RWTH Aachen University

Center for Computational Engineering Science – Mathematics Division (MathCCES)

— Prof. Dr. B. Stamm —

The Alternating Schwarz Method for solving Eigenvalue Problems by Inverse Power Iteration and Steepest Descent

Seminar project

Hendrik Borchardt
(344104)

Summer Semester 2019

Contents

1 Introduction

In the 19th century, one open question was whether the problem of finding solutions to Laplace's equation

$$\begin{aligned} &\text{Find } u \in C^2(\Omega) \cap C^0(\overline{\Omega}), \\ &\Delta u = 0 \text{ in } \Omega, \\ &u = g \text{ on } \partial\Omega \end{aligned} \tag{1.1}$$

is well-posed on an arbitrary domain Ω. For simple domains such as rectangles or circles, Fourier analysis can be used to prove the existence of a solution but the general case is tricky. In 1869, Hermann Schwarz developed the first domain decomposition method as a tool to solve this equation on the union of two shapes Ω_1 and Ω_2, given that there exists a solution on each of the two shapes for arbitrary boundary conditions $u_i = g_i$ on $\partial\Omega_i, i = 1, 2$ (see figure 2.1). In essence, Schwarz constructed a sequence of functions $(u_1^n)_{n\in\mathbb{N}} \subset C^2(\Omega) \cap C^0(\overline{\Omega}), (u_2^n)_{n\in\mathbb{N}} \subset C^2(\Omega) \cap C^0(\overline{\Omega})$ as follows:

Initialisation: arbitrary functions that fulfil the boundary conditions

$$\begin{aligned} &u_1^0 \in C^2(\Omega_1) \cap C^0(\overline{\Omega_1}) \text{ arbitrary with } u_1^0 = g \text{ on } \partial\Omega \cap \partial\Omega_1, \\ &u_2^0 \in C^2(\Omega_2) \cap C^0(\overline{\Omega_2}) \text{ arbitrary with } u_2^0 = g \text{ on } \partial\Omega \cap \partial\Omega_2. \end{aligned} \tag{1.2}$$

Iteration: For $n \in \mathbb{N}_0$, let

$$\begin{aligned} \Delta u_1^{n+1} &= 0 \text{ in } \Omega_1, & \Delta u_2^{n+1} &= 0 \text{ in } \Omega_2, \\ u_1^{n+1} &= u_2^n \text{ on } \Gamma_1 := \partial\Omega_1 \cap \Omega_2, & u_2^{n+1} &= u_1^{n+1} \text{ on } \Gamma_2 := \partial\Omega_2 \cap \Omega_1, \\ u_1^{n+1} &= g \text{ on } \partial\Omega_1 \setminus \Gamma_1, & u_2^{n+1} &= g \text{ on } \partial\Omega_2 \setminus \Gamma_2. \end{aligned} \tag{1.3}$$

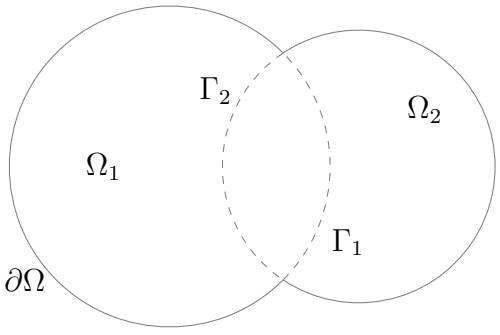

Figure 1.1: A composite domain $\Omega = \Omega_1 \cup \Omega_2$

This construction is called the **alternating Schwarz method**. To compute the iterates, it is required to solve the differential equation (1.1) on the first domain Ω_1 and then solve it on the second domain Ω_2 using the result of the former computation as a boundary condition, then repeating with new boundary conditions. Via the maximum principle, which states that a harmonic function either attains its maximum exclusively on the boundary or is constant, convergence of the sequence can be proven. In Section 2, we lay out the proof of convergence of the above mentioned sequence.

Eigenvalue problems for dense matrices can be solved using the QR decomposition, which returns all the eigenvalues as well as a system of eigenvectors. For large sparse matrices, the QR decomposition would yield non-sparse matrices which takes away the benefit of reduced storage requirements. This can render the QR decomposition less attractive, especially if one is not interested in all of the eigenvalues but only the smallest or largest ones. Inverse power iteration and steepest descent minimisation for the Rayleigh coefficient are iterative methods for finding the smallest eigenvalue that can work with sparse matrices.

As we discuss in Section 2, equations of the type (1.1) can be solved using an iterative scheme (1.3). A naive approach would be to try to solve an eigenvalue problem of the form

$$\begin{aligned} \text{Find } u \neq 0 : \Delta u = \lambda u \text{ in } \Omega, \\ u = g \quad \text{on } \partial\Omega \end{aligned} \tag{1.4}$$

with a scheme of the form

$$\begin{aligned} \Delta u_i^{n+1} &= \lambda_i^n u_i^n && \text{in } \Omega_i,\ i = 1, 2, \\ u_i^{n+1} &= u_i^n && \text{on } \partial\Omega_i \cap \Omega_{2-i},\ i = 1, 2, \\ u_i^{n+1} &= g && \text{on } \partial\Omega_i \setminus \partial\Omega_{2-i},\ i = 1, 2, \end{aligned} \tag{1.5}$$

similar to the alternating Schwarz method. Unfortunately, in this case it is not obvious to see how one would choose λ_i^n or whether the scheme would even converge.

Therefore, the main idea we follow is to apply an iterative eigenvalue solver to the problem (1.4), which yields a sequence $(u^n)_{n\in\mathbb{N}} \subset L^\infty(\Omega)$. $(u_n)_{n\in\mathbb{N}}$ can then be proven to converge in the function space $L^\infty(\Omega)$. Next, we make use of the domain decomposition approach to facilitate solving source problems that occur in the iterations of the scheme. In Section 3 we review the above mentioned well-known iterative methods for eigenvalue problems and discuss where suitable source problems arise.

In Section 4, we show the results of the implemented algorithms and discuss their properties such as the rate of convergence and optimal choice of the step size.

An implementation of a domain decomposition scheme requires solving PDEs on the subdomains. This includes a mesh generation algorithm as well as discretising the equation

and solving the generated linear system. For this we will use FreeFem++, which provides the above mentioned features and discretises the PDE using the finite element method. In Section 5, we show some implementation details that are useful for constructing the solutions and we provide the source code for the iterative schemes.

Acknowledgements

At this point I would like to thank Prof. Benjamin Stamm and especially Muhammad Hassan for their detailed feedback and helpful discussions. Also I am eternally grateful for my wife supporting me and keeping me from losing my path.

2 Domain Decomposition methods for source problems

In the following, let $\Omega \subset \mathbb{R}^2$ be an open, bounded domain with smooth boundary. As the operator domain we choose $H = C^2(\Omega) \cap C^0(\overline{\Omega})$ which is a Banach space.

Definition 1. *The Laplace equation reads: Let $g \in C^0(\partial\Omega)$.*

$$\textit{Find } u \in H : \Delta u = 0 \textit{ in } \Omega,\ u = g \textit{ on } \partial\Omega. \tag{2.1}$$

where

$$\Delta u(x) := \sum_{i=1}^{2} \frac{\partial^2 u_i(x)}{\partial^2 x_i} \tag{2.2}$$

Solutions $u \in H$ to the Laplace equation are called harmonic functions.

For continuous harmonic functions, we have the strong maximum principle:

Theorem 2. *Let $u \in H$ be a harmonic function on Ω. Then u either attains its maximum only on $\partial\Omega$ or is constant.*

Remark 3. *Applying this principle to $-u$ immediately shows that if u is not constant, u attains its minimum only on $\partial\Omega$ as well. In conclusion:*

$$\min_{x^* \in \partial\Omega} u(x^*) \leq u(x) \leq \max_{x^* \in \partial\Omega} u(x^*),\ \forall x \in \Omega,$$

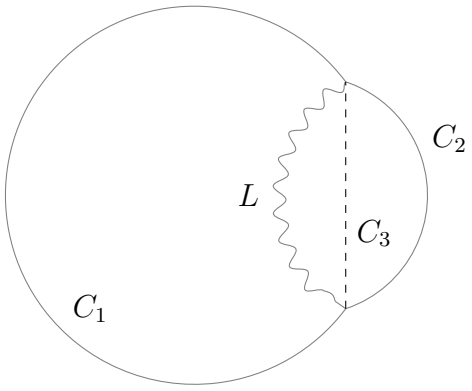

Figure 2.1: A curve L inside of a domain split by C_3

with equality only if u is constant. Therefore, if u is constant on $\partial\Omega$, then $\max_{\partial\Omega} u = \min_{\partial\Omega} u$ and u must be constant on Ω.

Similar maximum principles exist for solutions to other partial differential equations involving uniformly elliptic operators, see [Lio89].

