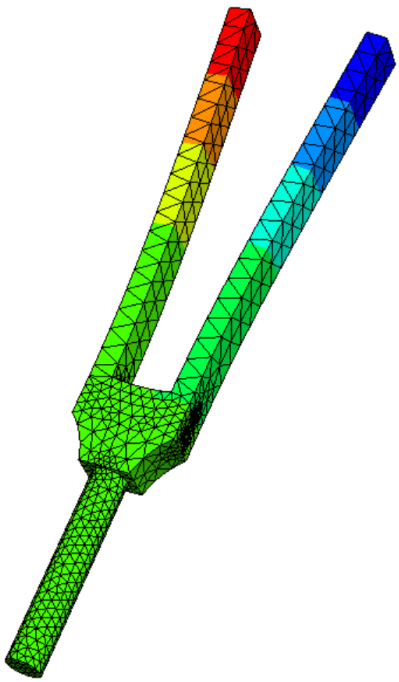
TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Computing Eigenfrequencies using Finite Element Methods

Sebastian Hirnschall
Christoph Leonhartsberger
Rafael Dorigo

Supervisor:
Prof. Dr. Joachim Schöberl



January 27, 2021

Abstract

We compare different iterative methods (including LOBPCG (A. Knyazev et al. 2007)) for computing eigenfrequencies and the corresponding eigenmodes in a finite element space. We analyze both numerical complexity and convergence of each algorithm and provide reference implementations using NGSolve.
The most suitable method is then used to analyze a clamped-free beam and a tuning fork with realistic material properties.
We compare our results to the Euler-Bernoulli beam-theory and measurements done on a real world model.

Keywords

symmetric eigenvalue problems, eigenfrequency, eigenmode, vibration, lobpcg, pinvit, steepest descent, lopsd, inverse iteration, preconditioner, ngsolve

Contents

1. Introduction

In the following paper we want to solve a generalized eigenvalue problem of the form

$$Au = \lambda Mu$$

using *Finite Element Method* (FEM).
Let us first take a look at a relatively simple example: Let $\Omega \subseteq \mathbb{R}^n$ be a bounded domain. Assume we have the following eigenvalue problem

$$-\Delta u = \lambda u \quad in\ \Omega \tag{1.1}$$
$$u = 0 \quad on\ \partial\Omega \tag{1.2}$$

Multiplying both sides of the eq. (1.1) with a bump function $v \in C_c^\infty(\Omega)$ [1] and using Green's first identity (Blümlinger 2019) results in the weak formulation

$$\underbrace{\int_\Omega \nabla u \nabla v d\mathcal{H}^n}_{A(u,v)} = \lambda \underbrace{\int_\Omega uv d\mathcal{H}^n}_{M(u,v)}$$

with $A : H^1 \times H^1 \to \mathbb{R}$ and $M : H^1 \times H^1 \to \mathbb{R}$ where H^1 is a Sobolev-Space.[2]
The goal is to find a solution $u \in H_0^1$ such that

$$A(u, v) = \lambda M(u, v), \quad \forall v \in H_0^1. \tag{1.3}$$

For more detailed information we refer to (Evans 2010).

We start approximating the geometry using a mesh which has a finite number of points. Now the solution can be approximated on a finite element space $V_h \subset H_0^1$, where $V_h := \text{span}\{\phi_1, \ldots, \phi_n\}$, $n = \dim V_h$ and h denotes the fineness of the grid(more precisely the diameter of the largest element in the mesh). As basis functions $(\phi_i)_{i\in\{1,\ldots,n\}}$ one can use hat functions such that

$$\phi_i(x_j) = \delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}$$

where x_j are the vertices of the mesh.
The new formulation on $V_h \subset H_0^1$ would be: Find $u_h \in V_h$ such that

$$A(u_h, v_h) = \lambda M(u_h, v_h), \quad \forall v_h \in V_h$$

[1] $C_c^\infty(\Omega) := \{\Phi \in C^\infty(\Omega) : \text{supp}\,\Phi \text{ compact in } \Omega\}$
[2] $H^1 := \{f \in L^2 : \nabla f \in L^2\}$ and $H_0^1 := \{v \in H^1 : v = 0 \text{ on } \partial\Omega\}$

where $u_h = \sum_{i=1}^{n} u_i \phi_i$. Using the representation of u_h and eq. (1.3) gives

$$A\left(\sum_{i=1}^{n} u_i \phi_i, \phi_j\right) = \lambda M\left(\sum_{i=1}^{n} u_i \phi_i, \phi_j\right), \quad \forall j \in \{1, \ldots, n\}$$

Due to the linearity of $A(\cdot,\cdot)$ and $M(\cdot,\cdot)$, the sums and operators can be interchanged. Defining the matrices A and M $\forall i, j \in \{1, \ldots, n\}$ as

$$\begin{aligned} a_{ij} &= A(\phi_i, \phi_j) \\ m_{ij} &= M(\phi_i, \phi_j) \end{aligned}$$

leads to

$$\sum_{j=1}^{n} a_{ij} u_j = \lambda \sum_{j=1}^{n} m_{ij} u_j, \quad \forall i \in \{1, \ldots, n\}$$

in compact form

$$Au = \lambda M u.$$

Remark. We will use eq. (1.1) as an example problem to compare complexity and convergence for the algorithms discussed in sections 3 to 5.

2. Motivation

Given a diagonalizable hermitian matrix $A \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ and $|\lambda_1| \leq, \ldots, < |\lambda_n|$, it is well known that the inverse iteration method (algorithm 1) can be used to approximate the eigenvalue λ_k of $Au = \lambda Mu$ closest to $\rho \in \mathbb{R}$. If A is positive semi definite, the smallest eigenvalue λ_1 can be approximated using $\rho = 0$.

Algorithm 1 The Shift-and-Inverse Iteration Method (Nannen 2019)

Input: $A, M \in \mathbb{K}^n \times n$, $u, \rho \in \mathbb{K}^n$

1: compute $LU = P(A - \rho M)Q$
2: **for** $i = 1, \ldots$ **do**
3: $\quad u = PMu$
4: $\quad$ Solve $Lz = u$
5: $\quad$ Solve $Uu = z$
6: $\quad u = Qx$
7: $\quad u = x / \|u\|$

Output: The approximation u to the eigenvector corresponding to the eigenvalue λ closest to ρ.

However, as the LU factorization of $A - \rho M$ in algorithm 1 has complexity $\mathcal{O}(n^3)$ this algorithm is not feasible for large matrices A and M.

3. Locally Optimal Preconditioned Steepest Descent - LOPSD

Let $A \in \mathbb{R}^{n\times n}$ be a symmetric, positive definite matrix and $M \in \mathbb{R}^{n\times n}$. We want to solve the generalized symmetric eigenvalue problem $Au = \lambda Mu$ by applying the *Locally Optimal Steepest Descent Method* (LOPSD) which is an iterative method to approximate the smallest solution λ_1 of $Au = \lambda Mu$ for matrices with large dimensions. Using a positive definite preconditioner $T \in \mathbb{R}^{n\times n}$ lowers the condition number for this problem. According to (A. V. Knyazev and Neymeyr 2003) the preconditioner T has to satisfy the constraint

$$\left\|I - T^{-1}A\right\|_A \leq \gamma, \quad 0 \leq \gamma < 1 \tag{3.1}$$

where I denotes the identity matrix and

$$\langle \cdot, \cdot \rangle_A \coloneqq \langle \cdot, A\cdot \rangle \tag{3.2}$$
$$\|\cdot\|_A \coloneqq \langle \cdot, \cdot \rangle_A. \tag{3.3}$$

Remark.
If M is symmetric, it is sufficient for A to be symmetric and positive semi definite. Let $A \in \mathbb{R}^{n\times n}$ be a matrix with eigenvalues $0 = \lambda_1 \leq \cdots \leq \lambda_n$, so we can shift all eigenvalues by a constant $s \in \mathbb{R}$ as follows:

$$\begin{aligned} Au &= \lambda Mu \\ Au + sMu &= \lambda Mu + sMu \\ (A + sM)u &= \underbrace{(\lambda + s)}_{=:\tilde{\lambda}} Mu. \end{aligned}$$

Therefore $(A + sM)$ is symmetric and positive definite with eigenvalues $s = \tilde{\lambda}_1 \leq \ldots \leq \tilde{\lambda}_n$ and the eigenvalues of A are $\forall i \in \{1, \ldots, n\} : \lambda_i = \tilde{\lambda}_i - s$.

3.1. Formulation of the iteration

Rewriting the problem from above we obtain the form $(A - \lambda M)u = 0$ where $A - \lambda M$ describes a regular pencil. Following this, we know that the constraint $(A - \lambda_i M)u_i = 0$ is satisfied by all real eigenvalues λ_i and their corresponding eigenvectors u_i. Given $A \in \mathbb{R}^{n\times n}$ and $M \in \mathbb{R}^{n\times n}$ being symmetric and positive

definite matrices, the minimum of the Rayleigh quotient for the generalized eigenvalue problem converges to λ_1. The Rayleigh quotient is defined as

$$\lambda(u) = \frac{\langle u, Au\rangle}{\langle u, Mu\rangle}, \quad \text{where } u \in \mathbb{R}^n, u \neq 0 \tag{3.4}$$

where $\langle \cdot, \cdot \rangle$ denotes the standard scalar product (Nannen 2019, p. 107). For the minimization of eq. (3.4) we generate a sequence of nonzero vectors by using the gradient of the Rayleigh quotient. Since A is symmetric we obtain by applying the product rule for derivatives:

$$\frac{d}{dx}(x^T Ax) = x^T A + x^T A^T = x^T(A + A^T) = 2x^T A \tag{3.5}$$

and the symmetry of A also yields that

$$x^T A = Ax. \tag{3.6}$$

The first equality of eq. (3.5) holds since the derivative of $x^T Ax$ in terms of the latter x equals $x^T A$ and for the derivative in terms of the former x we use the equality $x^T Ax = x^T A^T x$. Hence the second summand has to be $x^T A^T$.
Computing the gradient of $\lambda(u)$ by using eqs. (3.5) and (3.6) and the quotient rule for derivatives, leads to:

$$\begin{aligned}
\nabla\lambda(u) &= \nabla\left(\frac{u^T Au}{u^T Mu}\right) \\
&= \frac{\nabla(u^T Au)\cdot u^T Mu - \nabla(u^T Mu)\cdot(u^T Au)}{(u^T Mu)^2} \\
&= \frac{2u^T A\cdot u^T Mu - 2u^T M\cdot(u^T Au)}{(u^T Mu)^2} \\
&= \frac{2u^T}{u^T Mu}\cdot\left(A - \frac{Mu^T Au}{u^T Mu}\right) \\
&= \frac{2}{u^T Mu}\cdot(Au - \lambda(u)Mu)
\end{aligned}$$

