g. require the Arnoldi process to compute bases of Krylov spaces.

[18] If none of the residuals r^j, $0 \leq j \leq i-1$, vanishes, one easily sees that we have $\dim(\mathcal{K}^i(T, r^0)) = i$.

2. If x^{k+1} is not already the exact solution, we can then compute some vector $p^{k+1} \in \mathcal{K}^{k+2}(T, r^0) \cap \mathcal{K}^{k+1}(T, r^0)^{\perp_T}$ by combining r^{k+1} and p^k according to equation (5). We can then continue at step 1.

At the start of the iteration x^0 is given. We can then choose $p^0 := r^0$.

We have thus derived an iterative algorithm to compute the projection solutions x^k defined by the Galerkin condition

$$x^k \in x^0 + \mathcal{K}^k(T, r^0)$$
$$y - T(x^k) \perp \mathcal{K}^k(T, r^0)$$

To arrive at the standard formulation of the conjugate gradient method, we have to do some more work. Assume that the p^k are defined by the iteration above. Then there is some scalar λ^k with $x^{k+1} = x^k + \lambda^k p^k$. Then we have

$$r^{k+1} = y - T(x^{k+1}) = y - T(x^k) - \lambda^k T(p^k) = r^k - \lambda^k T(p^k). \tag{6}$$

Then condition (2) implies that we have

$$0 = \langle r^{k+1}, r^k \rangle = \langle r^k, r^k \rangle - \lambda^k \langle T(p^k), r^k \rangle \iff \lambda^k = \frac{\langle r^k, r^k \rangle}{\langle T(p^k), r^k \rangle}.$$

Furthermore we have $\langle T(p^k), r^k \rangle = \langle T(p^k), p^k \rangle$. For $k = 0$ this follows from $p^0 = r^0$, and for $k \geq 1$ we have by construction that p^{k-1} is T-orthogonal to p^k, and the claim follows from equation (5). We thus have $\lambda^k = \frac{\langle r^k, r^k \rangle}{\langle T(p^k), p^k \rangle}$.

If $r^k \neq 0$, we have seen that $p^k \neq 0$, and since T is positive definite it follows that $\lambda^k \neq 0$. Thus equation (6) gives us $T(p^k) = \frac{1}{\lambda^k}(r^k - r^{k+1})$, which leads to

$$\langle r^{k+1}, p^k \rangle_T = -\frac{\langle r^{k+1}, r^{k+1} \rangle}{\lambda^k} = -\frac{\langle r^{k+1}, r^{k+1} \rangle}{\langle r^k, r^k \rangle} \langle p^k, p^k \rangle_T.$$

We can thus rewrite equation (5) as

$$p^{k+1} = r^{k+1} + \frac{\langle r^{k+1}, r^{k+1} \rangle}{\langle r^k, r^k \rangle} p^k.$$

We have thus finally arrived at the conjugate gradient method in its standard form as it is described in algorithm 1 on page 9.

The convergence behavior of the conjugate gradient method is essentially determined by the spectrum of the linear operator T, as we will later see in a more general context. Let $\lambda_{\max}$ and $\lambda_{\min}$ denote the largest respectively smallest eigenvalue of T. Then $\kappa(T) = \frac{\lambda_{\max}}{\lambda_{\min}}$ is called the *condition* of T, and one observes (and can to some degree prove) that the conjugate gradient method generally converges faster for operators with smaller condition. One tries to exploit this by, instead of applying the conjugate gradient method to the equation $T(x) = y$ directly, applying the conjugate gradient method to the equation $S(T(x)) = S(b)$. For this assume that S is a positive definite operator so that $S \approx T^{-1}$. Then the condition $\kappa(S \circ T)$ is hopefully smaller than the condition $\kappa(T)$, and we can hope that the conjugate gradient algorithm for $S \circ T(x) = S(y)$ converges faster than the conjugate gradient algorithm for $T(x) = y$. It is customary to call the operator S^{-1} *preconditioner*, since its job is to improve the condition of the operator T.

For this to make sense we have to talk about a few things. First, $S \circ T$ is in general not positive definite for S, T positive definite, so at first sight it makes no sense to apply the conjugate gradient algorithm. But we can notice that

$$\langle x, y \rangle_{S^{-1}} := \langle S^{-1}(x), y \rangle$$

defines an inner product on H, and an easy calculation shows that $S \circ T$ is positive definite with respect to this inner product. One can thus apply the conjugate gradient method with respect to the inner product $\langle \cdot, \cdot \rangle_{S^{-1}}$ to the equation $S \circ T(x) = S(y)$ and can reap the potential benefits of an operator with small condition. It is then an easy algebraic exercise to derive an algorithm that computes the corresponding iterates while only using the standard inner product, and only needing to apply the operator S once per iteration. This iteration is then called the *preconditioned conjugate gradient method*. Details can be found in [14]. We do not give this derivation at this point since we will later derive it in a more general context.

We have thus derived the preconditioned conjugate gradient method. Looking at the derivation, we see that it was entirely based on the inner product space structure of the inner product space H over the field $\mathbb{K}$. It is thus natural to ask if there is some way to generalize this structure, and hope that we can use the above derivation as a template for the derivation of a generalized conjugate gradient method. This is now our goal. The first step in this direction is to discuss how to generalize the mathematical structure of an inner product space over the field $\mathbb{K}$.

4.1.2 Generalizing inner product spaces

In the following we always assume $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$. The mathematics is inspired by [41], [42] and [25].

The first step is to generalize the underlying field $\mathbb{K}$. The natural algebraic generalization of this field is an involutive algebra.

Definition 21 (Algebra)**.** Let A be a $\mathbb{K}$-vector space. A is then called a $\mathbb{K}$-*algebra* if A is equipped with a map

$$A \times A \to A; (a, b) \mapsto a \cdot b,$$

called *multiplication*, with the following properties.

1. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in A$, meaning the multiplication is *associative*.
2. The map $(a, b) \mapsto a \cdot b$ is bilinear.

If there is some element 1_A with $x \cdot 1_A = 1_A \cdot x = x$ for all $x \in A$ we call A *unital*. Note that such an element, if it exists, is necessarily unique. If A is unital and $x \in A$, we call x *invertible* if there exists $y \in A$ with $x \cdot y = y \cdot x = 1_A$.

If for all $x, y \in A$ we have $x \cdot y = y \cdot x$ we call the algebra *commutative*.

Definition 22 (Involutive algebra)**.** Let A be an algebra. A map $* : A \to A; x \mapsto x^*$ is called an *involution* if for all $a, b \in A$ and $\lambda \in \mathbb{K}$ we have

1. $(x^*)^* = x$
2. $(x + y)^* = x^* + y^*$
3. $(\lambda x)^* = \overline{\lambda} x^*$
4. $(x \cdot y)^* = y^* \cdot x^*$

An algebra A endowed with an involution is called an *involutive algebra*, or $*$-algebra.

We give a few examples.

Example 23. (1) The field $\mathbb{K}$ is itself an involutive $\mathbb{K}$-algebra with the usual product and with involution defined as complex conjugation.

(2) The set of $n \times n$-matrices with entries in $\mathbb{K}$ becomes an involutive $\mathbb{K}$-algebra when equipped with the usual matrix multiplication as multiplication and with the conjugate transpose operation as involution.

(3) More generally for a Hilbert space H the set $\mathcal{L}(H)$ of bounded linear operators on H becomes an involutive algebra with multiplication defined as composition of linear maps and involution defined as mapping a linear operator to its adjoint operator.

(4) If I is an index set and A_i is an involutive algebra for every $i \in I$, their cartesian product becomes an involutive algebra in the natural way. We call this algebra the *direct product* of the A_i. The subset of tuples that have only finitely many non-zero entries forms an involutive subalgebra, called the *direct sum* of the A_i. The direct sum is denoted by

$$\bigoplus_{i \in I} A_i.$$

Note that for finite sets I the direct sum and the direct product are identical.

(5) This next example is of no practical relevance for us, but it illustrates that the concept of an involutive algebra is quite general. We define $A = L^1(\mathbb{R}^n)$ to be the set of integrable $\mathbb{K}$-valued functions modulo functions differing on sets of measure zero. If we define multiplication via *convolution*, meaning that for $f, g \in L^1(\mathbb{R}^n)$ we set

$$f * g(x) = \int_{\mathbb{R}^n} f(t) g(x - t) \, d\lambda(t),$$

and we define the involution via $f^*(x) := \overline{f(-x)}$, one can see that A becomes a commutative involutive algebra.

In the following, we will be most interested in example (2) in combination with example (4). Notice that in this case, the corresponding algebra is in general not commutative.

One can think about the involution as a generalization of the complex conjugation. But note that the need for an involution is not a purely complex phenomenon. For instance in example (2) in the real case the involution is given by transposition.

Definition 24. Let A be an involutive algebra. We call $x \in A$ *self-adjoint* if $x = x^*$. Furthermore, we call self-adjoint element x *positive* if it can be written in the form $x = y^*y$. In this case we write $x \geq 0$. Furthermore, for $x, y \in A$ we say that $y \geq x$ if $y - x \geq 0$.

The self-adjoint elements in an algebra play the same role as the real numbers inside the complex numbers, and the positive elements play the same role as the nonnegative real numbers inside the complex numbers. Again, note that even for real involutive algebras these definitions are non trivial, as they for example in the case of real $n \times n$ real matrices correspond to symmetric and positive semidefinite matrices respectively.

Now we have the language to define the generalization of an inner product space.

Definition 25. Let $\mathcal{X}$ be a $\mathbb{K}$-vector space and let A be a $\mathbb{K}$-algebra. We call $\mathcal{X}$ a *(right) A-module* if there is a map $\mathcal{X} \times A \to \mathcal{X}; (x, a) \mapsto x \cdot a$ so that for all $x, y \in \mathcal{X}$ and $a, b \in A$ and $\lambda \in \mathbb{K}$ we have

1. $(x + y) \cdot a = x \cdot a + y \cdot a$
2. $x \cdot (a + b) = x \cdot a + x \cdot b$
3. $(x \cdot a) \cdot b = x \cdot (a \cdot b)$
4. $\lambda \cdot (x \cdot a) = (\lambda \cdot x) \cdot a = x \cdot (\lambda \cdot a)$

If A is unital we also require

5. $x \cdot 1_A = x$

Note that the requirements for the multiplication $X \times A \to X$ are formally the same as the requirements for vector spaces. Thus we can view A-modules as generalizations of vector spaces.

Now we generalize the inner product.

Definition 26. Let $\mathcal{X}$ be an A-module. Furthermore assume that A is endowed with an involution. Then we call $\mathcal{X}$ an *inner product A-module* if there is a map $\langle \cdot, \cdot \rangle : \mathcal{X} \times \mathcal{X} \to A$ so that for all $x, y, z \in \mathcal{X}, a \in A$ and $\lambda, \mu \in \mathbb{K}$ we have

1. $\langle x, \lambda y + \mu z \rangle = \lambda \langle x, y \rangle + \mu \langle x, z \rangle$
2. $\langle x, y \cdot a \rangle = \langle x, y \rangle \cdot a$
3. $\langle x, y \rangle^* = \langle y, x \rangle$
4. $\langle x, x \rangle \geq 0$ (as an element of A)
5. $\langle x, x \rangle = 0 \implies x = 0$

Remark. Let $\mathcal{X}$ be an inner product A-module. Then the inner product is conjugate linear in the first variable, since

$$\langle \lambda x + \mu y, z \rangle = \langle z, \lambda x + \mu y \rangle^* = (\lambda \langle z, x \rangle + \mu \langle z, y \rangle)^* = \bar{\lambda} \langle x, z \rangle + \bar{\mu} \langle y, z \rangle.$$

Furthermore we have

$$\langle x \cdot a, y \rangle = \langle y, x \cdot a \rangle^* = (\langle y, x \rangle a)^* = a^* \langle x, y \rangle.$$

Our overarching goal is to solve equations of the form $A \cdot X = B$, where A is a symmetric positive definite matrix and X, B are $n \times k$ matrices. We are thus interested in endowing the set $\mathcal{X} = \mathbb{K}^{n \times k}$ of $n \times k$-matrices with entries in $\mathbb{K}$ with the structure of an inner product module.

Example 27. **(1):** Let $A = \mathbb{K}$. Then the inner product A-modules are exactly the inner product spaces.

(2): Let $A = \mathbb{K}$. Then $\mathcal{X} = \mathbb{K}^{n \times k}$ becomes an inner product A-module with the standard scalar mutliplication and inner product defined by

$$\langle X, Y \rangle := \operatorname{Tr}(X^* Y).$$

Here X^* is the conjugate transpose of the matrix X. Notice that this is simply the standard euclidean inner product if we identify the $n \times k$ matrices with the $n \cdot k$-dimensional vectors.

(3): Let $A = \mathbb{K}^{k\times k}$ be the $k \times k$-matrices. Then $\mathcal{X} = \mathbb{K}^{n\times k}$ becomes a right A-module by standard matrix multiplication $(x, a) \mapsto x \cdot a$. If we further define

$$\langle X, Y\rangle = X^*Y,$$

where X^* is the conjugate transpose of the matrix X, $\mathcal{X}$ becomes in inner product A-module.

Proof. It is obvious that condition 1) - 3) of Definition 26 are true. To show 4), notice that for $X \in \mathcal{X}$ and $v \in \mathbb{K}^k$ we have

$$\langle X^*Xv, v\rangle = \langle Xv, Xv\rangle \geq 0.$$

Hence X^*X is positive semidefinite. By standard theory this implies that all eigenvalues of X^*X are nonnegative. By the spectral theorem we can thus write $X^*X = S\,\mathrm{diag}(\lambda_1, .., \lambda_k)S^*$ with $\lambda_1, ..., \lambda_k \geq 0$ and an unitary matrix S. If we then set $M = S\,\mathrm{diag}(\sqrt{\lambda_1}, ..., \sqrt{\lambda_k})S^*$, we have $M^* = M$ and $M^*M = M^2 = X^*X$. Thus X^*X is of the form M^*M with $M \in A$ and hence positive in A.

To show 5), simply notice that the entries of X^*X are the standard inner products in $\mathbb{K}^n$ of the columns of X, and in particular the diagonal values of X^*X are the euclidean norms of the columns of X. Hence $X^*X = 0$ implies $X = 0$. □

Once we have derived the generalized conjugate gradient method we will see that this choice of inner product module structure leads to the variant of the block conjugate gradient algorithm as it was originally described by O'Leary in [35]. We thus call this algebra the *classical algebra* and denote it with $\mathbb{S}_{\mathrm{cl}}$. The corresponding inner product is called the *classical inner product* and denoted by $\langle\cdot,\cdot\rangle_{\mathrm{cl}}$.

(4): Let $\mathcal{X} = \mathbb{K}^{n\times k}$ and $A = \mathbb{K}^k$. A becomes an involutive algebra by componentwise multiplication and complex conjugation. Then we can define for $X = (x_1, ..., x_k) \in \mathcal{X}$ and $a = (a_1, ..., a_k) \in \mathbb{K}^k$

$$X \cdot a = (a_1 \cdot x_1, ..., a_k \cdot x_k),$$

where the multiplication on the right hand side is simply the standard scalar multiplication in $\mathbb{K}^n$. This gives $\mathcal{X}$ the structure of an A-module. We can now define an inner product via

$$\langle X, Y\rangle = \mathrm{diag}(X^*Y),$$

where X^* is again the conjugate transpose. With this definition, $\mathcal{X}$ becomes an inner product A-module.

Proof. Notice that for $X = (x_1, ..., x_k)$ and $Y = (y_1, ..., y_k)$ we have

$$\langle X, Y\rangle = (\langle x_1, y_1\rangle, ..., \langle x_k, y_k\rangle),$$

and one can immediately see that 1) - 3) from definition 26 are true. The positive elements in this algebra are the elements of the form a^*a, meaning the set of vector $z \in \mathbb{K}^k$ so that $z = (z_1, ..., z_k)$, where each z_i is of the form $z_i = \bar{a_i}a_i$. This is simply the set of vectors $z \in \mathbb{K}^k$ where every component is a nonnegative real number. Thus we get for $X = (x_1, ..., x_k)$

$$\langle X, X\rangle = (\langle x_1, x_1\rangle, ..., \langle x_k, x_k\rangle) \geq 0 \in \mathbb{K}^k,$$

and we see that 4) is true. 5) is true by the same reasoning as in example (3). □

We call this algebra the *parallel algebra* and denote it $\mathbb{S}_{\mathrm{pl}}$.

(5): Let $\mathcal{X} = \mathbb{K}^{n\times k}$. Let $k = k_1 + ... + k_m$, where $k_1, ..., k_m \in \mathbb{N}$. Let $\mathbb{K}^{k_i\times k_i}$ denote the $k_i \times k_i$-matrices with entries from $\mathbb{K}$. We can then set

$$A = \bigoplus_{i=1}^{m} \mathbb{K}^{k_i\times k_i}.$$

If $X \in \mathcal{X}$ we can write $X = (X_1, ..., X_m)$, where $X_i \in \mathbb{K}^{n\times k_i}$. If we now have $a = (a_1, ..., a_m) \in A$, we can define

$$X \cdot a = (X_1 \cdot a_1, ..., X_m \cdot a_m),$$

where the multiplications on the right hand side are standard matrix multiplications. Then $\mathcal{X}$ becomes an A-module. Note that A is a unital algebra with $1_A = (E_{k_1}, ..., E_{k_m})$, where E_j is the identity matrix of dimension j. If $X = (X_1, ..., X_m)$ and $Y = (Y_1, ..., Y_m)$ we can then define

$$\langle X, Y\rangle = (X_1^*Y_1, ..., X_m^*Y_m),$$

where X_i^* on the right hand side denotes the conjugate transpose. With these definitions $\mathcal{X}$ becomes an inner product A-module.

Proof. One directly sees that conditions 1) - 3) from definition 26 are true. 4) follows immediately from the observation that for algebras $A_i, i \in I$ an element $z \in \bigoplus_{i\in I} A_i$ is positive if and only if every component of z is positive, and the proof from example (3). 5) also follows from the proof of example (3). □

Now assume that we have $k = p * q$. Then in the formulation above we can choose $m = q$ and $k_1 = ... = k_q = p$. The corresponding involutive algebra is called the *hybrid algebra* and denoted $\mathbb{S}^{p,q}_{\text{hy}}$. Note that $\mathbb{S}^{1,k}_{\text{hy}} = \mathbb{S}_{\text{pl}}$ and $\mathbb{S}^{k,1}_{\text{hy}} = \mathbb{S}_{\text{cl}}$.

(6): Let $\mathcal{X} = \mathbb{K}^{n\times k}$. Assume again that we have $k = p \cdot q$. Then let $A = \mathbb{K}^{p\times p}$. If $X \in \mathcal{X}$, we can again write $X = (X_1, ..., X_q)$ with $X_i \in \mathbb{K}^{n\times p}$. For $a \in A$ we can then define

$$X \cdot a = (X_1 \cdot a, ..., X_q \cdot a).$$

This gives $\mathcal{X}$ the structure of an A-module. If we now have $Y = (Y_1, ..., Y_q) \in \mathcal{X}$, we can define

$$\langle X, Y\rangle = \sum_{j=1}^{q} X_j^* Y_j.$$

This gives $\mathcal{X}$ the structure of an inner product A-module.

Proof. 1) - 3) are again clear. To show 4) it suffices to show that $\langle X, X\rangle$ defines a positive semidefinite matrix. The claim then follows from the proof of example (3). For $v \in \mathbb{K}^k$ we have

$$\langle\langle X, X\rangle v, v\rangle = \sum_{j=1}^{q} \langle X_j v, X_j v\rangle \geq 0.$$

Thus 4) is fulfilled. To show 5) assume that $\langle X, X\rangle = 0$. The computation above then shows that for all $v \in \mathbb{K}^k$ and for all $1 \leq j \leq q$ we have $\langle X_j v, X_j v\rangle = 0$. This shows $X_j = 0$ and hence $X = 0$. This shows 5). □

We thus see that there are various different inner product module structures on the set of $n \times k$-matrices. Once the generalized conjugate gradient algorithm has been derived, every single one of these structures will lead to a different iterative algorithm for solving the equation $A \cdot X = B$. The reader can already look forward to investigating the convergence properties of these algorithms.

We have now seen that the natural generalization of inner product spaces over the field $\mathbb{K}$ are inner product modules over involutive $\mathbb{K}$-algebras. We will now investigate what properties from inner product spaces carry over to the general case.

Definition 28. Let $\mathcal{X}$ and $\mathcal{Y}$ be inner product A-modules.

1. A map $T : \mathcal{X} \to \mathcal{Y}$ is called A-linear, if it is linear and for all $X \in \mathcal{X}$ and $a \in A$ we have $T(x \cdot a) = T(x) \cdot a$.
2. Let $X_1, ..., X_k \in \mathcal{X}$. We then define

$$\text{span}_A\{X_1, ..., X_k\} = \{\sum_{j=1}^{k} X_j a_j \mid a_1, ..., a_k \in A\} \subset X.$$

   Notice that $\text{span}_A\{X_1, ..., X_k\}$ is the smallest sub-module of $\mathcal{X}$ containing $X_1, ..., X_k$.
3. Let $X, Y \in \mathcal{X}$. Then we say X and Y are *orthogonal* if $\langle X, Y\rangle = 0$. If $\mathcal{Z} \subset \mathcal{X}$ we say that X is orthogonal to $\mathcal{Z}$ if X is orthogonal to every element of $\mathcal{Z}$. We write $X \perp Y$ and $X \perp \mathcal{Z}$.
4. Let $X_1, ..., X_k \in \mathcal{X}$. We say that $\{X_1, .., X_k\}$ is an *orthogonal basis* of X if we have $\text{span}_A\{X_1, ..., X_k\} = X$, for all $i \neq j$ we have $\langle X_i, X_j\rangle = 0$ and $\langle X_i, X_i\rangle$ is an invertible element of the algebra A. If for all $1 \leq i \leq k$ we have $\langle X_i, X_i\rangle = 1_A$ we call the basis *orthonormal*.

Note that orthogonality is a symmetric relation we have since $0^* = 0$. This means that $X \perp Y \iff Y \perp X$.

It is here that we encounter the first difference in the theory of inner product spaces and the theory of inner product modules. One statement central in Hilbert space theory is that every Hilbert space possesses an orthonormal basis. This is no longer true.

Remark. There exist inner product modules without an orthogonal basis.

Proof. Let $\mathcal{X} = \mathbb{R}^{3\times 2}$ and $A = \mathbb{R}^{2\times 2}$. Then $\mathcal{X}$ is an inner product A-module with respect to standard matrix multiplication and inner product

$$\langle X, Y\rangle = X^\intercal Y.$$

This is a special case of example 27 (3). Assume that there exists an orthogonal basis $X_1, ..., X_k$ of $\mathcal{X}$. By substituting X_i with $X_i \cdot \langle X_i, X_i\rangle^{-\frac{1}{2}}$ we can assume the basis to be orthonormal. Note that $\langle X_i, X_i\rangle^{-\frac{1}{2}}$ exists. To see this look at the proof of example 27 (3). The M constructed there is clearly invertible if X^*X is invertible, and hence M^{-1} exists and we can set $\langle X, X\rangle^{-\frac{1}{2}} := M^{-1}$.

Hence we can assume $X_1, ..., X_k$ to be an orthonormal basis of $\mathcal{X}$. We can write $X_j = (X_j^1, X_j^2)$ with each $X_j^i \in \mathbb{R}^3$. Since for $X = (X^1, X^2)$ and $Y = (Y^1, Y^2)$ we have

$$\langle X, Y\rangle = \begin{pmatrix} \langle X^1, Y^1\rangle & \langle X^1, Y^2\rangle \\ \langle X^2, Y^1\rangle & \langle X^2, Y^2\rangle \end{pmatrix},$$

where the inner products on the right hand side are the standard inner products on $\mathbb{R}^3$, the condition $\langle X_i, X_j\rangle = 0$ for $i \neq j$ implies that the columns of X_i are orthogonal to the columns of X_j. Since the basis is orthonormal we also have

$$\langle X_i, X_i\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Hence the columns of X_i form an orthonormal set. But this implies that the set $\{X_1^1, X_1^2, X_2^1, X_1^2, ..., X_k^1, X_k^2\}$ is orthonormal in $\mathbb{R}^3$. If $v \in \mathbb{R}^3$ is arbitrary we have $(v, 0) \in \mathbb{R}^{3\times 2}$, and hence we can write

$$(v, 0) = \sum_{j=1}^{k} X_j a_j$$

with $a_j \in \mathbb{R}^{2\times 2}$. Looking at the first column of this equation we see that we have $v \in \operatorname{span}\{X_1^1, X_1^2, ..., X_k^1, X_k^2\}$. Hence $\{X_1^1, X_1^2, ..., X_k^1, X_k^2\}$ is an orthonormal basis of $\mathbb{R}^3$, and we get

$$3 = \dim(\mathbb{R}^3) = 2k,$$

which is a contradiction. This shows that $\mathcal{X}$ possesses no orthogonal basis.