We now want to show the convergence result for the sequence (1.3) as given by Schwarz in his article from 1870 [Sch70]. One important proposition about a configuration such as the one in Figure 2.1 is the following:

Theorem 4. *Let the boundary $\partial\Omega_1$ be divided into two curves*

$$\begin{aligned} C_1 &= \partial\Omega_1 \setminus \overline{\partial\Omega_1 \cap \Omega}, \\ C_2 &= \partial\Omega_1 \cap \Omega \end{aligned}$$

and consider a harmonic function $u \in L^\infty(\Omega_1)$ that is continuous on $\overline{\Omega_1} \setminus (\overline{\partial\Omega_1 \cap \Omega} \cap \partial\Omega)$ with

$$\begin{aligned} \Delta u &= 0 \text{ in } \Omega_1, \\ u &= 0 \text{ on } C_1, \\ u &= 1 \text{ on } C_2. \end{aligned} \tag{2.3}$$

Let L be a curve that is in the interior of Ω_1, with the possible exception of the end points of L being on the end points of C_2. Then $q := \sup\{u(x) : x \in L\} \in (0, 1)$.

A proof for the case of circular domains can be found in [CG18].

Note that u is not explicitly defined at its isolated points of discontinuity.

Remark 5. *Consider the setting of Theorem 4. If we have a function w with*

$$\begin{aligned} \Delta w &= 0 \text{ in } \Omega_1, \\ w &= 0 \text{ on } C_1, \\ w &\leq G \text{ on } C_2, \end{aligned} \tag{2.4}$$

then $w(x) \leq Gu(x)$ for all $x \in \overline{\Omega_1}$ and therefore

$$\sup\{w(x) : x \in L\} \leq G \sup\{u(x) : x \in L\} = Gq \text{ where } q \in (0, 1). \tag{2.5}$$

Schwarz then used this result to prove the following theorem:

Theorem 6. *The sequences constructed in (1.3) converge to functions u_1, u_2. Moreover, the global function*

$$u := \begin{cases} u_1 & \text{in } \Omega_1, \\ u_2 & \text{in } \Omega_2, \\ u_1 = u_2 & \text{in } \Omega_1 \cap \Omega_2 \end{cases} \tag{2.6}$$

is well defined and solves the problem (2.1).

Proof. Let

$$u_- := \min_{x\in\partial\Omega} g(x), \quad u_+ := \max_{x\in\partial\Omega} g(x) \tag{2.7}$$

be the minimal and maximal value of g. For all n, we have $\Delta u_1^n = 0,\ \Delta u_2^n = 0$, therefore $u_i^{n+1} - u_i^n,\ i = 1, 2$, is a harmonic function. We apply Remark 5 for the first domain Ω_1, where we choose $L = \Gamma_2, C_2 = \Gamma_1$ and get a number $q_1 \in (0, 1)$. Conversely, when we apply Remark 5 for the other domain Ω_2, we have to choose $L = \Gamma_1, C_2 = \Gamma_2$ and obtain $q_2 \in (0, 1)$.

For the first iterate ($n = 1$), to complete the boundary condition on $\partial\Omega_1$ we impose

$$u_1^1\big|_{\Gamma_1}(x) = g_1(x) := u_-.$$

Therefore, on $\partial\Omega_1$ we have that

$$\{g(x) : x \in \partial\Omega_1 \setminus \Gamma_1\} \cup \{g_1(x) : x \in \Gamma_1\} \subset [u_-, u_+]$$

and by the maximum principle, $u_1^1(x) \in (u_-, u_+)$ for all $x \in \Omega_1$. In the same way, when imposing

$$u_2^1\big|_{\Gamma_2} = g_2 := u_1^1,$$

since $g_2(x) \in [u_-, u_+]$ for $x \in \Gamma_2$ and $g(x) \in [u_-, u_+]$ for $x \in \partial\Omega_2 \setminus \Gamma_2$, we have by the maximum principle $u_2^1(x) \in [u_-, u_+]$ for $x \in \Omega_2$.

For $n = 2$, with the same argument we get $u_1^2(x) \in [u_-, u_+]$ for $x \in \Omega_1$ and thereby $\|u_1^2 - u_1^1\|_\infty \le u_+ - u_- =: G$.

We proceed by induction. Assume that for $n \ge 2$ fixed, $\|u_1^n - u_1^{n-1}\|_\infty < G q_1^{n-2} q_2^{n-2}$. Then

$$\begin{aligned} u_2^{n+1} - u_2^n &= 0 && \text{on } \partial\Omega_2 \setminus \Gamma_2, \\ u_2^{n+1} - u_2^n &= u_1^{n+1} - u_1^n && \text{on } \Gamma_2 \end{aligned}$$

and by Remark 5

$$\|u_2^{n+1} - u_2^n\|_\infty < q_1 \|u_1^n - u_1^{n-1}\|_\infty = G q_1^n q_2^{n-1}.$$

Assume that for $n \ge 2$ fixed, $\|u_2^n - u_2^{n-1}\|_\infty < G q_1^{n-1} q_2^{n-2}$. Then similarly,

$$\begin{aligned} u_1^{n+1} - u_1^n &= 0 && \text{on } \partial\Omega_1 \setminus \Gamma_1, \\ u_1^{n+1} - u_1^n &= u_2^n - u_2^{n-1} && \text{on } \Gamma_1 \end{aligned}$$

and by Remark 5

$$\|u_1^{n+1} - u_1^n\|_\infty < q_2 \|u_2^n - u_2^{n-1}\|_\infty = G q_1^{n-1} q_2^{n-1}.$$

Alternating between the two arguments yields that for all $n \geq 2$,

$$\|u_1^n - u_1^{n-1}\|_\infty < G q_1^{n-2} q_2^{n-2},$$

$$\|u_2^n - u_2^{n-1}\|_\infty < G q_1^{n-1} q_2^{n-2}.$$

Writing the sequences as

$$u_1^n = u_1^1 + \sum_{i=2}^{n} (u_1^i - u_1^{i-1}),$$

$$u_2^n = u_2^1 + \sum_{i=2}^{n} (u_2^i - u_2^{i-1}),$$

we see that $(u_1^n)_{n\in\mathbb{N}}$ is a Cauchy sequence: For every $\varepsilon > 0$ there exists an $N \in \mathbb{N}$ such that for all $m, n \in \mathbb{N}$ with $N < m < n$ holds:

$$\|u_1^n - u_1^m\|_\infty = \left\| \sum_{i=m+1}^{n} (u_1^i - u_1^{i-1}) \right\|_\infty < G \sum_{i=m+1}^{n} (q_1 q_2)^i = G \frac{(q_1 q_2)^{n+1} - (q_1 q_2)^{m+1}}{q_1 q_2 - 1} < \varepsilon.$$

This is possible as q_1 and q_2 are less than 1 and therefore $(q_1 q_2)^{n+1} \overset{n\to\infty}{\to} 0$. As we consider a Banach space, completeness implies the convergence of $(u_1^n)_{n\in\mathbb{N}}$ to a function u_1. The same argument applies to $(u_2^n)_{n\in\mathbb{N}}$ converging to u_2.

By the continuity of the Laplace operator, u_1 and u_2 are harmonic functions. As u_1 and u_2 coincide on Γ_1 and on Γ_2, which form a closed curve, for their difference we have

$$(u_1 - u_2)\big|_{\Gamma_1 \cup \Gamma_2} = 0.$$

By the maximum principle (3) we have

$$u_1 - u_2 = 0 \text{ in } \Omega_1 \cap \Omega_2 \Rightarrow u_1 = u_2 \text{ in } \Omega_1 \cap \Omega_2.$$

For this reason, the piecewise function

$$u := \begin{cases} u_1 & \text{in } \Omega_1, \\ u_2 & \text{in } \Omega_2, \\ u_1 = u_2 & \text{in } \Omega_1 \cap \Omega_2 \end{cases}$$

is well defined and $\Delta u = 0$. Therefore, u is a solution to the problem (2.1). □

P. L. Lions extended this argument to weakly overlapping domains, meaning that the

boundaries of the two domains can intersect in more than two points. He also remarks that the Laplace operator $-\Delta$ can be replaced by other uniformly elliptic partial differential operators. [Lio89] Using a variational approach, he extends these results to spaces with less regularity like the Sobolev space $H_0^1(\Omega)$. [Lio88]

3 Iterative methods for eigenvalue problems

The methods and derivations in this chapter are adapted from [Arb] where they are detailed for the matrix case.