Note that the $\lambda(u)$ on the right hand side is the generalized eigenvalue problem $Au - \lambda Mu = 0$. So the eigenvalues are $\lambda = \lambda(u)$ with corresponding eigenvector u. According to Knyazev, Neymeyr the gradient of the Rayleigh quotient on a T-based scalar product can be formulated as:

$$\nabla_{T^{-1}}\lambda(u) = \frac{2}{u^T Mu}T^{-1}(Au - M\lambda(u)u) \tag{3.7}$$

Using a two-term gradient minimization leads to the formulation:

$$u^{(i+1)} = u^{(i)} - \alpha^{(i)} T^{-1}(Au^{(i)} - M\lambda^{(i)}u^{(i)}) \tag{3.8}$$

where $\alpha^{(i)}$ is a scalar step size. If $\forall i \in \mathbb{N} : \alpha^{(i)} = 1$ we obtain the so called *Preconditioned Inverse Iteration* (PINVIT). A better choice for $\alpha^{(i)}$ is a minimization factor of the Rayleigh quotient on the two-dimensional subspace $\text{span}\{u^{(i)}, \omega^{(i)}\}$ according to the Rayleigh-Ritz method, where $\omega^{(i)} \coloneqq T^{-1}(Au^{(i)} - \lambda(u^{(i)})Mu^{(i)})$. Hence we have to solve a 2×2 generalized eigenvalue problem of the form $ay = \lambda my$ with matrices

$$a = \begin{pmatrix} \langle u^{(i)}, u^{(i)} \rangle_A & \langle u^{(i)}, \omega^{(i)} \rangle_A \\ \langle \omega^{(i)}, u^{(i)} \rangle_A & \langle \omega^{(i)}, \omega^{(i)} \rangle_A \end{pmatrix}, m = \begin{pmatrix} \langle u^{(i)}, u^{(i)} \rangle_M & \langle u^{(i)}, \omega^{(i)} \rangle_M \\ \langle \omega^{(i)}, u^{(i)} \rangle_M & \langle \omega^{(i)}, \omega^{(i)} \rangle_M \end{pmatrix}.$$

So $u^{(i+1)}$ can be written equivalently to eq. (3.8) as:

$$u^{(i+1)} = \xi^{(i)} u^{(i)} + \delta^{(i)} \omega^{(i)} \tag{3.9}$$

where $y = (\xi^{(i)}, \delta^{(i)})$ is the eigenvector regarding the smaller eigenvalue. The solutions to the mentioned problem are the roots of the quadratic characteristic equation (A. V. Knyazev and Neymeyr 2003). The Rayleigh-Ritz method is summarized in the following algorithm.

Algorithm 2 The Rayleigh-Ritz Method (Schofield, Chelikowsky, and Saad 2012)

Input: Hamiltonian matrix $A \in \mathbb{R}^{N\times N}$, device to compute an orthogonal basis

1: compute a orthogonal basis $V \in \mathbb{R}^{N\times m}$ approximating the eigenspace corresponding to m eigenvectors;
2: $\tilde{A} = V^H A V$;
3: $\tilde{M} = V^H M V$;
4: solve $\tilde{A}v = \tilde{\lambda}\tilde{M}v$;
5: compute $\tilde{u} = Vv$;

Output: The approximation $\tilde{\lambda}$ and $\tilde{u}$ to the eigenvalue λ and corresponding eigenvector u.

As our goal is implementing an eigenvalue solver according to eq. (3.8), we have to define a stopping criterion. Therefore it is convenient to choose the preconditonernorm

$$\|\cdot\|_{T^{-1}} \coloneqq \langle \cdot, T^{-1}\cdot \rangle \tag{3.10}$$

$$\implies \left\|r^{(i)}\right\|_{T^{-1}} = \langle r^{(i)}, T^{-1}r^{(i)} \rangle = \langle r^{(i)}, \omega^{(i)} \rangle = r^{(i)}\omega^{(i)} \tag{3.11}$$

of the residual $r^{(i)} = Au^{(i)} - \lambda^{(i)} Mu^{(i)}$. Hence we iterate until $\left\|r^{(i)}\right\|_{T^{-1}} < \tau$ for a given tolerance τ. Formulating an algorithm for the LOPSD Method can be done as follows:

Algorithm 3 The LOPSD Method

Input: starting vector $u^{(0)}$ and corresponding $\lambda^{(0)}$ tolerance τ, devices to compute: Au, Mu, and $T^{-1}u$ for a given vector u, the vector inner product $\langle u, v \rangle$, and $\|r\|_{T^{-1}}$

1: select $u^{(0)}$;
2: **for** $i = 0, \ldots,$MaxIterations **do**
3: $\quad r^{(i)} := Au^{(i)} - \lambda^{(i)} Mu^{(i)}$;
4: $\quad \omega^{(i)} := T^{-1} r^{(i)}$;
5: $\quad$ **if** $\|r\|_{T^{-1}} < \tau$ **then**
6: $\quad\quad$ break;
7: $\quad$ Use the Rayleigh–Ritz method for the pencil $A - \lambda M$ on the trial subspace $\text{span}\{\omega^{(i)}, u^{(i)}\}$
8: $\quad \lambda^{(i+1)} := \tilde{\lambda}^{(i+1)}$; // the smallest Ritz value
9: $\quad u^{(i+1)} := \xi^{(i)} u^{(i)} + \delta^{(i)} \omega^{(i)}$; // the Ritz vector corresponding to $\lambda^{(i)}$

Output: The approximation $\lambda^{(k)}$ and $u^{(k)}$ to the smallest eigenvalue λ and corresponding eigenvectors.

3.2. Complexity

The computational complexity of the Rayleigh-Ritz Method in algorithm 2 equals $\mathcal{O}(Nm^2)$, hence we obtain $\mathcal{O}(N)$ since in our case m is constant. Now we consider the cost of algorithm 3. The overall complexity of this algorithm is dependent on the matrix-vector multiplications in line 3. If A is sparse, this multplication can be realized with $\mathcal{O}(N)$. Therefore the LOPSD Method can be implemented with complexity $\mathcal{O}(N)$.

3.3. Convergence of PINVIT

In this section we want to provide a convergence estimate of the PINVIT. So we choose $\alpha = 1$, $M = I$ to simplify the proof and for convenience we write u, u' instead of $u^{(i)}, u^{(i+1)}$. Let the preconditioner T satisfy the constraint (3.1) as in the previous subsection. According to Theorem 1.1 of (Neymeyr 2001b) if $\lambda_k < \lambda < \lambda_{k+1}$ where $\lambda = \lambda(u)$ for some iterate u, the Rayleigh quotient takes its maximal value at $\lambda_{k,k+1}(\lambda, \gamma)$ whereas:

$$\begin{aligned}\lambda_{i,j}(\lambda,\gamma) &= \lambda\lambda_i\lambda_j(\lambda_i+\lambda_j-\lambda)^2 \\ &\quad\times[\gamma^2(\lambda_j-\lambda)(\lambda-\lambda_i)(\lambda\lambda_j+\lambda\lambda_i-\lambda_i^2-\lambda_j^2) \\ &\quad-2\gamma\sqrt{\lambda_i\lambda_j}(\lambda-\lambda_i)(\lambda_j-\lambda) \\ &\quad\times\sqrt{\lambda_i\lambda_j+(1-\gamma^2)(\lambda-\lambda_i)(\lambda_j-\lambda)} \\ &\quad-\lambda(\lambda_i+\lambda_j-\lambda)(\lambda\lambda_j+\lambda\lambda_i-\lambda_i^2-\lambda_i\lambda_j-\lambda_j^2)]^{-1} \end{aligned} \tag{3.12}$$

The inequality $\lambda' := \lambda(u') \leq \lambda_{k,k+1}(\lambda, \gamma)$ is sharp for a certain vector u and preconditioner T (A. V. Knyazev and Neymeyr 2003).

Theorem 3.3.1
"Let $u \in \mathbb{R}^n$ and let $\lambda = \lambda(u) \in [\lambda_1, \lambda_n[$ be its Rayleigh quotient where $\lambda_1 \leq \cdots \leq \lambda_n$ are the eigenvalues of A. The preconditioner is assumed to satisfy (3.1) for some $\gamma \in [0, 1[$. If $\lambda = \lambda(u) \in [\lambda_k, \lambda_{k+1}[$, then it holds for the Rayleigh quotient $\lambda' := \lambda(u')$ with u' computed by eq. (3.8) with $\alpha = 1$ that either $\lambda' < \lambda_k$ (unless $k = 1$), or $\lambda' \in [\lambda_k, \lambda[$. In the latter case,

$$\frac{\lambda'-\lambda_k}{\lambda_{k+1}-\lambda'} \leq (q(\gamma,\lambda_k,\lambda_{k+1}))^2\frac{\lambda-\lambda_k}{\lambda_{k+1}-\lambda}, \tag{3.13}$$

where

$$q(\gamma,\lambda_k,\lambda_{k+1}) = \gamma+(1-\gamma)\frac{\lambda_k}{\lambda_{k+1}} = 1-(1-\gamma)\left(1-\frac{\lambda_k}{\lambda_{k+1}}\right) \tag{3.14}$$

is the convergence factor." (A. V. Knyazev and Neymeyr 2003)

Proof

As a reformulation of eq. (3.13) we obtain:

$$\frac{\lambda' - \lambda_k}{\lambda_{k+1} - \lambda'} \frac{\lambda_{k+1} - \lambda}{\lambda - \lambda_k} \leq (q(\gamma, \lambda_k, \lambda_{k+1}))^2$$

Now we want to show that this inequality holds $\forall \lambda, \lambda'$. Therefore we consider the maximum of the function on left hand side:

$$\frac{\lambda' - \lambda_k}{\lambda_{k+1} - \lambda'} \frac{\lambda_{k+1} - \lambda}{\lambda - \lambda_k}$$

We know that $\lambda \in [\lambda_k, \lambda_{k+1}[$ and $\lambda' \in [\lambda_k, \lambda[$. So we directly obtain

$$\lambda' - \lambda_k \geq 0, \quad \lambda_{k+1} - \lambda' \geq 0$$

Furthermore $\lambda' \leq \lambda_{k,k+1}(\lambda, \gamma)$ (sharp), hence we get the inequality

$$\frac{\lambda_{k,k+1} - \lambda_k}{\lambda_{k+1} - \lambda_{k,k+1}} \frac{\lambda_{k+1} - \lambda}{\lambda - \lambda_k} \geq \frac{\lambda' - \lambda_k}{\lambda_{k+1} - \lambda'} \frac{\lambda_{k+1} - \lambda}{\lambda - \lambda_k}$$