The same proof in fact shows that the $n \times l$-matrices as a module over the $l \times l$-matrices with the inner product $\langle X, Y\rangle = X^\intercal Y$ can only possess an orthogonal basis if n is a multiple of l. In fact, this condition is not only necessary, but also sufficient, as can be seen by splitting an arbitrary orthogonal basis of $\mathbb{R}^n$ into $\frac{n}{l}$ blocks of size l. □

Typically one constructs bases using the Gram Schmidt process. We want to investigate how this could be generalized.

Lemma 29. Let $\mathcal{X}$ be an inner product A-module. Let $X, Y_1, ..., Y_n \in \mathcal{X}$. Let $Y_1, ..., Y_n$ be pairwise orthogonal and assume that for $1 \leq i \leq n$ there exists some $a_i \in A$ with $\langle Y_i, X\rangle = \langle Y_i, Y_i\rangle a_i$. Then

$$Z := X - \sum_{i=1}^{n} Y_i \cdot a_i$$

is orthogonal to $\{Y_1, ..., Y_n\}$ and we have

$$\operatorname{span}_A\{Y_1, ..., Y_n, X\} = \operatorname{span}_A\{Y_1, ..., Y_n, Z\}.$$

Proof. Let $1 \leq j \leq n$. Then we have

$$\begin{aligned} \langle Y_j, Z\rangle &= \langle Y_j, X\rangle - \langle Y_j, \sum_{i=1}^{n} Y_i \cdot a_i\rangle \\ &= \langle Y_j, X\rangle - \langle Y_j, Y_j\rangle a_j \\ &= 0. \end{aligned}$$

The second claim about the span follows directly from the definition of Z. □

Remark. If in the above theorem $\langle X_i, X_i\rangle$ is invertible for all i, the only possible choice for a_i is

$$a_i = \langle X_i, X_i\rangle^{-1}\langle X_i, Y\rangle.$$

Corollary 30. Let $\mathcal{X}$ be an inner product A-module and let $\mathcal{Y} \subset \mathcal{X}$ be a submodule of $\mathcal{X}$. Assume that $\mathcal{Y}$ possesses a finite orthogonal basis. Then for all $X \in \mathcal{X}$ there exists a unique $Y \in \mathcal{Y}$ with

$$X - Y \perp \mathcal{Y}.$$

We call Y the *orthogonal projection of X onto $\mathcal{Y}$*.

Proof. Let $\{Y_1, ..., Y_n\}$ be an orthogonal basis for $\mathcal{Y}$. We can then define

$$Y := \sum_{i=1}^{n} Y_i \langle Y_i, Y_i \rangle^{-1} \langle Y_i, X \rangle.$$

By lemma 29 we then have $X - Y \perp \mathcal{Y}$. If $\widetilde{Y} \in \mathcal{Y}$ with $X - \widetilde{Y} \perp \mathcal{Y}$, we have that

$$\begin{aligned} 0 &= \langle X - \widetilde{Y}, Y - \widetilde{Y} \rangle = \langle X - Y, Y - \widetilde{Y} \rangle + \langle Y - \widetilde{Y}, Y - \widetilde{Y} \rangle \\ &= \langle Y - \widetilde{Y}, Y - \widetilde{Y} \rangle. \end{aligned}$$

But this implies $Y - \widetilde{Y} = 0$, and hence the uniqueness of the orthogonal projection. □

Remark. (1) The theorem above remains true if instead of an orthogonal basis there is some pairwise orthogonal subset $\{Y_1, .., .Y_n\}$ spanning $\mathcal{Y}$, so that for all $X \in \mathcal{X}$ there exists $a_i \in A$ with $\langle Y_i, X \rangle = \langle Y_i, Y_i \rangle a_i$.

(2) Let $p_{\mathcal{Y}}$ be the map that associates to $X \in \mathcal{X}$ its orthogonal projection onto $\mathcal{Y}$. One immediately sees that p is A-linear and idempotent. We call this map the *orthogonal projection onto Y*.

Corollary 31 (Gram-Schmidt)**.** Let $\mathcal{X}$ be an inner product A-module. Let $X_1, ..., X_n \in \mathcal{X}$. We want to compute an orthogonal basis of $\text{span}_A\{X_1, ..., X_n\}$.

Define $Y_1 = X_1$. If for $1 \leq k < n$ the elements $Y_1, ..., Y_k$ have been defined and for all $1 \leq i \leq n$ we have that $\langle Y_i, Y_i \rangle$ is invertible, we define

$$Y_{k+1} := X_{k+1} - \sum_{i=1}^{k} Y_i \langle Y_i, Y_i \rangle^{-1} \langle Y_i, X_{k+1} \rangle$$

If $Y_{k+1} = 0$ we know that $X_{k+1} \in \text{span}_A\{Y_1, ..., Y_k\}$ and hence can be ignored for computing a basis. In this case we can delete X_{k+1} from the list $X_1, ..., X_n$ and continue normally with X_{k+2}.

If $Y_{k+1} \neq 0$ and $\langle Y_{k+1}, Y_{k+1} \rangle$ is not invertible we say that the algorithm *breaks down at step $k+1$*. We call this algorithm the *(generalized) Gram-Schmidt process*.

Assume that the algorithm does not break down up to step k. Then, after potentially deleting dependent elements, we have for all $1 \leq i \leq k$

$$\text{span}_A\{X_1, ..., X_i\} = \text{span}_A\{Y_1, ..., Y_i\},$$

and $\{Y_1, ..., Y_i\}$ is an orthogonal basis for $\text{span}_A\{X_1, ..., X_i\}$.

Proof. We iteratively apply lemma 29. The theorem then follows by a simple induction. □

We need to generalize a few more concepts.

Definition 32. Let $\mathcal{X}$ be an inner product A-module. Let $T : \mathcal{X} \to \mathcal{X}$ be an A-linear operator. We call T *self-adjoint* (or *hermitian*), if for all $X, Y \in \mathcal{X}$ we have

$$\langle T(X), Y \rangle = \langle X, T(Y) \rangle.$$

A self-adjoint opertor T is further called *positive definite* if for all $X \in \mathcal{X}$ we have

$$\langle T(X), X \rangle \geq 0,$$

and $\langle T(X), X \rangle = 0$ if and only if $X = 0$.

Remark. If T is a positive definite operator one immediately sees that

$$\langle X, Y \rangle_T := \langle T(X), Y \rangle$$

defines an inner product on $\mathcal{X}$ with respect to the given module structure.

4.1.3 The block conjugate gradient method

We will now finally derive the block conjugate gradient method.

In the following let $\mathcal{X}$ be an inner product A-module and let $T : \mathcal{X} \to \mathcal{X}$ be an invertible positive definite A-linear map. Let $Y \in \mathcal{X}$ and assume we want to solve the equation $T(X) = Y$. Let $X = T^{-1}(Y)$ be the exact solution and $R^0 = Y - T(X^0)$ the initial residual. Furthermore, we denote by $\langle \cdot, \cdot \rangle_T$ the inner product induced by T.

Our starting point is the definition of the conjugate gradient method via projection solutions. The natural generalization of the projection approach is as follows. Let $\mathcal{K}, \mathcal{L} \subset \mathcal{X}$ be submodules, called the *ansatz space* and *test space* respectively. Assume we have an initial guess $X^0 \in \mathcal{X}$. Then the corresponding *projection solution* $X_{\mathcal{K},\mathcal{L}}$ is given by

$$\begin{aligned} X_{\mathcal{K},\mathcal{L}} &\in X^0 + \mathcal{K} \\ Y - T(X_{\mathcal{K},\mathcal{L}}) &\perp \mathcal{L} \end{aligned}$$

Lemma 33. Let $\mathcal{L} = \mathcal{K}$. Assume that $\mathcal{K}$ possesses a finite orthogonal basis. Then there exists a unique projection solution.

Proof. Let $Z \in \mathcal{X}$. Then Z is a projection solution if and only if $Z - X^0$ is the orthogonal projection of $X - X^0$ onto $\mathcal{K}$ with respect to the $\langle \cdot, \cdot \rangle_T$ inner product. The statement of the lemma then follows from corollary 30. □

Since $T : X \to X$ is in particular linear, the Cayley-Hamilton theorem still implies that the inverse of T can be expressed as a polynomial in T. Thus we can still use the heuristic of choosing an ansatz space that contains the elements $\{R^0, T(R^0), ..., T^k(R^0)\}$. This motivates the following definition.

Definition 34. We define the k-th *block Krylov space* to be

$$\mathcal{K}_A^k(T, R^0) = \mathrm{span}_A\{R^0, T(R^0), ..., T^{k-1}(R^0)\}.$$

Our goal is now to compute projection solutions with respect to the choice $\mathcal{K} = \mathcal{L} = \mathcal{K}_A^k(T, R^0)$. Our derivation follows the one given in 4.1.1, with the difference that we do not have access to the same strong existence results as in the derivation discussed there.

Assume that we have $P^0, ..., P^{k-1} \in \mathcal{X}$ so that for all $1 \leq i \leq k$ the set $\{P^0, ..., P^{i-1}\}$ is an orthogonal basis of $\mathcal{K}_A^i(T, R^0)$. By lemma 33 there is a unique projection solution X^i with respect to $K_A^i(T, R^0)$ for all $1 \leq i \leq k$, and this X^i is given as X^0 plus the orthogonal projection of $X - X^0$ onto $K_A^i(T, R^0)$ with respect to $\langle \cdot, \cdot \rangle_T$, and by Corollary 30 we can compute this via

$$X^i = X^0 + \sum_{j=0}^{i-1} P^j \langle P^j, P^j \rangle_T^{-1} \langle P^j, X - X^0 \rangle_T.$$

But this implies for all $1 \leq i \leq k$

$$X^i = X^{i-1} + P^{i-1} \langle P^{i-1}, P^{i-1} \rangle_T^{-1} \langle P^{i-1}, X - X^0 \rangle_T.$$

Hence there exists some $a_{i-1} \in A$ so that we can compute X^i from X^{i-1} via a simple update

$$X^i = X^{i-1} + P^{i-1} \cdot a_{i-1}.$$

The question is how to construct $P^0, ..., P^{k-1}$ as described above. Since the only restriction on P^0 is that $\mathrm{span}_A\{P^0\} = \mathrm{span}_A\{R^0\}$ and that $\langle P^0, P^0 \rangle$ is invertible, we can simply set $P^0 = R^0$ if $\langle R^0, R^0 \rangle$ is invertible.

Let $R^k = Y - T(X^k)$ be the *residual* of X^k. By the same argument as in lemma 20 we see that $R^k \in \mathcal{K}_A^{k+1}(T, R^0)$, and by definition of the projection solution we have $R^k \perp \mathcal{K}_A^k(T, R^0)$. If $P^0, ..., P^{k-1}$ have already been constructed as desired, meaning in particular that they form an orthogonal basis of $\mathcal{K}_A^k(T, R^0)$, it is thus natural to extend this basis by applying the Gram Schmidt process to the vector R^k, meaning we can define

$$P^k = R^k - \sum_{j=0}^{k-1} P^j \langle P^j, P^j \rangle_T^{-1} \langle P^j, R^k \rangle_T.$$

By assumption we have $P^i \in \mathcal{K}_A^{i+1}(T, R^0)$. Since $T(\mathcal{K}_A^{i+1}(T, R^0)) \subset \mathcal{K}_A^{i+2}(T, R^0)$ we get for $i \leq k-2$

$$\langle P^i, R^k \rangle_T = \langle T(P^i), R^k \rangle = 0,$$

since $R^k \perp \mathcal{K}_A^k(T, R^0)$. Thus we can compute $P^k \perp \mathcal{K}_A^k(T, R^0)$ via

$$P^k = R^k - P^{k-1} \langle P^{k-1}, P^{k-1} \rangle_T^{-1} \langle P^{k-1}, R^k \rangle_T,$$

meaning we can compute P^k via a simple update $P^k = R^k + P^{k-1} b_{k-1}$, for some $b_{k-1} \in A$.

We thus know how to construct P^k so that it is orthogonal to $\{P^0, ..., P^{k-1}\}$. It is not immediately clear if the set $\{P^0, ..., P^k\}$ then actually spans the block Krylov space. In the next theorem we show that, given certain assumptions, this is indeed the case.

Theorem 35. Let $\mathcal{X}$ be an inner product A-module. Let $T : X \to X$ be an invertible, positive definite, A-linear map. Assume we want to solve the equation $T(X) = Y$, and we have some initial guess $X^0 \in \mathcal{X}$. Let $R^0 = Y - T(X^0)$ and $P^0 = R^0$.

We now iteratively construct R^k, P^k, X^k according to the following scheme.

1. If $\langle T(P^k), P^k \rangle$ or $\langle R^k, R^k \rangle$ is not invertible, the iteration breaks down.
2. Otherwise, define $a_k = \langle T(P^k), P^k \rangle^{-1} \cdot \langle R^k, R^k \rangle$ and
$$X^{k+1} = X^k + P^k \cdot a_k \tag{7}$$
$$R^{k+1} = R^k - T(P^k) \cdot a_k \tag{8}$$
3. We then define $b_k = \langle R^k, R^k \rangle^{-1} \cdot \langle R^{k+1}, R^{k+1} \rangle$, and further
$$P^{k+1} = R^{k+1} + P^k \cdot b_k. \tag{9}$$

If at some point $\langle T(P^k), P^k \rangle$ or $\langle R^k, R^k \rangle$ is not invertible we say that the iteration *breaks down at step* k.

Now assume that the iteration does not break down up to step k. Then we have the following.

1. For all $0 \leq i \leq k$ we have that
$$\mathcal{K}_A^{i+1}(T, R^0) = \text{span}_A\{R^0, ..., R^i\} = \text{span}_A\{P^0, ..., P^i\}, \tag{10}$$
and $\{P^0, ..., P^i\}$ forms an $\langle \cdot, \cdot \rangle_T$-orthogonal basis of $\mathcal{K}_A^{i+1}(T, R^0)$.
2. R^i is the residual associated to X^i, meaning $R^i = Y - T(X^i)$ for all $0 \leq i \leq k$.
3. For all $1 \leq i \leq k$ the element X^i is the unique projection solution associated to $\mathcal{K}_A^i(T, R^0)$. More concretely, we have
$$X^i \in X^0 + \mathcal{K}_A^i(T, R^0)$$
$$Y - T(X^i) \perp \mathcal{K}_A^i(T, R^0)$$
4. For all $0 \leq i \leq k$ we have
$$\langle P^i, P^i \rangle_T = \langle P^i, R^i \rangle_T = \langle R^i, P^i \rangle_T \tag{11}$$
$$\langle R^i, R^i \rangle = \langle P^i, R^i \rangle = \langle R^i, P^i \rangle \tag{12}$$

Proof. We first proof the claims by induction on k. For $k = 0$ statements 1), 2) and 4) are trivial and for 3) there is nothing to show. So assume they are true for some k. We first show (10). By (9) and the induction hypotheses, we have

$$\text{span}_A\{P^0, ..., P^{k+1}\} = \text{span}\{R^0, ..., R^{k+1}\}.$$

Furthermore, since $T(\mathcal{K}_A^{k+1}(T, R^0)) \subset \mathcal{K}_A^{k+2}(T, R^0)$, we have by the induction hypotheses that $T(P^k) \in \mathcal{K}_A^{k+2}(T, R^0)$, and hence by (8) we have

$$\text{span}_A\{R^0, ..., R^{k+1}\} \subset \mathcal{K}_A^{k+2}(T, R^0).$$

To show 1) for $k+1$ it thus suffices to show that $T^{k+1}(R^0) \in \text{span}_A\{R^0, ..., R^{k+1}\}$. To see this, note that $T^k(R^0) \in \text{span}_A\{P^0, ..., P^k\}$, meaning we have

$$T^k(R^0) = \sum_{j=0}^{k} P^j c_j,$$

for some $c_j \in A$. This gives

$$T^{k+1}(R^0) = T(P^k)c_k + \sum_{j=0}^{k-1} T(P^j)c_j.$$

For $j \leq k-1$ we know that

$$\begin{aligned} T(P^j) &\subset \mathcal{K}_A^{j+2}(T, R^0) \subset \mathcal{K}_A^{k+1}(T, R^0) = \operatorname{span}_A\{R^0, ..., R^k\} \\ &\subset \operatorname{span}_A\{R^0, ..., R^{k+1}\}. \end{aligned}$$

It thus suffices to show that $T(P^k) \in \operatorname{span}_A\{R^0, ..., R^{k+1}\}$. Since the iteration does not break down at step k, we know that $a_k = \langle T(P^k), P^k\rangle^{-1} \cdot \langle R^k, R^k\rangle$ is invertible, and hence we have by (8)

$$T(P^k) = R^k \cdot a_k^{-1} - R^{k+1} \cdot a_k^{-1} \in \operatorname{span}_A\{R^0, ..., R^{k+1}\}.$$

We have thus proven (10) for $k+1$.

Furthermore, we have

$$\begin{aligned} R^{k+1} &= R^k - T(P^k) \cdot a_k = Y - T(X^k) - T(P^k) \cdot a_k \\ &= Y - T(X^k + P^k \cdot a_k) = Y - T(X^{k+1}). \end{aligned}$$

Hence R^{k+1} is the actual residual associated to X^{k+1}, which was the claim of 2).

We now show 3). By our induction hypothesis we know that the set $\{P^0, ..., P^k\}$ has the property that for all $0 \leq i \leq k$ the set $\{P^0, ..., P^i\}$ is an orthonormal basis for $\mathcal{K}_A^{i+1}(T, R^0)$. Then by lemma 33 there exists a unique projection solution $Z^{k+1} \in X^0 + \mathcal{K}_A^{k+1}(T, R^0)$, and by the computations preceding this theorem and the induction hypothesis, we know that we can compute Z^{k+1} via an update

$$Z^{k+1} = X^k + P^k\lambda^k,$$

for some $\lambda^k \in A$. Since Z^{k+1} is a projection solution, we know that

$$0 = \langle R^k, Y - T(Z^{k+1})\rangle = \langle R^k, R^k\rangle - \langle R^k, T(P^k)\rangle\lambda^k.$$

By our induction hypothesis for claim 4) we know that $\langle R^k, T(P^k)\rangle = \langle P^k, T(P^k)\rangle$, and this gives

$$\lambda^k = \langle T(P^k), P^k\rangle^{-1}\langle R^k, R^k\rangle = a_k,$$

and hence we see that $X^{k+1} = Y^{k+1}$ is the unique projection solution corresponding to $\mathcal{K}_A^{k+1}(T, R^0)$.

The considerations just before this theorem have shown that for R^{k+1} the residual of the projection solution X^{k+1}, we can compute a vector Q^{k+1} with $Q^{k+1} \perp_T \{P^0, ..., P^k\}$ via an update

$$Q^{k+1} = R^{k+1} + P^k \cdot \mu^k$$

for some $\mu^k \in A$. This gives

$$\begin{aligned} 0 &= \langle P^k, R^{k+1} + P^k\mu^k\rangle_T = \langle T(P^k), R^{k+1} + P^k\mu^k\rangle \\ &= \langle \left(R^k - R^{k+1}\right) \cdot a_k^{-1}, R^{k+1} + P^k\mu^k\rangle \\ &= -\left(a_k^{-1}\right)^* \left(\langle R^{k+1}, R^{k+1}\rangle + \langle R^k, P^k\rangle\mu^k\right), \end{aligned}$$

and since by induction hypothesis we have $\langle R^k, P^k\rangle = \langle R^k, R^k\rangle$ we get $\mu^k = \langle R^k, R^k\rangle^{-1}\langle R^{k+1}, R^{k+1}\rangle$. This shows that $P^{k+1} = Q^{k+1} \perp_T \{P^0, ..., P^k\}$, and hence since the iteration does not break down until step $k+1$ we have shown that $\{P^0, ..., P^{k+1}\}$ is an orthogonal basis for $\mathcal{K}_A^{k+2}(T, R^0)$. This shows that 1) is true for $k+1$.

It only remains to show 4). Using (9) we can compute

$$\langle P^{k+1}, P^{k+1}\rangle_T = \langle P^{k+1}, R^{k+1} + P^k b_k\rangle_T = \langle P^{k+1}, R^{k+1}\rangle_T,$$

Furthermore we can compute

$$\langle R^{k+1}, R^{k+1}\rangle = \langle R^{k+1}, P^{k+1} - P^k \cdot b_k\rangle = \langle R^{k+1}, P^{k+1}\rangle.$$

Taking the adjoints of both of these equations we see that 4) is true for $k+1$

This proves the theorem. □

Algorithm 2 Block Conjugate Gradient Method

Require: - Inner product module $\mathcal{X}$ over involutive algebra A
- Invertible positive definite $T : \mathcal{X} \to \mathcal{X}$
- Right hand side $Y \in \mathcal{X}$
- Initial guess $X^0 \in \mathcal{X}$

$R^0 \leftarrow Y - T(X^0)$
$P_0 \leftarrow R_0$
for $k = 0, 1, 2, ...$ until convergence **do**
$\alpha^k \leftarrow \langle T(P^k), P^k \rangle^{-1} \cdot \langle R^k, R^k \rangle$
$X^{k+1} \leftarrow X^k + P^k \cdot \alpha^k$
$R^{k+1} \leftarrow R^k - T(P^k) \cdot \alpha^k$
$\beta^k \leftarrow \langle R^k, R^k \rangle^{-1} \cdot \langle R^{k+1}, R^{k+1} \rangle$
$P^{k+1} \leftarrow R^{k+1} + P^k \cdot \beta^k$
end for

We have thus derived an algorithm to compute the projection solutions with respect to the block Krylov subspaces. We summarize the result of our derivation in algorithm 2.

If at some point one of the elements $\langle T(P^k), P^k \rangle$ or $\langle R^k, R^k \rangle$ is not invertible, the algorithm cannot proceed, at least in its current form. We call this a *breakdown*. In certain special cases it is still possible to continue the algorithm to produce further projection solutions. There are multiple approaches to try to overcome breakdowns. One class of these are the *deflation methods*, which we will study later on.

We have now generalized the conjugate gradient method and have arrived at an algorithm that computes approximations to equations of the form $T(X) = Y$ in inner product modules over involutive algebras. Due to historical reasons, and since we are mainly interested in applying this algorithm to $\mathcal{X} = \mathbb{K}^{n \times k}$ endowed with some suitable inner product module structure, we call this algorithm the *block conjugate gradient method*, or *BCG* for short.

We have derived the algorithm by enforcing an algebraic Galerkin condition. It is now natural to ask if the projection solutions defined by this algebraic condition are close to the exact solution in a topological sense. This is what we want to study in the next subsection.

4.2 Optimality results

Assume we have some Hilbert space H and a positive definite operator $T : H \to H$. Further assume we want to solve a linear system $T(x) = y$ and have an initial guess x^0. Let $x^1, x^2, ...$ be the iterates produced by the conjugate gradient algorithm. By theorem 19, we now that x^k is the best approximation in the ansatz space to the exact solution in the $\| \cdot \|_T$-norm, meaning

$$x^k = \underset{z \in x^0 + \mathcal{K}^k(T, r^0)}{\arg\min} \|x - z\|_T.$$

In this sense, we can thus say that x^k is the optimal approximation to the exact solution we can produce inside the ansatz space $x^0 + \mathcal{K}^k(T, r^0)$.

We now want to investigate if we can derive a similar optimality property for the block conjugate gradient algorithm. We first introduce a few concepts.

Definition 36. Let A be an involutive algebra. A linear functional $f : A \to \mathbb{K}$ is called *positive*, if for all $a \in A$ we have

$$a \geq 0 \implies f(a) \geq 0.$$

Furthermore, we say that a positive functional f is *faithful*, if for all $a \in A$ with $a \geq 0$ we have

$$f(a) = 0 \implies a = 0.$$

We can now state the main result of this subsection.

Theorem 37. Let $\mathcal{X}$ be an inner product A-module. Assume there exists a faithful positive functional f on A with $f(a^*) = \overline{f(a)}$ for all $a \in A$. Let $T : \mathcal{X} \to \mathcal{X}$ be a positive definite operator. Then there exists an inner product $\ll \cdot, \cdot \gg : \mathcal{X} \to \mathbb{K}$, so that the induced norm $\|x\|_{\text{ind}} := \sqrt{\ll x, x \gg}$ has the following property.

For any equation $T(X) = Y$ with initial guess X^0, let X^k be a projection solution with respect to $\mathcal{K}_A^k(T, R^0)$, meaning X^k fulfills

$$X^k \in X^0 + \mathcal{K}_A^k(T, R^0)$$
$$Y - T(X^k) \perp \mathcal{K}_A^k(T, R^0)$$

Then X^k is an optimal approximation to the exact solution with respect to $\|\cdot\|_{\text{ind}}$, meaning

$$X^k = \underset{Z \in X^0 + \mathcal{K}_A^k(T,R^0)}{\arg\min} \|X - Z\|_{\text{ind}}.$$

The inner product can be defined via

$$\ll X, Y \gg = f(\langle T(X), Y\rangle),$$

meaing the norm is given by $\|X\|_{\text{ind}} = \sqrt{f(\langle T(X), X\rangle)}$.