3.1 Spectral theorem for compact operators and their inverse

Let H be a Hilbert space. Let $K : H \to H$ be a linear compact self-adjoint operator, then the spectral theorem [Wer18, Kap6] states that there exists an orthonormal system $\{\varphi_1, \varphi_2, \dots\}$ with an associated sequence $(\lambda_1, \lambda_2, \dots)$ converging to zero, such that

$$Kx = \sum_{k=1}^{\infty} \lambda_n \langle x, \varphi_n \rangle \varphi_n \ \forall x \in H \tag{3.1}$$

If the operator is defined on an infinite dimensional Banach space H, compact operators cannot have a linear, bounded inverse T, because then $T \circ K$ would be compact, but the identity operator Id_H is not compact. In contrast, if we have an unbounded linear operator, it is possible that its inverse is compact. One example are Sturm-Liouville differential operators that have an inverse operator of Hilbert-Schmidt type, which are compact operators, when defined on an appropriate space. In this case, for an eigenvector x_n of K with eigenvalue $\lambda_n \neq 0$ we have

$$\begin{aligned} x_n = T(H(x_n)) = T(\lambda_n x_n) = \lambda_n T(x_n) \\ \Leftrightarrow T(x_n) = \lambda_n^{-1} x_n \end{aligned} \tag{3.2}$$

and we see that x_n is an eigenvector of T for the eigenvalue λ_n^{-1}.

3.2 Inverse power iteration

One can observe that the sequence $(x_n)_{n \in \mathbb{N}} \subset H$,

$$x_n = \frac{A^n u_0}{||A^n u_0||}$$

for a matrix A and a vector u_0 converges to an eigenvector for the largest eigenvalue of A. Intuitively, this is due to the largest eigenvalue having the greatest effect on the magnitude of Au_0. Therefore, we can define the iterative scheme called **power iteration** as follows: Starting from an initial guess x_0 that is not an eigenvector, $Ax_0 \neq \lambda x_0 \, \forall \lambda \in \mathbb{R}$, let

$$x_{n+1} = \frac{Ax_n}{||x_n||}. \tag{3.3}$$

Then the sequence $(x_n)_{n\in\mathbb{N}}$ converges to an eigenvector with the largest eigenvalue of A.

The spectral theorem for comapct operators justifies using the power iteration for such operators as well. If we know how to evaluate a compact operator K which is inverse to the operator T, we can find the smallest eigenvalue of T by applying the power iteration (3.3) to K. This is called **inverse power iteration**. One step of the power iteration can be written as

$$u_{n+1} = \frac{Ku_n}{||u_n||} \Leftrightarrow Tu_{n+1} = \frac{u_n}{||u_n||} \tag{3.4}$$

where u_n is known. Therefore, this step has the form of a source problem.

The resulting algorithm will be:

Algorithm 1 Inverse Power iteration

Input: Number of iterations N_{outer}

1: Choose start value x_0.
2: **for** $n \in (0, 1, 2, \ldots, N_{\text{outer}} - 1)$ **do**
3: Obtain x_{new} by solving $Lx_{\text{new}} = x_n$.
4: Set $\lambda_{n+1} = ||x_{\text{new}}||$.
5: Set $x_{n+1} := x_{\text{new}}/\lambda_{n+1}$.

Output: Eigenpair approximation $(\lambda_{N_{\text{outer}}}, x_{N_{\text{outer}}})$

3.3 Steepest descent

In the following, consider a linear self-adjoint operator T with a compact inverse operator K. Then the largest eigenvalue of K can be determined by maximizing the Rayleigh coefficient ρ_K (cf. [Wer18, Kap. 6]):

$$\rho_K(x) := \frac{\langle Kx, x\rangle}{\langle x, x\rangle}, \tag{3.5}$$

$$\lambda_{\max} := \max\{|\lambda| : \exists x \in H \setminus \{0\} : Kx = \lambda x\} \tag{3.6}$$

$$\Leftrightarrow \lambda_{\max} = \max_{x\in H\setminus\{0\}} \rho_K(x) = \max_{x\in H\setminus\{0\}} \frac{\langle Kx, x\rangle}{\langle x, x\rangle} \tag{3.7}$$

It is easy to see that, by the same argument, minimizing ρ_T yields the smallest eigenvalue of T with the minimizer being an eigenvector.

For minimization by the steepest descent method, the next search direction $d(x)$ is the negative of the gradient of the target function $\rho_T(x)$, which turns out to be

$$\begin{aligned} d(x) &= -\nabla\rho_T(x) = -\frac{\nabla\langle Tx,x\rangle\langle x,x\rangle - \langle Tx,x\rangle\nabla\langle x,x\rangle}{\langle x,x\rangle^2} \\ &= -\frac{2Tx\langle x,x\rangle - \langle Tx,x\rangle\,(2x)}{\langle x,x\rangle^2} \\ &= -2\left(Tx - \frac{\langle Tx,x\rangle}{\langle x,x\rangle}x\right)(\langle x,x\rangle)^{-1}. \\ &= -2\frac{Tx-\rho_T(x)x}{\langle x,x\rangle} \end{aligned} \tag{3.8}$$

This allows us to construct an iterative method with step width $\gamma > 0$:

$$x_{n+1} = x_n + \gamma d(x_n) = x_n - 2\gamma\frac{Tx_n - \rho(x_n)x_n}{||x_n||^2} \tag{3.9}$$

The step width γ can be determined in various ways. See Section 3.4 for an optimized choice.

If the spectrum of T is spread out, more precisely if $\kappa_2(T) = \frac{\lambda_{\max}}{\lambda_{\min}}$ is large, convergence of this method is quite slow. Especially for the case of infinitely many eigenvalues that converge to 0 or ∞, we would have $\kappa_2(T) = \infty$. Slow convergence is an effect which is also observed in the case of solving a linear system $Ax = b$: If the spectrum of A is spread out, a large condition number $\kappa_2(A)$ means that solving the system is numerically inaccurate. In the linear system case, the condition number can be improved by multiplying with a *preconditioner* P such that

$$\begin{aligned} &Ax = b \\ \Leftrightarrow\ &PAx = Pb \end{aligned}$$

and $\kappa_2(PA) << \kappa_2(A)$. The best choice for the preconditioner is $P = A^{-1}$, since it yields $PA = Id$ with $\kappa_2(Id) = 1$ which is the minimal condiniton number. This choice makes applying the preconditioner as hard as solving the original problem, which is why, usually, approximations to A^{-1} are used. One example would be $P = D^{-1}$ where $D = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$.

In the iterative scheme, assuming that it converges to an eigenvector x_*, we have

$$\begin{aligned} x_* &= x_* - 2\gamma \frac{Ax_* - \rho(x_*)x_*}{||x_*||^2} \\ \Leftrightarrow 0 &= -2\gamma \frac{Ax_* - \rho(x_*)x_*}{||x_*||^2} \end{aligned}$$

Applying a linear invertible operator P to both sides as a preconditioner and reintroducing x_* gives:

$$x_* = x_* - 2\gamma P \left(\frac{Ax_* - \rho(x_*)x_*}{||x_*||^2} \right) \tag{3.10}$$

and going back to the iterative form:

$$x_{n+1} = x_n + \gamma d(x_n) = x_n - 2\gamma P \left(\frac{Ax_n - \rho(x_n)x_n}{||x_n||^2} \right) \tag{3.11}$$

This shows that x_* is the limit of both schemes, with and without the preconditioner.

If at this point we choose $P = A^{-1}$, applying the preconditioner amounts to solving the following source problem:

$$\begin{aligned} P \left(\frac{Ax_n - \rho(x_n)x_n}{||x_n||^2} \right) &= y_{\text{pre}} \\ \Leftrightarrow Ay_{\text{pre}} &= \frac{Ax_n - \rho(x_n)x_n}{||x_n||^2} \end{aligned}$$

Using a domain decomposition method allows us to apply the best preconditioner at a reduced cost.

The resulting algorithm is:

Algorithm 2 Steepest descent iteration - fixed step width

Input: Number of iterations N_{outer}, step width γ

1: Choose start value $x_0 \neq 0$.
2: **for** $n \in (0, 1, 2, \ldots, N_{\text{outer}} - 1)$ **do**
3: Calculate $x_{\text{res}} = Lx_n - \frac{\langle Lx_n, x_n \rangle}{\langle x_n, x_n \rangle} x_n$.
4: Obtain x_{pre} by solving $Lx_{pre} = x_{\text{res}}$.
5: Set $x_{\text{new}} = x_n - 2\gamma x_{\text{pre}}$.
6: Set $x_{n+1} := x_{\text{new}}/||x_{\text{new}}||$.

Output: Eigenpair approximation $(\rho(x_{N_{\text{outer}}}), x_{N_{\text{outer}}})$

3.4 Steepest descent with optimal stepwidth

The steepest descent method seeks to minimize the Rayleigh coefficient in order to find an approximation of an eigenvector. If we determine the direction p_n of steepest descent for the Rayleigh coefficient according to equation (3.8), we can minimize $\rho(x_n + \gamma p_n)$ with respect to γ along the line in this direction p_n to find the best next guess on that line.