Now we want to show that the maximum $\forall \lambda \in [\lambda_k, \lambda_{k+1}[$ of

$$\frac{\lambda_{k,k+1} - \lambda_k}{\lambda_{k+1} - \lambda_{k,k+1}} \frac{\lambda_{k+1} - \lambda}{\lambda - \lambda_k} \tag{3.15}$$

is equal to $(q(\gamma, \lambda_k, \lambda_{k+1}))^2$. It is rather easy to see that the maximum of eq. (3.15) is attained if λ approaches λ_k because $\lambda \in [\lambda_k, \lambda_{k+1}[$ leads to the conlusion:

$$\lambda_{k+1} - \lambda \geq 0, \quad \lambda - \lambda_k \geq 0$$

As it is not convenient to prove this statement with the defintion (3.12) of $\lambda_{k,k+1}$ we choose another method using "mini-dimensional analysis" in $\text{span}\{u_k, u_{k+1}\}$ (ibid.). The rest of this proof is a reproduction of Knyazef's and Neymeyr's proof of theorem 3.3.1 adopting the notations of (Neymeyr 2001a) and choosing $k = 1, k+1 = 2$ (w.l.o.g). According to Lemma 2.3 of (ibid.) the set of all iterates E_γ, for a fixed vector u and preconditioner T satisfying (3.1) is a ball in the A-based scalar product. Intersecting this ball with the two-dimensional subspace $\text{span}\{u_k, u_{k+1}\}$ results in a disc. In the following r will denote the radius of the disc, c the center x, y the cartesian coordinates of the center given by A-orthonormal eigenvectors u_1, u_2 of A, where $\text{span}\{u_1, u_2\}$ has to coincide with $\text{span}\{u_k, u_{k+1}\}$.

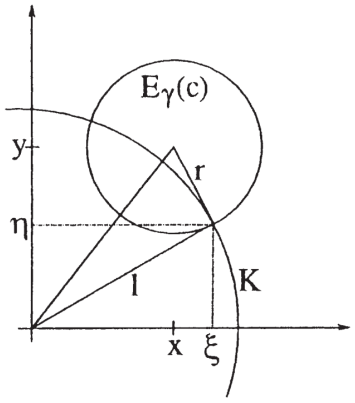


Figure 1: Geometric representation (Neymeyr 2001a)

The formulas for x, y and r illustrated in fig. 1 are taken from (Neymeyr 2001a):

$$x = \sqrt{\frac{\lambda(\lambda - \lambda_1)}{\lambda_2(\lambda_2 - \lambda_1)}}, \quad y = \sqrt{\frac{\lambda(\lambda_2 - \lambda)}{\lambda_1(\lambda_2 - \lambda_1)}}, \quad r = \gamma\sqrt{\frac{(\lambda - \lambda_1)(\lambda_2 - \lambda)}{\lambda_1\lambda_2}} \tag{3.16}$$

Neymeyr proved, that the Rayleigh quotient attains its unique maximum on the disc $\text{span}\{u_k, u_{k+1}\} \cap E_\gamma$. He obtained the formula:

$$\lambda_{1,2}(\lambda, \gamma) = \frac{\eta^2 + \xi^2}{\eta^2/\lambda_1 + \xi^2/\lambda_2} \tag{3.17}$$

where the intersection point maximizing the Rayleigh quotient has the form:

$$(\eta, \xi) = \left(\sqrt{l^2 - \xi^2}\,,\ \frac{xl^2 + ryl}{x^2 + y^2}\right), \quad l = \sqrt{x^2 + y^2 - r^2} \tag{3.18}$$

Considering eq. (3.17) we have an equivalent but more convenient form of eq. (3.12). For given x, y and r (dependent on $\gamma, \lambda, \lambda_1, \lambda_2$) we are able to evaluate $\lambda_{1,2}(\lambda, \gamma)$ with the just derived formula. From eq. (3.17) we obtain

$$\frac{\lambda_{1,2} - \lambda_1}{\lambda_2 - \lambda_{1,2}} = \frac{\frac{\eta^2+\xi^2}{\eta^2/\lambda_1+\xi^2/\lambda_2} - \lambda_1}{\lambda_2 - \frac{\eta^2+\xi^2}{\eta^2/\lambda_1+\xi^2/\lambda_2}} = \frac{\lambda_1\xi^2(\frac{1}{\lambda_1} - \frac{1}{\lambda_2})}{\lambda_2\eta^2(\frac{1}{\lambda_1} - \frac{1}{\lambda_2})} = \frac{\lambda_1\xi^2}{\lambda_2\eta^2} = \frac{\lambda_1}{\lambda_2}\frac{(xl + ry)^2}{(x^2 + y^2)^2 - (xl + ry)^2},$$

where

$$(x^2 + y^2)^2 - (xl + ry)^2 = (yl - xr)^2.$$

The denominator $yl - xr$ is positive because of $y > r$. As a result we get

$$\frac{\lambda_{1,2} - \lambda_1}{\lambda_2 - \lambda_{1,2}} = \frac{\lambda_1}{\lambda_2} \frac{(xl + ry)^2}{(yl - xr)^2} \tag{3.19}$$

Furthermore the expressions for x and y from eq. (3.16) imply

$$\frac{\lambda_2 - \lambda}{\lambda - \lambda_1} = \frac{y^2 \lambda_1}{x^2 \lambda_2} \tag{3.20}$$

Using eqs. (3.19) and (3.20), the convergence factor

$$\frac{\lambda_{1,2} - \lambda_1}{\lambda_2 - \lambda_{1,2}} \frac{\lambda_2 - \lambda}{\lambda - \lambda_1} = \frac{\lambda_1^2 y^2}{\lambda_2^2 x^2} \frac{(xl + ry)^2}{(yl - xr)^2} = (q[\lambda])^2$$

can be written as

$$q[\lambda] = \frac{\lambda_1 y(xl + ry)}{\lambda_2 x(yl - xr)} = \frac{\lambda_1}{\lambda_2} \frac{1 + yr/xl}{1 - xr/yl} > 0. \tag{3.21}$$

Now can compute yr/xl and xr/yl directly by using eq. (3.16):

$$\frac{yr}{xl} = \gamma(\lambda_2 - \lambda) \left(\frac{\lambda_2}{\lambda_1} \right)^{1/2} z^{-1/2} \tag{3.22}$$

$$\frac{xr}{yl} = \gamma(\lambda - \lambda_1) \left(\frac{\lambda_1}{\lambda_2} \right)^{1/2} z^{-1/2} \tag{3.23}$$

with $z = \gamma(\lambda_1 - \lambda)(\lambda_2 - \lambda) + \lambda(\lambda_1 + \lambda_2 - \lambda) > 0$. With eqs. (3.22) and (3.23) we obtain from eq. (3.21)

$$q[\lambda] = \frac{\sqrt{\lambda_1/\lambda_2} z^{1/2} + \gamma(\lambda_2 - \lambda)}{\sqrt{\lambda_2/\lambda_1} z^{1/2} - \gamma(\lambda - \lambda_1)} \tag{3.24}$$

Having a simplified expression for $q[\lambda]$ our next goal is to find a sharp upper bound for q which is not dependent of λ. Therefore we want show that

$$q'[\lambda] < 0$$

Taking the derivative with respect to λ this is equivalent to

$$\gamma \sqrt{\lambda_1 \lambda_2}(\lambda_2 - \lambda_1) < (\lambda_2 - \lambda_1) z^{1/2} + \left(\frac{d}{d\lambda} z^{1/2} \right) (\lambda_2(\lambda_2 - \lambda) + \lambda_1(\lambda - \lambda_1)).$$

By squaring both sides and inserting z and $\frac{d}{d\lambda}z^{1/2}$ we obtain that the last inequality holds if

$$(1-\gamma^2)(\lambda_2-\lambda_1)^2(\lambda_1+\lambda_2-\lambda)^2[(1+\gamma)\lambda_1+(1-\gamma)\lambda_2] \cdot[(1-\gamma)\lambda_1+(1+\gamma)\lambda_2]$$

is positive. This is true under the assumptions $0 \leq \gamma < 1$ and $0 < \lambda_1 \leq \lambda < \lambda_2$. As $q[\lambda]$ is strictly decreasing, it takes its largest value at $\lambda = \lambda_1$:

$$q[\lambda_1] = \frac{\sqrt{\lambda_1/\lambda_2}z^{1/2}+\gamma(\lambda_2-\lambda_1)}{\sqrt{\lambda_2/\lambda_1}z^{1/2}-\gamma(\lambda_1-\lambda_1)} = \frac{\sqrt{\lambda_1/\lambda_2}\sqrt{\lambda_1\lambda_2}+\gamma(\lambda_2-\lambda_1)}{\sqrt{\lambda_2/\lambda_1}\sqrt{\lambda_1\lambda_2}}$$

$$= \frac{\lambda_1+\gamma(\lambda_2-\lambda_1)}{\lambda_2} = \frac{\lambda_1}{\lambda_2}+\gamma\left(1-\frac{\lambda_1}{\lambda_2}\right)$$

$$= 1-(1-\gamma)\left(1-\frac{\lambda_1}{\lambda_2}\right)$$

Now we have obtained eq. (3.14) for $k = 1$ and $k+1 = 2$ (A. V. Knyazev and Neymeyr 2003). □