We first prove a useful lemma.

Lemma 38. Let $\mathcal{X}$ be an inner product module over A. Let f be a faithful positive functional on A with $f(a^*) = \overline{f(a)}$ for all $a \in A$. Then we can define

$$\widetilde{\ll \cdot, \cdot \gg} : X \times X \to \mathbb{K}; (x, y) \mapsto f(\langle x, y\rangle).$$

Then $\widetilde{\ll \cdot, \cdot \gg}$ is an inner product on $\mathcal{X}$. If $T : \mathcal{X} \to \mathcal{X}$ is positive definite with respect to $\langle \cdot, \cdot \rangle$, then it is also positive definite with respect to $\widetilde{\ll \cdot, \cdot \gg}$.

Proof. Since $\langle \cdot, \cdot \rangle$ is linear in the second component, the same is true for the map $\widetilde{\ll \cdot, \cdot \gg}$. Since $\overline{f(a)} = f(a^*)$ it follows that $\widetilde{\ll \cdot, \cdot \gg}$ is hermitian (resp. symmetric). Furthermore, we have for $X \in \mathcal{X}$ that $\ll \widetilde{X, X} \gg = f(\langle X, X\rangle) \geq 0$, and $\ll \widetilde{X, X} \gg = 0$ implies $\langle X, X\rangle = 0$, since f is faithful, and this implies $X = 0$. This shows that $\widetilde{\ll \cdot, \cdot \gg}$ defines an inner product. Furthermore, T is in particular linear, and we have for $X, Y \in \mathcal{X}$

$$\ll \widetilde{T(X), Y} \gg = f(\langle T(X), Y\rangle) = f(\langle X, T(Y)\rangle) = \ll \widetilde{X, T(Y)} \gg.$$

Hence T is self adjoint with respect to $\widetilde{\ll \cdot, \cdot \gg}$. Furthermore we have

$$\ll T(X), X \gg = f(\langle T(X), X\rangle) \geq 0,$$

and $\ll \widetilde{T(X), X} \gg = 0$ implies $\langle T(X), X\rangle = 0$, since f is faithful, and since T is positive definite with respect to $\langle \cdot, \cdot \rangle$ this implies $X = 0$. Hence T is positive definite with respect to $\widetilde{\ll \cdot, \cdot \gg}$. □

Proof of Theorem 37. We use the same notation as in the theorem. According to lemma 38 we can define an inner product via $\ll \widetilde{X, Y} \gg = f(\langle X, Y\rangle)$, and T is positive definite with respect to this inner product. Hence

$$\ll X, Y \gg = \ll \widetilde{T(X), Y} \gg = f(\langle T(X), Y\rangle)$$

also defines an inner product on $\mathcal{X}$ and hence a norm $\|X\|_{\text{ind}} = \sqrt{\ll X, X \gg}$. But then

$$X^k \in X^0 + \mathcal{K}_A^k(T, R^0)$$
$$Y - T(X^k) \perp_{\langle \cdot, \cdot \rangle} \mathcal{K}_A^k(T, R^0)$$

implies that

$$X^k \in X^0 + \mathcal{K}_A^k(T, R^0)$$
$$Y - T(X^k) \perp_{\widetilde{\ll \cdot, \cdot \gg}} \mathcal{K}_A^k(T, R^0)$$

and the same argument as in theorem 19 shows that we then have

$$X^k = \underset{Z \in X^0 + \mathcal{K}_A^k(T,R^0)}{\arg\min} \|X - Z\|_{\text{ind}}.$$

□

We have seen that the block conjugate gradient method produces optimal approximations to the exact solution in a certain norm, given that we can define a faithful positive functional on A with $f(a^*) = \overline{f(a)}$ for all $a \in A$. We want to briefly discuss how such functionals might look. We first take a look at a special case.

Lemma 39. Let $A = \mathbb{K}^{k\times k}$ be the involutive algebra of $k \times k$-matrices. Then the trace

$$\mathrm{Tr} : \mathbb{K}^{k\times k}; (m_{i,j})_{1\leq i,j\leq n} \mapsto \sum_{i=1}^{n} m_{i,i}$$

is a faithful positive functional with $\mathrm{Tr}(a^*) = \overline{\mathrm{Tr}(a)}$.

Proof. The trace is obviously linear, and it is easily checked that $\mathrm{Tr}(a^*) = \overline{\mathrm{Tr}(a)}$. It is well known and otherwise quickly computed that for matrices a, b we have $\mathrm{Tr}(ab) = \mathrm{Tr}(ba)$. Now every positive element $a^*a \in \mathbb{K}^{k\times k}$ is by the spectral theorem unitarily equivalent to a diagonal matrix, and since we have $\mathrm{Tr}(ab) = \mathrm{Tr}(ba)$, we see that $\mathrm{Tr}(a^*a)$ is simply the sum of the eigenvalues of a^*a. Since a^*a is symmetric positive semidefinite, it has nonnegative eigenvalues, implying $\mathrm{Tr}(a^*a) \geq 0$. Furthermore, $\mathrm{Tr}(a^*a) = 0$ implies that all eigenvalues of a^*a are zero, hence a^*a is unitarily equivalent to the zero matrix and hence zero itself, meaning $a^*a = 0$. Hence Tr is positive and faithful. □

If $A_1, ..., A_n$ are involutive algebras and $f_1, ..., f_n$ are faithful positive functionals with $f_i(a_i^*) = \overline{f_i(a_i)}$, we can define a faithful positive functional

$$\bigoplus_{i=1}^{n} f_i : \bigoplus_{i=1}^{n} A_i \to \mathbb{K}; (a_1, ..., a_n) \mapsto \sum_{i=1}^{n} f_i(a_i),$$

and we obviously also have $\bigoplus_{i=1}^{n} f_i(a^*) = \overline{\bigoplus_{i=1}^{n} f_i(a)}$.

This consideration together with lemma 39 implies that the block conjugate gradient algorithm applied to all of the examples discussed in 27 produces "optimal" iterates with respect to suitable norms, as all algebras discussed there can be constructed as direct sums of matrix algebras.

4.3 Convergence analysis

We have seen in theorem 37 that, for involutive algebras A with a faithful positive functional $f : A \to \mathbb{K}$ so that $f(a^*) = \overline{f(a)}$, we can define a norm on the module $\mathcal{X}$ so that the block conjugate gradient method produces optimal approximations with respect to this norm in the translated block Krylov spaces. In the following, we will call such algebras A *admissable*. We are interested in statements about the convergence rate of the block conjugate gradient algorithm.

In line with our previous approach we first take a look at the ordinary conjugate gradient method. Let $T : H \to H$ be a positive definite opertor and assume we want to solve the equation $T(x) = y$ using the conjugate gradient method. Let $x^0, ..., x^k, ...$ be the iterates produced by the conjugate gradient method. Let $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ be the *condition* of the operator T, where $\lambda_{\max}$ is the largest eigenvalue and $\lambda_{\min}$ is the smallest eigenvalue of T. Let $e^k = x - x^k$ be the error in the k-th iteration. Then a standard result about the conjugate gradient method is that we have

$$\|e^k\|_T \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)\|e^0\|_T.$$

This result will be a special case of a theorem we will prove later, and hence we will not give a proof here. But we want to use this theorem as a template to derive convergence results for the block conjugate gradient method. This presentation is based on the discussion of the convergence properties of the BCG method given in [43].

In the following, let $\mathcal{X}$ be an inner product module over an admissable algebra A. Let $T : \mathcal{X} \to \mathcal{X}$ be invertible positive definite with respect to $\langle\cdot,\cdot\rangle$. Let $\ll \cdot,\cdot \gg$ be the inner product inducing the norm with respect to which the BCG method produces optimal iterates, as it was constructed in theorem 37. Denote this norm by $\|\cdot\|_{\mathrm{ind}}$.

Lemma 40. The operator $T : \mathcal{X} \to \mathcal{X}$ is positive definite with respect to $\ll \cdot,\cdot \gg$.

Proof. Since T is A-linear, it is in particular linear. Furthermore, we have for $X, Y \in \mathcal{X}$

$$\ll T(X), Y \gg = f(\langle T^2(X), Y\rangle) = f(\langle T(X), T(Y)\rangle) = \ll X, T(Y) \gg,$$

and hence T is self adjoint with respect to $\ll \cdot, \cdot \gg$. In addition, we have

$$\ll T(X), X \gg = f(\langle T(X), T(X) \rangle) \geq 0,$$

and $\ll T(X), X \gg = 0$ implies $\langle T(X), T(X) \rangle = 0$ since f is faithful, and by definition of an inner product module this implies $T(X) = 0$, and since T is invertible this implies $X = 0$. □

Before we prove the following theorem, let us take a look at a linear operator L. Let v be an eigenvector with eigenvalue λ with respect to L, meaning we have $L(v) = \lambda v$. If p is then a polynomial, we have

$$p(L)v = p(\lambda)v.$$

We use this in the following theorem.

Theorem 41. Assume the algebra A contains an identity and that $\mathcal{X}$ is finite dimensional. Let $\lambda_1, ..., \lambda_m$ be the eigenvalues of T viewed as a linear operator $T : \mathcal{X} \to \mathcal{X}$. Assume we want to solve the equation $T(X) = Y$ using the BCG method, and that the BCG method does not break down until the k-th step. Let $X^0, X^1, ...$ be the iterates produced by the BCG method, and let $E^i = X - X^i$ be the error in the k-th step. Let P_k be the space of polynomials of degree at most k. Then we have

$$\|E^k\|_{\text{ind}} \leq \left(\inf_{p \in P^k, p(0)=1} \left(\max_{i=1,...,m} |p(\lambda_i)| \right) \right) \|E^0\|_{\text{ind}}.$$

Proof. Since T is positive definite with respect to $\ll \cdot, \cdot \gg$, the spectral theorem implies that there is an orthonormal basis of eigenvectors of T with respect to this inner product. Let $U_1, ..., U_l$ denote this orthonormal basis. Then we have for all polynomials $q = \sum_{j=0}^{k-1} b_j x^j \in P_{k-1}$

$$\begin{aligned}
\|E^k\|_{\text{ind}} = \|X - X^k\|_{\text{ind}} &= \underset{X^0 + \mathcal{K}_A^k(T, R^0)}{\arg\min} \|X - Z\|_{\text{ind}} \\
&\leq \|X - (X^0 + \sum_{j=0}^{k-1} T^j(R^0) \cdot (b_j 1_A))\|_{\text{ind}} = \|E^0 - \sum_{j=0}^{k-1} b_j T^j(R^0)\|_{\text{ind}} \\
&= \|E^0 - \sum_{j=0}^{k-1} b_j T^{j+1}(E^0)\|_{\text{ind}} = \| \left(\text{id} - \sum_{j=1}^{k} b_{j-1} T^j \right) E^0 \|_{\text{ind}} \\
&= \|p(T)E^0\|_{\text{ind}},
\end{aligned}$$

where $p(T)$ is a polynomial of degree k with $p(0) = 1$. We can now write the initial error E^0 in the orthonormal basis, meaning we have $E^0 = \sum_{j=1}^{l} \mu_j U_j$. Let $\lambda_{\sigma(j)}$ be the eigenvalue corresponding to the eigenvector U_j. The theorem of Pythagoras then implies

$$\begin{aligned}
\|p(T)E^0\|_{\text{ind}}^2 = \| \sum_{j=1}^{l} p(\lambda_{\sigma(j)}) \mu_j U_j \|_{\text{ind}}^2 &= \sum_{j=1}^{l} |p(\lambda_{\sigma(j)})|^2 \mu_j^2 \\
&\leq \left(\max_{i=1,...,m} |p(\lambda_i)|^2 \right) \sum_{j=1}^{l} \mu_j^2 = \left(\max_{i=1,...,m} |p(\lambda_i)|^2 \right) \|E^0\|_{\text{ind}}^2,
\end{aligned}$$

and hence

$$\|E^k\|_{\text{ind}} \leq \left(\max_{i=1,...,m} |p(\lambda_{\sigma(i)})| \right) \|E^0\|_{\text{ind}}. \tag{13}$$

If p is an arbitrary polynomial of degree k with $p(0) = 1$, there is a polynomial q of degree $k - 1$ so that

$$p = 1 - q \cdot x,$$

where x is the monomial of degree 1. Hence the upper bound (13) is valid for all such polynomials p. This implies the claim of the theorem. □

Hence we see that the eigenvalue distribution of the operator T plays a role in studying the convergence rate of the block conjugate gradient method. In particular, if the spectrum of the operator T consists of few tightly packed clusters of eigenvalues we can expect the method to converge fast.

For the following theorem we need a few facts about the *Chebyshev polynomials* T_k. We have assembled the necessary statements in appendix A. The question we want to study is how small the expression

$$\inf_{p\in P^k, p(0)=1} \left(\max_{i=1,\dots,m} |p(\lambda_i)| \right)$$

is guaranteed to be. Note that since T is positive definite, all eigenvalues are contained inside the interval $(0,\infty)$. Let $\lambda_{\max}$ denote the maximal eigenvalue and $\lambda_{\min}$ the minimal eigenvalue of T, and $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ the condition. If $\lambda_{\min} = \lambda_{\max}$ the operator T must be a multiple of the identity, and the block conjugate gradient algorithm converges in the first step. Hence in the following we assume $\lambda_{\min} < \lambda_{\max}$.

Lemma 42. We have

$$\inf_{p\in P^k, p(0)=1} \left(\max_{i=1,\dots,m} |p(\lambda_i)| \right) \leq \frac{1}{T_k\left(\frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{min}}\right)} \leq \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k.$$

Proof. Since $\max_{i=1,\dots,m} |p(\lambda_i)|$ is bounded above by the supremum norm of p on the interval $[\lambda_{\min}, \lambda_{\max}]$, the first inequality is a direct consequence of theorem 66. For the second inequality is then a direct application of corollary 62. □

We can summarize this discussion in the following theorem.

Theorem 43. We use the same assumptions and notations as in theorem 41 and lemma 42. Then we have

$$\|E^k\|_{\text{ind}} \leq \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|E^0\|_{\text{ind}}.$$

Taking a close look at the proof of theorem 41, we see that this bound is based entirely on approximations inside the subspace

$$X^0 + \mathcal{K}^k_{\mathbb{K}}(T, R^0) \subset X^0 + \mathcal{K}^k_A(T, R^0).$$

Hence, depending on the structure of the algebra A, we can expect the approximations produced by the block conjugate gradient method to be even better than what theorem 41 or theorem 43 guarantees.

We want to take a closer look at some choices for A and try to improve the generic estimates given above in these cases.

Our motivation for introducing the BCG method was our interest in solving systems $AX = B$, where A is a positive definite $n \times n$-matrix and X, B are $n \times k$-matrices. As described in example 27 (3) we can endow the set of $n \times k$-matrices with entries from $\mathbb{K}$ with the structure of an inner product module over the $k \times k$-matrices $\mathbb{K}^{k\times k}$ via standard matrix multiplication and inner product defined by

$$\langle X, Y\rangle = X^*Y,$$

where X^* is the conjugate transpose of X. Note that

$$T_A : \mathbb{K}^{n\times k} \to \mathbb{K}^{n\times k}; X \mapsto A \cdot X,$$

defines an $\mathbb{K}^{k\times k}$-linear map. For $X, Y \in \mathbb{K}^{n\times k}$ we have

$$\langle T_A(X), Y\rangle = (A \cdot X)^* \cdot Y = X^* \cdot A \cdot Y = \langle X, T_A(Y)\rangle,$$

and hence T_A is self-adjoint. Furthermore

$$\langle T_A(X), X\rangle = X^*AX \geq 0 \in \mathbb{K}^{k\times k},$$

which follows exactly as in the proof of example 27 (3). Notice that the entries of X^*AX are simply the inner products of the columns of X with respect to the inner product on $\mathbb{K}^n$ induced by A. Hence $\langle T_A(X), X\rangle = 0$ implies $X = 0$. Since the matrix A is invertible, the map T_A is invertible with inverse $X \mapsto A^{-1} \cdot X$. Hence T_A is invertible positive definite with respect to $\langle\cdot,\cdot\rangle$. This means we can apply the BCG method to solve equations of the form $AX = B$ using the $k \times k$-matrices as an algebra. The corresponding algorithm is called the *classical block conjugate gradient algorithm*, as this is the algorithm O'Leary originally published in 1980 in [35].

We proved in lemma 39 that $\mathbb{K}^{k\times k}$ is an admissable algebra, and the trace is a corresponding positive faithful functional. Then by theorem 37, we see that if X^k is the iterate produced by the k-th step of the conjugate gradient algorithm that X^k is the best approximation to the exact solution in the norm

$$\|Z\|^2 = \mathrm{Tr}(Z^*AZ) = \sum_{j=1}^{k} \|Z_j\|_A^2, \tag{14}$$

where Z_j is the j-th column of Z and $\|\cdot\|_A$ is the norm on $\mathbb{K}^n$ induced by A. In the following, if X is a matrix we always denote by X_j its j-th column.

In the following let X be the exact solution of the equation $AX = B$ we want to solve. Furthermore, let $p_j : \mathbb{K}^{n\times k} \to \mathbb{K}^n$ denote the projection onto the j-th column, meaning $Z = (p_1(Z), ..., p_k(Z))$. Taking a close look at the definition we see that for

$$\mathcal{K}(m, A, R^0) := \sum_{j=1}^{k} \mathcal{K}_{\mathbb{K}}^m(A, R_j^0) \subset \mathbb{K}^n$$

we have

$$\mathcal{K}_{\mathbb{K}^{k\times k}}^m(T_A, R^0) = \mathcal{K}(m, A, R^0)^k,$$

where the power on the right hand side means the cartesian product. Let X^0 be an initial guess and let X^m be the iterate produced by the classical BCG method in the m-th iteration. Then (14) implies that X_j^m is the best approximation to X_j in the norm $\|\cdot\|_A$ inside the space

$$p_j(X^0 + \mathcal{K}_{\mathbb{K}^{k\times k}}^m(T_A, R^0)) = X_j^0 + \mathcal{K}(m, A, R^0).$$

We summarize this discussion.

Lemma 44. Assume that the classical BCG method does not break down up to step m. Let X^m be the iterate produced by the classical BCG method. Then for all $1 \le j \le k$ we have that

$$\begin{aligned}\|X_j - X_j^m\|_A &= \min_{z \in X_j^0 + \mathcal{K}(m,T,R^0)} \|X_j - z\|_A \\ &= \min_{q_1,...,q_k \in P_{m-1}} \|X_j - X_j^0 - \sum_{l=1}^{k} q_l(A)(R_l^0)\|_A\end{aligned}$$

We thus see that for the classical BCG method, the m-th iterate does not only minimize the error in the j-th component over the classical Krylov space for the residual R_j^0, but instead over the sum of the Krylov spaces for all residuals.

We want to show that, given certain conditions, we can use this enriched optimization space to prove a version of theorem 43 that is less dependent on the distribution of the eigenvalues of the operator. We first study under what conditions the classical BCG algorithm breaks down. We remind the reader that we assume $k < n$.

Lemma 45. The classical BCG method breaks down at step m if and only if the m-th residual $R^m = Y - X^m \in \mathbb{K}^{n\times k}$ does not have full rank.

Proof. The algorithm breaks down at step m if $\langle R^m, R^m\rangle$ or $\langle T_A(P^m), P^m\rangle$ are not invertible. Since $\langle R^m, R^m\rangle$ is square, it is invertible if and only if it is surjective. By standard linear algebra, we have

$$\mathrm{rank}(R^m) = \mathrm{rank}((R^m)^* R^m) = \mathrm{rank}(\langle R^m, R^m\rangle),$$

and hence R^m has full rank if and only if $\langle R^m, R^m\rangle$ is invertible. Since A is positive definite the same argument shows that P^m has full rank if and only if $\langle T_A(P^m), P^m\rangle$ is invertible. By the rank-nullity theorem it thus suffices to show that if R^m has trivial kernel, then so does P^m. But by theorem 35 (4) we have $(R^m)^* P^m = (R^m)^* R^m$, and hence $P^m v = 0$ implies $(R^m)^* R^m v = 0$ and hence $R^m v = 0$. □

Hence we see that if the algorithm breaks down at step m, then there is some vector $0 \neq v \in \mathbb{K}^n$ with $0 = R^m v = Y \cdot v - A \cdot X^m \cdot v$, which means that we have $A \cdot (X^m v) = Yv$. Thus the algorithm breaks down if and only if we have solved the linear system $Ax = b$ where b is a non trivial linear combination of the block right hand side B.

We now want to discuss how to improve the bound given in theorem 43. On the way to prove that theorem, we constructed polynomials in such a way that they were small on the interval $[\lambda_{\min}, \lambda_{\max}]$. But looking at

lemma 44, we now do not only have access to polynomials in the residual of the initial approximation, but also to polynomials in all other residuals. We thus have $k-1$ extra "degrees of freedom". The idea is to use these degrees of freedom to eliminate the influence of $k-1$ eigenvalues in the derivation of the bound, thus giving a bound that does not depend on the whole spectrum, but only on a part of the spectrum, and using this smaller part we can then derive a tighter bound. We need a few conditions for this to work. We will discuss them first.

Definition 46. Let A be a positive definite $n \times n$-matrix. Let $[a,b] \subset (0,\infty)$ be an interval and let $R \in \mathbb{K}^{n\times l}$ with $l < n$. Let $0 < \lambda_1 \leq ... \leq \lambda_n$ be the eigenvalues of A. We then call the interval $[a,b]$ *admissable with respect to* R, if we have

1. Counting multiplicities, at most l eigenvalues of A are not contained inside the interval $[a,b]$.
2. If U is the direct sum of the eigenspaces of the eigenvalues not contained in the interval $[a,b]$, then the orthogonal projection onto U
$$p_U : \text{span}\{R_1, ..., R_l\} \to U$$
is surjective.[19]

We then define the *spectral spread of A with respect to R* as

$$\kappa(A,R) = \inf\{\frac{b}{a} \mid [a,b] \text{ is admissable with respect to } R\}$$

Remark. (1) Let $\lambda_{\max}$ denote the maximal eigenvalue of A and $\lambda_{\min}$ the minimal eigenvalue of A. Then $[\lambda_{\min}, \lambda_{\max}]$ is always an admissable interval of A with respect to R. Hence we always have

$$\kappa(A,R) \leq \kappa = \frac{\lambda_{\max}}{\lambda_{\min}},$$

where κ denotes the condition of A.

(2) An obvious candidate is the interval $[\lambda_{l+1}, \lambda_n]$. Then condition 1. is true, and if additionally 3. is true we get

$$\kappa(A,R) \leq \frac{\lambda_n}{\lambda_{l+1}}$$

(3) We generally expect condition 3. to be true. This heuristic is inspired by the easily verified fact that, given some space U with $\dim(U) \leq l$, the set of all matrices $R \in \mathbb{K}^{n\times l}$ for which 3. fails is nowhere dense in the set of all $n \times l$ matrices.

(4) There exists an admissable interval $[a,b]$ with $\kappa(A,R) = \frac{b}{a}$. This follows from the fact that if we have an admissable interval $[a,b]$ and λ_i is the smallest eigenvalue of A at least as large as a, and λ_j is the largest eigenvalue of A that is at most as large as b, then the interval $[\lambda_i, \lambda_j]$ is also an admissable interval leading to a bound as least as good as $\frac{b}{a}$. Hence we only need to consider a finite set of candidates for the computation of $\kappa(A,R)$.

We can now finally prove the desired theorem. The proof is based on the argument given in [35], theorem 5. In the following, if R is a $n \times k$-matrix, we denote by $R(j)$ the $n \times (k-1)$-matrix obtained by deleting the j-th column of R.

Theorem 47. Assume we want to solve the equation $AX = B$, where A is positive definite, using the classical BCG method. Let B be a $n \times k$-matrix. Denote by X^m the iterates produced by the BCG method, and let $R^m = B - AX^m$ denote their residuals and $E^m = X - X^m$ the error in the m-th iteration. Let $1 \leq j \leq k$. Then there is a constant c so that for all $m \geq 0$ for which the iterates are defined we have

$$\|E_j^m\|_A \leq c \left(\frac{\sqrt{\kappa(A, R^0(j))} - 1}{\sqrt{\kappa(A, R^0(j))} + 1} \right)^m.$$

Proof. Let $[a,b]$ be an admissable interval with $\kappa(A, R^0(j)) = \frac{b}{a}$. Let $\lambda_1, ..., \lambda_s$ be the eigenvalues of A not contained inside the interval $[a,b]$, repeated according to multiplicity. Let $u_1, ..., u_s$ be orthonormal eigenvectors of A to the eigenvalues $\lambda_1, ..., \lambda_s$. Let $\lambda_{s+1}, ..., \lambda_n$ be the eigenvalues of A that are contained inside the interval $[a,b]$, again repeated according to multiplicity, and let $u_{s+1}, ..., u_n$ again be corresponding orthonormal eigenvectors. This is possible by the spectral theorem, as this theorem states that $\mathbb{K}^n$ decomposes into the orthogonal direct sum of the eigenspaces of A. Then by construction we have $s \leq k-1$.