To do this, consider for the self-adjoint operator A

$$\begin{aligned}\rho\left(x_n + \gamma p_n\right) &= \frac{\langle Ax_n, x_n\rangle + 2\gamma\langle Ap_n, x_n\rangle + \gamma^2\langle Ap_n, p_n\rangle}{\langle x_n, x_n\rangle + 2\gamma\langle p_n, x_n\rangle + \gamma^2\langle p_n, p_n\rangle} \\ &= \frac{\begin{pmatrix} 1 \\ \gamma \end{pmatrix}^\top \begin{pmatrix} \langle Ax_n, x_n\rangle & \langle Ap_n, x_n\rangle \\ \langle Ax_n, p_n\rangle & \langle Ap_n, p_n\rangle \end{pmatrix} \begin{pmatrix} 1 \\ \gamma \end{pmatrix}}{\begin{pmatrix} 1 \\ \gamma \end{pmatrix}^\top \begin{pmatrix} \langle x_n, x_n\rangle & \langle p_n, x_n\rangle \\ \langle x_n, p_n\rangle & \langle p_n, p_n\rangle \end{pmatrix} \begin{pmatrix} 1 \\ \gamma \end{pmatrix}}.\end{aligned}$$

which is the Rayleigh coefficient for the generalized eigenvector problem for two real 2×2 matrices:

$$\begin{pmatrix} \langle Ax_n, x_n\rangle & \langle Ap_n, x_n\rangle \\ \langle Ax_n, p_n\rangle & \langle Ap_n, p_n\rangle \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda \begin{pmatrix} \langle x_n, x_n\rangle & \langle p_n, x_n\rangle \\ \langle x_n, p_n\rangle & \langle p_n, p_n\rangle \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \tag{3.12}$$

This means that obtaining an eigenvector $(\alpha, \beta)^T$ to the smallest eigenvalue λ of (3.12) and scaling it to get

$$(1, \gamma)^T := (\alpha/\alpha, \beta/\alpha)^T$$

gives a local minimum at $\rho(x_n + \gamma p_n)$.

3.5 Application of the alternating Schwarz method

We use the alternating Schwarz method in the algorithms to solve the source problems of the form $Lx = f$. This gives us two new algorithms, Algorithm 3 and 4.

Algorithm 3 Inverse Power iteration with alternating Schwarz method

Input: Number of iterations N_{outer}

1: Choose start value x_0.
2: **for** $n \in (0, 1, 2, \ldots, N_{\text{outer}} - 1)$ **do**
3: Obtain x_{new} by solving $Lx_{\text{new}} = x_n$ with the alternating Schwarz method (1.3).
4: Set $\lambda_{n+1} = ||x_{\text{new}}||$.
5: Set $x_{n+1} := x_{\text{new}}/\lambda_{n+1}$.

Output: Eigenpair approximation $(\lambda_{N_{\text{outer}}}, x_{N_{\text{outer}}})$

Algorithm 4 Steepest descent iteration - fixed step width

Input: Number of iterations N_{outer}, step width γ

1: Choose start value $x_0 \neq 0$.
2: **for** $n \in (0, 1, 2, \ldots, N_{\text{outer}} - 1)$ **do**
3: Calculate $x_{\text{res}} = Lx_n - \frac{\langle Lx_n, x_n \rangle}{\langle x_n, x_n \rangle} x_n$.
4: Obtain x_{pre} by solving $Lx_{pre} = x_{\text{res}}$ with the alternating Schwarz method (1.3).
5: Set $x_{\text{new}} = x_n - 2\gamma x_{\text{pre}}$.
6: Set $x_{n+1} := x_{\text{new}}/||x_{\text{new}}||$.

Output: Eigenpair approximation $(\rho(x_{N_{\text{outer}}}), x_{N_{\text{outer}}})$

4 Numerical results

4.1 Questions

In this chapter we want to answer the following questions:

1. Are the inverse power iteration and steepest descent method viable algorithms for calculating eigenvalues and eigenvectors of partial differential operators?

2. Does using domain decomposition instead of solving source problems on the whole domain decrease the speed of convergence?

3. Is the optimized step width from Section 3.4 a better choice than a fixed step width?

4. How many inner iterations in the domain decomposition algorithm have to be done?

5. How does the grid mesh width affect the performance of the algorithms?

6. How do the algorithms perform on different grids? Does the addition of a constant term $V \neq 0$ to the operator decrease the performance of the algorithms?

4.2 Approach

We consider the eigenvalue problem (1.4) for a differential operator of the form

$$L := -\Delta + V \text{ with } V \in \{0, 1\}$$

and zero boundary conditions.

The experiments are conducted on a rectangular domain with $\frac{1}{3}$ overlap, which is to say that the overlapping region is as large as each of the non-overlapping parts (cf. Figure 4.1). On this region, the smallest eigenvalue and its eigenvector is known to be [Ber]

$$u_{\text{exact}} = \sin(\frac{\pi}{3}x)\sin(\pi y),\ \lambda_{\text{exact}} = \left(\frac{\pi}{3}\right)^2 + \pi^2, \tag{4.1}$$

which can therefore be used as an exact reference solution.

We also consider an L-shaped domain as shown in Figure 4.2. More specifically, Ω_1 is a rectangle with side lengths 2 by 1 units. Ω_2 is a copy of Ω_1 rotated 90°.

Note that the rectangles in these configurations are weakly overlapping, because $\partial\Omega_1 \cap \partial\Omega_2$ contains not only isolated points but two of the edges of the overlapping region as well.

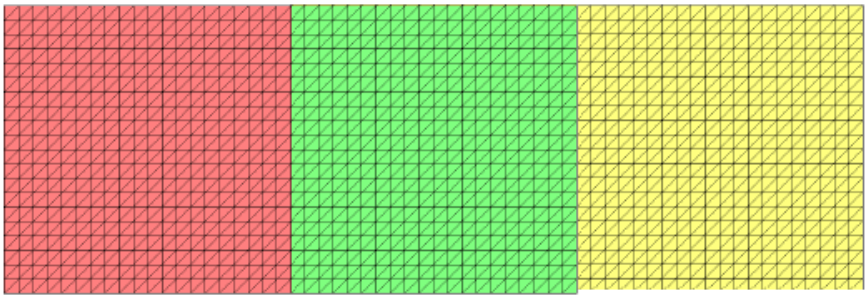

Figure 4.1: The rectangular mesh. Ω_1: red and green area, Ω_2: yellow and green area.

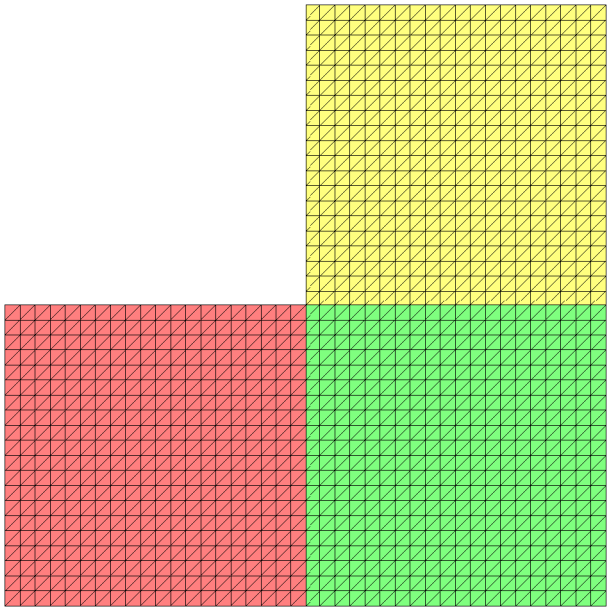

Figure 4.2: The L shaped mesh. Ω_1: red and green area, Ω_2: yellow and green area.

We implement an alternating Schwarz method for the source problem:

$$-\Delta u + Vu = f \text{ in } \Omega, \tag{4.2}$$
$$u = 0 \text{ on } \partial\Omega. \tag{4.3}$$

We then use this method in the inverse power algorithm with domain decomposition, Algorithm 3, and the steepest descent algorithm with domain decomposition, Algorithm 4, to find eigenpairs of the operator

$$L : C_0^2(\Omega) \to C_0^0(\Omega), \tag{4.4}$$
$$Lu := -\Delta u + Vu. \tag{4.5}$$

The operator L is elliptic if $V \geq 0$, and we have a maximum principle for functions u satisfying $Lu = 0$.

For L it is known that there exists a Green's function G such that for any $f \in L^2(\Omega)$,

$$u(x) := \int_\Omega G(x, x') f(x') \, \mathrm{d}x'$$

satisfies $Lu = f$ in Ω. Therefore, source problems (4.2) for this operator are well-posed.

Errors are calculated relative to the norm of the reference solution:

$$\mathrm{err}_{L^2}(u) = \frac{||u - u_{\text{exact}}||_{L^2}}{||u_{\text{exact}}||_{L^2}} \tag{4.6}$$

$$\mathrm{err}(\lambda) = \frac{|\lambda - \lambda_{\text{exact}}|}{|\lambda_{\text{exact}}|} \tag{4.7}$$

4.3 Test cases

We address the questions stated above. If not specified otherwise, we use Algorithms 3 and 4 with $\gamma = 1$, $V = 0$, $N_{\text{inner}} = 6$, $N_{\text{outer}} = 20$, $N_{\text{unit}} = 100$.