3.4. Implementation in NGSolve

An implementation of the block version of the LOPSD method is provided below. If $num = 1$ it is equivalent to the LOPSD method discussed in section 3.

```
def lopsd(mesh,fes,num,A,M,pre,maxNumIterations):
    u = fes.TrialFunction()
    v = fes.TestFunction()

    lam = EigenValues_Preconditioner(mat=A.mat, pre=pre)

    u = GridFunction(fes, multidim=num)

    # using multivectors for better performance
    uvecs = MultiVector(u.vec, num)
    vecs = MultiVector(u.vec, 2 * num)

    for v in vecs[0:num]:
        v.SetRandom()
    uvecs[:] = pre * vecs[0:num]
    lams = Vector(num * [1])
    res = Vector((maxNumIterations) * [1])
    numIterations = 0
```

```
19
    # weird python do while loop
21  while True:
        numIterations += 1
23      vecs[0:num] = A.mat * uvecs[0:num] - (M.mat * uvecs[0:num]).
    Scale(lams)
        vecs[num:2 * num] = pre * vecs[0:num]
25
        #T-norm res
27      r = InnerProduct(vecs[num], vecs[0])
        for i in range(1, num):
29          tmp = InnerProduct(vecs[num + i], vecs[i])
            if (r < tmp):
31              r = tmp
        res[numIterations-1] = r
33
        vecs[0:num] = uvecs[0:num]
35
        vecs.Orthogonalize()
37
        asmall = InnerProduct(vecs, A.mat * vecs)
39      msmall = InnerProduct(vecs, M.mat * vecs)

41      ev, evec = scipy.linalg.eigh(a=asmall, b=msmall)
        prev = lams
43      lams = Vector(ev[0:num])
        print(numIterations, ":", [l for l in lams])
45
        uvecs[0:num] = vecs * Matrix(evec[:, 0:num])
47
        print("res:", res[numIterations-1], "\n")
49
        if (res[numIterations-1] < 10e-16 or numIterations >=
    maxNumIterations):
51          break

53  for j in range(num):
        u.vecs[j][:] = 0.0
55      u.vecs[j].data += uvecs[j]

57  Draw(u, mesh, "mode")
    SetVisualization(deformation=True)
59  return res,numIterations
```

Listing 1: Implementation of LOBPSD in NGSolve

4. Locally Optimal Preconditioned Conjugate Gradient - LOPCG

To get a faster converging preconditioned eigensolver, the cg-method (Hirnschall and Dorigo 2020) can be used instead of the steepest descent method described in section 3 to minimize the rayleigh-quotient $\lambda(u)$. Instead of minimizing in the direction of steepest descent, the search directions

$$r^{(i)} := \nabla\lambda(u^{(i)}) \tag{4.1}$$

$$p^{(0)} := -r^{(0)} \tag{4.2}$$

$$p^{(i)} := -r^{(i)} + \underbrace{\frac{r^{(i)*}Mr^{(i)}}{r^{(i-1)*}Mr^{(i-1)}}}_{=:\beta^{(i)}} p^{(i-1)} \tag{4.3}$$

are used. Therefore consecutive search directions are A-orthogonal (Feng and Owen 1996):

$$(p^{(i)}, p^{(i-1)})_A = 0. \tag{4.4}$$

Using eqs. (3.8) and (4.3), the next iterate u^{i+1} is defined as

$$\begin{aligned}\lambda\left(u^{(i+1)}\right) &= \min_{\xi^{(i)},\delta^{(i)}} \left\{\lambda\left(\xi^{(i)}u^{(i)} - \delta^{(i)}p^{(i)}\right)\right\} \\ &= \min_{\xi^{(i)},\delta^{(i)}} \left\{\lambda\left(\xi^{(i)}u^{(i)} - \delta^{(i)}(r^{(i)} + \beta^{(i)}p^{(i-1)})\right)\right\}.\end{aligned} \tag{4.5}$$

Replacing the residual $r^{(i)}$ in eq. (4.5) with the preconditioned residual $\omega^{(i)}$ results in

$$\lambda\left(u^{(i+1)}\right) = \min_{\delta^{(i)},\xi^{(i)}} \left\{\lambda\left(\xi^{(i)}u^{(i)} - \delta^{(i)}(\omega^{(i)} + \beta^{(i)}p^{(i-1)})\right)\right\}. \tag{4.6}$$

This method can potentially be improved by optimizing all three parameters ξ, δ and γ at once as

$$\begin{aligned}&\min_{\xi^{(i)},\delta^{(i)},\gamma^{(i)}} \left\{\lambda\left(\xi^{(i)}u^{(i)} - \delta^{(i)}\omega^{(i)} + \gamma^{(i)}p^{(i-1)}\right)\right\} \leq \\ &\min_{\xi^{(i)},\delta^{(i)}} \left\{\lambda\left(\xi^{(i)}u^{(i)} - \delta^{(i)}(\omega^{(i)} + \beta^{(i)}p^{(i-1)})\right)\right\}.\end{aligned} \tag{4.7}$$

The new procedure is now locally optimal. (A. Knyazev 2000)

Using the Rayleigh-Ritz-Method (algorithm 2) a new iterate is computed on a three-dimensional trial-subspace span$\{\omega^{(i)}, u^{(i)}, p^{(i)}\}$. The subspace used is therefore equal to the one in section 3 expanded by the previous search direction $p^{(i)}$, leading to the following, slightly modified version of algorithm 3:

Algorithm 4 The LOPCG Method (A. Knyazev 2000)

Input: starting vector $u^{(0)}$, tolerance τ, devices to compute: Au, Mu, and $T^{-1}u$ for a given vector u, the vector inner product $\langle u, v\rangle$, and $\|r\|_{T^{-1}}$

1: select $u^{(0)}$, and set $p^{(0)} = 0$;
2: **for** $i = 0, \ldots,$MaxIterations **do**
3: $r := Au^{(i)} - \lambda^{(i)} Mu^{(i)}$;
4: $\omega^{(i)} := T^{-1}r$;
5: **if** $\|r\|_{T^{-1}} < \tau$ **then**
6: break;
7: Use the Rayleigh–Ritz method for the pencil $A - \lambda M$ on the trial subspace span$\{\omega^{(i)}, u^{(i)}, p^{(i)}\}$;
8: $\lambda^{(i+1)} := \tilde{\lambda}^{(i+1)}$; //the smallest Ritz value
9: $u^{(i+1)} := \alpha^{(i)}\omega^{(i)} + \tau^{(i)}u^{(i)} + \gamma^{(i)}p^{(i)}$; //the Ritz vector corresponding to $\lambda^{(i)}$
10: $p^{(i+1)} := \alpha^{(i)}\omega^{(i)} + \gamma^{(i)}p^{(i)}$;

Output: The approximation $\lambda^{(k)}$ and $u^{(k)}$ to the smallest eigenvalue λ and corresponding eigenvector.

Remark. In order to make the implementation of algorithm 4 easier, we are using $p^{(i)}$ and $p^{(i+1)}$ instead of $p^{(i-1)}$ and $p^{(i)}$ as described in eqs. (4.2) and (4.3). By initializing $p^{(0)} = 0$, $p^{(1)}$ is calculated as $p^{(1)} = \alpha^{(0)}\omega^{(0)}$ which is equivalent to eq. (4.2).

4.1. Complexity

As the computational complexity of the Rayleigh-Ritz method scales with $\mathcal{O}(n)$, the above mentioned alteration of algorithm 3 as seen in algorithm 4 does not change the overall complexity. In fig. 2 the time per iteration for both LOPSD and LOPCG can be seen together with $\mathcal{O}(n)$ for comparison. The two dashed lines marked lopsd and lopcg are first degree polynomials fitted to the actual data using $np.polyfit(data, 1)$.

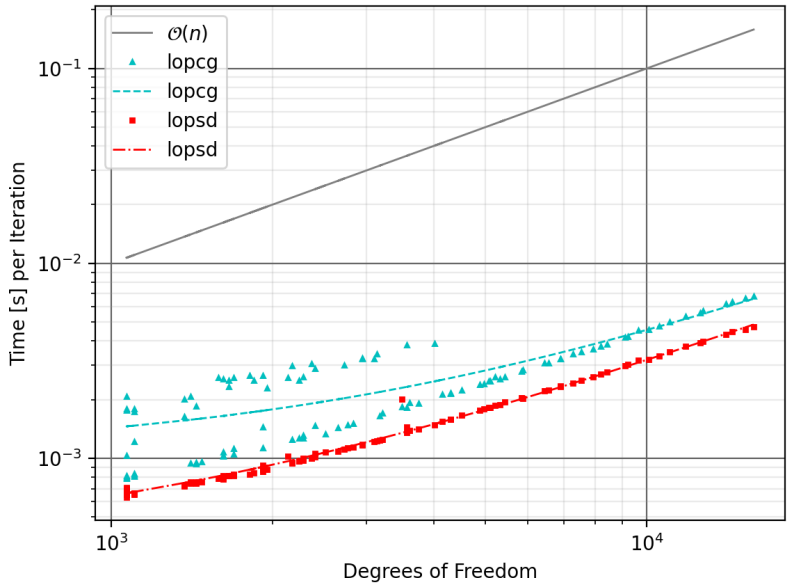
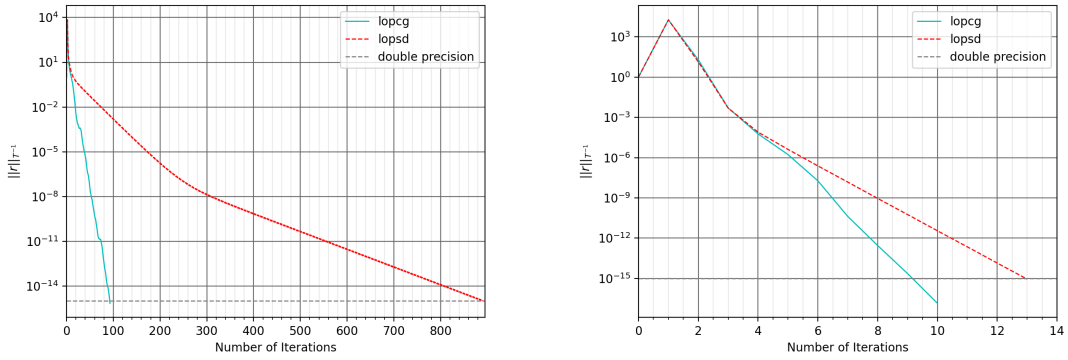


Figure 2: Comparing the computational complexity of algorithms 3 and 4

4.2. Convergence

Since the trial subspace used in the LOPCG method includes the one used in section 3, the rate of convergence is naturally at least as fast as described in section 3.3.
An adequate theory of convergence for the LOPCG method (or similar methods) is not yet known (A. Knyazev 2000). Therefore we will settle with a numerical comparison of algorithms 3 and 4. In our tests, the LOPSD method seems to be much more preconditioner dependent than the LOPCG method.



(a) Local (b) Multigrid

Figure 3: Comparing the rate of convergence of algorithms 3 and 4 using different preconditioners

4.3. Block Iteration - LOBPCG

Until now we only approximated the smallest eigenvalue λ_1 and corresponding eigenvector of a symmetric eigenproblem $Au = \lambda Mu$. If, however, we care about the m smallest eigenvalues/vectors, algorithm 4 can be adapted to use the $3m$ dimensional subspace $\text{span}\{\omega_1^{(i)}, \ldots, \omega_m^{(i)}, u_1^{(i)} \ldots u_m^{(i)}, p_1^{(i)}, \ldots p_m^{(i)}\}$. Using the Rayleigh-Ritz method, the k^{th} smallest Ritzvalue will be used as an approximation to the k^{th} smallest eigenvalue λ_k. The resulting procedure is called *Locally Optimal Block Preconditioned Conjugate Gradient Method* (LOBPCG).