[19] Notice that, by 1., we have $\dim(U) \leq l$.

Now let $U = \text{span}\{u_1, ..., u_s\}$ and $W = \text{span}\{u_{s+1}, ..., u_n\}$. By construction $U^\perp = W$. Let

$$\mathcal{R}(j) = \text{span}\{R_1^0, ..., R_{j-1}^0, R_{j+1}^0, ..., R_k^0\}.$$

Then by construction the orthogonal projection

$$p_U : \mathcal{R}(j) \to U$$

is surjective. Now let $1 \leq i \leq s$. Because of the surjectivity there exists some $v \in \mathcal{R}(j)$ with $p_U(v) = u_i$. Let p_W denote the orthogonal projection onto W. Since $v = p_U(v) + p_W(v)$, we see that there exist $\mu_{s+1,i}, ..., \mu_{n,i} \in \mathbb{K}$ so that

$$u_i + \sum_{l=s+1}^{n} \mu_{l,i} \cdot u_l = v \in \mathcal{R}(j).$$

Let T_m be the m-th Chebyshev polynomial. Then define a polynomial

$$q_i^m(x) = \frac{T_{m-1}\left(\frac{b+a-2x}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)}.$$

Then q_i^m is a polynomial of degree $m-1$ and we have

$$\begin{aligned} q_i^m(A)\left(u_i + \sum_{l=s+1}^{n} \mu_{l,i} \cdot u_l\right) &= q_i(\lambda_i)u_i + \sum_{l=s+1}^{n} \mu_{l,i} \cdot q_i(\lambda_l)u_l \\ &= u_i + \sum_{l=s+1}^{n} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} u_l. \end{aligned}$$

Since $v \in \mathcal{R}(j)$ we thus see that we have

$$g_{i,m} := u_i + \sum_{l=s+1}^{n} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} u_l \in \bigoplus_{\substack{r=1 \\ r \neq j}}^{k} \mathcal{K}_{\mathbb{K}}^m(A, R_r^0). \tag{15}$$

Now we take a look at E_j^0, the initial error in the j-th component. We can write this in the eigenbasis of A, meaning there are $\xi_{1,j}, ..., \xi_{n,j}$ so that

$$E_j^0 = \sum_{t=1}^{n} \xi_{t,j} u_t.$$

Now we can construct the polynomials needed to prove the bound in the theorem. The definitions might seem cryptic at first, but they will become clear when we compute the bound later on. First, we define a polynomial $p_{j,m}$ via

$$1 - p_{j,m}(x)x = T_{m,[a,b],0},$$

where $T_{m,[a,b],0}$ is defined as in theorem 66. Since $T_{m,[a,b],0}(0) = 1$, this is well defined and gives a polynomial $p_{j,m}$ of degree $m-1$. Using equation (15) we then see that

$$\sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i)\xi_{i,j} g_{i,m} \in \bigoplus_{\substack{r=1 \\ r \neq j}}^{k} \mathcal{K}_{\mathbb{K}}^m(A, R_r^0),$$

and hence there exists polynomials $p_{1,m}, ..., p_{j-1,m}, p_{j+1,m}, ..., p_{k,m}$ of degree at most $m-1$ so that

$$\sum_{\substack{r=1 \\ r \neq j}}^{k} p_{r,m}(A)R_r^0 = \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i)\xi_{i,j} g_{i,m}.$$

Using lemma 44 we get

$$\|E_j^m\|_A \leq \min_{q_1,...,q_k \in P_{m-1}} \|E_j^0 - \sum_{l=1}^{k} q_l(A)R_l^0\|_A \leq \|E_j^0 - \sum_{l=1}^{k} p_{l,m}(A)R_l^0\|_A.$$

We can now compute

$$
\begin{aligned}
&E_j^0 - \sum_{l=1}^{k} p_{l,m}(A) R_l^0 = E_j^0 - p_{j,m}(A) A E_j^0 - \sum_{\substack{l=1 \\ r\neq j}}^{k} p_{r,m}(A) R_l^0 \\
&= T_{m,[a,b],0}(A) E_j^0 - \sum_{\substack{l=1 \\ r\neq j}}^{k} p_{r,m}(A) R_l^0 = \sum_{t=1}^{n} T_{m,[a,b],0}(\lambda_t) \xi_{t,j} u_t - \sum_{\substack{l=1 \\ r\neq j}}^{k} p_{r,m}(A) R_l^0 \\
&= \sum_{t=1}^{n} T_{m,[a,b],0}(\lambda_t) \xi_{t,j} u_t - \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} g_{i,m} \\
&= \sum_{t=1}^{n} T_{m,[a,b],0}(\lambda_t) \xi_{t,j} u_t - \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \left(u_i + \sum_{l=s+1}^{n} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} u_l \right) \\
&= \sum_{t=s+1}^{n} T_{m,[a,b],0}(\lambda_t) \xi_{t,j} u_t - \sum_{i=1}^{s} \sum_{l=s+1}^{n} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} u_l \\
&= \sum_{l=s+1}^{n} T_{m,[a,b],0}(\lambda_l) \xi_{l,j} u_l - \sum_{l=s+1}^{n} \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} u_l \\
&= \sum_{l=s+1}^{n} \left(T_{m,[a,b],0}(\lambda_l) \xi_{l,j} - \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} \right) u_l.
\end{aligned}
$$

Since $u_1, ..., u_n$ are A-orthogonal with $\|u_i\|_A = \sqrt{\lambda_i}$, the Pythagorean theorem then implies

$$
\begin{aligned}
\|E_j^m\|_A^2 &\leq \sum_{l=s+1}^{n} \left| T_{m,[a,b],0}(\lambda_l) \xi_{l,j} - \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} \right|^2 \lambda_l \\
&= \sum_{l=s+1}^{n} |T_{m,[a,b],0}(\lambda_l) \xi_{l,j}|^2 \lambda_l + \sum_{l=s+1}^{n} \left| \sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} \right|^2 \lambda_l \\
&\quad - 2 \sum_{l=s+1}^{n} \operatorname{Re} \left(\sum_{i=1}^{s} \overline{T_{m,[a,b],0}(\lambda_l) \xi_{l,j}} T_{m,[a,b],0}(\lambda_i) \xi_{i,j} \mu_{l,i} \frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)} \right) \lambda_l
\end{aligned}
$$

We now separately bound the three summands above. Using theorem 66 and the fact that for $l \geq s+1$ we have $\lambda_i \in [a,b]$, we get for the first term

$$
\sum_{l=s+1}^{n} |T_{m,[a,b],0}(\lambda_l) \xi_{l,j}|^2 \lambda_l \leq \frac{\sum_{l=s+1}^{n} |\xi_{l,j}|^2 \lambda_l}{T_m\left(\frac{b+a}{b-a}\right)^2} \leq \frac{\|E_j^0\|_A^2}{T_m\left(\frac{b+a}{b-a}\right)^2}.
$$

To bound the second term we need another quick fact about Chebyshev polynomials. For $m \geq 1$ we have for all $|x| \geq 1$ that $\left|\frac{T_m(x)}{T_{m-1}(x)}\right| \leq 2|x|$. For $m = 1$ this is clear, and for $m \geq 2$ we have by definition 60

$$
\left|\frac{T_m(x)}{T_{m-1}(x)}\right| = \left|\frac{2xT_{m-1}(x) - T_{m-2}(x)}{T_{m-1}(x)}\right| = \left|2x - \frac{T_{m-2}(x)}{T_{m-1}(x)}\right| \leq 2|x|,
$$

since Chebyshev polynomials are positive on $[1,\infty)$, T_{m-2} and T_{m-1} have different signs on $(-\infty,-1]$, as was for example shown in the proof of theorem 66, and we have $\left|\frac{T_{m-2}(x)}{T_{m-1}(x)}\right| \leq 1$ for $|x| \geq 1$, which can be easily proven

using induction and the recursive definition of the Chebyshev polynomials.. For bounding the second term, we first compute

$$\left|\sum_{i=1}^{s} T_{m,[a,b],0}(\lambda_i)\xi_{i,j}\mu_{l,i}\frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)}\right| \leq \sum_{i=1}^{s}|\xi_{i,j}\mu_{l,j}|\left|\frac{T_m\left(\frac{b+a-2\lambda_i}{b-a}\right)}{T_m\left(\frac{b+a}{b-a}\right)}\frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)}\right|$$
$$\leq \frac{\sum_{i=1}^{s}|\xi_{i,j}\mu_{l,i}|\cdot 2\left|\frac{b+a-2\lambda_i}{b-a}\right|}{\left|T_m\left(\frac{b+a}{b-a}\right)\right|} \leq \frac{c_l}{T_m\left(\frac{b+a}{b-a}\right)},$$

where c_l is some constant that does not depend on m. Note that we have used $\left|T_m\left(\frac{b+a-2\lambda_l}{b-a}\right)\right| \leq 1$, which follows from $\frac{b+a-2\lambda_l}{b-a} \in [-1,1]$, which in turn follows from $\lambda_l \in [a,b]$. This shows that the second term is bounded in absolute value by an expression of the form

$$\frac{d}{T_m\left(\frac{b+a}{b-a}\right)^2},$$

where the constant d does not depend on m.

The absolute value of the third term is bounded by

$$2\sum_{l=s+1}^{n}\sum_{i=1}^{s}\left|\lambda_l\xi_{l,j}\xi_{i,j}\mu_{l,i}T_{m,[a,b],0}(\lambda_l)T_{m,[a,b],0}(\lambda_i)\frac{T_{m-1}\left(\frac{b+a-2\lambda_l}{b+a}\right)}{T_{m-1}\left(\frac{b+a-2\lambda_i}{b-a}\right)}\right|$$
$$\leq \frac{2\sum_{l=s+1}^{n}\sum_{i=1}^{s}|\lambda_l\xi_{l,j}\xi_{i,j}\mu_{l,i}|\,2\left|\frac{b+a-2\lambda_i}{b-a}\right|}{T_m\left(\frac{b+a}{b-a}\right)^2}$$

We see that again the numerator does not depend on m. In total, we thus see that we have

$$\|E_j^m\|_A \leq \frac{c}{T_m\left(\frac{b+a}{b-a}\right)} \leq 2c\left(\frac{\sqrt{\frac{b}{a}}-1}{\sqrt{\frac{b}{a}}+1}\right)^k,$$

where the last inequality follows from corollary 62. This proves the theorem. □

Theorem 47 nicely illustrates the theoretical advantage of the BCG method over the CG method. While the convergence bound for the conjugate gradient method is based on the whole spectrum in the form of the condition, the estimate in theorem 47 is more dependent on how tightly the eigenvalues of the matrix A are clustered, and is independent of outliers at the edge of the spectrum.

We now want to take a look at other instances of the block conjugate gradient method. In example 27 we took a look at the matrix algebra $\bigoplus_{i=1}^{l}\mathbb{K}^{k_i\times k_i}$, where $k_1+...+k_l=k$. Then the $n\times k$-matrices become an inner product module with respect to multiplication

$$X\cdot a = (X_1\cdot a_1,...,X_l\cdot a_l),$$

and the inner product

$$\langle X,Y\rangle = (X_1^*Y_1,...,X_l^*Y_l).$$

If A is a positive definite matrix, define the map

$$T_A:\mathcal{X}\to\mathcal{X}; X\mapsto A\cdot X$$

Then T_A is $\bigoplus_{i=1}^{l}\mathbb{K}^{k_i\times k_i}$-linear, invertible and positive definite. This can be seen in the same way as for the algebra $\mathbb{K}^{k\times k}$, which we discussed above. Hence we can apply the BCG method to solve the system $AX=B$ using this inner product structure.

Taking a close look at algorithm 2 we see (and can prove using a simple induction) that the conjugate gradient algorithm for this algebra is equivalent to performing the classical block conjugate gradient method for the systems $AX_i=B_i$, where $B=(B_1,...,B_l)$ and B_i is a $k_i\times k_i$-matrix, in parallel. More concretely, this means that if X^m is the iterate produced by the block conjugate gradient method for the algebra $\bigoplus_{i=1}^{l}\mathbb{K}^{k_i\times k_i}$ and we write $X^m=(X_1^m,...,X_l^m)$, then X_i^m is the m-th iterate produced by the classical block conjugate gradient

method applied to the system $AX_i = B_i$. We call this instance of the BCG method the *hybrid block conjugate gradient algorithm.*

We can describe the breakdown behavior of the algorithm.

Lemma 48. Assume we want to solve the equation $AX = B$. Let X^m be the m-th iterate produced by the block conjugate gradient method with respect to the algebra $\bigoplus_{i=1}^m \mathbb{K}^{k_i \times k_i}$, and let R^m be the corresponding residual. Let $R^m = (R_1^m, ..., R_l^m)$. Then the algorithm breaks down at step m if and only if any of the matrices $R_1^m, ..., R_l^m$ fails to have full rank.

Proof. This is a direct consequence of lemma 45. □

Finally, we want to discuss the BCG method for the inner product structure introduced in example 27 (6). In this case we wrote $k = p \cdot q$ and choose the algebra $\mathbb{K}^{p\times p}$. The multiplication is given by

$$X \cdot a = (X_1 \cdot a, ..., X_q \cdot a)$$

and the inner product is given by

$$\langle X, Y \rangle = \sum_{j=1}^{q} X_j^* Y_j.$$

We again see that the map

$$T_A : \mathbb{K}^{n\times k} \to \mathbb{K}^{n\times k}; X \mapsto A \cdot X$$

is $\mathbb{K}^{p\times p}$-linear, invertible and positive definite. The corresponding calculations are straightforward. The only potentially non-obvious fact needed is that the sum of positive semidefinite matrices is again positive semidefinite. This can immediately be seen by looking at the characterization of positive semidefiniteness via the inner product. Hence we can apply the BCG method with respect to this algebra to solve the equation $A \cdot X = B$.

We want to describe the corresponding algorithm. First note that we have $\mathbb{K}^{n\times k} \simeq \mathbb{K}^{(n\cdot q)\times p}$ via

$$\Phi : \mathbb{K}^{n\times k} \to \mathbb{K}^{(n\cdot q)\times p}; \begin{pmatrix} X_1 & \dots & X_q \end{pmatrix} \mapsto \begin{pmatrix} X_1 \\ \vdots \\ X_q \end{pmatrix}. \tag{16}$$

Furthermore, define $A_q \in \mathbb{K}^{(n\cdot q)\times(n\cdot q)}$ to be

$$A_q = \begin{pmatrix} A & & \\ & \ddots & \\ & & A \end{pmatrix}.$$

Then we have $\Phi(A \cdot X) = A_q \cdot \Phi(X)$. This implies

$$AX = B \iff A_q \Phi(X) = \Phi(B).$$

Notice that A_q is a positive definite matrix on $\mathbb{K}^{n\cdot q}$. Hence we can solve the system $A \cdot X = B$ by applying the classical block conjugate gradient method to the equation $A_q\Phi(X) = \Phi(B)$.

Let $\widetilde{X}^m \in \mathbb{K}^{(n\cdot q)\times p}$ denote the iterates produced by the classical BCG method applied to the equation $A_q\Phi(X) = \Phi(B)$. Furthermore, let $X^m \in \mathbb{K}^{n\times k}$ denote the iterates produced by the BCG method with respect to the algebra $\mathbb{K}^{p\times p}$ and the inner product structure defined above. Let X^0 be the initial guess for the latter, and $\Phi(X^0)$ the initial guess for the former. Then we have $\Phi(X^m) = \widetilde{X}^m$. This can be shown directly via an induction, using the definition of algorithm 2. We call this approach the *global block conjugate gradient algorithm.*

We see that the global block conjugate gradient algorithm is essentially given by solving the system $A_q\Phi(X) = \Phi(B)$ using the classical block conjugate gradient algorithm.

We can again describe the breakdown behavior of the method.

Lemma 49. Assume we want to solve the equation $AX = B$. Let X^m be the m-th iterate produced by the global block conjugate gradient method corresponding to the partition $k = p \cdot q$. Let R^m denote the corresponding residuals. Then the algorithm breaks down at step m if and only if the matrix $\Phi(R^m) \in \mathbb{K}^{(n\cdot q)\times p}$ fails to have full rank, where Φ is defined as in (16).

Proof. This is a direct consequence of lemma 45 and the fact that the global BCG method is equivalent to a classical BCG method for the equation $A_q\Phi(X) = \Phi(B)$. □

We now want to compare the different instances of the BCG method. For this, we first need to discuss norms on $\mathbb{K}^{n\times k}$.

Remark. We have now discussed three inner product structures on $\mathbb{K}^{n\times k}$, namely

1. $\langle X, Y\rangle_{\text{cl}} = X^*Y \in \mathbb{K}^{k\times k}$
2. $\langle X, Y\rangle_{\text{hy}} = (X_1^*Y_1, ..., X_l^*Y_l) \in \bigoplus_{i=1}^{l} \mathbb{K}^{k_i\times k_i}$
3. $\langle X, Y\rangle_{\text{gl}} = \sum_{i=1}^{q} X_i^*Y_i \in \mathbb{K}^{p\times p}$

All three classes of algebras mentioned above are admissable with respect to the functionals $\text{Tr}_k : \mathbb{K}^{k\times k} \to \mathbb{K}$, $\bigoplus_{i=1}^{l} \text{Tr}_{k_i} : \bigoplus_{i=1}^{l} \mathbb{K}^{k_i\times k_i} \to \mathbb{K}$ and $\text{Tr}_p : \mathbb{K}^{p\times p} \to \mathbb{K}$. This follows from lemma 39 and the discussion thereafter.

Hence by theorem 37, we see that the different instances of the BCG method produce optimal approximations inside the respective block Krylov spaces with respect to the norm

$$\|X\|_{\text{ind}} = \sqrt{f(\langle T_A(X), X\rangle)},$$

where f is the corresponding functional mentioned above. In the first case we have

$$\|X\|^2_{\text{ind, cl}} = \text{Tr}_k(X^*AX) =: \|X\|^2_A.$$

For the second case we have

$$\|X\|^2_{\text{ind, hy}} = \sum_{i=1}^{l} \text{Tr}_{k_i}(X_i^*AX_i) = \text{Tr}_k(X^*AX) = \|X\|^2_A,$$

and in the third case

$$\|X\|^2_{\text{ind, gl}} = \text{Tr}_p\left(\sum_{i=1}^{q} X_i^*AX_i\right) = \sum_{i=1}^{q} \text{Tr}_p(X_i^*AX_i) = \|X\|^2_A.$$

Hence the classical, hybrid and global BCG methods *all produce iterates that are optimal with respect to the norm* $\|\cdot\|_A$.

Theorem 50. Let A be positive definite. Let $B \in \mathbb{K}^{n\times k}$, and let $k = p \cdot q$. Assume we want to solve the equation $A \cdot X = B$, and assume we have an initial guess $X^0 \in \mathbb{K}^{n\times k}$. Let X^m_{cl} denote the iterates produced by the classical BCG method, let X^m_{hy} denote the iterates produced by the hybrid BCG method for the algebra $\bigoplus_{i=1}^{q} \mathbb{K}^{p\times p}$ and let X^m_{gl} denote the iterates produced by the global BCG method for the algebra $\mathbb{K}^{p\times p}$. Then we have

$$\|X - X^m_{\text{cl}}\|_A \le \|X - X^m_{\text{hy}}\|_A \le \|X - X^m_{\text{gl}}\|_A.$$

Proof. Since all methods produce iterates that are best approximations with respect to the $\|\cdot\|_A$-norm inside the respective shifted block Krylov spaces, it suffices to show

$$\mathcal{K}^m_{\mathbb{K}^{p\times p}}(T_A, R^0) \subset \mathcal{K}^m_{\bigoplus_{i=1}^{q} \mathbb{K}^{p\times p}}(T_A, R^0) \subset \mathcal{K}^m_{\mathbb{K}^{k\times k}}(T_A, R^0).$$

To show the first inclusion it suffices to show that for $X \in \mathbb{K}^{n\times k}$ and $a \in \mathbb{K}^{p\times p}$, there exists some $b \in \bigoplus_{i=1}^{q} \mathbb{K}^{p\times p}$ with $X \cdot a = X \cdot b$. But this can be achieved by setting $b = (a, ..., a)$. To show the second inclusion it suffices to show that for $b = (b_1, ..., b_q) \in \bigoplus_{i=1}^{q} \mathbb{K}^{p\times p}$, there exists some $c \in \mathbb{K}^{k\times k}$ with $X \cdot b = X \cdot c$. This in turn can be achieved by setting

$$c = \begin{pmatrix} b_1 & & \\ & \ddots & \\ & & b_q \end{pmatrix}.$$

□

We remind the reader that we called the algebra $\bigoplus_{i=1}^{q} \mathbb{K}^{p\times p}$ a *hybrid algebra* and denoted it by $\mathbb{S}^{p,q}_{\text{hy}}$. Furthermore, we call the algebra $\mathbb{K}^{p\times p}$ which we intend to use for a global BCG method a *global algebra* and denote it by $\mathbb{S}^{p,q}_{\text{gl}}$.

Theorem 51. We make the same assumptions as in theorem 50. Assume that we have $k = p_1 \cdot q_1 = p_2 \cdot q_2$. Furthermore assume that p_2 divides p_1. Let X^m_{hy,p_1} be the iterates produced by the hybrid BCG method for the algebra $\mathbb{S}^{p_1,q_1}_{\mathrm{hy}}$. Let X^m_{hy,p_2}, X^m_{gl,p_1} and X^m_{gl,p_2} be defined in the same way. Then we have

$$\|X - X^m_{\mathrm{hy},p_1}\|_A \leq \|X - X^m_{\mathrm{hy},p_2}\|_A$$

and

$$\|X - X^m_{\mathrm{gl},p_1}\|_A \leq \|X - X^m_{\mathrm{gl},p_2}\|_A.$$

Proof. Just as in the proof of theorem 50, it suffices to show that given $X \in \mathbb{K}^{n\times k}$ and $a \in \mathbb{S}^{p_2,q_2}_{\mathrm{hy}}$ and $b \in \mathbb{S}^{p_2,q_2}_{\mathrm{gl}}$, there exist $c \in \mathbb{S}^{p_1,q_1}_{\mathrm{hy}}$ and $d \in \mathbb{S}^{p_1,q_1}_{\mathrm{gl}}$ with $X \cdot a = X \cdot c$ and $X \cdot b = X \cdot d$. Let $p_1 = kp_2$. Looking at the definition of the multiplication, we see that this can be achieved by setting

$$c = \big(\mathrm{diag}(a_1, ..., a_k), ..., \mathrm{diag}(a_{q_2(k-1)+1}, ..., a_{q_2})\big)$$

and

$$d = \mathrm{diag}(\underbrace{b, ..., b}_{k \text{ times}}).$$

□

We thus see that, given a fixed amount of iterations, the classical BCG method produces iterates at least as good as the hybrid method, and the hybrid method produces iterates as least as good as the global method. Moreover, increasing the matrix size in the hybrid and global methods is guaranteed to lead to iterates that are at least as good as the approximations produced by algebras based on smaller matrices.

Furthermore, letting $k = p \cdot q$ and ignoring the question if surjective projections exist, theorem 47 then guarantees a convergence bound given by the tightest cluster of $n - k + 1$ eigenvalues of A for the classical BCG method. As the hybrid BCG method is equivalent to performing q instances of the classical BCG method for matrices with p right hand sides in parallel, theorem 47 gives us a convergence bound that is essentially determined by the tightest cluster of $n - p + 1$ eigenvalues. Furthermore, the global BCG method is equivalent to solving a system $A_q\Phi(X) = \Phi(B)$. One easily sees that the eigenvalues of A_q are given by the eigenvalues of A, whose multiplicity is mulitplied by q. Hence choosing a cluster of l eigenvalues for A corresponds to choosing a cluster of $l \cdot q$ eigenvalues of A_q. By theorem 47, the convergence bound is essentially given by the tightest cluster of $n \cdot q - p + 1$ eigenvalues of A_q, which means the convergence bound is essentially given by the tightest cluster of $n - \left\lfloor \frac{p-1}{q} \right\rfloor$ eigenvalues of A. We summarize this in table 4.