4.3.1 Question 1

Calculate the eigenvalue and eigenvector using Algorithms 1 and 2 and compare them to the exact solution (4.1).

The eigenvalue errors converge to the same value (cf. Figure 4.4), but the steepest descent method result differs in the L^2 norm. Inspecting a plot of the deviation (Figure 4.5)

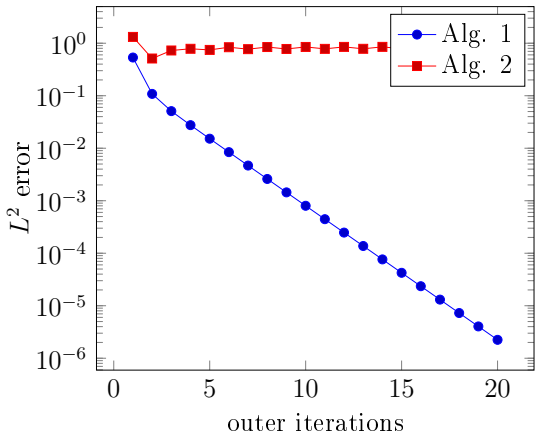

Figure 4.3: L^2 error

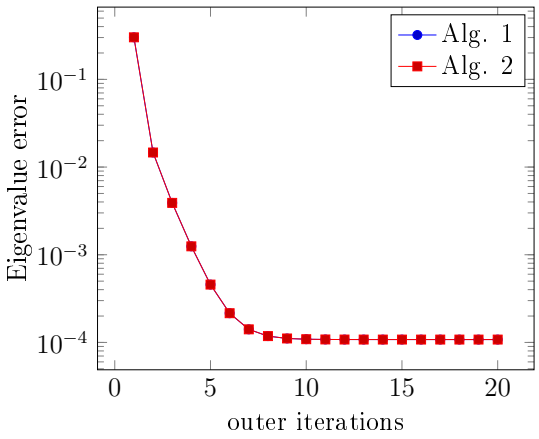

Figure 4.4: Eigenvalue error

shows that oscillations occurred at the corners. Therefore the algorithm was not stable.

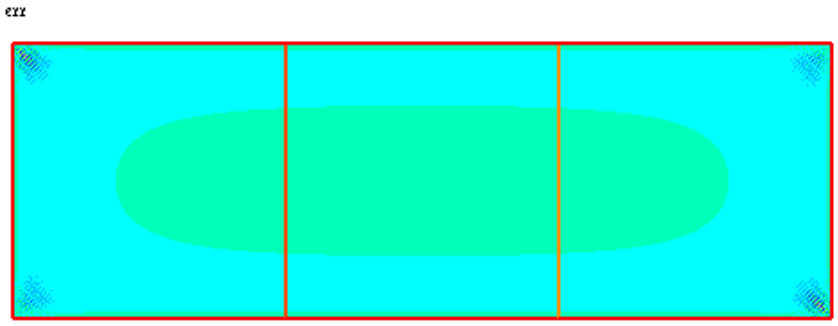

Figure 4.5: Steepest descent method producing oscillations in the corners

4.3.2 Question 2

Compare the errors of estimates from Algorithms 3 and 4 with those from Algorithms 1 and 2.

For the inverse power iteration, there is no additional error introduced by using the Alternating Schwarz Method (Figure 4.6). The steepest descent algorithm benefits from the Alternating Schwarz Method, as it now converges in the L^2 norm (Figure 4.7).

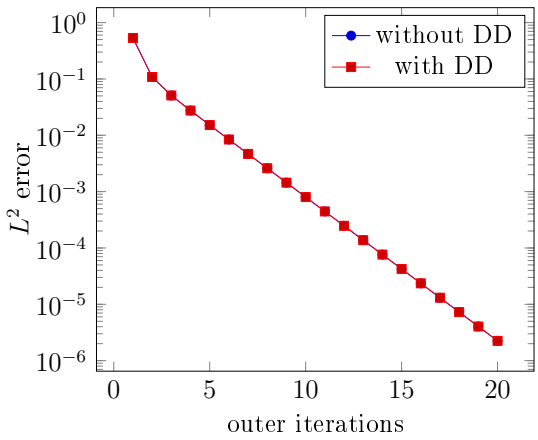

Figure 4.6: Inverse Power Iteration error

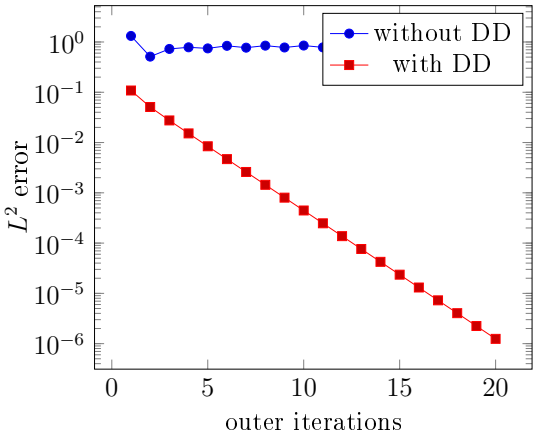

Figure 4.7: Steepest descent method error

4.3.3 Question 3

Try Algorithm 4 with a step width of $\gamma = 1$ and with a step width calculated using equation (3.12).

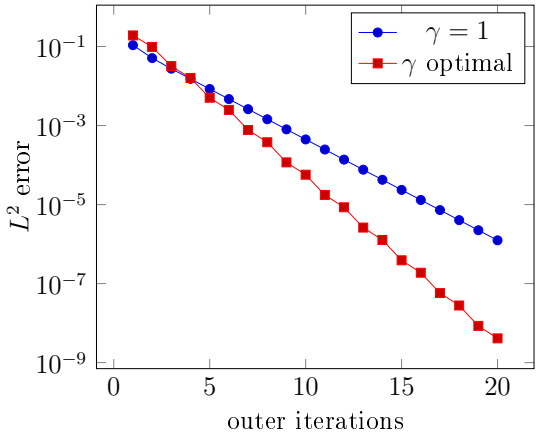

Figure 4.8: Steepest descent method error for different step widths

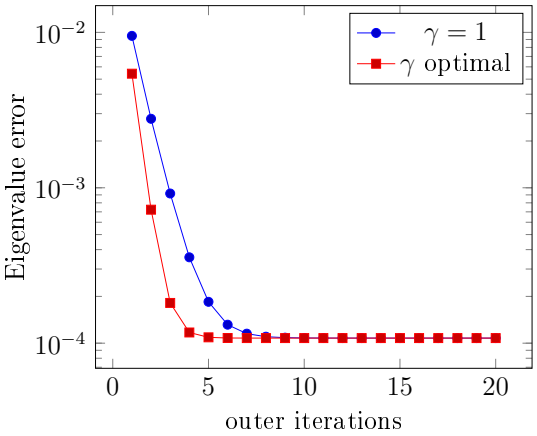

Figure 4.9: Steepest descent method eigenvalue error

We see that the speed of convergence is increased by using the optimal step width (Figure 4.8). For the (somewhat arbitrary choice) $\gamma = 1$, there is an average reduction of the error by 22%, the optimized step width yields an average reduction by 32% per step.

4.3.4 Question 4

For Algorithms 3 and 4, calculate the error for $N_{\text{inner}} = 1, \ldots, 5$.

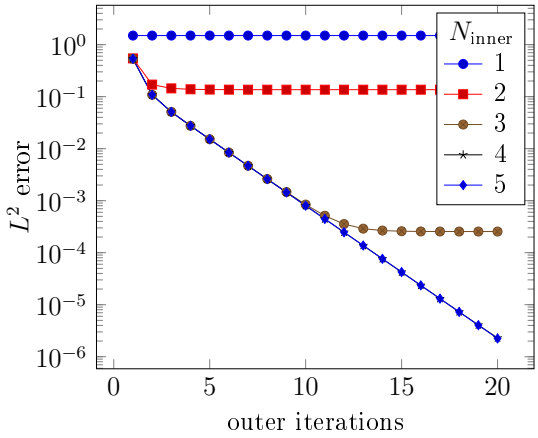

Figure 4.10: Inverse power iteration method error

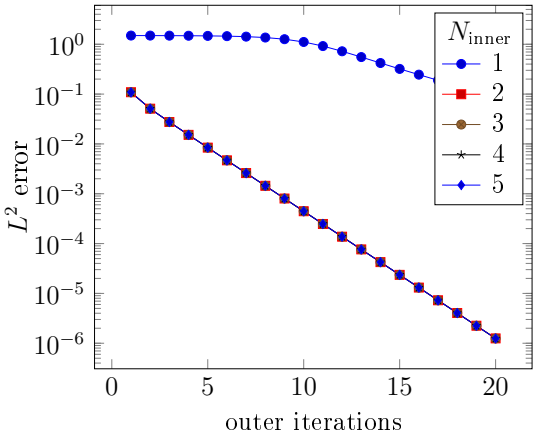

Figure 4.11: Steepest descent method error

The inverse power method needs 4 inner iterations to reach a relative error of 10^{-6} (Figure 4.10). Less iterations result in a higher error after convergence of the algorithm. The rate of convergence is not affected by the number of inner iterations, only the lowest possible error. For the steepest descent method, more than 2 inner iterations don't benefit either the rate of convergence or the lowest possible error after 20 iterations (Figure 4.11).