Algorithm 5 The LOBPCG Method (A. Knyazev et al. 2007)

Input: m starting vectors $u_1^{(0)}, \ldots, u_m^{(0)}$, tolerance τ, devices to compute: Au, Mu, and $T^{-1}u$ for a given vector u, the vector inner product $\langle u, v \rangle$, and $\|r\|_{T^{-1}}$

1: set $J \coloneqq \{1, \ldots, m\}$
2: select $u_j^{(0)}$, and set $p_j^{(0)} = 0$, $j \in J$
3: **for** $i = 0, \ldots,$MaxIterations **do**
4: $\quad r_j \coloneqq Au_j^{(i)} - \lambda_j^{(i)} M u_j^{(i)}$, $j \in J$;
5: $\quad \omega_j^{(i)} \coloneqq T^{-1} r_j$, $j \in J$;
6: $\quad$ **if** $\max\left\{\|r_j\|_{T^{-1}} : j \in J\right\} < \tau$ **then**
7: $\quad\quad$ break;
8: $\quad$ Use the Rayleigh–Ritz method for the pencil $A - \lambda M$ on the trial subspace $\mathrm{Span}\{\omega_1^{(i)}, \ldots, \omega_m^{(i)}, u_1^{(i)} \ldots u_m^{(i)}, p_1^{(i)}, \ldots p_m^{(i)} : j \in J\}$;
9: $\quad \lambda_j^{(i+1)} \coloneqq \tilde{\lambda}_j^{(i+1)}$; //the jth smallest Ritz value
10: $\quad u_j^{(i+1)} \coloneqq \sum_{k=1,\ldots,m} \alpha_k^{(i)} \omega_k^{(i)} + \tau_k^{(i)} u_k^{(i)} + \gamma_k^{(i)} p_k^{(i)}$, $j \in J$; //the Ritz vector corresponding to $\lambda_j^{(i+1)}$
11: $\quad p_j^{(i+1)} \coloneqq \sum_{k=1,\ldots,m} \alpha_k^{(i)} \omega_k^{(i)} + \gamma_k^{(i)} p_k^{(i)}$, $j \in J$;

Output: The approximation $\lambda_j^{(k)}$ and $u_j^{(k)}$ to the smallest eigenvalue λ_j and corresponding eigenvectors, $j = 1, \ldots, m$.

4.4. Choosing a Block Size

According to Knyazev (A. Knyazev et al. 2007) the block size should be chosen, if possible, to provide a large gap between the first m eigenvalues and the rest of the spectrum as this typically leads to better convergence.
However, as algorithm 5 uses the Rayleigh-Ritz method which requires solving a $3m \times 3m$ eigenproblem each iteration-step, the computational cost will rise if the number of eigenvalues m is increased.

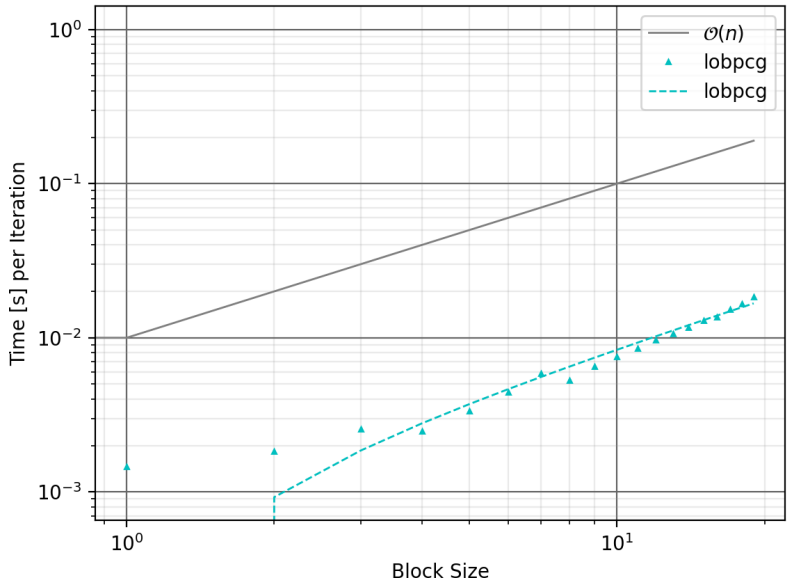


Figure 4: Comparing time per iteration for different block sizes

4.5. Locking Eigenvectors

As not all eigenvectors converge with the same rate we will now take a look at how to reduce the computational complexity by ignoring eigenvectors that have already reached a given exactness.
This can be done using an index set J containing the indices of eigenvectors that have not yet converged adequately. The next guess $u^{(i+1)}$ is then calculated using the Rayleigh-Ritz method on the trial subspace span$\{\omega_j^{(i)}, u_1^{(i)} \dots u_m^{(i)}, p_1^{(i)}, \dots p_m^{(i)} : j \in J\}$. As even "ignored" eigenvectors can still change from one iteration to the next, this method is called *soft locking*, leading to algorithm 6:

Algorithm 6 The LOBPCG Method with Soft Locking (A. Knyazev et al. 2007)

Input: m starting vectors $u_1^{(0)}, \ldots, u_m^{(0)}$, tolerance τ, devices to compute: Au, Mu, and $T^{-1}u$ for a given vector u, the vector inner product $\langle u, v \rangle$, and $\|r_j\|_{T^{-1}}$

set $J \coloneqq \{1, \ldots, m\}$
select $u_j^{(0)}$, and set $p_j^{(0)} = 0$, $j \in J$
for $i = 0, \ldots,$MaxIterations **do**
    $r_j \coloneqq Au_j^{(i)} - \lambda_j^{(i)} M u_j^{(i)}$, $j \in J$;
    $\omega_j^{(i)} \coloneqq T^{-1} r_j$, $j \in J$;
    remove j from J if $\|r_j\|_{T^{-1}} < \tau$, $j \in J$;
    if $|J| = 0$ **then**
        break;
    Use the Rayleigh–Ritz method for the pencil $A - \lambda M$ on the trial subspace Span$\{\omega_j^{(i)}, u_1^{(i)} \ldots u_m^{(i)}, p_1^{(i)}, \ldots p_m^{(i)} : j \in J\}$;
    $\lambda_j^{(i+1)} \coloneqq \tilde{\lambda}_j^{(i+1)}$; //the jth smallest Ritz value
    $u_j^{(i+1)} \coloneqq \sum_{k=1,\ldots,m} \alpha_k^{(i)} \omega_k^{(i)} + \tau_k^{(i)} u_k^{(i)} + \gamma_k^{(i)} p_k^{(i)}$, $j \in J$; //the Ritz vector corresponding to $\lambda_j^{(i+1)}$
    $p_j^{(i+1)} \coloneqq \sum_{k=1,\ldots,m} \alpha_k^{(i)} \omega_k^{(i)} + \gamma_k^{(i)} p_k^{(i)}$, $j \in J$;

Output: The approximation $\lambda_j^{(k)}$ and $u_j^{(k)}$ to the smallest eigenvalue λ_j and corresponding eigenvectors, $j = 1, \ldots, m$.

Note that soft locking does not improve the rate of convergence, the computational complexity is reduced as more and more eigenvectors get locked.

4.6. Implementation in NGSolve

Although algorithms 4 to 6 get progressively better, the use of NGSolve multivectors allowed us to implement algorithm 5 such that it is overall faster than algorithm 6. Therefore an implementation of algorithm 5 is provided in Listing 2:

```
def lobpcg(mesh,fes,num,A,M,pre,maxNumIterations):
    u = fes.TrialFunction()
    v = fes.TestFunction()

    lam = EigenValues_Preconditioner(mat=A.mat, pre=pre)

    u = GridFunction(fes, multidim=num)

    # using multivectors for better performance
    uvecs = MultiVector(u.vec, num)
    vecs = MultiVector(u.vec, 2 * num)
```

```
    for v in vecs[0:num]:
        v.SetRandom()
    uvecs[:] = pre * vecs[0:num]
    lams = Vector(num * [1])
    res = Vector((maxNumIterations) * [1])
    numIterations = 0

    # weird python do while loop
    while True:
        numIterations += 1
        vecs[0:num] = A.mat * uvecs[0:num] - (M.mat * uvecs[0:num]).
   Scale(lams)
        vecs[num:2 * num] = pre * vecs[0:num]

        #T-norm res
        r = InnerProduct(vecs[num], vecs[0])
        for i in range(1, num):
            tmp = InnerProduct(vecs[num + i], vecs[i])
            if (r < tmp):
                r = tmp
        res[numIterations-1] = r

        vecs[0:num] = uvecs[0:num]

        vecs.Orthogonalize()

        asmall = InnerProduct(vecs, A.mat * vecs)
        msmall = InnerProduct(vecs, M.mat * vecs)

        ev, evec = scipy.linalg.eigh(a=asmall, b=msmall)
        prev = lams
        lams = Vector(ev[0:num])
        print(numIterations, ":", [l for l in lams])

        if (numIterations==1):
            tmp = MultiVector(u.vec, 2 * num)
            tmp[0:2*num] = vecs
            vecs = MultiVector(u.vec, 3 * num)
            vecs[0:2*num] = tmp[0:2 * num]

        uvecs[0:num] = vecs * Matrix(evec[:, 0:num])

        #todo: use span{w^i,x^i,p^i} instead of span{w^i,x^i,x^{i-1}}
    for better stability
        vecs[2 * num:3 * num] = vecs[0:num]

        print("res:", res[numIterations], "\n")

        if (res[numIterations] < 10e-16 or numIterations >=
```

```
    maxNumIterations):
            break

    for j in range(num):
        u.vecs[j][:] = 0.0
        u.vecs[j].data += uvecs[j]

    Draw(u, mesh, "mode")
    SetVisualization(deformation=True)
    return res,numIterations
```

Listing 2: Implementation of LOBPCG in NGSolve

5. Preconditioner

Choosing a optimal preconditioner results in a faster convergence. Given a preconditioner T, the inverse of T should approximate A^{-1} in the best way possible. Consider the simplified eigenvalue problem $Au = \lambda u$. Assume that the preconditioner T is symmetric and positive definite and approximates the matrix A such that

$$\left\|I - T^{-1}A\right\|_A \leq \gamma, \quad 0 \leq \gamma < 1$$

Since both A and T are symmetric positive definite it holds that

$$\delta_0\langle u, T^{-1}u\rangle \leq \langle u, Au\rangle \leq \delta_1\langle u, T^{-1}u\rangle, \quad 0 < \delta_0 \leq \delta_1$$

Faster convergence can be achieved by a smaller ratio of δ_1/δ_0 that can be viewed as the spectral condition number $\kappa(T^{-1}A)$ [3] of the preconditioned matrix. Choosing δ_0 and δ_1 or scaling T properly we obtain (A. V. Knyazev and Neymeyr 2003)

$$\gamma = \frac{\kappa(T^{-1}A) - 1}{\kappa(T^{-1}A) + 1}$$

It is known, that the same preconditioner used for linear systems $Ax = b$ can also be used to solve eigenvalue problems (A. Knyazev 2000). Hence from now on we restrict ourselves to linear systems.