BCG variant	classical	hybrid	global
convergence bound given by tightest cluster of # EV	$n - k + 1$	$n - p + 1$	$n - \left\lfloor \frac{p-1}{q} \right\rfloor$

Table 4: Dependence of convergence estimate on eigenvalues

We thus see that the convergence bound for the classical method in general relies on fewer eigenvalues of A than the hybrid method, which itself relies on fewer eigenvalues than the global method. Hence we see that going from the classical to the hybrid to the global method, the algorithm becomes more dependent on the distribution of the eigenvalues of the matrix.

In total we thus expect the classical method to need fewer iterations to converge than the hybrid method, and we expect the hybrid method to need fewer iterations to converge than the global method. However, this does not have to mean that the classical method converges in less time than the hybrid or global method. While in total fewer iterations are needed, we see by looking at the algorithm that the classical method needs to perform operations with respect to the algebra of $k \times k$-matrices, while the hybrid method is based on operations using q-tuples of matrices of size $p \times p$, whereas the global method is based on operations using $p \times p$ matrices. Since matrix operations become increasingly expensive the larger the matrices they are performed on, we expect a single iteration of the classical method to need more work than an iteration of the hybrid method, which itself needs more work than an iteration of the global method. Thus the tradeoff for fewer iterations is a higher workload per iteration. It is thus not immediately obvious which method performs best with regard to the time needed to converge. If the matrices operated on become too large, the cost of performing operations like matrix multiplication and inversion will make the algorithm inefficient. If the matrices are too small on the other hand, we might lose out on a larger search space that would lead to a quick convergence, or we might not be able to fully utilize vectorization. The proper choice of method thus requires some thought, and we will discuss this in more depth later on.

One question we have largely ignored up to now is what to do if the algorithm breaks down. We have discussed in the lemmas 45, 48 and 49 under what conditions such a breakdown occurs, and have seen that, essentially, a breakdown indicates that a linear combination of the iterates has converged. But typically, we do not want to only solve some linear combination, but instead the whole system $A \cdot X = B$. In this case, we need some way to handle the breakdown.

4.4 Deflation Methods

In 4.3 we saw that studying the classical, hybrid and global method for solving the system $AX = B$ with A positive definite comes down to studying the classical method. Hence we will now study how to handle breakdowns in the classical BCG method, and these ideas can then directly be generalized to the hybrid method, which is essentially a parallel version of the classical BCG, and the global method, which is essentially the classical BCG after rearranging the underlying matrices.

For the rest of this subsection we will thus study the BCG method for the system $AX = B$ with respect to the inner product $\langle X, Y\rangle = X^*Y$. For this, we first need a few facts.

Lemma 52. Let $X, Y \in \mathbb{K}^{n\times k}$. Denote by $\mathrm{Im}(Z)$ the image of a matrix Z, which is simply the linear span of its columns. Denote by Z^* the conjugate transpose of a matrix Z. Let M be a positive definite matrix. Then we can define an inner product via $\langle X, Y\rangle_M = X^*MY$. Then we have the following.

1. $\mathrm{Im}(X^*MX) = \mathrm{Im}(X^*)$
2. There exists some $a \in \mathbb{K}^{k\times k}$ so that
$$\langle X, Y\rangle_M = \langle X, X\rangle_M \cdot a.$$
3. We can write $X = \overline{X} \cdot \sigma$, where $\overline{X} \in \mathbb{K}^{n\times \mathrm{rank}(X)}$, $\sigma \in \mathbb{K}^{\mathrm{rank}(X)\times kt}$ and $\overline{X}^*\overline{X}$ is the identity matrix of dimension $\mathrm{rank}(X)$. In this case we write $(\overline{X}, \sigma) = \mathrm{Orth}(X)$. For any such decomposition, we have $\mathrm{rank}(\sigma) = \mathrm{rank}(X) = \mathrm{rank}(\overline{X})$ and σ^* is injective.

Proof. It is well known and otherwise easy to show that $\ker(X) = \ker(X^*MX)$. Since $\mathrm{Im}(X^*MX) \subseteq \mathrm{Im}(X^*)$, the first claim then follows from $\mathrm{rank}(X^*) = \mathrm{rank}(X)$ and the rank-nullity theorem. To see the second claim, note that
$$\langle X, Y\rangle = \langle X, X\rangle \cdot a \iff X^*MY = X^*MXa,$$
which we can read column-wise as ordinary linear systems $X^*MY_i = X^*MXa_i$, which are solvable by the first claim. The existence in the third claim can easily be established by choosing an orthonormal basis of $\mathrm{Im}(X)$ and writing the columns of X in this basis. The claim about the ranks follows directly from the fact that for matrices S, T we have $\mathrm{rank}(ST) \leq \min\{\mathrm{rank}(S), \mathrm{rank}(T)\}$, which in turn implies the surjectivity of σ, which itself implies the injectivity of σ^*. □

The goal of the block conjugate gradient method was to compute projection solutions with respect to the block Krylov spaces. The first natural question, then, is whether such solutions actually exist. In the classical case, this can be answered positively.

Corollary 53. Let $\mathcal{Y}$ be a submodule of $\mathbb{K}^{n\times k}$ and let $X \in \mathbb{K}^{n\times k}$. Let M be a positive definite matrix. Then there exists a unique orthogonal projection of X onto $\mathcal{Y}$ with respect the inner product $\langle X, Y\rangle_M = X^*MY$.

Proof. By the remark following Corollary 30 and lemma 52.2 it suffices to show that there is an M-orthogonal spanning set of $\mathcal{Y}$. But since $\mathcal{Y}$ is finitely generated as a vector space, it is also finiteley generated as a submodule. Iteratively applying lemma 29 and lemma 52 then produces such an orthogonal spanning set. □

Corollary 54. Assume we want to solve the equation $AX = B$ with $X \in \mathbb{K}^{n\times k}$. Denote by $\mathcal{K}^m(A, R^0)$ the block Krylov space with respect to the algebra $\mathbb{K}^{k\times k}$.[20]. Then there exists a unique projection solution of $AX = B$ with respect to $\mathcal{K}^m(A, R^0)$, meaning there is a unique X^m so that
$$X^m \in X^0 + \mathcal{K}^m(A, R^0)$$
$$B - AX^m \perp_{\langle\cdot,\cdot\rangle} \mathcal{K}^m(A, R^0)$$

Proof. Note that X^m is the corresponding projection solution if and only if $X^m - X^0$ is the orthogonal projection of $X - X^0$ onto $\mathcal{K}^m(A, R^0)$ with respect to $\langle\cdot,\cdot\rangle_A$. The claim then follows from corollary 53. □

[20]This should not lead to any confusion with respect to the ordinary Krylov space, as here R^0 is a matrix, while the corresponding argument in the ordinary Krylov space is a vector. And if $R^0$is a vector, the two definitions agree.

Hence there exist unique projection solutions, and potential breakdowns of the classical BCG method are not due to a lack of existence of the solutions. Hence the only thing we are missing at the moment is a way to compute them in the breakdown case.

We now want to show that a natural modification of the BCG algorithm, as it is described in algorithm 2, can achieve this. We remind the reader that we denoted the classical algebra by $\mathbb{S}_{\text{cl}}$.

Theorem 55. Assume we want to solve the equation $AX = B$, where $X \in \mathbb{K}^{n\times k}$. Let X^0 be an initial guess. Then define $R^0 = B - AX^0$ and $P^0 = R^0$. Assuming that $R^0, ..., R^m, P^0, ...P^m, X^0, ..., X^m$ have already been defined, we continue the construction as follows.

1. Choose $\alpha^m \in \mathbb{K}^{k\times k}$ so that
$$\langle R^m, R^m\rangle = \langle AP^m, P^m\rangle\alpha^m$$
2. Set
$$X^{m+1} = X^m + P^m\alpha^m$$
and
$$R^{m+1} = R^m - AP^m\alpha^m$$
3. Choose $\beta^m \in \mathbb{K}^{k\times k}$ so that
$$\langle R^{k+1}, R^{k+1}\rangle = \langle R^k, R^k\rangle\beta^k$$
4. Set
$$P^{m+1} = R^{m+1} + P^m\beta^m.$$

Then we have the following.

1. The construction as described above is always possible.
2. For all $m \geq 0$ we have
   (a) $\mathcal{K}^{m+1}(A, R^0) = \text{span}_{\mathbb{S}_{\text{cl}}}\{R^0, ..., R^m\} = \text{span}_{\mathbb{S}_{\text{cl}}}\{P^0, ..., P^m\}$.
   (b) $\langle R^m, R^m\rangle = \langle P^m, R^m\rangle = \langle R^m, P^m\rangle$
   (c) $\langle P^m, P^m\rangle_A = \langle R^m, P^m\rangle_A = \langle P^m, R^m\rangle_A$
   (d) $\text{rank}(R^m) = \text{rank}(P^m)$ and $\ker(R^m) = \ker(P^m)$
   (e) For $i < m$ we have $\langle R^m, R^i\rangle = 0$ and $\langle P^m, P^i\rangle_A = 0$.
3. For $m \geq 1$ we have that X^m is the unique projection solution with respect to $\mathcal{K}^m(A, R^0)$ and initial guess X^0.

Proof. By definition, claim 2. is true for $m = 0$. To show the claim of the theorem it suffices to show that if claim 2. is true for some m, then the construction of X^{m+1}, R^{m+1} and P^{m+1} succeeds, 2. is true for $m+1$ and X^{m+1} is the projection solution.

So assume 2. is true for some m. Since $\langle R^m, R^m\rangle = \langle P^m, R^m\rangle$ and $\text{Im}((P^m)^*) = \text{Im}((P^m)^*AP^m)$, there exists some α^m with
$$\langle R^m, R^m\rangle = \langle AP^m, P^m\rangle\alpha^m.$$
We then have
$$\langle R^m, R^{m+1}\rangle = \langle R^m, R^m\rangle - \langle R^m, AP^m\rangle\alpha^m = 0,$$
and for $j < m$
$$\langle R^j, R^{m+1}\rangle = -\langle R^j, AP^m\rangle\alpha^m = 0,$$
where we have used that $\langle P^i, P^m\rangle_A = 0$ for $i < m$ and $R^j \in \text{span}_{\mathbb{S}_{\text{cl}}}\{P^0, ..., P^j\}$. Since $\mathcal{K}^{m+1}(A, R^0) = \text{span}_{\mathbb{S}_{\text{cl}}}\{R^0, ..., R^m\}$ this implies that X^{m+1} is the unique projection solution.

We now want to prove that $(\alpha^m)^*$ is injective on $\text{Im}((P^m)^*)$. By construction we have
$$(\alpha^m)^*\langle AP^m, P^m\rangle = \langle R^m, R^m\rangle,$$

and by lemma 52 and the induction hypothesis we then have

$$\begin{aligned}\operatorname{rank}((\alpha^m)^*\langle AP^m, P^m\rangle) &= \operatorname{rank}(\langle R^m, R^m\rangle) = \operatorname{rank}(R^m) = \operatorname{rank}(P^m)\\ &= \operatorname{rank}(\langle AP^m, P^m\rangle),\end{aligned}$$

and $\operatorname{rank}(ST) = \operatorname{rank}(T)$ implies by the rank-nullity theorem that $\ker(ST) = \ker(T)$, which implies that S is injective on the range of T. This implies that $(\alpha^m)^*$ is injective on the range of $\langle AP^m, P^m\rangle$, but by lemma 52 this is the same as $\operatorname{Im}((P^m)^*)$. But then we have for $\gamma \in \mathbb{K}^{k\times k}$ using the construction of R^{k+1}

$$\begin{aligned}-\langle P^m, R^{m+1}\rangle_A &= \langle P^m, P^m\rangle_A\gamma\\ \iff -(P^m)^*AR^{m+1} &= (P^m)^*AP^m\gamma\\ \iff -(\alpha^m)^*(P^m)^*AR^{m+1} &= (\alpha^m)^*(P^m)^*AP^m\gamma\\ \iff \langle R^{m+1}, R^{m+1}\rangle &= \langle R^m, R^m\rangle\gamma.\end{aligned}$$

Since $\operatorname{Im}((P^m)^*) = \operatorname{Im}(\langle P^m, P^m\rangle_A)$ we see that there exists some β^m with

$$\langle R^{m+1}, R^{m+1}\rangle = \langle R^m, R^m\rangle\beta^m.$$

We thus see that the construction of X^{m+1}, R^{m+1} and P^{m+1} as described is possible. Furthermore, we have

$$\langle P^m, P^{m+1}\rangle_A = \langle P^m, R^{m+1}\rangle_A + \langle P^m, P^m\rangle_A\beta^m = 0,$$

and for $j < m$ we have $\langle P^j, P^{m+1}\rangle_A = 0$ since $AP^j \in \mathbb{K}^{m+1}_{\mathbb{S}_{\mathrm{cl}}}(A, R^0)$ and we have $R^{m+1} \perp \mathbb{K}^{m+1}_{\mathbb{S}_{\mathrm{cl}}}(A, R^0)$, as shown above. Furthermore, computing $\langle P^{m+1}, P^{m+1}\rangle_A$ gives $(P^{m+1})^*AP^{m+1} = (P^{m+1})^*AR^{m+1}$, and hence

$$\ker(R^{m+1}) \subset \ker((P^{m+1})^*AP^{m+1}) = \ker(P^{m+1}).$$

Similarly, we have $\langle R^{m+1}, R^{m+1}\rangle = \langle R^{m+1}, P^{m+1}\rangle$, and hence we also have $\ker(P^{m+1}) \subset \ker(R^{m+1})$. This implies $\ker(P^{m+1}) = \ker(R^{m+1})$, and the rank-nullity theorem then implies $\operatorname{rank}(R^{m+1}) = \operatorname{rank}(P^{m+1})$. These computations effectively also show claims 2 (b) and (c) for $m+1$.

Thus it only remains to show 2 (a). Exactly as in the proof of theorem 35 we see that

$$\operatorname{span}_{\mathbb{S}_{\mathrm{cl}}}\{P^0, ..., P^{m+1}\} = \operatorname{span}_{\mathbb{S}_{\mathrm{cl}}}\{R^0, ..., R^{m+1}\} \subseteq \mathbb{K}^{m+2}_{\mathbb{S}_{\mathrm{cl}}}(A, R^0),$$

Moreover, just as in the proof of theorem 35, to show the reverse inclusion it suffices to show that $AP^m \in \operatorname{span}_{\mathbb{S}_{\mathrm{cl}}}\{R^0, ..., R^{m+1}\}$. This is equivalent to showing that the columns of AP^m are linear combinations of the columns of $R^0, ..., R^{m+1}$. Now the defining equality

$$\langle R^m, R^m\rangle = \langle AP^m, P^m\rangle\alpha^m$$

implies that $\operatorname{rank}(R^m) = \operatorname{rank}(AP^m\alpha^m)$, and since A is invertible we thus have $\operatorname{rank}(AP^m) = \operatorname{rank}(R^m) = \operatorname{rank}(AP^m\alpha^m)$. But this implies $\operatorname{Im}(AP^m) = \operatorname{Im}(AP^m\alpha^m) = \operatorname{Im}(R^{m+1} - R^m)$. This proves the claim. □

Theorem 55 gives a strategy for computing projection solutions. This strategy does not rely on assumptions about the rank of the residuals R^m, as algorithm 2 does for contrast. Now it only remains to construct an algorithm that actually implements this strategy. Our approach is based on [44], section 6.

The central idea is to orthogonalize the residual R^m, which means writing it in the form $R^m = \overline{R^m}\sigma^m$, where $\operatorname{rank}(\overline{R^m}) = \operatorname{rank}(R^m)$ and the columns of $\overline{R^m}$ form an orthonormal set, as described by lemma 52.3. We can then use this decomposition to reduce the problem of finding α^m with

$$\langle R^m, R^m\rangle = \langle AP^m, P^m\rangle\alpha^m$$

to a problem with potentially fewer dimensions we can then actually solve. More concretely, let $(\overline{R^m}, \sigma^m) = \operatorname{Orth}(R^m)$. Since $R^m = \overline{R^m}\sigma^m$, we have the inclusion $\ker(\sigma^m) \subset \ker(R^m)$. We now by lemma 52 that $\operatorname{rank}(\sigma^m) = \operatorname{rank}(R^m)$, and for dimensionality reasons we thus have $\ker(\sigma^m) = \ker(R^m) = \ker(P^m)$. Hence, by the fundamental theorem of homomorphisms, there is a unique full rank matrix $\overline{P^m}$ with $P^m = \overline{P^m}\sigma^m$. Now we can compute

$$\langle R^m, R^m\rangle = \langle AP^m, P^m\rangle\alpha^m \iff (\overline{P^m}^*A\overline{P^m})^{-1}\sigma^m = \sigma^m\alpha^m,$$

where we have used the fact that $(\sigma^m)^*$ is injective. But this implies

$$\begin{aligned}X^{m+1} &= X^m + P^m\alpha^m\\ &= X^m + \overline{P^m}(\overline{P^m}^*A\overline{P^m})^{-1}\sigma^m.\end{aligned}$$

If we can thus devise an easy way to compute $\overline{R^m}, \overline{P^m}$ and σ^m, we can compute the projection solution X^{m+1} by the formula defined above. For now, assume we know how to compute $(\overline{R}, \sigma) = \mathrm{Orth}(R)$. We will later discuss this in more depth.

Now assume we already know $(\overline{R^m}, \sigma^m) = \mathrm{Orth}(R^m)$ and a $\overline{P^m}$ as described above. Then we have

$$R^{m+1} = R^m - AP^m\alpha^m = \left(\overline{R^m} - A\overline{P^m}(\overline{P^m}^* A\overline{P^m})^{-1}\right)\sigma^m = \widetilde{R^{m+1}}\sigma^m.$$

Then let $(\overline{R^{m+1}}, \gamma^{m+1}) = \mathrm{Orth}(\widetilde{R^{m+1}})$. Define $\sigma^{m+1} = \gamma^{m+1}\sigma^m$. We claim that we then have $(\overline{R^{m+1}}, \sigma^{m+1}) = \mathrm{Orth}(R^{m+1})$. Since the columns of $\overline{R^{m+1}}$ are orthonormal by construction, we need to show that $\mathrm{rank}(\overline{R^{m+1}}) = \mathrm{rank}(R^{m+1})$. By construction this is equivalent to $\mathrm{rank}(\widetilde{R^{m+1}}) = \mathrm{rank}(R^{m+1})$. But this is a consequence of the surjectivity of σ^m and the equality $R^{m+1} = \widetilde{R^{m+1}}\sigma^m$. Thus, given $\overline{R^m}, \overline{P^m}$ and σ^m, we can compute a decomposition $(\overline{R^{m+1}}, \sigma^{m+1}) = \mathrm{Orth}(R^{m+1})$ without needing to explicitely compute R^{m+1}. Furthermore, we have

$$\langle R^{m+1}, R^{m+1}\rangle = \langle R^m, R^m\rangle\beta^m \iff (\gamma^{m+1})^*\sigma^{m+1} = \sigma^m\beta^m,$$

where we have again used the injectivity of $(\sigma^m)^*$. This implies

$$\begin{aligned} P^{m+1} &= R^{m+1} + P^m\beta^m \\ &= \left(\overline{R^{m+1}} + \overline{P^m}(\gamma^{m+1})^*\right)\sigma^{m+1}, \end{aligned}$$

and hence we can compute the factorization of P^{m+1} through σ^{m+1} via the equation $\overline{P^{m+1}} = \overline{R^{m+1}} + \overline{P^m}(\gamma^{m+1})^*$. We thus see that we never need to actually compute R^m, P^m, α^m and β^m. We summarize this discusion in algorithm 3.

Algorithm 3 Deflated Block Conjugate Gradient Method

Require: - Positive definite matrix $A \in \mathbb{K}^{n\times n}$
- Right hand side $B \in \mathbb{K}^{n\times k}$
- Initial guess $X^0 \in \mathbb{K}^{n\times k}$

$R^0 \leftarrow B - AX^0$
$(\overline{R^0}, \sigma^0) \leftarrow \mathrm{Orth}(R^0)$
$\overline{P^0} \leftarrow \overline{R^0}$
for $m = 0, 1, 2, ...$ until convergence **do**
  $\overline{\alpha^m} \leftarrow \left(\overline{P^m}^* A\overline{P^m}\right)^{-1}$
  $X^{m+1} \leftarrow X^m + \overline{P^m}\cdot\overline{\alpha^m}\cdot\sigma^m$
  $\widetilde{R^{m+1}} \leftarrow \overline{R^m} - A\overline{P^m}\cdot\overline{\alpha^m}$
  $(\overline{R^{m+1}}, \gamma^{m+1}) \leftarrow \mathrm{Orth}(\widetilde{R^{m+1}})$
  $\sigma^{m+1} \leftarrow \gamma^{m+1}\sigma^m$
  $\overline{P^{m+1}} \leftarrow \overline{R^{m+1}} + \overline{P^m}\cdot\left(\gamma^{m+1}\right)^*$
end for

Our considerations above prove the following theorem.

Theorem 56. The sequence X^m, $m = 1, 2, ...$ produced by Algorithm 3 is the sequence of projection solutions of the equation $AX = B$ with respect to the initial guess X^0 and the spaces $\mathcal{K}^m_{\mathbb{S}_{c1}}(A, R^0)$, $m = 1, 2, ...$

Note that we have shown above that the constructed $\overline{P^m}$ has full rank, and hence $\overline{P^m}^* A\overline{P^m}$ is invertible. Hence the algorithm above cannot break down, at least in exact arithmetic.

Note that we can check if we have achieved the desired residual reduction without needing to compute R^m explicitly. By construction, we have $R^m = \overline{R^m}\sigma^m$, and since the columns of $\overline{R^m}$ are orthonormal we have by the Pythagorean theorem for the columns of R^m

$$\|R^m_j\|_2 = \|\sigma^m_j\|_2.$$

Hence we can check via the columns of σ^m if the iteration has converged to the desired degree.

We have thus derived an algorithm that can handle rank deficient residuals in the iteration. We now want to shortly talk about the orthonormalization step inside the algorithm, which means we want to talk about how we can compute $(\overline{R}, \sigma) = \mathrm{Orth}(R)$. We actually want to talk about this in a slightly more general setting. Let

M be a positive definite matrix. We then say $(\overline{R}, \sigma) = \mathrm{Orth}(R, M)$ if the columns of $\overline{R}$ are M-orthonormal, $\mathrm{rank}(\overline{R}) = \mathrm{rank}(R)$ and $R = \overline{R}\sigma$.

The implementation inside dune-istl[21] follows the example of [45], where it is advised to use the (repeated) choleskyQR algorithm to compute such a decomposition for tall and skinny matrices. As, at least to the author of this thesis, the application of this method to rank deficient matrices is non-trivial, we want to say a few words about it.

The basic idea is quite simple. Assume that $X \in \mathbb{K}^{n\times k}$ has full rank. Then $X^*MX \in \mathbb{K}^{k\times k}$ has full rank and is thus positive definite. Hence we can write $X^*MX = LL^*$, where L is an invertible lower triangular matrix. We can then set $Q = X(L^*)^{-1}$ and $R = L^*$. Then we have $QR = X$, and

$$Q^*MQ = L^{-1}X^*MX(L^*)^{-1} = E_k.$$

Hence we have computed a QR decomposition of X with respect to the orthogonality defined by M.

We want to generalize this construction to rank deficient matrices. We first have to talk about the Cholesky decomposition in this case. We have the following theorem.

Theorem 57 (Cholesky Decomposition)**.** Let Z be self-adjoint and positive semidefinite. Then there exists a unique lower triangular matrix L so that $Z = LL^*$ with the following properties.

1. The diagonal entries of L are nonnegative.
2. For every diagonal entry we have either $L_{i,i} \neq 0$ or the entire i-th column is 0.

Then we have

$$\mathrm{rank}(Z) = \#\{i \mid L_{i,i} \neq 0\}.$$

A proof can be found in [46], 3.2.2. This decomposition can be computed easily by just looking at the equation $Z = LL^*$. Looking entry-wise at this equation directly produces an algorithm for computing L, either column-wise or row-wise.