4.3.5 Question 5

Try the four algorithms for $N_{\text{unit}} = 30, 50, 75$.

There is no difference in the ability to approximate the discretized exact solution between the grid sizes (Figures 4.12, 4.13)). In contrast, the eigenvalue can be determined with a higher precision on finer grids (Figures 4.14, 4.15).

4.3.6 Question 6

Compare the errors obtained on a rectangular domain with those on an L shaped domain. On the L shape there is no closed form of the solution, therefore we use the eigenvalue solver implemented in FreeFem++ to calculate the generalized eigenvalue and eigenvector of the stiffness and mass matrices of the FEM problem on a grid with $N_{\text{unit}} = 250$

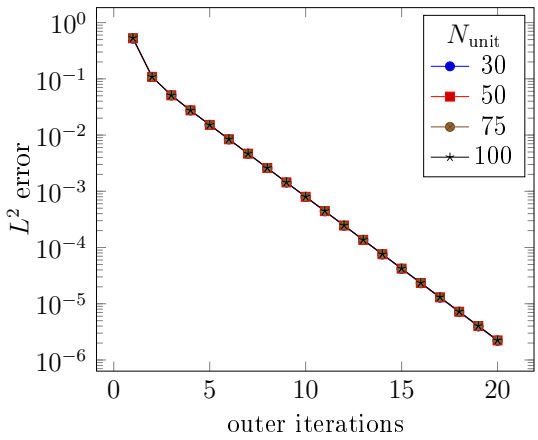

Figure 4.12: Inverse power iteration method error

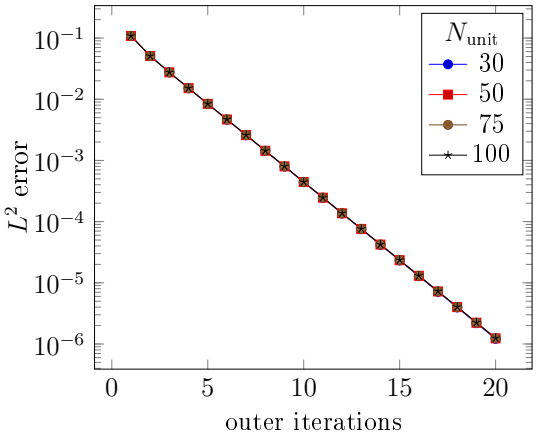

Figure 4.13: Steepest descent method error

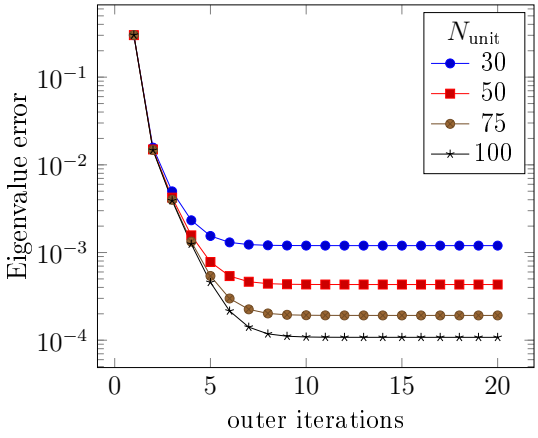

Figure 4.14: Inverse power iteration method error

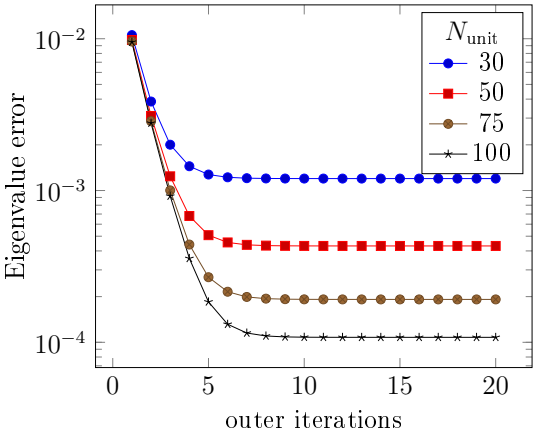

Figure 4.15: Steepest descent method error

points which is then projected on the grid with $N_{\text{unit}} = 100$ points where the algorithms are executed. The choice $V = 1$ only makes a slight difference (Figures 4.16, 4.17). On the L shape, the inverse power iteration only achieves an error of 0.1% (Figure 4.16), whereas the steepest descent method approaches the solution even quicker (Figure 4.17). The eigenvalue on the L shape can be calculated up to 0.03% (Figures 4.18, 4.19).

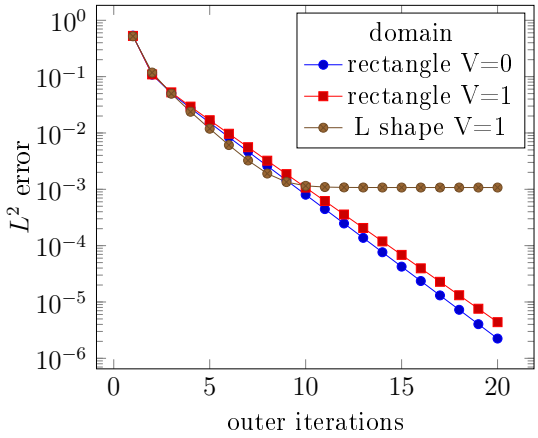

Figure 4.16: Inverse power iteration method error

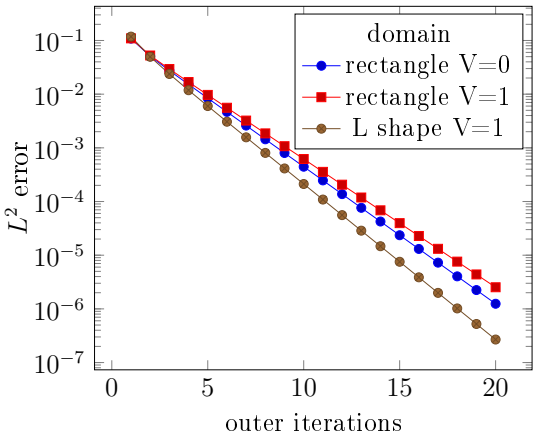

Figure 4.17: Steepest descent method error

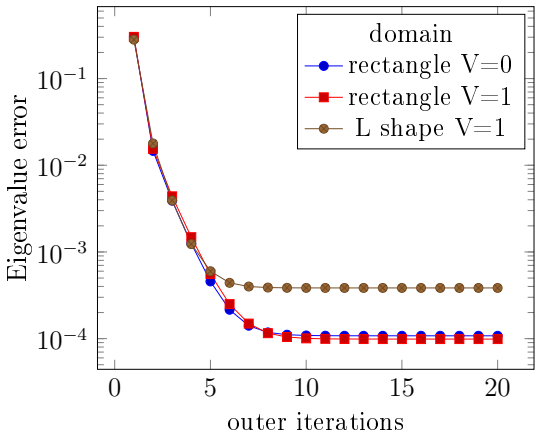

Figure 4.18: Inverse power iteration method error

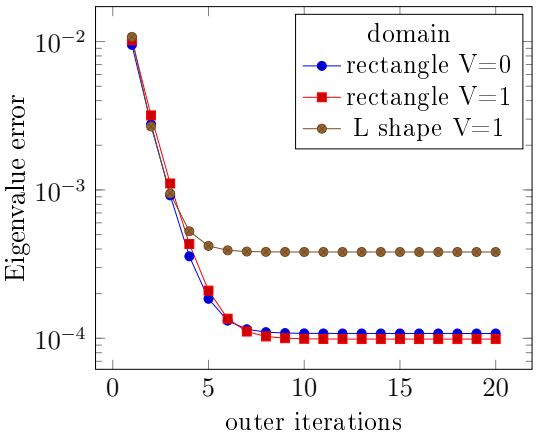

Figure 4.19: Steepest descent method error

5 Implementation details in FreeFem++

FreeFem++ is a C++ program used to solve partial differential equations in variational form, presented in [Hec12]. It features a C-like programming language, handles mesh generation and provides access to the underlying stiffness and mass matrices if desired. This makes it a good tool to develop algorithms related to PDEs.

Here we provide code-level details of the implementation, as the provided snippets might be of interest for other implementers.

5.1 Mesh generation

Rectangular meshes are constructed using the `square` function. It is important to generate the overlapping region separately, as creating two triangulations of the overlap might lead to different triangulations that can't be combined. Also, we need to label the boundaries of the regions to be able to impose boundary conditions later. We choose the three labels `outerEdge`, `innerEdge1` for the vertical inner edge and `innerEdge2` for the horizontal inner edge.