5.1. Direct

A direct approach to solve a linear system is to calculate the inverse matrix to get $x = A^{-1}b$. For large matrices the inverse calculation is very expensive and thus not feasible. As in NGSolve a direct solver uses spare factorization, e.g. Incomplete LU decomposition, Incomplete Cholesky factorization and others.

5.1.1. Incomplete LU Decomposition

The normal LU decomposition splits the matrix $A = LU$ into a normalized lower triangular matrix L and an upper triangular matrix U. The decomposition of a sparse matrix is usually not sparse. Rewriting the problem gives $Ax = LUx = b$. The solution can be found using a forward substitution on $Ly = b$ to determine y and a backward substitution on $Ux = y$. We can determine the decomposition using algorithm 6.6 (Nannen 2019). The overall complexity is $1/3n^3 + \mathcal{O}(n^2)$.

[3] $\kappa(A) = |\lambda_{max}(A)/\lambda_{min}(A)|$, with respective eigenvalues.

Due to the high complexity, we try the iterative way using the ILU method. The incomplete factorization seeks triangular matrices L, U such that $A \approx LU$. The ILU(0) decomposition only considers non zero entries of the matrix A. Using a compressed sparse row format will reduce the computation time from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$. The ILU(0) decomposition can be computed as follows

Algorithm 7 Incomplete LU decomposition (Taylor 2007)

Input: Matrix A

```
for i = 2, ..., n do;
    for j = 1, ..., i − 1 do;
        if a_ik ≠ 0 and a_kk ≠ 0 then;
            a_ik = a_ik / a_kk;
            for j = k + 1, ..., n do;
                if a_ij ≠ 0 then;
                    a_ij = a_ij − a_ik a_kj
```

Output: Matrix A, U is the upper triangular part of the matrix and L the lower part with ones on the diagonal

In the end one gets $A = LU + R$ where R is an error matrix. For better approximations the ILU(p) [4] factorization is used, where each element gets a level of fill assigned (elements that become nonzero after the decomposition) . Larger values of p result in a slower but more accurate factorization. For the calculation of ILU(p), a determination of the sparsity structure of the matrix is needed. The sparsity pattern S needs to fulfill

$$S \subseteq \{1, \ldots, n\}^2, \quad \{(i,i) : 1 \leq i \leq n\}, \quad G(A) := \{(i,j) \in \mathbb{N}^2 : a_{ij} \neq 0\} \subseteq S$$

and the decomposition $A = LU + R$ satisfies

$$L_{ij} = U_{ij} = 0 \ \forall (i,j) \notin S, \quad R_{ij} = 0 \ \forall (i,j) \in S.$$

Afterwards a level of fill matrix must be created, that indicates which zero elements can become a nonzero entry after the factorization. A possible determination of the level of fill matrix would be

[4] The exact factorization is achieved if $p = n - 1$, with n being the dimension of the matix

Algorithm 8 Level Of Fill Matrix (Meerbergen 2007)

Input: Matrix A

1: $Lev(a_{i,j}) = 0,\ \forall (i,j) \in S$ and $Lev(a_{i,j}) = \infty,\ \forall (i,j) \notin S$;
2: **for** $k = 1, \ldots, n$ **do**;
3: **for** all (i,j) **do**;
4: $Lev(a_{i,j}) = \min(Lev(a_{i,j}), \max(Lev(a_{i,k}), Lev(a_{k,j})) + 1)$

Output: Level of fill matrix of A

For the ILU(p) algorithm we refer to (Taylor 2007). Using a preconditioner T and rewriting the linear system results in

$$Ax = b \Leftrightarrow Tx = Tx - Ax + b \Leftrightarrow x = \underbrace{(I - T^{-1}A)}_{B} x + \underbrace{T^{-1}b}_{z}$$

As discussed before the iteration converges if $\|B\| \leq \gamma < 1$ (Banach Fixed-point theorem assures that) (Nannen 2019). T^{-1} should be easy to calculate. Using the preconditioner $T = LU$, where LU is result the incomplete LU factorization we get

$$T^{-1}Ax = T^{-1}b \Leftrightarrow (LU)^{-1}(LU + R)x = (I + (LU)^{-1}R)x = (LU)^{-1}b$$

We see, that $(LU)^{-1}$ is a proper approximation to A^{-1} if $\|(LU^{-1})R\|$ is close to zero. Hence the iteration

$$x^{(t+1)} = -(LU)^{-1}Rx^{(t)} + (LU)^{-1}b$$

converges if $\|B\| = \|(LU^{-1})R\| \leq \gamma < 1$

5.1.2. Incomplete Cholesky Decomposition

Since we are considering symmetric positive definite matrices it is recommended to use an incomplete Cholesky decomposition over ILU. The complete Cholesky decomposition can be found in (ibid.). The same idea applies as in section 5.1.1. We want to find a lower triangular matrix L such that $A \approx LL^*$, in other words $A = LL^* + R$ with an error matrix R. A simple approach is to just consider the nonzero elements of the spare matrix which would result in the IC(0). If a better approximation is needed, than a sparsity pattern and a level of fill matrix must be computed. Since the procedure is analog to the incomplete LU decomposition we refer to (T.-Z. Huang and Li 2018) for more details.
The iteration using the result of the decomposition LL^* as the preconditioner becomes

$$x^{(t+1)} = -(LL^*)^{-1}Rx^{(t)} + (LL^*)^{-1}b.$$

Convergence is again achieved if $\|(LL^*)^{-1}R\| < 1$

5.2. Local (Jacobi)

The local preconditioner is the simplest to compute but not often used since the convergence can be very slow as we will see. Decomposing the A into $A = L+D+U$

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} = \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ a_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n-1} & 0 \end{pmatrix} + \begin{pmatrix} a_{1,1} & & & \\ & \ddots & & \\ & & & a_{n,n} \end{pmatrix} + \begin{pmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & a_{n-1,n} \\ 0 & \cdots & \cdots & 0 \end{pmatrix}$$

Using $T = D$ results in the Jacobi preconditioning. Starting from a vector $x^{(0)}$ and assuming the diagonal elements are non-zero define the Jacobi iteration as

$$x^{(t+1)} = -D^{-1}(L+U)x^{(t)} + D^{-1}b$$

The iteration converges if $\|D^{-1}(L+U)\| \leq \gamma < 1$

Remark. Note that for diagonal dominant matrices the Jacobi iteration converges fast (Nannen 2019) since γ is close to zero. For problems deriving from discretization of elliptic differential equations, γ is often close to 1, hence the iteration convergence is slow and not advisable but the iteration is often using in other algorithms as we will see in the following.

5.3. Multigrid

Multigrid is a method to accelerate the convergence of a iterative method on a fine mesh using several levels of coarser meshes. The basic idea is to reduce the high frequency errors by solving a coarse problem. The resulting error will be interpolated and added to approximated solution.
Consider the finite element space V_h introduced in section 1. We will restrict ourselves to a coarse grid V_{2h}. The key ingredients are the restriction and the interpolation matrix. The restriction matrix $R_h^{2h} : V_h \to V_{2h}$ transfers vector from the fine grid to the coarse grid, whereas the $I_{2h}^h : V_{2h} \to V_h$ does the opposite. The matrix A_h on the fine grid is approximated by $A_{2h} = R_h^{2h} A_h I_{2h}^h$ on the coarse grid. The matrices are constructed such that $I_{2h}^h = (R_h^{2h})^T$. A calculation of the

matrices can be found at (Tatebe 2018).
The 3 main multigrid methods are the V-Cycle, W-Cycle and F-Cycle.

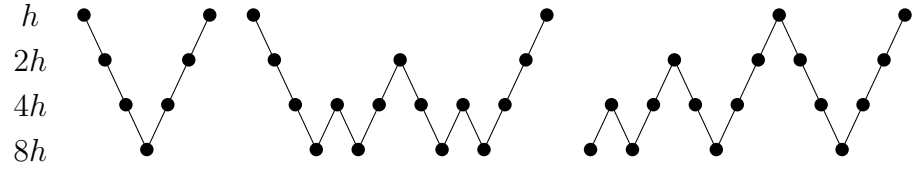


Figure 5: V-Cycle, W-Cycle and F-Cycle

The most used cycle is the V-Cycle that will become clear soon. Remember that the residual is $r_h = b_h - A_h x_h = A_h(x - x_h)$. Now that we have the necessary equipment, we can define the multigrid method as

Algorithm 9 Implementation of the multigrid method (Strang 2006)

Input: Matrix A_h, b_h, h

1: Iterate on $A_h x = b_h$ to reach x_h (i.e after 3 iterations using the Jacobi or Gauss-Seidel [5] preconditioner);
2: Restrict the residual $r_h = b_h - A_h x_h$ to the coarse grid by $r_{2h} = R_h^{2h} r_h$;
3: Repeat the above steps until $A_{2h} E_{2h} = r_{2h}$ can be exactly computed (i.e using LU or Cholesky factorization);
4: Interpolate E_{2h} to the finer grid by $E_h = I_{2h}^h E_{2h}$. Add E_h to x_h;
5: Iterate 3 more time on $A_h x = b_h$ starting from the improved $x_h + E_h$

Output: Iterated approximated solution x of the problem

Note that algorithm 9 represents a V-Cycle, where the initial grid gets restricted to a coarser grid where the linear system can be solved directly follow by the backwards interpolation to the finer grid. The algorithm is repeated until convergence. For visual purposes and better understanding of the procedure we include the following graphic