Now let $X \in \mathbb{K}^{n\times k}$ and let $s = \mathrm{rank}(X)$. Then X^*MX is a self-adjoint positive semidefinite $k \times k$-matrix, and hence possesses a Cholesky decomposition as described in theorem 57. Let $X^*MX = LL^*$ be this decomposition. Let $i_1 < i_2 < ... < i_s$ be the columns of L with non-zero diagonal entries. Let $\overline{L}$ be the $k \times s$ matrix obtained from L by deleting all zero columns, and let $\hat{L}$ be the $k \times k$-matrix obtained from L by setting the zero diagonal entries to 1. Let $\iota = (e_{i_1}, ..., e_{i_s}) \in \mathbb{K}^{k\times s}$ and $p = \iota^*$, where e_i denotes the i-th unit vector. Then $\hat{L}$ is lower triangular with non-zero diagonal entries, and hence invertible. One can then easily see that $\hat{L}\iota = \overline{L}$ and $p\iota = E_s$. This implies $\hat{L}^{-1}\overline{L} = \iota$ and $\overline{L}p\hat{L}^{-1}|_{\mathrm{Im}(\overline{L})} = \mathrm{id}_{\mathrm{Im}(\overline{L})}$. If we thus define $Q = X(p\hat{L}^{-1})^* \in \mathbb{K}^{n\times \mathrm{rank}(X)}$ and $\sigma = \overline{L}^*$, we have

$$Q\sigma = X(p\hat{L}^{-1})^*L^* = (Lp\hat{L}^{-1}X^*)^* = (X^*)^* = X,$$

where we have used $\mathrm{Im}(X^*) = \mathrm{Im}(X^*X) = \mathrm{Im}(LL^*) = \mathrm{Im}(L) = \mathrm{Im}(\overline{L})$. Furthermore, we have $\overline{L}\,\overline{L}^* = LL^*$,[22] and hence

$$\begin{aligned} Q^*Q &= p\hat{L}^{-1}X^*MX(\hat{L}^{-1})^*\iota = p\hat{L}^{-1}LL^*(\hat{L}^{-1})^*\iota \\ &= p\hat{L}^{-1}\overline{L}\,\overline{L}^*(\hat{L}^{-1})^*\iota = p\iota p\iota = E_s. \end{aligned}$$

This shows that we have $(Q, \sigma) = \mathrm{Orth}(X, M)$. We have thus derived an algorithm to compute $\mathrm{Orth}(X, M)$ that relies on the Cholesky decomposition of X^*MX. We call this algorithm the *choleskyQR algorithm*. Note that the actual implementation inside dune-istl cited above is slightly more complex, as the algorithm there is implemented to support the version described in [45]. More concretely, [45] suggests to compute the decomposition by essentially applying the ordinary choleskyQR algorithm 3 times to different matrices and build the total factorization from the results of these separate instances of the choleskyQR algorithm. The corresponding algorithm is called choleskyQR3. In [45] there is also a variant based on two instances of the choleskyQR method, called *choleskyQR2*. The implementation in dune-istl supports all three methods. In the current implementation of the computation of the transfer matrices inside duneuro we always use the ordinary choleskyQR algorithm to compute the factorization.

To finish the discussion about the deflated BCG method, we shortly want to talk about the matrix inversion $\left(\overline{P^m}^* A \overline{P^m}\right)^{-1}$ needed in each iteration. In our examples for transfer matrix computations we typically have at most $k \approx 600$, and hence $\overline{P^m}^* A \overline{P^m}$ is a square matrix of at most ca. 600 rows, and hence can be effectively inverted using a direct method. In dune-common this is implemented via a pivoted LU decomposition.[23]

[21] `https://gitlab.dune-project.org/core/dune-istl/-/blob/blockkrylov/dune/istl/blockkrylov/blockinnerproduct.hh`
[22] this follows from the fact that for $Z = (z_1, ..., z_m)$ we have $ZZ^* = \sum_{i=1}^m z_i z_i^*$
[23] `https://gitlab.dune-project.org/core/dune-common/-/blob/master/dune/common/densematrix.hh`

Remark. Using the deflated BCG method, we do not need to worry about breakdowns anymore. While the theoretical mathematician might thus deem the deflated method superior and call it a day, the author wants to point out that for example for the computation of the EEG and MEG transfer matrices using the BCG method both for the spherical head model given in [3] and the realistic head model given in [15], with various choices of matrix algebras, preconditioning and selections of electrodes and coil positions, in the computations the author performed there was **not a single breakdown** before the desired residual reduction was achieved. The author thus thinks it is justified to deliberately accept the possibility of a breakdown. One could for example simply hope that no breakdown occurs and directly use algorithm 2. Another approach, that is actually able to continue the algorithm should a breakdown occur, is to *restart* the algorithm. To see how this would be done, notice that the BCG method for $AX = B$ with initial guess X^0 is equivalent to performing the BCG method for $AX = R^0$ with initial guess 0 and adding X^0 to the iterates this produces. Now assume we can write $R^0 = W\rho$. One can then easily show that if $\widetilde{X}^m$ are the iterates produced by the BCG method applied to the equation $AX = W$, the projection solution of $AX = B$ with respect to $\mathcal{K}^m(A, R^0)$ and initial guess X^0 is given by $X^0 + \widetilde{X}^m\rho$. We can thus proceed as follows.

1. Let A, B and X^0 be given and let A be self-adjoint and positive definite. Assume we want to solve $AX = B$. Let $R^0 = W\rho$, where ρ has full rank.
2. Apply the BCG method to the equation $AX = W$ with initial guess 0. Notice that we can directly compute the projection solution X^m of $AX = B$ with respect to the initial guess X^0 in each step of the iteration by a simple update.
3. If the algorithm breaks down at step m, go to step 1 with X^m in place of X^0 and R^m in place of R^0.

We call this algorithm the *restarted block conjugate gradient method*.

We have now derived and discussed a breakdown-free algorithm for computing projection solutions with respect to block Krylov spaces corresponding to the classical matrix algebra. This also gives algorithms for the hybrid algebras, as the hybrid algebra algorithm is essentially given by parallel classical BCG methods, and global algebra algorithms, which can be viewed as classical algorithms after rearranging matrices. But before we arrive at an algorithm that is actually feasible in practice, we need to introduce one more concept. We saw in theorem 47 that the convergence bounds we were able to derive depended on the eigenvalue distribution of the underlying matrix A. Essentially, the tighter the eigenvalues are clustered, the better the convergence bound. On the other hand, if the eigenvalues are spread far apart, our bounds degrade. In this case we say that the matrix is *ill conditioned*.[24] And this is not only of theoretical importance. For example, [14] points out that directly applying Krylov methods to problems arising from practical applications like FEM typically leads to slow convergence, as the matrices are oftentimes ill conditioned. It is common knowledge that Krylov methods only get really effective if some measure is taken to remedy the ill conditioning of the stiffness matrix. Such measures are known as *preconditioning*. We now want to discuss how to incorporate preconditioning into the deflated BCG method.

4.5 Preconditioning

We want to devise a way to solve the equation $AX = B$ using the deflated BCG method, which attenuates the slow convergence coming from the ill conditioning of the matrix A. The central idea is the following. Let S be an invertible matrix. We then have

$$AX = B \iff SAS^*\left((S^*)^{-1}X\right) = SB.$$

Since the matrix $\hat{A} = SAS^*$ is again self adjoint and positive definite, we can thus solve the equation $\hat{A}Y = SB$ and then compute X via $X = S^*Y$. If we now choose S in a sensible way, we can hope that the eigenvalue distribution of $\hat{A}$ is much more tightly clustered than the eigenvalue distribution of A, leading to faster convergence. Furthermore, if we have $\widetilde{Y} \approx Y$, we also have $S^*\widetilde{Y} \approx X$, and an algorithm that produces approximates for Y thus also produces approximates for X. We thus arrive at the following approach.

1. Solve the equation $\hat{A}Y = SB$ using the deflated BCG method. This produces iterates $Y^0, Y^1, ...$
2. Compute approximations to the solution of $AX = B$ via $X^i = S^*Y^i$.

This immediately raises two questions.

1. **How do we choose S so that $\hat{A} = SAS^*$ has a more favourable spectrum than A?**
2. **How can we implement the algorithm outlined above efficiently?**

[24] Classically, *ill conditioned* referes to the *condition number* of the matrix A, which is given by $\|A\|\|A^{-1}\|$. We want to use the term in a more vague manner to simply describe a matrix whose spectrum is very spread out.

We first question can be answered easily. By standard spectral theory, we have for invertible matrices L, K that $\sigma(LK) = \sigma(KL)$. A proof can for example be found in [47], appendix A.1. Hence we have $\sigma(\hat{A}) = \sigma(S^*SA)$. Hence if we choose $S^*S \approx A^{-1}$, we have $S^*SA \approx E_n$ and $\sigma(\hat{A})$ is more tightly clustered. The central question is thus how well S^*S approximates the inverse of A, where better approximations generally lead to better convergence of the BCG method for $\hat{A}$. Due to this we call the matrix M defined by $M^{-1} = S^*S$ a *preconditioner*. Notice that we then have $M \approx A$. Furthermore, if we have a self adjoint positive definite M, we can always write $M = S^*S$. So if we have such M for which we know that $M \approx A$, we can associate it to an approach based on some SAS^*.

The next question is how we can design an efficient algorithm. From a practical point of view, the typical preconditioning approaches produce the matrices M or M^{-1} described above, and generally not a factor S with $S^*S = M$. Our algorithm should thus not need to compute S, or its actions against right hand sides, explicitly, and only rely on M. Furthermore, we generally want to avoid unnecessary computations. We are now going to derive such an algorithm. We first need a small lemma.

Lemma 58. Let M be a self adjoint and positive definite $n \times n$-matrix. Let $M = S^*S$, where S is also a $n \times n$-matrix. Let $X \in \mathbb{K}^{n\times k}$. Then for $\overline{X}, \sigma$ the following are equivalent.

1. $(\overline{X}, \sigma) = \mathrm{Orth}(X, M)$
2. $(S\overline{X}, \sigma) = \mathrm{Orth}(SX, E_n)$¸

Proof. Since S is invertible, we have $\mathrm{rank}(X) = \mathrm{rank}(SX)$ and $\mathrm{rank}(\overline{X}) = \mathrm{rank}(S\overline{X})$. Hence it suffices to show that the columns of $\overline{X}$ are M-orthonormal if and only if the columns of $S\overline{X}$ are orthonormal in the ordinary sense. But this is a direct consequence of

$$(S\overline{X})^*(S\overline{X}) = \overline{X}^* M X.$$

□

Now assume we want to solve the equation $AX = B$, and we are given some initial guess 0. Assume we have some preconditioner M, meaning M is self adjoint and positive definite. Then we can write $M^{-1} = S^*S$. Then let $\hat{A} = SAS^*$. Let $Y^0 = (S^*)^{-1}X^0$. We now want to apply the deflated BCG method to the equation $\hat{A}Y = SB$ with initial guess Y^0. Let $R_Y^0 = SB - \hat{A}Y^0$ and $R_X^0 = B - AX^0$. We then have

$$R_Y^0 = SB - SAS^*(S^*)^{-1}X^0 = SR_X^0.$$

Hence if we have $(\overline{R_X^0}, \sigma^0) = \mathrm{Orth}(R_X^0, M^{-1})$, we also have $(S\overline{R_X^0}, \sigma^0) = \mathrm{Orth}(R_Y^0, E_n)$, and hence we can choose $\overline{R_Y^0} = S\overline{R_X^0}$. We now define $\overline{P_X^0} := S^*\overline{P_Y^0} = M^{-1}\overline{R_X^0}$.

Now assume the BCG algorithm has already run for m iterations, and let $\overline{P^m}_Y, \overline{R^m}_Y, \sigma^m$ be the corresponding iterates. Furthermore, assume we have constructed $\overline{P^m}_X, \overline{R^m}_X$ so that $\overline{R^m}_Y = S\overline{R^m}_X$ and $\overline{P^m}_X = S^*\overline{P^m}_Y$. We then have

$$\overline{\alpha^m} = \left(\overline{P^m}_Y^* \hat{A} \overline{P^m}_Y\right)^{-1} = \left(\overline{P^m}_X A \overline{P^m}_X\right)^{-1},$$

and furthermore

$$Y^{m+1} = Y^m + \overline{P^m}_Y \overline{\alpha^m} \sigma^m.$$

Since our approximations to the original equation are defined by $X^l := S^*Y^l$, multiplying by S^* thus gives

$$X^{m+1} = X^m + \overline{P^m}_X \overline{\alpha^m} \sigma^m.$$

This gives

$$\widetilde{R^{m+1}}_Y = \overline{R^m}_Y - \hat{A}\overline{P^m}_Y \overline{\alpha^m} = S(\overline{R^m}_X - A\overline{P^m}_X \overline{\alpha^m}),$$

and if we define $\widetilde{R^{m+1}}_X = \overline{R^m}_X - A\overline{P^m}_X \overline{\alpha^m}$, we have $\widetilde{R^{m+1}}_Y = S\widetilde{R^{m+1}}_X$. Hence, if we compute $(\overline{R^{m+1}}_X, \gamma^{m+1}) = \mathrm{Orth}(\widetilde{R^{m+1}}_X, M^{-1})$, we know by lemma 58 that we have $(S\overline{R^{m+1}}_X, \gamma^{m+1}) = \mathrm{Orth}(\widetilde{R^{m+1}})_Y, E_n)$. We can thus choose $\overline{R^{m+1}}_Y = S\overline{R^{m+1}}_X$. This gives

$$\overline{P^{m+1}}_Y = \overline{R^{m+1}}_Y + \overline{P^m}_Y(\gamma^{m+1})^* = S\overline{R^{m+1}}_X + \overline{P^m}_Y(\gamma^{m+1})^*.$$

If we now define

$$\overline{P^{m+1}}_X = M^{-1}\overline{R^{m+1}}_X + \overline{P^m}_X(\gamma^{m+1})^*,$$

we have $\overline{P^{m+1}}_X = S^*\overline{P^{m+1}}_Y$. This means we can compute the iterate X^{m+1}, $\overline{R^{m+1}}_X$ and $\overline{P^{m+1}}_X$ with the desired properties without needing to explicitely compute $\overline{R^{m+1}}_Y$ or $\overline{P^{m+1}}_Y$ and without needing excess to S or S^*.

Algorithm 4 Preconditioned Deflated Block Conjugate Gradient Method

Require: - Positive definite matrix $A \in \mathbb{K}^{n\times n}$
- Right hand side $B \in \mathbb{K}^{n\times k}$
- Initial guess $X^0 \in \mathbb{K}^{n\times k}$
- Positive definite preconditioner $M \in \mathbb{K}^{n\times n}$

$R^0 \leftarrow B - AX^0$
$(\overline{R^0}, \sigma^0) \leftarrow \mathrm{Orth}(R^0, M^{-1})$
$\overline{P^0} \leftarrow M^{-1}\overline{R^0}$
for $m = 0, 1, 2, \ldots$ until convergence **do**
$\overline{\alpha^m} \leftarrow \left(\overline{P^m}^* A \overline{P^m}\right)^{-1}$
$X^{m+1} \leftarrow X^m + \overline{P^m} \cdot \overline{\alpha^m} \cdot \sigma^m$
$\widetilde{R^{m+1}} \leftarrow \overline{R^m} - A\overline{P^m} \cdot \overline{\alpha^m}$
$(\overline{R^{m+1}}, \gamma^{m+1}) \leftarrow \mathrm{Orth}(\widetilde{R^{m+1}}, M^{-1})$
$\sigma^{m+1} \leftarrow \gamma^{m+1}\sigma^m$
$\overline{P^{m+1}} \leftarrow M^{-1}\overline{R^{m+1}} + \overline{P^m} \cdot \left(\gamma^{m+1}\right)^*$
end for

We have thus derived the *preconditioned deflated block conjugate gradient method*, which we summarize in algorithm 4.

To implement this algorithm efficiently, note that during the computation of $(\overline{X}, \sigma) = \mathrm{Orth}(X, W)$, for W self-adjoint and positive definite, we can compute $\overline{X}$ and $W\overline{X}$, while only needing one application of W in total. To see how, assume that we have already stored X. We can then compute WX, and can then compute X^*WX. Then we compute a Cholesky decomposition $X^*WX = LL^*$, and can then compute $\overline{X}$ by setting $\overline{X} = X(P\hat{L}^{-1})^*$. But since we have already computed WX, we can then also directly compute $W\overline{X}$ via $W\overline{X} = WX(P\hat{L}^{-1})^*$, meaning we only need to rightmultiply WX with a small matrix. Hence we can implement algorithm 4 in such a way that each iteration only needs one application of the preconditioner M^{-1}. The same is true for the setup step.

Additionally note that by construction the algorithm cannot fail, at least in exact arithmetic. This is a consequence of the fact that we can write $\overline{P^m} = S^*\overline{P^m}_Y$, where $\overline{P^m}_Y$ is an iterate from the deflated BCG method, which was proven to always have full rank. Hence we know that $\overline{P^m}$ also has full rank.

Furthermore, we have for all $m \geq 0$ that $R^m := B - AX^m = \overline{R^m}\sigma^m$. For $m = 0$ this is so by construction, and by induction we have for $m + 1$

$$R^{m+1} = R^m - A\overline{P}^m\overline{\alpha^m}\sigma^m = \left(\overline{R^m} - A\overline{P}^m\overline{\alpha^m}\right)\sigma^m = \overline{R^{m+1}}\gamma^{m+1}\sigma^m = \overline{R^{m+1}}\sigma^{m+1}.$$

In the deflated BCG method without preconditioning we tested for convergence by checking the column norms of σ^m, which by the Pythagorean theorem were the same as the euclidean norms of R^m. By the same argument we have for the preconditioned case with preconditioner M that the euclidean column norms of σ^m are the same as the column norms of R^m in the norm induced by M^{-1}, since by construction the columns of $\overline{R^{m+1}}$ are M^{-1}-orthonormal. Hence we can also check for convergence by computing the euclidean column norms of σ^m and stop once the desired residual reduction is achieved. But note that when doing this **we are computing the residual reduction in the norm $\|x\|_{M^{-1}} = \sqrt{\langle M^{-1}x, x\rangle}$, and not in the euclidean norm.** Speaking from my own experience, the desired residual reduction in the norm $\|\cdot\|_{M^{-1}}$ tends to be achieved in fewer iterations than are needed for the corresponding residual reduction in the euclidean norm. Later we are going to see a concrete example of this behavior.

Looking in the literature, e.g. [14], there are typically two ways to introduce preconditioning for the ordinary conjugate gradient method. The first one is the one we have discussed above, which was derived by going to the equation $SAS^*Y = SB$. The other is given by noticing that $\langle V, W\rangle_M = V^*MW$ defines an inner product module structure on the $n \times k$-matrices with respect to the classical algebra, and with respect to this inner product the matrix $M^{-1}A$ defines a $\mathbb{S}_{\mathrm{cl}}$-linear, self-adjoint positive definite operator on the $n \times k$-matrices via left multiplication. Hence we can apply the BCG method to the equation $M^{-1}AX = M^{-1}B$. Taking a close look at the proof of theorem 47, we see that it is also valid for the inner product $\langle\cdot,\cdot\rangle_M$, which essentially follows from the fact that the spectral theorem is true for all self-adjoint maps on finite dimensional Hilbert spaces. Hence we see that the spectrum of $M^{-1}A$ determines the convergence bounds we can derive, and if $M \approx A$ we can hope for a tightly clustered spectrum and hence a fast convergence. Perhaps surprisingly, these two methods for constructing a system with a better condition turn out to produce the same iterates.

Theorem 59. Assume we want to solve $AX = B$ with A self adjoint and positive definite. Assume we have an initial guess X^0, and a preconditioner M. Let then $X^1, X^2, ...$ be the sequence produced by the preconditioned deflated BCG method. Then X^m is the projection solution of the equation $M^{-1}AX = M^{-1}B$ with respect to the initial guess X^0, the ansatz space $\mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$ and the inner product $\langle \cdot, \cdot \rangle_M$. Furthermore, we have that

$$X^m = \underset{Z \in X^0 + \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)}{\arg\min} \|X - Z\|_A,$$

where $\|Z\|_A^2 = \text{Tr}\,(Z^*AZ)$ and X is the exact solution.

Proof. To show that X^m is the projection solution we have to show that

$$X^m \in X^0 + \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$$
$$M^{-1}B - M^{-1}AX^m \perp_{\langle \cdot, \cdot \rangle_M} \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$$

Firstly, we have $X^m = S^*Y^m$, where Y^m is the projection solution corresponding to the equation $SAS^*Y = SB$ with initial guess $Y^0 = (S^*)^{-1}X^0$. Since we have $Y^m \in Y^0 + \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(\hat{A}, SR^0)$, the easily shown equality $S^*\hat{A}^nSR^0 = (M^{-1}A)^nM^{-1}R^0$ implies

$$X^m = S^*Y^m \in X^0 + S^*\mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(\hat{A}, SR^0) = X^0 + \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0).$$

Now denote by R^m_Y the residual associated to Y^m, meaning we have $R^m_Y = SR^m$. To show that $M^{-1}B - M^{-1}AX^m$ is M-orthogonal to $\mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$, it suffices to show that it is M-orthogonal to $S^*R^n_Y$ for all $n < m$, by theorem 55. By construction, we have $\overline{R^m}_Y = S\overline{R^m}$, and mutliplying by σ^m implies that we have $R^m_Y = SR^m$. This implies

$$\begin{aligned}\langle M^{-1}(B - AX^m), S^*R^n_Y \rangle_M &= (R^m)^*M^{-1}MS^*R^n_Y = (R^m_Y)^*(S^{-1})^*M^{-1}MS^*R^n_Y \\ &= (R^m_Y)^*(S^{-1})^*S^*SS^{-1}(S^{-1})^*S^*R^n_Y \\ &= (R^m_Y)^*R^n_Y = 0.\end{aligned}$$

This implies that X^m is the projection solution of $M^{-1}AX = M^{-1}B$ with respect to the ansatz space $\mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$, the inner product $\langle \cdot, \cdot \rangle_M$ and the initial guess X^0.

Since the trace is a faithful positive functional with $\text{Tr}(a^*) = \overline{\text{Tr}(a)}$, as was proven in lemma 39, we thus know by theorem 37 that X^m is the best approximation inside $X^0 + \mathcal{K}^m_{\mathbb{S}_{\text{cl}}}(M^{-1}A, M^{-1}R^0)$ to the exact solution with respect to the norm defined by

$$\|Z\|_A^2 = \text{Tr}(\langle M^{-1}AZ, Z \rangle_M) = \text{Tr}(Z^*AZ).$$

□

We have thus derived an algorithm that has the potential to attenuate the ill conditioning of the stiffness matrix arising from a FEM approach, and have seen that this algorithm still satisfies an optimality property. We want to finish the discussion of the BCG method with some remarks.

Remark. (1) : We want to use the preconditioned deflated BCG method for the computation of transfer matrices inside duneuro. The duneuro toolbox is based on the DUNE framework, as it is for example described in [48]. Hence we want to use the SIMD functionalities build inside DUNE for vectorization. These are currently implemented in such a way that the length of the vectors we want to operate on needs to be known at compile time. For example, if we want to work on $n \times k$ matrices, the block length k needs to be known at compile time. This raises the problem that the deflated method potentially needs to work on matrices of size $n \times s$ and $s \times s$, where s is the rank of the corresponding residual, but the ranks of the residual matrices are not known at compile time. The way this is currently dealt with is to store the iterates from algorithm 4 in matrices of fixed sizes $n \times k$ and $k \times k$, and just fill the superfluous matrix entries with arbitrary values. The computations inside the algorithm are then set up in such a way that the superfluous entries have no influence on the final result.

(2) : In this subsection, we derived a preconditioned version of the deflated BCG method. Using exactly the same approach, meaning going to the equation $SAS^*Y = SB$ or going to the equation $M^{-1}AX = M^{-1}B$ with the corresponding inner product, one can derive preconditioned versions of the other variants of the BCG method we discussed.

We have now derived the BCG method and studied its optimality and convergence properties deeply. We have seen that the BCG method is theoretically well founded. The next natural question, which some readers probably find way more interesting than the preceding discussion about the abstract properties of the BCG method, is then of course how well the algorithm performs in a realistic setting. We will study this in the next section.

5 Implementation and Numerical Experiments

In section 4 we derived a number of different algorithms, collectively referred to as *block conjugate gradient methods*. We are particularly interested in solving equations of the form $AX = B$, where A is self-adjoint positive definite, and $X, B \in \mathbb{K}^{n\times k}$, as this is the type of equation defining the transfer matrices. In section 4 we derived three classes of approaches for this kind of problem, which we called the *classical*, *hybrid* and *global* approach. Note that this is not an exclusive classification, as the classical approach is an edge case of the hybrid approach as well as of the global approach.

Our experiments are thus focused on the computation of transfer matrices. One goal of this thesis was to implement the computation of the transfer matrices using the BCG method into the software toolbox *duneuro*. An introduction to this toolbox can be found in [3]. We summarize the main details of the implementation.

As duneuro is based on the DUNE framework, we decided to use the dune-istl (short for ***I**terative **S**olver **T**emplate **L**ibrary*) module for linear algebra. In particular, we based our implementation on a branch of dune-istl in which the BCG method was implemented.[25] We could not use this concrete branch due to the reason that we wanted to use an AMG preconditioner. For details on this preconditioner we refer to [49]. This preconditioner essentially smoothes the problem, projects it into a coarser space, "solves" it there, and then prolongs it back to the original space and does some postsmoothing. The "solving" can then again consist of smoothing, solving a coarser system and prolonging, and this can be iterated a few times. Once the resulting linear system is small enough, it is then solved exactly. For this DUNE uses either UMFPACK[26], SUPERLU[27] or a BiCGSTAB solver[28]. Since our whole approach is based on using vectorization using DUNE's build in SIMD data types, we cannot use UMFPACK or SUPERLU, as in the current implementation these direct solvers can only work on vector right hand sides, and not on matrix right hand sides. It would surely be possible to adapt these solvers to be able to work on matrix right hand sides, but this thesis needs to be finished at some point, and hence this will be a task for future work. Hence we use the fallback BiCGSTAB solver as a coarse solver. The problem with this approach is that the default implementation only solves on the coarse grid up to a residual reduction of 10^{-4}, which leads to slow convergence or even failure to converge of the BCG solver using the AMG preconditioner. The author thus recommends changing this value to a smaller one. In the following numerical experiments, we always used a residual reduction factor of 10^{-14} for the solver on the coarse grid.