The code to create the L shaped mesh is given in the following:

Listing 5.1: LMesh.idp

```
// requires outerEdge, innerEdge1, innerEdge2, resolution
// yields Th, Th0, Th1
mesh Th, Th0, Th1;
{
  mesh Tha = square(resolution, resolution, [x,y]);
  int[int] ra = [0,1];
  int[int] la = [1,outerEdge,2,innerEdge1,3,outerEdge,4,outerEdge];
  Tha = change(Tha, region=ra, label=la);

  mesh Thb = square(resolution, resolution, [x+1,y]);
  int[int] rb = [0,2];
  int[int] lb = [1,outerEdge,2,outerEdge,3,innerEdge2,4,innerEdge1];
  Thb = change(Thb, region=rb, label=lb);

  mesh Thc = square(resolution, resolution, [x+1,y+1]);
  int[int] lc = [1,innerEdge2,2,outerEdge,3,outerEdge,4,outerEdge];
  int[int] rc = [0,3];
  Thc = change(Thc, region=rc, label=lc);
```

```
20    Th = Tha + Thb;
21    Th = Th + Thc;
22    Th0 = Tha + Thb;
23    Th1 = Thb + Thc;
24  }
```

5.2 Region indicator function

To be able to piece together two functions, we need a function that tells us whether a point is inside or outside of a certain region. For this problem we can use the `chi(mesh)(x,y)` function which returns `true` if `(x,y)` is part of the mesh `mesh`. As we want to write something like $u = u_1 + u_2$, we construct a function that returns the number of regions a point is in:

Listing 5.2: RegionIndicator.idp

```
1 // requires Th0, Th1
2 // yields regionIndicator
3 func real regionIndicator() {
4   real nregions = 0;
5   if (chi(Th0)(x,y)) { nregions += 1; }
6   if (chi(Th1)(x,y)) { nregions += 1; }
7   return nregions;
8 }
```

Then we can write `u=(u1+u2)/regionIndicator()`, where the indicator divides by 2 in the overlap.

5.3 Restriction macro

During the algorithms, we have to switch forward and backward between different meshes. Going from a big mesh to a submesh is a simple assignment:

To extend a function on a submesh to a function on the big mesh, we can't use an assignment, as FreeFem++ tries to extend the function continuously. Therefore, we make use of the following macro:

Listing 5.3: RestrictMacro.idp

```
1 // This macro creates a V1 function uo that is
2 // ui on V2 and 0 elsewhere
3 macro restrict(ui, V2, uo, V1)
4 V1 uo;
```

```
{
  V2 ui2 = ui;
  matrix ip = interpolate(V1, V2, inside=1);
  uo[] = ip*ui2[];
}//
```

This macro copies the points of the function where it is defined and sets it to 0 outside of the submesh. This further helps us with writing expressions like `u=(u1+u2)/regionIndicator()`, as outside of the overlap, the non-defined function is now 0.

5.4 Solving a source problem

The syntax to solve a source problem

$$\begin{aligned} Lu &= f \text{ in } \Omega \\ f &= g \text{ on outerEdge} \end{aligned}$$

on a mesh `Th` with $P1$ finite elements is as follows:

Listing 5.4: Source problem

```
fespace Vh(Th, P1);
Vh u,v;
solve ProblemName(u,v) = int2d(Th)(Lu*v) - int2d(Th)(f*v)
+ on(outerEdge, u=g);
```

5.5 Domain Decomposition algorithm implementation

Listing 5.5: SolveDD.idp

```
// requires Th0, Th1, Th, Vh0, Vh1, Vh, fh, outerEdge, innerEdge1,
//   innerEdge2, regionIndicator, Ninner, V
// yields uges
Vh uges;
{
  Vh0 u0=0,v0=0;
  Vh1 u1=0,v1=0;
  for(int i=0; i < Ninner; i++) {
    solve Prob1(u0,v0) = int2d(Th0)(grad(u0)'*grad(v0) + V*u0*v0)
      - int2d(Th0)(fh*v0) + on(outerEdge, u0=0) + on(innerEdge2, u0=u1);
    solve Prob2(u1,v1) = int2d(Th1)(grad(u1)'*grad(v1) + V*u1*v1)
      - int2d(Th1)(fh*v1) + on(outerEdge, u1=0) + on(innerEdge1, u1=u0);
  }

  // assemble solution
```

```
16    // Extend result with zeros
17    restrict(u0, Vh0, ua, Vh);
18    restrict(u1, Vh1, ub, Vh);
19    uges = (ua + ub);
20    uges = uges/regionIndicator();
21  }
```

5.6 Inverse power iteration algorithm

Listing 5.6: DDInvPower.idp

```
1  macro grad(u) [dx(u), dy(u)] //EOM
2  //verbosity = 1;
3  wait= 1;
4
5  // Problem parameters
6  func V = 1;
7  //int Nouter = 20;
8  //int Ninner = 3;
9  // parameters for fine Mesh LMesh.idp
10 int outerEdge = 1;
11 int innerEdge1 = 2;
12 int innerEdge2 = 3;
13 mesh ThFine;
14 {
15   int resolution = fineResolution;
16
17   // requires outerEdge, innerEdge1, innerEdge2, resolution
18   // yields Th, Th0, Th1
19   include "LMesh.idp"
20   ThFine = Th;
21 }
22 fespace VhFine(ThFine, P1);
23
24 // parameters for LMesh.idp
25 int resolution = coarseResolution;
26
27 // requires outerEdge, innerEdge1, innerEdge2, resolution
28 // yields Th, Th0, Th1
29 include "LMesh.idp"
30 fespace Vh(Th, P1);
31 fespace Vh0(Th0, P1);
32 fespace Vh1(Th1, P1);
33
34 // requires Th0, Th1
35 // yields regionIndicator
36 include "RegionIndicator.idp"
37
38 // yields restrict
39 include "RestrictMacro.idp"
```

```
41 // exact solution
42 // requires ThFine, VhFine, outerEdge, V
43 // yields uhExact
44 include "SolveExact.idp"

46 // alternatively on the rectangle:
47 //func uExact = sin(pi*x/3)*sin(pi*y);
48 //Vh uhExact = uExact;
49 //real normExact = sqrt(int2d(Th)(uhExact*uhExact));
50 //uhExact = uhExact/normExact;
51 //real lambdaExact = (pi/3)*(pi/3) + (pi)*(pi);

53 for(int Ninner=NinnerMin; Ninner < NinnerMax+1; Ninner++) {
54 // open files for logging
55 ofstream log(logName + Ninner + ".csv");
56 ofstream logValue(logName + Ninner + "Eigenvalue.csv");

58 // initialize as constant 1
59 Vh uhCurrent = 1;
60 for(int j=0; j < Nouter; j++) {
61   // Lu_{k+1} = u_k
62   // RHS is current u
63   Vh fh = uhCurrent;

65   Vh unew;
66   if (withDD) {
67     // DD solution of -Laplace u + Vu = fh
68     // requires Th0, Th1, Th, Vh0, Vh1, Vh, fh, outerEdge, innerEdge1,
69     //   innerEdge2, regionIndicator, Ninner, V
70     // yields uges
71     include "SolveDD.idp"
72     unew = uges;
73   } else {
74     Vh ugesExact, v1;
75     solve ExactSource(ugesExact, v1) = int2d(Th)(grad(ugesExact)'*grad(v1)
76       + V*ugesExact*v1) - int2d(Th)(fh*v1) + on(outerEdge, ugesExact=0);
77     unew = ugesExact;
78   }
79   Vh uges = unew;

81   // norm of solution
82   real Norm = sqrt(int2d(Th)(uges*uges));
83   real sign = int2d(Th)(uges);
84   uges = uges/Norm*sign/abs(sign);

86   uhCurrent = uges;

88   // Error statistics
89   Vh uerr;
90   if (compareExact) {
91     //restrict(uhCurrent, Vh, uhCurrentFine, VhFine);
```

```
92      //uerr = uhCurrentFine - uhExactFine;
93      uerr = uhCurrent - uhExact;
94    } else {
95      // ...
96    }
97    real errorNorm = sqrt((int2d(Th)(uerr*uerr))/normExact);
98    real rho = 1/Norm;
99    real errorValue = abs(rho - lambdaExact)/lambdaExact;
100   log << j+1 << ";" << errorNorm << endl;
101   logValue << j+1 << ";" << errorValue << endl;
102   cout << "Outer it. " << j << ": Error " << errorNorm << endl;
103 }
104 Vh uerr = uhCurrent - uhExact;
105 if (doPlot) {
106   plot(uhCurrent, dim=3, fill=1, wait=wait, value=1, cmm="Current");
107   plot(uhExact, dim=3, fill=1, wait=wait, value=1, cmm="Exact");
108   plot(uerr, dim=3, fill=1, wait=wait, value=1, cmm="Error");
109 }
110 }
```