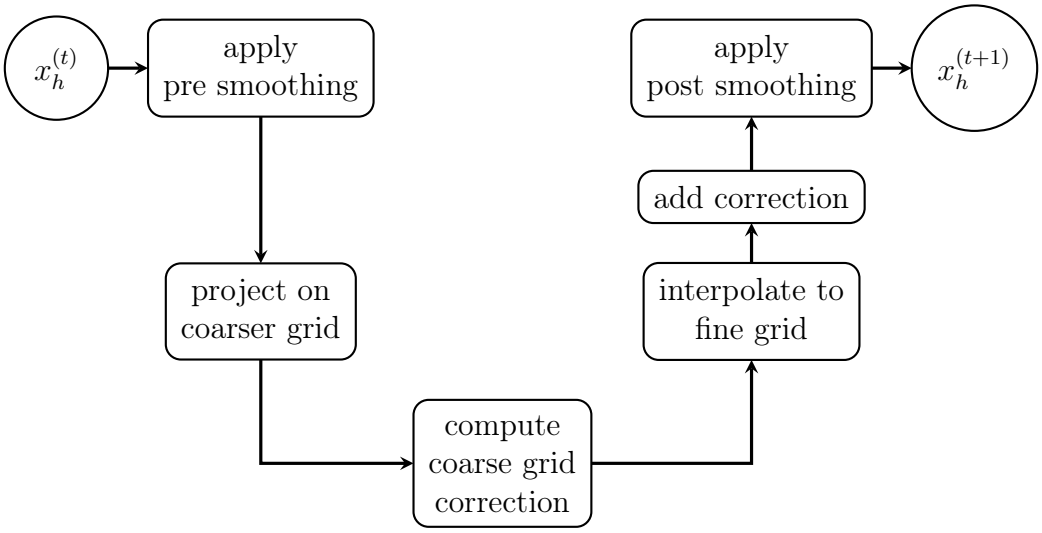


Figure 6: Representaion of algorithm 9

5.4. Comparison

We will compare the different preconditioners based on the eigenvalue problem described in section 1 on the unit square, calculating the first eigenvalue.

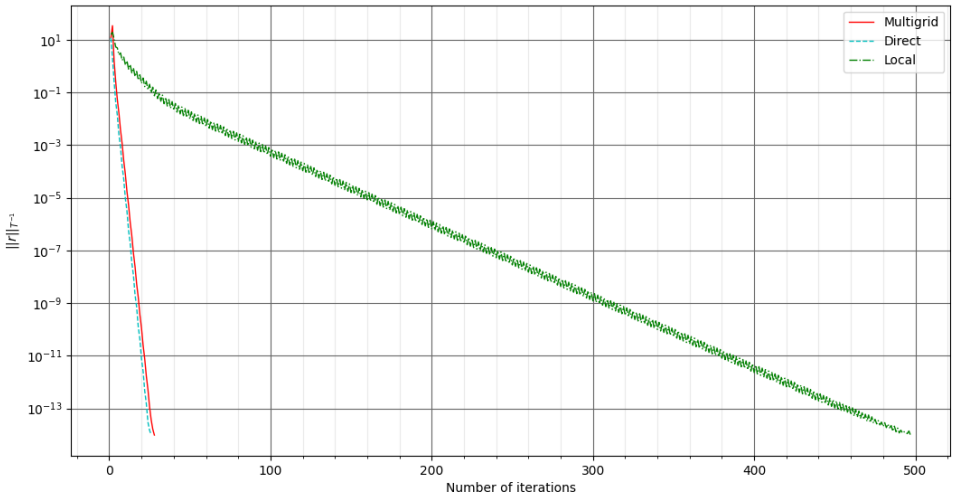


Figure 7: Convergence of LOPSD using different preconditioners

Note that the convergence using a local preconditioner is slow, due to the poor

approximation of A^{-1}, which leads to the factor γ being close to 1. On the other hand the direct and multigrid preconditioners need a similar number of iterations until the convergence of the residuum is achieved.

6. Computing Eigenfrequencies and Eigenmodes

As discussed in sections 1 and 4, algorithm 6 can be used to calculate the m smallest eigenfrequencies $f_1, \ldots, f_m$ and corresponding eigenmodes $\eta_1, \ldots, \eta_m$ of a bounded domain $\Omega \subseteq \mathbb{R}^n$ by solving the eigenvalue problem

$$A\eta = (2\pi f)^2 M\eta$$

where A is a stiffness matrix and $M \coloneqq \int_\Omega \rho uv d\mathcal{H}^n$ is a mass matrix. The density ρ and stiffness matrix A are material dependent.
As we want to analyze metal parts which deform linearly (up to a point) we will use the model of linear elasticity (Schöberl 2011) to define A with Lamé parameters μ and λ as

$$A \coloneqq \int_\Omega 2\mu\varepsilon(u) : \varepsilon(v) + \lambda \mathrm{div}(u)\mathrm{div}(v) d\mathcal{H}^n \tag{6.1}$$

where $\varepsilon(u) \coloneqq \frac{1}{2}\left(\nabla u + \nabla u^T\right)$ and $C : D \coloneqq \sum_{ij} C_{ij} D_{ij} = tr(C : D^T)$.

Remark. All used material parameters can be found in appendix A.1

We will use the following theorem to obtain a reference value to compare the FEM results to.

Theorem 6.0.1
The natural frequency of a cuboid clamped-free beam with length L can be approximated using the Euler-Bernoulli model (Han, Benaroya, and Wei 1999):

$$f_n = \frac{a_n^2}{2\pi L^2}\sqrt{\frac{E}{\rho}}\frac{b}{\sqrt{12}} \tag{6.2}$$

where E denotes Young's modulus, ρ the density of the used material and b the width of the beam in the direction of motion. The five smallest wave numbers a_n are displayed in table 1.

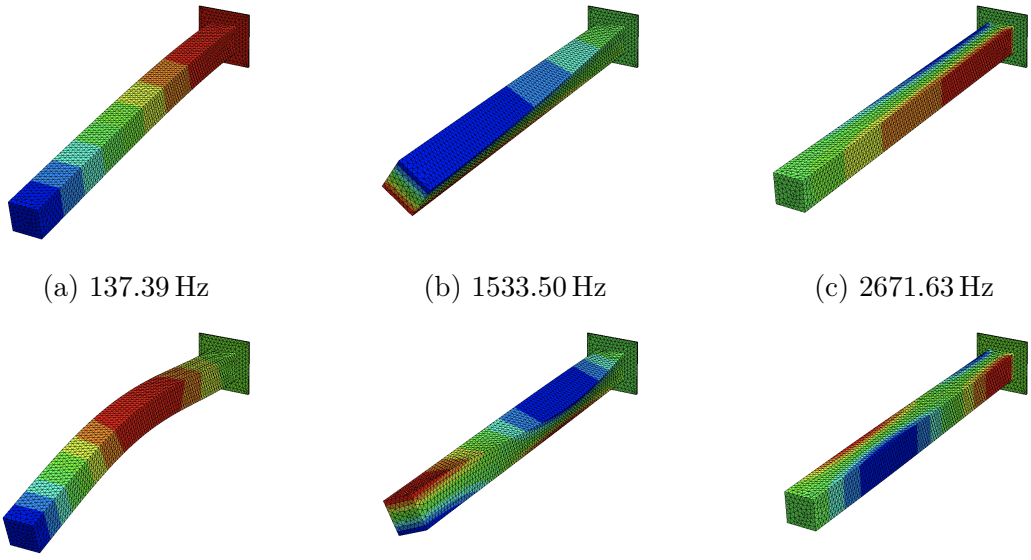
	The frequency equation	a_1	a_2	a_3	a_4	a_5
c-f	$\cos(a)\cosh(a) + 1 = 0$	1.875	4.694	7.855	10.996	14.137

Table 1: The first five wave numbers of the Euler-Bernoulli model.

Remark. By replacing $\frac{b}{\sqrt{12}}$ in eq. (6.2) with $\frac{r}{2}$, theorem 6.0.1 can be adapted for cylindrical beams with radius r.

6.1. Clamped-Free Beam

We start with the analysis of a simple clamped-free beam made out of steel. First, a mesh approximating the geometry is constructed. Then, as mentioned above, eq. (6.1) is used together with algorithm 5 to calculate both eigenfrequencies and eigenmodes.
The complete code used in this section can be found in appendix A.
The first six mode shapes corresponding to movement in different directions and their frequencies can be seen in fig. 8. Note that the horizontal movement displayed in figs. 8a and 8d also occurs laterally. This modeshape is not included in fig. 8 due to space constraints.



(a) 137.39 Hz (b) 1533.50 Hz (c) 2671.63 Hz

(d) 837.14 Hz (e) 4600.42 Hz (f) 8008.43 Hz

Figure 8: Six modeshapes of a clamped-free beam

A comparison of our results[6] to both theorem 6.0.1 and CATIA V5's analysis is displayed in table 2.

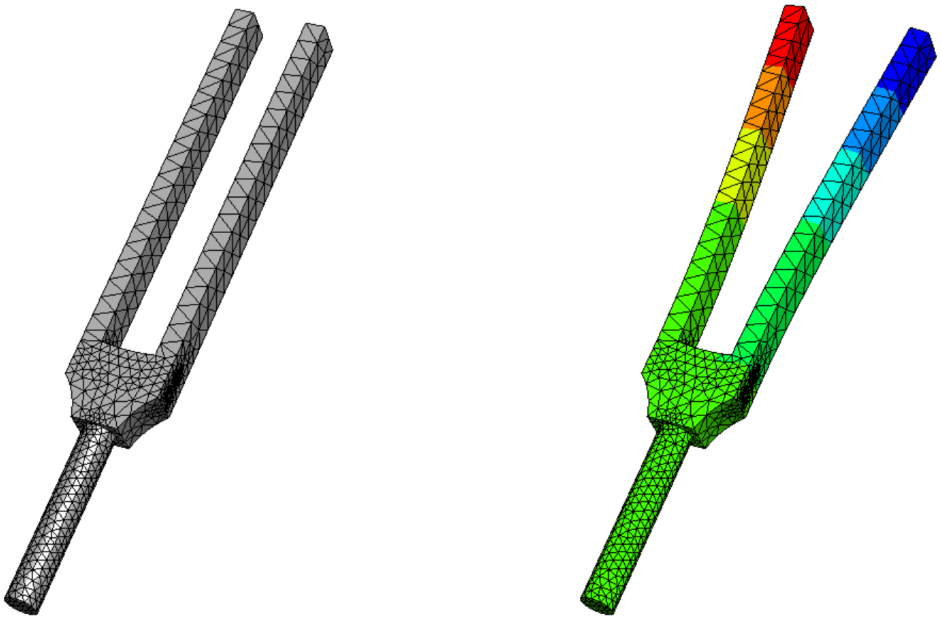
[6]The geometry used can be found in appendix B.1.

shape mode	NGSolve	Catia V5	Euler-Bernoulli
1	137.39 Hz	137.39 Hz	138.14 Hz
3	837.14 Hz	837.19 Hz	-
5	1533.50 Hz	1533.8 Hz	-
8	2671.63 Hz	2671.66 Hz	-
11	4600.42 Hz	4601.46 Hz	-
15	8008.43 Hz	8008.52 Hz	-

Table 2: Comparing our results to theorem 6.0.1 and frequency analysis in Catia

6.2. Tuning Fork

As the natural frequency of a tuning fork is well known it is a slightly more interesting part to analyze. Using the approach described in sections 6 and 6.1 we obtain the following results:



(a) Mesh

(b) Eigenmode corresponding to the second smallest eigenfrequency

Figure 9: Analyzing a Tuning Fork

Remark. The frequency we are interested in is the one corresponding to the eigenmode with both prongs swinging mirror-inverted in and out as seen in fig. 9b.