Furthermore, DUNE supports different forms of vectorization. We always used the LoopSIMD type with double entries.[29] In the current implementation, the dune-istl branch mentioned above only implements the hybrid matrix algebras $\mathbb{S}_{\text{hy}}^{p,q}$ for the BCG method. In particular, the global algebras are not implemented. This is mainly due to the fact that the global method for parameters p, q converges slower than the corresponding hybrid method, which is a consequence of theorem 50 and table 4, while only being a little less computationally expensive. In practice, it is thus better to choose the hybrid algebra instead of the global algebra given some fixed parameter p. For a more detailed discussion we refer to [25] and [43].

We have now implemented the computation of the EEG transfer matrix, the MEG transfer matrix and the combined computation of the EEG and MEG transfer matrices for the continuous Galerkin FEM approach inside duneuro. The implementation is based on the "2.7-related-changes" branch of duneuro. Furthermore, we added a branch to duneuro-tests that illustrates the usage of these methods.[30] A much more in depth discussion of the implementation can be found in the corresponding merge request.[31]

5.1 Investigating the Choice of Parameters

The first question we need to talk about is what version of the BCG method we want to use for the computation of the transfer matrices. Given some head model with e electrodes and c coils, we need to solve $e + c - 1 =: k$ linear systems, meaning we need to solve a system $AX = B$, where $A \in \mathbb{R}^{n\times n}$ and $X, B \in \mathbb{R}^{n\times k}$. Here n is the number of degrees of freedom of the underlying FEM approach. Via dune-istl we have access to the BCG methods for all hybrid algebras $\mathbb{S}_{\text{hy}}^{p,q}$, where $p \cdot q = k$. Since once p is given, we have $q = \frac{k}{p}$, it suffices to give the parameter p to describe the algebra. Thus for the rest of this section we simply write $\mathbb{S}_{\text{hy}}^{p}$ for the corresponding hybrid algebra. We thus need to choose some p for our implementation, and our first investigation will be on the topic of what might be a suitable choice for p.

[25] https://gitlab.dune-project.org/nils.dreier/dune-istl/-/tree/release2.7+blockkrylov
[26] [50]
[27] [51]
[28] [14], 7.4.2
[29] https://gitlab.dune-project.org/core/dune-common/-/blob/master/dune/common/simd/loop.hh
[30] https://gitlab.dune-project.org/duneuro/duneuro-tests/-/tree/blockkrylov-transfer-tests
[31] https://gitlab.dune-project.org/duneuro/duneuro/-/merge_requests/85

The literature on this topic is somewhat inconclusive, and what is there does not really fit the situation inside duneuro. In [25], this question was investigated for a matrix arising from a finite difference discretization of a 2D Poisson problem with 128 random right hand sides, and an incomplete Cholesky preconditioner was used. The experiments in this paper were run on an Intel Skylake-SP Xeon Gold 6148 CPU, which is the same CPU we are going to run our experiments on later. The results of this paper were that the choice $p = 16$ was optimal, and there was a massive difference between the different choices of p. For example, the algorithm for $p = 128$ ran about 4 times slower than the algorithm for $p = 16$. In [43], the different version of the BCG method, named *BCG with Adaptive Residual Re-Orthonormalization*, was tested. There, different values of p were tested for different matrices, and every time an incomplete Cholesky preconditioner was used, and the system was solved for 256 random right hand sides. In those experiments sometimes $p = 1$ performed best, sometimes $p = 256$ performed best, and most of the time $p = 32$ performed best. In the experiments there we again see a massive variance in the number of iterations and the time needed to converge, depending on p. For example, going from $p = 1$ to $p = 32$ reduced the number of iterations needed to achieve the desired residual reduction by a factor of ≈ 10 for most matrices.

While the authors cited above chose to employ an incomplete Cholesky preconditioner, we decided to use an AMG preconditioner for transfer matrix computations inside duneuro. This is due to the observation that multigrid methods, in general, tend to outperform all other forms of preconditioners, as it is for example described in [27]. But when using an AMG preconditioner, we expect a different dependence on the parameter p as it is observed in the literature above. We will discuss the "why" by looking at the classical method. We take a look at the k-th component of the error in iteration m.[32] Due to lemma 44, the classical BCG method produces an approximation that minimizes the error over all expressions of the form

$$P_k(A)E_k^m + \sum_{j=1}^{k-1} Q_j(A)R_j^0,$$

where P_k is a polynomial of degree m with $P_k(0) = 1$ and $Q_1, ..., Q_{k-1}$ are arbitrary polynomials. Compare this with the classical conjugate gradient method, which minimizes the error in the m-th iteration over all expressions of the form

$$P_K(A)E_k^m$$

where P_k is a polynomial of degree m with $P_k(0) = 1$. Thus when going from the ordinary CG method to the BCG method, we can improve potential approximations from $\mathcal{K}^m(A, R_k^0)$ by adding some optimal element from the (typically low dimensional) space $\sum_{j=1}^{k-1} \mathcal{K}^m(A, R_j^0)$. Thus, BCG can be thought of as improving the CG method by removing a low dimensional component from the error. This is also reflected in theorem 47, where we derived a convergence bound by showing that we could essentially ignore components of the error that correspond to outliers in the spectrum. This is in some way close to the functionality of an AMG preconditioner. This preconditioner essentially tries to smooth the error and then solves a low dimensional approximation of the linear problem. This low dimensional approximation is then used as a so-called *coarse grid correction* to improve the initial approximation. Thus the AMG preconditioner can be thought of as removing low dimensional error components that are difficult to smooth, and dampening the remaining components. But this statement could equally well be used to describe the proof of theorem 47. In total, we see that an AMG preconditioner and a BCG method perform a similar job. Considering this, we do not expect the different versions of the BCG method to have a huge impact on the convergence when using an AMG preconditioner, as the AMG preconditioner already introduces a low dimensional correction.

Furthermore, looking at the definition of the transfer matrix right hand sides in definition 16, we see that there are many right hand sides that are quite similar. The BCG method produces the best approximation possible inside the space $X_k^0 + \sum_{j=1}^{k} \mathcal{K}^m(A, R_j^0)$. Thus we can also interpret the BCG method as computing approximations in "enriched" Krylov spaces. But if the Krylov spaces are somewhat similar, this enrichment might not add much new information, and in this case we do not expect the approximation from the "enriched space" to be significantly better than the approximation from the non-enriched space.

Thus we do not expect the iteration count for different choices of p to vary as drastically as it did in the literature cited above. Even more generally, we do not expect the BCG method to depend strongly on the choice of p. In particular, we do not expect the BCG method to lead to a large speedup compared to the CG method because of faster convergence, as one might hope looking at theorem 47. Instead, we expect the BCG method to lead to a speedup because it can utilize vectorization and has a better cache usage than the CG method, as described in section 3.

[32] The choice of the k-th component is of course arbitrary.

But of course, we still expect some variance for different choices of p. And this is what we investigated. To do this, we computed the stiffness matrix corresponding to a continuous Galerkin FEM approach for the realistic tetrahedral head model given in [15]. The head model is illustrated in figure 5.

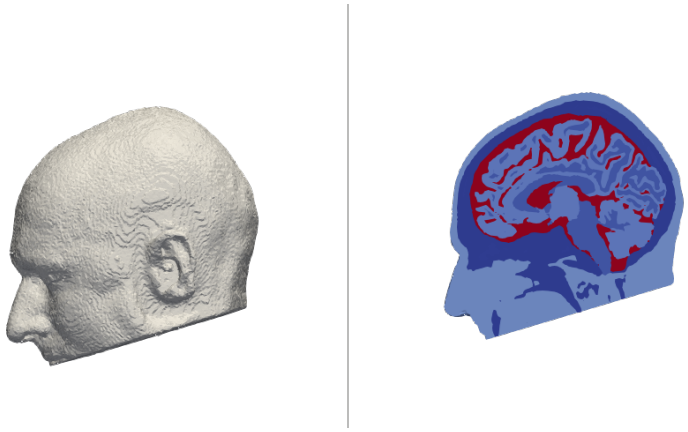

Figure 5: Realistic head model used to compute the stiffness matrix

Let A be the corresponding stiffness matrix. The corresponding FEM approach contains 885214 degrees of freedom. We then filled a matrix right hand side $B \in \mathbb{R}^{885214\times 512}$ and solved the equation $AX = B$. We did this first by solving the equation column-wise using a conjugate gradient method. Then we solved the system using the BCG method for $p \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$. We used an AMG preconditioner for every solver. The computations were performed on a single core of an Intel Skylake-SP Xeon Gold 6148 CPU. The results are illustrated in figure 6 and table 5.

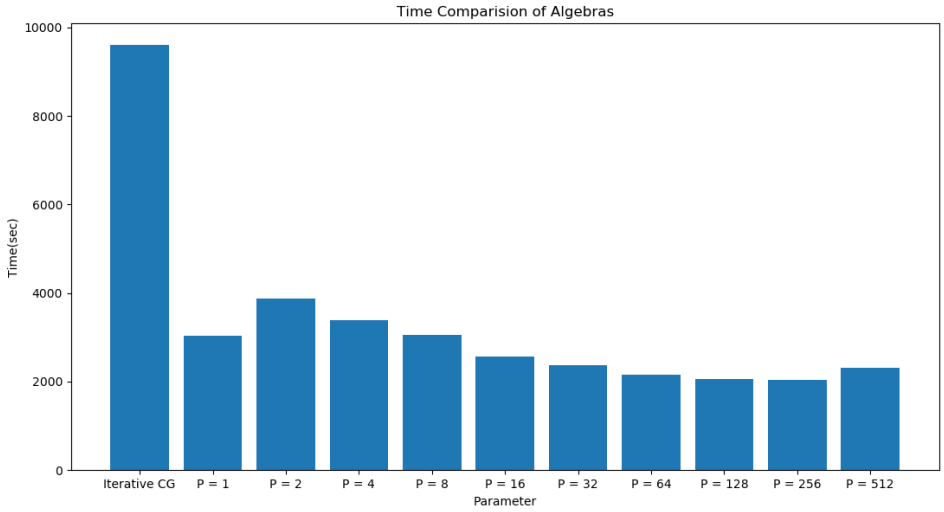

Figure 6: Comparision of different hybrid algebras, parametrized by p, and the iterative application of the ordinary CG method

Setting	CG	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$	$P = 32$	$P = 64$	$P = 128$	$P = 256$	$P = 512$
Iterations	31	25	23	21	19	17	15	14	13	12	11

Table 5: Comparision of Iterations needed to achieve a residual reduction of 10^{-10}

These results fit our expectations. We see that we have a massive speedup when going from iteratively applying the CG method to a variant of the BCG method, but there is not much variance in between the variants of the BCG method, at least when compared to [25] and [43]. We thus see that the main speed up comes from the utilization of vectorization and a more effective cache usage, which dominates the speed up from faster convergence. Furthermore, we see that $p = 128$ or $p = 256$ seems to be the optimal choice in this case. The author recommends to choose $p = 128$, since using $p = 256$ or greater can lead to a stack overflow on many

machines.[33] The system can even be solved a little bit faster by using the observation made in [25], that instead of solving $AX = B$ with X, B having 512 columns using the hybrid algebra $\mathbb{S}_{\text{hy}}^{128}$, it is a little bit more efficient to split $B = (B_1, B_2, B_3, B_4)$ with B_i having 128 columns each, and then solving $AX_i = B_i$ with the classical BCG approach. Notice that this is equivalent to solving $AX = B$ using the BCG method corresponding to the hybrid algebra $\mathbb{S}_{\text{hy}}^{128}$. The difference is that when the right hand side has fewer columns, more rows of the right hand side fit into the cache, leading to a more efficient matrix multiplication. A more detailed discussion of this behavior can be found in [25]. Using this splitting approach, the linear system is solved in 1984 seconds. As solving the system by iteratively applying the ordinary CG method took 9604 seconds, we see that using the BCG method leads to a speedup of a factor of 4.8×. We want to note that the concrete factor is only a rule of thumb, as the concrete execution time depends on a lot of different influences, e.g. what other processes are running.

Looking at table 5, we make two observations. First, we see that the number of iterations goes down as p goes up, in accordance with theorem 51 and table 4. Furthermore, just as we expect, the iteration count does not decrease drastically, but only goes down by 1 or 2 each time p is doubled, hence showing that the convergence does not depend as strongly on the choice of matrix algebra as it does when using a weaker preconditioner. Secondly, notice that the CG method needs 31 iterations on average to converge, while the $p = 1$ case converged in 25 iterations. Since the $p = 1$ case is just a data parallel version of iteratively applying the ordinary CG method, both methods produce the same iterates with the same residuals. The only difference is that the ordinary CG method wants to achieve the residual reduction in the euclidean norm, while the BCG method with $p = 1$ wants to achieve the residual reduction in the norm induced by the preconditioner. We already mentioned in section 4.5 that the residual reduction in the preconditioned norm tends to be achieved a few iterations before the corresponding residual reduction in the euclidean norm, and here we see an example of this behavior. Notice that this means that at least in the case $p = 1$ we can attribute a speedup factor of 1.24× simply to the fact that we perform fewer iterations. It would be an interesting question to investigate how many iterations the other choices for P would need to achieve the desired residual reduction in the euclidean norm instead of in the preconditioned norm, and hence how much of their speedup can be attributed to simply performing fewer iterations than an algorithm based on residual reduction in the euclidean norm would need to achieve the desired reduction. Due to time constraints, this will be a topic of future work.

The result of this investigation is the following heuristic for solving systems $AX = B$ using the BCG method with an AMG preconditioner on an Intel Skylake-SP Xeon Gold 6148 CPU with enough RAM. We split the right hand side $B = (B_1, ..., B_{l-1}, B_l)$, where $B_1, ..., B_{l-1}$ have 128 columns each an B_l has at most 128 columns. We then solve the systems $AX_i = B_i$, $i = 1, ..., l$ using the classical BCG method.

Note that this strongly depends on the machine the code is running on. For example on the author's laptop, this heuristic would lead to a terrible performance. This is because this laptop is only equipped with 8 GB of RAM. Now a single block vector $P \in \mathbb{R}^{885214\times 128}$, where the entries are stored as double values, takes up almost 1 GB of memory. Since multiple such vectors are needed for the BCG method, the memory required exceeds the memory available. The system then uses the swap space for storing data, and page faults dominate the runtime of the algorithm. Furthermore, if the system only supports vectorization with smaller vector length, the minima in figure 6 might not be at 128/256, but instead at some earlier value for P, since the increasing cost of performing the matrix computations might then start to dominate earlier. There is really no way to give a one fits all heuristic for choosing a variant of the BCG method. The most sound approach is to perform the same experiment as the one underlying figure 6 on the machine of interest and deduce the optimal choice of parameters from the results. But if the goal is to make duneuro and its usage accessible to a broad community, it might not be reasonable to expect the user to carry out such an experiment, and it might make sense to use a small value of p, perhaps even $p = 1$, as a default. Since the variance between the different p's is small, we do not lose too much efficiency by doing this, and we minimize the possibility of overusing the system's resources. One could then still leave in the option for the user to choose a p himself to finetune the method.

5.2 Comparision of CG and BCG

The next question we want to investigate is how well the BCG method performs for the computation of the transfer matrices. For this, we took the head model, electrode positions, coil positions and coil orientations given in [15]. The head model with the electrodes and coil positions is illustrated in figure 7.

This model consists of 73 electrode positions and 555 coil positions. Duneuro already implements methods to compute the EEG and MEG transfer matrices. We now want to compare these methods to our own implementations based on the BCG method. We always use an AMG preconditioner and employ the BCG method in accordance with the heuristic described above. We implemented methods for the separate computation of the

[33] If one wishes to use $p = 256$ or greater, one typically has to increase the stack size. This can for example be done from the command line using the *ulimit* command.

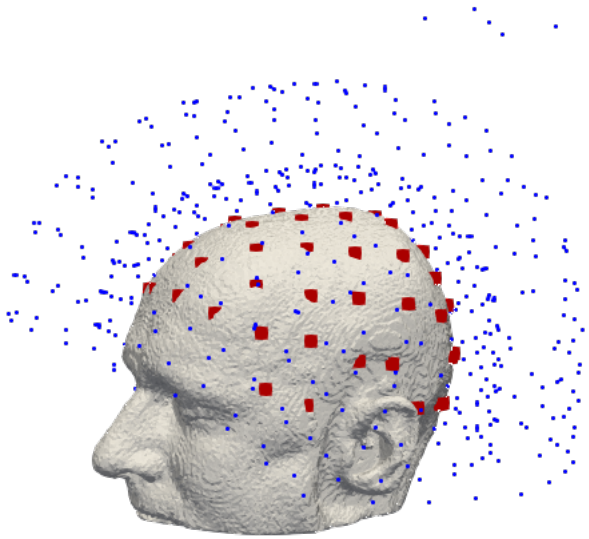

Figure 7: Head model used in the transfer matrix computations. Red points are electrode positions, blue points are coil positions.

EEG and MEG transfer matrices and the combined computation of the EEG and MEG transfer matrices. To differentiate, we call the methods already implemented in duneuro the *scalar* methods, as they operate on right hand sides whose rows are scalars, and the methods using the BCG algorithm *SIMD* methods. Furthermore, we only measure the time it takes to actually solve the linear system defining the transfer matrix. In particular, we do not measure "preprocessing" steps like reading the mesh and assembling the stiffness matrix and right hand sides. Meaning, once all variables for the system $AX = B$ are set up, we only measure the time it takes to solve this system. To compute the EEG and MEG transfer matrices with the scalar methods, we first computed the EEG transfer matrix and then the MEG transfer matrix using the corresponding scalar methods, as there is no scalar method implemented into duneuro to compute both transfer matrices at the same time. The calculations were performed on a single core of an Intel Skylake-SP Xeon Gold 6148 CPU. The results are shown in table 6.

	EEG transfer matrix	MEG transfer matrix	EEG and MEG transfer matrix
Scalar	1890	12665	14555
SIMD	373	2672	3008
Speed up	5.0×	4.7×	4.8×

Table 6: Performance comparision of the scalar and SIMD approaches for the computation of the transfer matrices. Times are given in seconds.

We thus see that in this example the time needed to compute the EEG and MEG transfer matrices gets reduced from about 4 hours to about 50 minutes. To give the reader a sense of scale, the computation of the stiffness matrix took about 20 seconds for both the scalar and the SIMD approach. The assembly of the EEG right hand sides took less than a second in both cases, which is clear when looking at the definition of these right hand sides in definition 16. The assembly of the MEG right hand sides took about 12 minutes for both the scalar and the SIMD approach, with the SIMD approach being slightly faster. Thus the assembly of the MEG right hand side makes up about 6% of the time needed to solve the linear system with the scalar approach, but it makes up about 25% percent of the time needed to solve the linear system using the SIMD approach. Thus it might now be a worthwhile occupation of one's time to try to optimize the MEG right hand side assembly. Note that for the computation of the EEG transfer matrix we assume that we already know the elements of the mesh containing the electrodes. Finding these for all electrodes takes about 20 seconds.

The computations mentioned before were performed on an Intel Skylake-SP Xeon Gold 6148 CPU. This is an expensive high end CPU supporting 512 bit vectorization. Many people might not have access to such a CPU,

and it is a natural question to ask if lower end CPUs are still able to benefit from the BCG method for the computation of transfer matrices. This is what we want to investigate now.

Almost all modern processors support some form of vectorization, although 512 bit vectorization is typically only found on high end Intel CPUs.[34] But even though many processors do not support 512 bit vectorization, the majority of processors, even those made for laptops, support 256 bit vectorization, and even the cheapest processors support 128 bit vectorization. As the main benefit from the BCG method is its possibility to utilize vectorization, we can expect the BCG method to lead to a speed up on almost all modern processors. For example the author's laptop is equipped with an Intel Core i5-6200U CPU. This CPU supports 256 bit vectorization. Performing the same experiment as in section 5.1 shows that in this case the parameter $p = 64$ is optimal. Comparing the scalar and SIMD approaches for computations of the transfer matrices then shows that a speed up of a factor of about $3\times$ is obtained. We thus see that the BCG method can also lead to a substantial speed up on lower end CPUs.

We now take a look at computing transfer matrices for the spherical head model given in [3]. As this paper only supplies electrode positions, we will only compare the computation of the EEG transfer matrix. Running the computation of the transfer matrices on a single core of an Intel Skylake-SP Xeon Gold 6148 CPU and on a single core of an Intel Core i5-6200U CPU, we get the following result.

	Xeon	i5-6200U
Scalar	52.61 sec	96.4 sec
SIMD	18.37 sec	36.74 sec
Speed up	2.9×	2.6×

Table 7: Comparision of CG and BCG for the computation of the EEG transfer matrix for a spherical head model.

We see that the speed up on the Xeon CPU is much smaller than what we would expect based on the realistic case discussed above. We suppose this is due to the fact the mesh used for the FEM approach is much smaller than the mesh for the realistic head model. The mesh consists of 54771 nodes, and the corresponding stiffness matrix contains 782093 nonzero entries. Thus the stiffness matrix takes up about 12.5 MB of memory when stored in the compressed row storage format. Furthermore, a vector of 54771 double entries takes up about 450 KB of memory. As the L3 cache of the Xeon CPU has a capacity of $27,5$ MB, we see that the ordinary CG method for this stiffness matrix can store all its data inside the L3 cache. Hence the improved cache efficiency of the BCG method does only play a reduced role, and the method can probably even utilize vectorization to some extent. This explains why employing the BCG method only leads to a comparatively small speed up when compared to a realistic head model. To test this hypothesis, we generated a mesh of the same sphere consisting of smaller tetrahedra. This new mesh consists of 993165 nodes, and thus contains more nodes than the realistic head model discussed above and the corresponding stiffness matrix cannot be contained inside the cache. We then used the same electrodes as in the computation underlying table 7. Measuring the time for the scalar and SIMD approaches, we get a speed up of a factor of $4.6\times$, which is comparable to the speedup in the realistic case. This supports our assumption that the relatively small speed up observed above is due to the small size of the underlying mesh and associated cache effects.

In total, we see that for the computation of the transfer matrices the BCG method leads to a substantial speedup over the ordinary CG method. This speedup can be achieved on high end and low end processors, and for spherical head models as well as for realistic head models.

6 Summary and Outlook

In this thesis, we gave a thorough introduction to the transfer matrix approach, including giving exact formulas for the linear systems defining these matrices and a discussion about the implementation of the right hand side assembly. Furthermore, we discussed the often overlooked problem of the stiffness matrix coming from a CG FEM approach not being invertible and investigated different approaches for dealing with this problem. After having described how to correctly define the transfer matrices, we discussed the current implementation of their computation inside duneuro, and illustrated why this implementation cannot exploit vectorization. For this, we gave an introduction to vectorization, and discussed at length what prohibits algorithms from utilizing the potential of SIMD instructions, and how algorithms that can utilize vectorization should be designed. We applied these principles to give a surface level introduction into the problem of effectively implementing matrix

[34] At least at the moment of writing this thesis.

multiplication, and saw that a properly implemented matrix multiplication algorithm can utilize vectorization. This motivated us to pursue the design of an algorithm for computing transfer matrices based on matrix-matrix operations. We found such an algorithm by taking a close look at the conjugate gradient method and generalizing its core features to a more general setting. In this general setting, we then derived the natural generalization of the conjugate gradient method, which we called the block conjugate gradient method, and saw that certain instances of this generalized conjugate gradient method are based on matrix-matrix operations, thus giving us an algorithm of the desired form. We then put a lot of work into analyzing the theoretical properties of the block conjugate gradient algorithm, and showed that it shares many properties with the ordinary conjugate gradient method. We then introduced special instances of the BCG method, specifically tailored to the transfer matrix problem, and studied their convergence properties in more detail. A highlight of this discussion was the derivation of a convergence bound depending on the clustering of the eigenvalues, but independent of outlier eigenvalues. We then discussed how to handle the classical problem of rank deficient residuals, arriving at a deflated algorithm. At the end of the theoretical considerations of the BCG method, we then derived a preconditioned version of the deflated BCG algorithm. After having motivated, derived and studied the BCG algorithm we then tested it in a practical setting. For this, we first investigated how to choose the parameter of the hybrid BCG method, which we can set in dune-istl. We then compared the BCG method for the computation of the transfer matrices with the methods already implemented inside duneuro, which are based on iteratively applying the ordinary CG method, and saw that the BCG method leads to a significant speedup on almost all machines and for spherical head models as well as for realistic head models.