5.7 Steepest descent iteration algorithm

Listing 5.7: DDStDescent.idp

```
1  macro grad(u) [dx(u), dy(u)] //EOM
2  wait= 0;

4  // Problem parameters
5  func V = 1.0;

7  // parameters for fine Mesh LMesh.idp
8  int outerEdge = 1;
9  int innerEdge1 = 2;
10 int innerEdge2 = 3;
11 mesh ThFine;
12 {
13   int resolution = fineResolution;

15   // requires outerEdge, innerEdge1, innerEdge2, resolution
16   // yields Th, Th0, Th1
17   include "LMesh.idp"
18   ThFine = Th;
19 }
20 fespace VhFine(ThFine, P1);

22 // parameters for LMesh.idp
23 int resolution = coarseResolution;

25 // requires outerEdge, innerEdge1, innerEdge2, resolution
26 // yields Th, Th0, Th1
```

```
include "LMesh.idp"
fespace Vh(Th, P1);
fespace Vh0(Th0, P1);
fespace Vh1(Th1, P1);

// requires Th0, Th1
// yields regionIndicator
include "RegionIndicator.idp"

// yields restrict
include "RestrictMacro.idp"

// exact solution
// requires Th, Vh, outerEdge, V
// yields uhExact
include "SolveExact.idp"

// alternatively on the rectangle:
//func uExact = sin(pi*x/3)*sin(pi*y);
//Vh uhExact = uExact;
//real normExact = sqrt(int2d(Th)(uhExact*uhExact));
//uhExact = uhExact/normExact;
//real lambdaExact = (pi/3)*(pi/3) + (pi)*(pi);


//int Ninner = 3;
for(int Ninner=NinnerMin; Ninner < NinnerMax+1; Ninner++) {
// open files for logging
ofstream log(logName + Ninner + ".csv");
ofstream logValue(logName + Ninner + "Eigenvalue.csv");
ofstream logGamma(logName + Ninner + "Gamma.csv");

// Starting value that is non-zero and satisfies the boundary condition
Vh uhCurrent = 0;
real rho = 0;
Vh Ax, v0;
func initialFunc = 1;
Vh fh = initialFunc;
Vh unew;
if (withDD) {
// DD solution of -Laplace u + Vu = fh
// requires Th0, Th1, Th, Vh0, Vh1, Vh, fh, outerEdge, innerEdge1,
//   innerEdge2, regionIndicator, Ninner, V
// yields uges
include "SolveDD.idp"
unew = uges;
} else {
Vh ugesExact, v1;
solve ExactSource(ugesExact, v1) = int2d(Th)(grad(ugesExact)'*grad(v1)
  + V*ugesExact*v1) - int2d(Th)(fh*v1) + on(outerEdge, ugesExact=0);
unew = ugesExact;
}
```

```
uhCurrent = unew;

// pre-calculation of initial Ax and rho(uhCurrent)
{
  solve Residual(Ax, v0) = int2d(Th)(grad(uhCurrent)'*grad(v0)
    + V*uhCurrent*v0) - int2d(Th)(Ax*v0) + on(outerEdge, Ax=0);
  // Ax = Lu_{k} - (u_{k}, Lu_{k})_{L2}/||u_{k}||^2 * u_{k}
  real xAx = int2d(Th)(uhCurrent*Ax);
  real normuhCurrent= int2d(Th)(uhCurrent*uhCurrent);
  rho = xAx/normuhCurrent;
}

for(int j=0; j < Nouter; j++) {
  // Calculate residual
  Vh uhRes = Ax - rho*uhCurrent;
  // RHS is residual
  Vh fh = uhRes;

  Vh ugesExact, v1;
  solve ExactSource(ugesExact, v1) = int2d(Th)(grad(ugesExact)'*grad(v1)
    + V*ugesExact*v1) - int2d(Th)(fh*v1) + on(outerEdge, ugesExact=0);

  Vh unew;
  if (withDD) {
    // DD solution of -Laplace u + Vu = fh
    // requires Th0, Th1, Th, Vh0, Vh1, Vh, fh, outerEdge, innerEdge1,
    //   innerEdge2, regionIndicator, Ninner, V
    // yields uges
    include "SolveDD.idp"
    unew = uges;
  } else {
    unew = ugesExact;
  }
  if (Ninner==0) {
    unew = uhRes;
  }
  Vh udderr = ugesExact - unew;
  real udderrnorm = int2d(Th)(udderr*udderr);
  cout << "DD error: " << udderrnorm << endl;

  real gamma;
  if (optStepWidth) {
    // determine best step length
    real a11 = int2d(Th)(grad(uhCurrent)'*grad(uhCurrent)
      + V*uhCurrent*uhCurrent);
    real a12 = int2d(Th)(grad(uhCurrent)'*grad(unew) + V*uhCurrent*unew);
    real a22 = int2d(Th)(grad(unew)'*grad(unew) + V*unew*unew);
    matrix Ax =[[a11, a12],[a12, a22]];
    set(Ax,solver=sparsesolver);

    real b11 = int2d(Th)(uhCurrent*uhCurrent);
    real b12 = int2d(Th)(uhCurrent*unew);
```

```
131     real b22 = int2d(Th)(unew*unew);
132     matrix Bx =[[b11, b12],[b12, b22]];
133     set(Bx,solver=sparsesolver);

135     int nev = 1;
136     real[int] evx(2);
137     real[int,int] eVx(2,nev);
138     real solveps=1e-14;
139     int k = EigenValue(Ax,Bx,sym=true,sigma=0,value=evx,rawvector=eVx,
140       tol=solveps,maxit=0,nev=nev);
141     gamma = eVx(1,0)/eVx(0,0);
142   } else {
143     gamma = -stepWidth;
144   }

146   uhCurrent = uhCurrent + gamma*unew;

148   // norm of solution
149   real Norm = sqrt(int2d(Th)(uhCurrent*uhCurrent));
150   real sign = int2d(Th)(uhCurrent);
151   cout << "Norm: " << Norm << endl;
152   uhCurrent = uhCurrent/Norm*sign/abs(sign);

154   // calc Ax and rho(uhCurrent)
155   {
156     solve Residual(Ax, v0) = int2d(Th)(grad(uhCurrent)'*grad(v0)
157       + V*uhCurrent*v0) - int2d(Th)(Ax*v0) + on(outerEdge, Ax=0);
158     // Ax = Lu_{k} - (u_{k}, Lu_{k})_{L2}/||u_{k}||^2 * u_{k}
159     real xAx = int2d(Th)(uhCurrent*Ax);
160     real normuhCurrent= int2d(Th)(uhCurrent*uhCurrent);
161     rho = xAx/normuhCurrent;
162   }

164   // Error statistics
165   Vh uerr;
166   if (compareExact) {
167     //restrict(uhCurrent, Vh, uhCurrentFine, VhFine);
168     uerr = uhCurrent - uhExact;
169   } else {
170     // ...
171   }
172   if (doPlot) {
173     plot(uerr, cmm="err", dim=3, fill=1, value=1, wait=wait);
174   }
175   real errorNorm = sqrt((int2d(Th)(uerr*uerr))/normExact);
176   real errorValue = abs(rho - lambdaExact)/lambdaExact;
177   log << j+1 << ";" << errorNorm << endl;
178   logValue << j+1 << ";" << errorValue << endl;
179   logGamma << j+1 << ";" << gamma << endl;
180   cout << "Outer it. " << j << ": Error " << errorNorm << endl;
181 }
182 } //end for
```

Bibliography

[Arb] Arbenz, Peter: *Lecture Notes on Solving Large Scale Eigenvalue Problems.* https://people.inf.ethz.ch/arbenz/ewp/Lnotes/lsevp.pdf

[Ber] Berg, Tobias: *Eigenwerte des Laplaceoperators auf einem Rechteck.* http://www.alt.mathematik.uni-mainz.de/Members/kostrykin/Lehre/hauptseminar-ss2009-vortrag01

[CG18] Ciaramella, G. ; Gander, M. J.: Analysis of the Parallel Schwarz Method for Growing Chains of Fixed-sized Subdomains: Part II. In: *SIAM Journal on Numerical Analysis* 56 (2018), jan, Nr. 3, S. 1498–1524. http://dx.doi.org/10.1137/17m1115885. – DOI 10.1137/17m1115885

[Hec12] Hecht, F.: New development in FreeFem++. In: *J. Numer. Math.* 20 (2012), Nr. 3-4, S. 251–265. – ISSN 1570–2820

[Lio88] Lions, Pierre-Louis: On the Schwarz alternating method. I. In: *First international symposium on domain decomposition methods for partial differential equations* Bd. 1 Paris, France, 1988, S. 42

[Lio89] Lions, Pierre-Louis: On the Schwarz alternating method II. In: *Domain decomposition methods* 628 (1989), S. 47–70

[Sch70] Schwarz, Hermann A.: *Ueber einen Grenzübergang durch alternirendes Verfahren.* Zürcher u. Furrer, 1870

[Wer18] Werner, Dirk: Funktionalanalysis. (2018). http://dx.doi.org/10.1007/978-3-662-55407-4. – DOI 10.1007/978–3–662–55407–4