During this thesis, a number of questions and tasks arose which could be the topic of future work. We want to note a few that the author finds particularly interesting.

- In section 2.3 we saw that the BiCGSTAB solver on the coarse grid of an AMG preconditioner failed to converge for one electrode. While working on this thesis there were actually a couple of times that this coarse solver failed, for different matrices and while serving as a preconditioner for different solvers. It was suggested to the author to instead try a GMRES solver on the coarse grid. Furthermore one could adapt the direct solvers to also be able to handle block right hand sides. It should be investigated if this solves the issue.
- In definition 16 the MEG transfer matrix was defined for general quadrature rules on the coils. At the moment, the BCG method for computing the MEG transfer matrix only works for one point quadrature rules. At some point in the future, the general definition of the MEG transfer matrix should also be implemented.
- In theorem 37 we showed that the iterates produced by the BCG method are optimal approximations to the exact solution with respect to some pre-Hilbert space norm, given that the underlying algebra admits a faithful positive functional f with $f(a^*) = \overline{f(a)}$. On all algebras that were relevant for this thesis it was straightforward to construct such a functional, but it would be interesting to investigate under what conditions such a functional is guaranteed to exist. It is for example a direct corollary of [47] 3.7.2 that every separable complex C^*-algebra admits such a functional.
- At the moment the computation of the transfer matrices using the BCG method is only implemented into duneuro for the continuous Galerkin FEM approach. At some point in the future, it should also be made accessible for the other FEM approaches implemented in duneuro.

At the moment of writing this thesis, it is not clear if the blockkrylov branch of dune-istl, which is the branch implementing the BCG method, will be merged into the master branch of dune-istl. This presents us with a problem, as at some point in the future the blockkrylov branch might become incompatible with more recent versions of DUNE, or it might miss some new features. One possible workaround might be to fall back to the case $P = 1$, which is essentially a data parallel version of the ordinary CG method, and can be implemented without needing to implement the sophisticated algebraic background of the BCG method, as it is done inside the blockkrylov branch. Another approach might be to maintain a version of the BCG method inside duneuro, should it happen that modern versions of dune-istl no longer support it. How to deal with this possibility might be a topic for future discussion.

A Chebyshev polynomials

For the convergence analysis of the block conjugate gradient algorithm, we need a few facts about Chebyshev polynomials. All necessary information about these polynomials is assembled in this appendix. Our presentation is based on [14].

Definition 60. We define $T_0(x) = 1$ and $T_1(x) = x$. For $k \geq 2$ we define

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x). \tag{17}$$

Then we call T_k the *k-th Chebyshev polynomial*.

Notice that T_k is a polynomial of degree k. Furthermore, T_k has the same parity as k.

We first give a closed expression for the Chebyshev polynomials.

Theorem 61. For $|x| \leq 1$ we have

$$T_k(x) = \cos(k \cdot \arccos(x)).$$

For $|x| \geq 1$ we have

$$T_k(x) = \frac{1}{2}\left(\left(x + \sqrt{x^2-1}\right)^k + \left(x + \sqrt{x^2-1}\right)^{-k}\right).$$

Proof. Let $S_k(x) = \cos(k \cdot \arccos(x))$. Then $S_0 = T_0$ and $S_1 = T_1$ on $[-1, 1]$. If we can show that the recurrence relation 17 is also valid for S_k, this implies that $S_k = T_k$ on $[-1, 1]$ for all k. We thus have to show that for all $k \geq 2$ and $x \in [-1, 1]$ we have

$$S_k(x) = 2xS_{k-1}(x) - S_{k-2}(x),$$

which is equivalent to

$$S_k(x) + S_{k-2}(x) = 2xS_{k-1}(x).$$

It is a standard fact that for all $\varphi, \psi \in \mathbb{R}$ we have

$$\cos(\varphi + \psi) = \cos(\varphi)\cos(\psi) - \sin(\varphi)\sin(\psi).$$

Together with $\sin(-x) = -\sin(x)$ this implies

$$\begin{aligned} S_k(x) + S_{k-2}(x) &= \cos(k \arccos(x)) + \cos((k-2)\arccos(x)) \\ &= \cos((k-1)\arccos(x) + \arccos(x)) + \cos((k-1)\arccos(x) - \arccos(x)) \\ &= 2x\cos((k-1)\arccos(x)) = 2xS_{k-1}(x). \end{aligned}$$

We have thus proven the first claim in the theorem.

Noticing that $\left(x + \sqrt{x^2-1}\right)^{-1} = x - \sqrt{x^2-1}$, the second claim is clear for $k = 0$ and $k = 1$. For $k \geq 1$, notice that

$$\begin{pmatrix} T_{k+1}(x) \\ T_k(x) \end{pmatrix} = \begin{pmatrix} 2x & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} T_k(x) \\ T_{k-1}(x) \end{pmatrix},$$

and hence

$$\begin{pmatrix} T_{k+1}(x) \\ T_k(x) \end{pmatrix} = \begin{pmatrix} 2x & -1 \\ 1 & 0 \end{pmatrix}^k \cdot \begin{pmatrix} T_1(x) \\ T_0(x) \end{pmatrix}.$$

Assume $|x| > 1$, as the case $|x| = 1$ is obvious. Let $\lambda_1 = x + \sqrt{x^2-1}$ and $\lambda_2 = x - \sqrt{x^2-1}$. Diagonalizing in the standard way yields that for

$$S = \begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix}$$

we have

$$S^{-1} = \frac{1}{\lambda_1 - \lambda_2}\begin{pmatrix} 1 & -\lambda_2 \\ -1 & \lambda_1 \end{pmatrix},$$

and

$$\begin{pmatrix} 2x & -1 \\ 1 & 0 \end{pmatrix} = S\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}S^{-1}.$$

This implies

$$\begin{pmatrix} T_{k+1}(x) \\ T_k(x) \end{pmatrix} = \frac{1}{\lambda_1 - \lambda_2}\begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} \lambda_1^k & 0 \\ 0 & \lambda_2^k \end{pmatrix}\begin{pmatrix} 1 & -\lambda_2 \\ -1 & \lambda_1 \end{pmatrix}\begin{pmatrix} x \\ 1 \end{pmatrix}.$$

Computing the product on the right hand side gives the second claim. This computation is straightforward, using the easily verifyable facts that $x\lambda_1 - 1 = \frac{\lambda_1 - \lambda_2}{2}\lambda_1$ and $1 - x\lambda_2 = \frac{\lambda_1 - \lambda_2}{2}\lambda_2$. □

We can use these explicit representations to derive all necessary facts about Chebyshev polynomials.

Corollary 62. For $|x| \geq 1$ we have for all $k \geq 0$

$$T_k(x) \geq \frac{\left(x + \sqrt{x^2 - 1}\right)^k}{2}.$$

In particular, for $0 < a < b$ we have

$$T_k\left(\frac{a+b}{a-b}\right) \geq \frac{1}{2}\left(\frac{\sqrt{\frac{b}{a}} + 1}{\sqrt{\frac{b}{a}} - 1}\right)^k.$$

Proof. This first statement is a direct consequence of the second claim of theorem 61. Applying this to $x = \frac{a+b}{b-a}$ yields

$$T_k\left(\frac{a+b}{a-b}\right) \geq \frac{1}{2}\left(\frac{a+b}{b-a} + \sqrt{\left(\frac{a+b}{b-a}\right)^2 - 1}\right)^k.$$

Computing the term inside the braces on the right hand side gives the second claim. □

For some intervale $[a, b] \subset \mathbb{R}$, let $\|f|_{[a,b]}\|_\infty = \sup_{x\in[a,b]} |f(x)|$ denote the supremum norm on $[a, b]$. In the convergence analysis of the (block) conjugate gradient method, the following question naturally arises.

Let $[a, b] \subset \mathbb{R}$ be an interval and let $\gamma \notin [a, b]$. Let p be a polynomial of degree $\leq k$ with $p(\gamma) = 1$. What is the smallest possible value of $\|p|_{[a,b]}\|_\infty$?

This question can be answered using Chebyshev polynomials. The goal for the rest of this appendix is to derive this answer.

Lemma 63. We have $\|T_k|_{[-1,1]}\| = 1$. Furthermore, if we define for $j = 0, ..., k$

$$x_j = \cos\left(\frac{k-j}{k}\pi\right),$$

then $-1 = x_0 < x_1 < ... < x_k = 1$ and

$$T_k(x_j) = (-1)^{k-j}.$$

Proof. Using theorem 61, we have for $|x| \leq 1$

$$|T_k(x)| = |\cos(k \arccos(x))| \leq 1,$$

and hence $\|T_k|_{[-1,1]}\| \leq 1$. Since $\cos : [0, \pi] \to [-1, 1]$ is strictly decreasing, we have $-1 = x_0 < x_1 < ... < x_k = 1$. This implies

$$T_k(x_j) = \cos(k \arccos(\cos\left(\frac{k-j}{k}\pi\right))) = \cos((k-j)\pi) = (-1)^{k-j}.$$

This implies $\|T_k|_{[-1,1]}\| = 1$. □

Corollary 64. For $|x| > 1$ we have $T_k(x) \neq 0$.

Proof. Lemma 63 together with the intermediate value theorem implies that T_k has k roots inside the interval $[-1, 1]$. As T_k is a polynomial of degree k, these have to be all roots of T_k. □

Lemma 65. Let P_k denote the space of all polynomials of degree at most k. Let $\gamma \notin [-1, 1]$. Then we have

$$\inf_{p\in P_k, p(\gamma)=1} \|p|_{[-1,1]}\|_\infty = \frac{1}{|T_k(\gamma)|}.$$

This infimum is realized by $\frac{T_k}{T_k(\gamma)}$.

Proof. First note that by lemma 63 we have $\|\frac{T_k}{T_k(\gamma)}|_{[-1,1]}\|_\infty = \frac{1}{|T_k(\gamma)|}$. This shows that

$$\inf_{p\in P_k, p(\gamma)=1} \|p|_{[-1,1]}\|_\infty \leq \frac{1}{|T_k(\gamma)|}.$$

Now assume there exists some $p \in P_k$ with $p(\gamma) = 1$ and $\|p|_{[-1,1]}\|_\infty < \frac{1}{|T_k(\gamma)|}$. Then let $q = T_k(\gamma) \cdot p$. Then $\|q|_{[-1,1]}\|_\infty < 1$, $q(\gamma) = T_k(\gamma)$ and q is a polynomial of degree k. Let $x_0, ..., x_k$ be defined as in lemma 63. Let $l = T_k - q$. Then l is a polynomial of degree at most k, and by lemma 63 and since $|q(x_i)| < 1$, l changes its sign between x_i and x_{i+1} for $0 \leq i \leq k-1$, and $l(x_i) \neq 0$ for all $0 \leq i \leq k$. Hence by the intermediate value theorem there has to be a root ξ_i of l inside the interval (x_i, x_{i+1}) for all $0 \leq i \leq k-1$. Hence l has k roots inside the interval $[-1, 1]$. Furthermore we have $l(\gamma) = T_k(\gamma) - q(\gamma) = 0$, and hence l is a polynomial of degree at most k with at least $k+1$ roots. Hence we have $T_k = q$, which is a contradiction to $\|q|_{[-1,1]}\|_\infty < 1 = \|T_k|_{[-1,1]}\|_\infty$. □

Now we can finally prove the desired theorem. But first, let us note that given some interval $[a, b]$ with $a < b$ the maps

$$[-1, 1] \to [a, b]; x \mapsto \frac{b-a}{2}x + \frac{b+a}{2}$$

and

$$[a, b] \to [-1, 1]; x \mapsto \frac{2x-b-a}{b-a}$$

are bijections and inverses of each other.

Theorem 66. Let $[a, b]$ with $a < b$ be an interval, and let $\gamma \notin [a, b]$. Let P_k denote the polynomial of degree at most k. Define

$$T_{k,[a,b],\gamma}(x) = \frac{T_k\left(\frac{2x-b-a}{b-a}\right)}{T_k(\frac{2\gamma-b-a}{b-a})}.$$

Then $T_{k,[a,b],\gamma}$ is a polynomial of degree k with $T_{k,[a,b],\gamma}(\gamma) = 1$, and we have

$$\inf_{p\in P_k, p(\gamma)=1} \|p|_{[a,b]}\|_\infty = \|T_{k,[a,b],\gamma}|_{[a,b]}\|_\infty = \frac{1}{|T_k(\frac{2\gamma-b-a}{b-a})|}.$$

Furthermore, let $d = \max\{a - \gamma, \gamma - b\} > 0$. Then

$$\|T_{k,[a,b],\gamma}|_{[a-d,b+d]}\|_\infty = 1.$$

Proof. Let $f : A \to B$ be a bijection. Then, for some $g : B \to \mathbb{R}$, we obviously have

$$\sup_{x\in A} |g \circ f(x)| = \sup_{y\in B} |g(y)|.$$

Then by construction we have

$$\inf_{p\in P_k, p(\gamma)=1} \|p|_{[a,b]}\|_\infty \leq \|T_{k,[a,b],\gamma}|_{[a,b]}\|_\infty = \sup_{x\in[-1,1]} \left|\frac{T_k(x)}{T_k(\frac{2\gamma-b-a}{b-a})}\right| = \frac{1}{|T_k(\frac{2\gamma-b-a}{b-a})|}.$$

Now let $p \in P_k$ with $p(\gamma) = 1$. Then $q(x) = p\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)$ is a polynomial of degree at most k with $q\left(\frac{2\lambda-b-a}{b-a}\right) = p(\gamma) = 1$, and $\frac{2\gamma-b-a}{b-a} \notin [-1, 1]$. Hence by lemma 65 that

$$\|p|_{[a,b]}\|_\infty = \|q|_{[-1,1]}\|_\infty \geq \frac{1}{|T_k(\frac{2\gamma-b-a}{b-a})|}.$$

This implies

$$\inf_{p\in P_k, p(\gamma)=1} \|p|_{[a,b]}\|_\infty = \frac{1}{|T_k(\frac{2\gamma-b-a}{b-a})|}.$$

Now let $\mu = \frac{2\gamma-b-a}{b-a}$. Then, under the map

$$\varphi : \mathbb{R} \to \mathbb{R}; x \mapsto x \mapsto \frac{2x-b-a}{b-a},$$

the interval $[a - d, b + d]$ is mapped to the interval $[-|\mu|, |\mu|]$. This implies

$$\|T_{k,[a,b],\gamma}|_{[a-d,b+d]}\|_\infty = \frac{1}{|T_k(\mu)|}\|T_k|_{[-|\mu|,|\mu|]}\|_\infty$$

Now let $x_0, ..., x_k$ be defined as in lemma 63. Then, according to that lemma, the map T_k attains local extrema at the points $x_1, ..., x_{k-1}$. Hence the derivative of T_k vanishes at these points, and since T_k' is a polynomial of degree $k-1$ these are all roots of T_k'. Since the leading coefficient of T_k is positive, as can immediately be seen by looking at the recursive definition, the map $x \mapsto T_k(x)$ is strictly increasing on $[1, \infty)$ and for k even strictly decreasing on $(-\infty, -1]$ and for k odd strictly increasing on $(-\infty, -1]$. Since $T_k(1) = 1$ this implies that for $1 \leq t_1 \leq t_2$ we have $1 \leq T_k(t_1) \leq T_k(t_2)$, and since $T_k(-1)$ is 1 for k even and -1 for k odd, we have for $t_2 \leq t_1 \leq -1$ that $1 \leq |T_k(t_1)| \leq |T_k(t_2)|$. Since T_k is either even or odd, we have $|T_k(x)| = |T_k(-x)|$ for all x, and since for all $|z| > 1$ and $|x| \leq 1$ we have $|T(x)| \leq 1 \leq |T(z)|$, we can thus conclude

$$\|T_k|_{[-|\mu|,|\mu|]}\|_\infty = |T_k(\mu)|.$$

We have thus shown that

$$\|T_{k,[a,b],\gamma}|_{[a-d,b+d]}\|_\infty = 1.$$

□

References

[1] S. N. Makarov, G. M. Noetscher, T. Raij, and A. Nummenmaa, "A quasi-static boundary element approach with fast multipole acceleration for high-resolution bioelectromagnetic models," IEEE Transactions on Biomedical Engineering, vol. 65, no. 12, pp. 2675–2683, 2018.

[2] M. Hämäläinen, R. Hari, R. J. Ilmoniemi, J. Knuutila, and O. V. Lounasmaa, "Magnetoencephalography—theory, instrumentation, and applications to noninvasive studies of the working human brain," Rev. Mod. Phys., vol. 65, pp. 413–497, Apr 1993.

[3] S. Schrader, A. Westhoff, M. C. Piastra, T. Miinalainen, S. Pursiainen, J. Vorwerk, H. Brinck, C. H. Wolters, and C. Engwer, "Duneuro—a software toolbox for forward modeling in bioelectromagnetism," PLOS ONE, vol. 16, pp. 1–21, 06 2021.

[4] C. Wolters, L. Grasedyck, and W. Hackbusch, "Efficient computation of lead field bases and influence matrix for the fem-based eeg and meg inverse problem," Inverse Problems, vol. 20, pp. 1099–1116, 08 2004.

[5] W. Nolting, Grundkurs Theoretische Physik 3: Elektrodynamik. 01 2007.

[6] F. Drechsler, C. Wolters, T. Dierkes, H. Si, and L. Grasedyck, "A full subtraction approach for finite element method based source analysis using constrained delaunay tetrahedralisation," NeuroImage, vol. 46, pp. 1055–65, 04 2009.

[7] C. Engwer, J. Vorwerk, J. Ludewig, and C. H. Wolters, "A discontinuous galerkin method to solve the eeg forward problem using the subtraction approach," SIAM Journal on Scientific Computing, vol. 39, no. 1, pp. B138–B164, 2017.

[8] T.-R. Erdbrügger, "Cutfem forward modeling for geometries with touching surfaces in bioelectromagnetism," Master's thesis, University of Muenster, 2021.

[9] D. Braess, Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics. Cambridge University Press, 3 ed., 2007.

[10] T. Miinalainen, A. Rezaei, D. us, A. Nüßing, C. Engwer, C. Wolters, and S. Pursiainen, "A realistic, accurate and fast source modeling approach for the eeg forward problem," NeuroImage, vol. 184, 08 2018.

[11] J. Vorwerk, A. Hanrath, C. H. Wolters, and L. Grasedyck, "The multipole approach for eeg forward modeling using the finite element method," NeuroImage, vol. 201, p. 116039, 2019.

[12] C. Wolters, H. Köstler, C. Möller, J. Härdtlein, L. Grasedyck, and W. Hackbusch, "Numerical mathematics of the subtraction method for the modeling of a current dipole in eeg source reconstruction using finite element head models," SIAM J. Scientific Computing, vol. 30, pp. 24–45, 01 2007.

[13] W. Rudin, Functional Analysis. McGraw-Hill, 2 ed., 1991.

[14] Y. Saad, Iterative Methods for Sparse Linear Systems. 01 2003.

[15] M. C. Piastra, S. Schrader, A. Nüßing, M. Antonakakis, T. Medani, A. Wollbrink, C. Engwer, and C. H. Wolters, "The WWU DUNEuro reference data set for combined EEG/MEG source analysis," June 2020. The research related to this dataset was supported by the German Research Foundation (DFG) through project WO1425/7-1 and the EU project ChildBrain (Marie Curie Innovative Training Networks, grant agreement 641652).

[16] C. H. Wolters, "Direkte Methoden zur Berechnung dipolinduzierter elektrischer und magnetischer Felder und inverse Strategien zur quellenlokalisation im Gehirn," 1997.

[17] A. Rezaei, A. Koulouri, and S. Pursiainen, "Randomized multiresolution scanning in focal and fast e/meg sensing of brain activity with a variable depth," Brain Topography, vol. 33, pp. 161 – 175, 2020.

[18] U. Aydin, J. Vorwerk, P. Küpper, M. Heers, H. Kugel, A. Galka, L. Hamid, J. Wellmer, C. Kellinghaus, S. Rampp, and C. H. Wolters, "Combining eeg and meg for the reconstruction of epileptic activity using a calibrated realistic volume conductor model," PLOS ONE, vol. 9, pp. 1–17, 03 2014.

[19] J. Silc, B. Robic, and T. Ungerer, Processor architecture - from dataflow to superscalar and beyond. 01 1999.

[20] M. Flynn, "Very high-speed computing systems," Proceedings of the IEEE, vol. 54, no. 12, pp. 1901–1909, 1966.

[21] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, 2021.

[22] M. Gottschlag and F. Bellosa, "Reducing avx-induced frequency variation with core specialization," 03 2019.

[23] Intel, "Intel intrinsics guide." `https://software.intel.com/sites/landingpage/IntrinsicsGuide`. Accessed: 2021-08-16.

[24] J. Hofmann, C. L. Alappat, G. Hager, D. Fey, and G. Wellein, "Bridging the architecture gap: Abstracting performance-relevant properties of modern server processors," ArXiv, vol. abs/1907.00048, 2020.

[25] N.-A. Dreier and C. Engwer, "Strategies for the vectorized block conjugate gradients method," 2019.

[26] D. Etiemble, "45-year cpu evolution: one law and two equations," 03 2018.

[27] O. Sander, DUNE — The Distributed and Unified Numerics Environment. Springer International Publishing, 2020.

[28] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE Transactions on Computers, vol. 44, no. 5, pp. 609–623, 1995.

[29] P. J. Denning, "The locality principle," Commun. ACM, vol. 48, p. 19–24, July 2005.

[30] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), pp. 283–292, 2008.

[31] V. Strassen, "Gaussian elimination is not optimal," Numerische Mathematik, vol. 13, pp. 354–356, 1969.

[32] M. Lam, E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in ASPLOS IV, 1991.

[33] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," ACM Trans. Math. Softw., vol. 34, May 2008.

[34] D. Yan, T. Wu, Y. Liu, and Y. Gao, "An efficient sparse-dense matrix multiplication on a multicore system," in 2017 IEEE 17th International Conference on Communication Technology (ICCT), pp. 1880–1883, 2017.

[35] D. P. O'Leary, "The block conjugate gradient algorithm and related methods," Linear Algebra and its Applications, vol. 29, pp. 293–322, 1980. Special Volume Dedicated to Alson S. Householder.

[36] A. Frommer, K. Lund, and D. Szyld, "Block krylov subspace methods for functions of matrices," ETNA - Electronic Transactions on Numerical Analysis, vol. 47, pp. 100–126, 01 2017.

[37] A. Frommer, K. Lund, and D. B. Szyld, "Block krylov subspace methods for functions of matrices ii: Modified block fom," SIAM Journal on Matrix Analysis and Applications, vol. 41, no. 2, pp. 804–837, 2020.

[38] K. Lund, A NEW BLOCK KRYLOV SUBSPACE FRAMEWORK WITH APPLICATIONS TO FUNCTIONS OF MATRICES ACTING ON MULTIPLE VECTORS. PhD thesis, Bergischen Universität Wuppertal, 2018.

[39] G. Fischer, Lineare Algebra. Springer Spektrum, 2014.

[40] F. Hirzebruch, Einführung in die Funktionalanalysis. Spektrum Akademischer Verlag, 1991.

[41] J. Dixmier, C*-Algebras. North-Holland Publishing Company, 1977.

[42] I. Raeburn and D. Williams, "Morita equivalence and continuous-trace c*-algebras," 1998.

[43] N.-A. Dreier, Hardware-Oriented Krylov Methods for High-Performance Computing. dissertation, University of Muenster, 2020.

[44] A. Dubrulle, "Retooling the method of block conjugate gradients.," 2001.

[45] T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto, and Y. Yanagisawa, "Shifted cholesky qr for computing the qr factorization of ill-conditioned matrices," SIAM J. Sci. Comput., vol. 42, pp. A477–A503, 2020.

[46] J. E. Gentle, Numerical Linear Algebra for Applications in Statistics. 1998.

[47] D. Olesen, S. Eilers, and G. Pedersen, C*-Algebras and their Automorphism Groups, 2nd Edition. 09 2018.

[48] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, and O. Sander, "The dune framework: Basic concepts and recent developments,"

Computers & Mathematics with Applications, vol. 81, pp. 75–112, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.

[49] M. Blatt, "A parallel algebraic multigrid method for elliptic problems with highly discontinuous coefficients," 01 2010.

[50] T. A. Davis, "Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method," ACM Trans. Math. Softw., vol. 30, p. 196–199, June 2004.

[51] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," toms, vol. 31, pp. 302–325, September 2005.

Declaration of Academic Integrity

I hereby confirm that this thesis on "Efficient Computation of Transfer Matrices using the Block Conjugate Gradient Method" is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

_____

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

_____