The frequency corresponding to fig. 9b is 440.22 Hz. The 3d-model used (appendix B.2) was modeled after a real 440 Hz tuning fork.

A. Code

A.1. Materials

```
class Material():
    E=0
    nu=0
    mu=0
    lam=0
    rho=0
    type="linear"
    def __init__(self, type="linear", E=0, nu=0,rho=0):
        self.E=E
        self.nu=nu
        self.type=type
        if(abs(E)>10e-15 and abs(nu)>10e-15 and abs(nu+1)>10e-15 and
   abs(nu-1/2)>10e-15 and abs(rho)>10e-15):
            self.mu=E / 2 / (1+nu)
            self.lam=E * nu / ((1+nu)*(1-2*nu))
            self.rho=rho


#definging default materials:
#type=Material(type of model (e.g. "linear"), E (=young-module),nu (=
   poisson number), rho (=density))
aluminium=Material("linear", 69 * (10**(6)), 0.33, 2700 * (10**(-9)))
aluminum=Material("linear", 69 * (10**(6)), 0.33, 2700 * (10**(-9)))
steel=Material("linear", 220 * (10**(6)), 0.28, 7700 * (10**(-9)))
```

Listing 3: materials.py

A.2. Geometry

```
from materials import *

import netgen.gui
import netgen.geom2d as geom2d
from netgen.csg import *
from netgen.geom2d import unit_square
from ngsolve import *
import math
import scipy.linalg
from scipy import random
from netgen.NgOCC import *

class Geometry():
    material=steel
    geometry="" #3d model
    maxh=1
    order=0
    dirichlet=0
    low_order_space=True
    mesh=0
    fes=0
    u=0   #TrialFunction
    v=0   #TestFunction
    A=0   #StiffnessMatrix
    M=0   #MassMatrix
    shift=0
    precond=False   #preconditioner
    def __init__(self, material, geometryPath, maxh, order, dirichlet
    , shift, low_order_space):
        self.shift=shift
        self.maxh=maxh
        self.order=order
        self.dirichlet=dirichlet
        self.low_order_space=low_order_space
        if(geometryPath!=""):
            self.material=material
            self.geometry=OCCGeometry(geometryPath)


    def generateMesh(self):
        self.mesh = Mesh(self.geometry.GenerateMesh(maxh=self.maxh))
        self.mesh.Curve(2)
        self.fes = VectorH1(self.mesh, order=self.order, dirichlet=
    self.dirichlet, low_order_space=self.low_order_space)
        self.u = self.fes.TrialFunction()
        self.v = self.fes.TestFunction()
        # StiffnessMatrix depending on material type:
```

```
46          if (self.material.type == "linear"):
                self.A = BilinearForm(self.fes)
48              self.A += 2 * self.material.mu * InnerProduct(1 / 2 * (
    grad(self.u) + grad(self.u).trans),
                                                      1 / 2 * (grad(
    self.v) + grad(self.v).trans)) * dx
50              self.A += self.material.lam * div(self.u) * div(self.v) *
     dx
                self.A += self.shift * self.material.rho * self.u * self.
    v * dx
52          # else if(1==1):
            # other material types
54
            self.M = BilinearForm(self.fes)
56          self.M += self.material.rho * self.u * self.v * dx

58      def pre(self,type,inverse=""):
            self.precond = Preconditioner(self.A, type, inverse=inverse)
60
        def draw(self):
62          Draw(mesh)

64      def assemble(self):
            self.A.Assemble()
66          self.M.Assemble()
```

Listing 4: geometry.py

A.3. EVSolver

```
from ngsolve.la import EigenValues_Preconditioner
from geometry import *


import netgen.gui
import netgen.geom2d as geom2d
from netgen.csg import *
from netgen.geom2d import unit_square
from ngsolve import *
import math
import scipy.linalg
from scipy import random
from netgen.NgOCC import *
from numpy import linalg as la
import time

# finding eigenvalues of   A u = lambda C u
# condition number = largest/smallest
class EVSolver():

    def __init__(self):
        pass

    def getF(self,l,shift=0):
        return (sqrt(abs(l - shift)) / (2 * math.pi))


    def lobpsd(self, geo , num, maxNumIterations=50, eps= 10e-6):
        lam = EigenValues_Preconditioner(mat=geo.A.mat, pre=geo.
    precond)

        u = GridFunction(geo.fes, multidim=num)

        #using multivectors for better performance
        uvecs = MultiVector(u.vec, num)
        vecs = MultiVector(u.vec, 2 * num)

        for v in vecs[0:num]:
            v.SetRandom()
        uvecs[:] = geo.precond * vecs[0:num]
        lams = Vector(num * [1])
        numIterations=0
        res=[1]*(maxNumIterations)

        #weird python do while loop
        while True:
            numIterations+=1
```

```
            vecs[0:num] = geo.A.mat * uvecs - (geo.M.mat * uvecs).
    Scale(lams)
            vecs[num:2 * num] = geo.precond * vecs[0:num]

            # T-norm res
            r = InnerProduct(vecs[num], vecs[0])
            for i in range(1, num):
                tmp = InnerProduct(vecs[num + i], vecs[i])
                if (r < tmp):
                    r = tmp
            res[numIterations-1] = r

            vecs[0:num] = uvecs

            vecs.Orthogonalize()

            asmall = InnerProduct(vecs, geo.A.mat * vecs)
            msmall = InnerProduct(vecs, geo.M.mat * vecs)

            ev, evec = scipy.linalg.eigh(a=asmall, b=msmall)
            prev = lams
            lams = Vector(ev[0:num])
            print(numIterations, ":", [self.getF(l, geo.shift) for l
    in lams])
            print("res:", res[numIterations-1],"\n")

            uvecs[:] = vecs * Matrix(evec[:, 0:num])


            if(abs(res[numIterations-1]) < eps or numIterations>=
    maxNumIterations):
                break

        for j in range(num):
            u.vecs[j][:] = 0.0
            u.vecs[j].data += uvecs[j]

        Draw(u, geo.mesh, "mode")
        SetVisualization(deformation=True)
        return ev,evec,res


    def lobpcg(self, geo , num, maxNumIterations=50, eps= 10e-6):
        lam = EigenValues_Preconditioner(mat=geo.A.mat, pre=geo.
    precond)

        u = GridFunction(geo.fes, multidim=num)

        # using multivectors for better performance
```

```
        uvecs = MultiVector(u.vec, num)
93      vecs = MultiVector(u.vec, 2 * num)

95      for v in vecs[0:num]:
            v.SetRandom()
97      uvecs[:] = geo.precond * vecs[0:num]
        lams = Vector(num * [1])
99      numIterations = 0
        res=[1]*(maxNumIterations)
101
        # weird python do while loop
103     while True:
            numIterations += 1
105         vecs[0:num] = geo.A.mat * uvecs[0:num] - (geo.M.mat *
    uvecs[0:num]).Scale(lams)
            vecs[num:2 * num] = geo.precond * vecs[0:num]
107
            # T-norm res
109         r = InnerProduct(vecs[num], vecs[0])
            for i in range(1, num):
111             tmp = InnerProduct(vecs[num + i], vecs[i])
                if (r < tmp):
113                 r = tmp
            res[numIterations-1] = r
115
            vecs[0:num] = uvecs[0:num]
117
            vecs.Orthogonalize()
119
            asmall = InnerProduct(vecs, geo.A.mat * vecs)
121         msmall = InnerProduct(vecs, geo.M.mat * vecs)

123         ev, evec = scipy.linalg.eigh(a=asmall, b=msmall)
            prev = lams
125         lams = Vector(ev[0:num])
            print(numIterations, ":", [self.getF(l, geo.shift) for l
    in lams])
127         print("res:", res[numIterations-1], "\n")

129         if (numIterations==1):
                tmp = MultiVector(u.vec, 2 * num)
131             tmp[0:2*num] = vecs
                vecs = MultiVector(u.vec, 3 * num)
133             vecs[0:2*num] = tmp[0:2 * num]

135         uvecs[0:num] = vecs * Matrix(evec[:, 0:num])

137         #todo: use span{w^i,x^i,p^i} instead of span{w^i,x^i,x^{i
    -1}} for better stability
```

```
            vecs[2 * num:3 * num] = vecs[0:num]

            if (abs(res[numIterations-1]) < eps or numIterations >=
    maxNumIterations):
                break

        for j in range(num):
            u.vecs[j][:] = 0.0
            u.vecs[j].data += uvecs[j]

        Draw(u, geo.mesh, "mode")
        #SetVisualization(deformation=True)
        return ev, evec, res
```

Listing 5: evsolver.py

A.4. Eigenfrequencies

```
from materials import *
from geometry import *
from evsolver import *
from ngsolve.la import EigenValues_Preconditioner
import netgen.gui
import netgen.geom2d as geom2d
from netgen.csg import *
from netgen.geom2d import unit_square
from ngsolve import *
import math
import scipy.linalg
from scipy import random
from netgen.NgOCC import *

import matplotlib.pyplot as plt

evsolver=EVSolver()

#Geometry(material, geometryPath, maxh, order, dirichlet, shift,
    low_order_space)
tuningfork=Geometry(steel, "../3d-models/fork/tuning-fork.stp", 5, 2,
     [1,2,3], 0, True)

tuningfork.generateMesh()
Draw(tuningfork.mesh)

ngsglobals.msg_level=5

#pre(type, inverse)
tuningfork.pre("multigrid", "sparsecholesky")
tuningfork.assemble()


ev,evec,res = evsolver.lobpcg(tuningfork, 10, 500, 10e-8)


input("Press any key to continue..")
```

Listing 6: eigenfrequencies.py

B. Geometry

B.1. Clamped-Free Beam

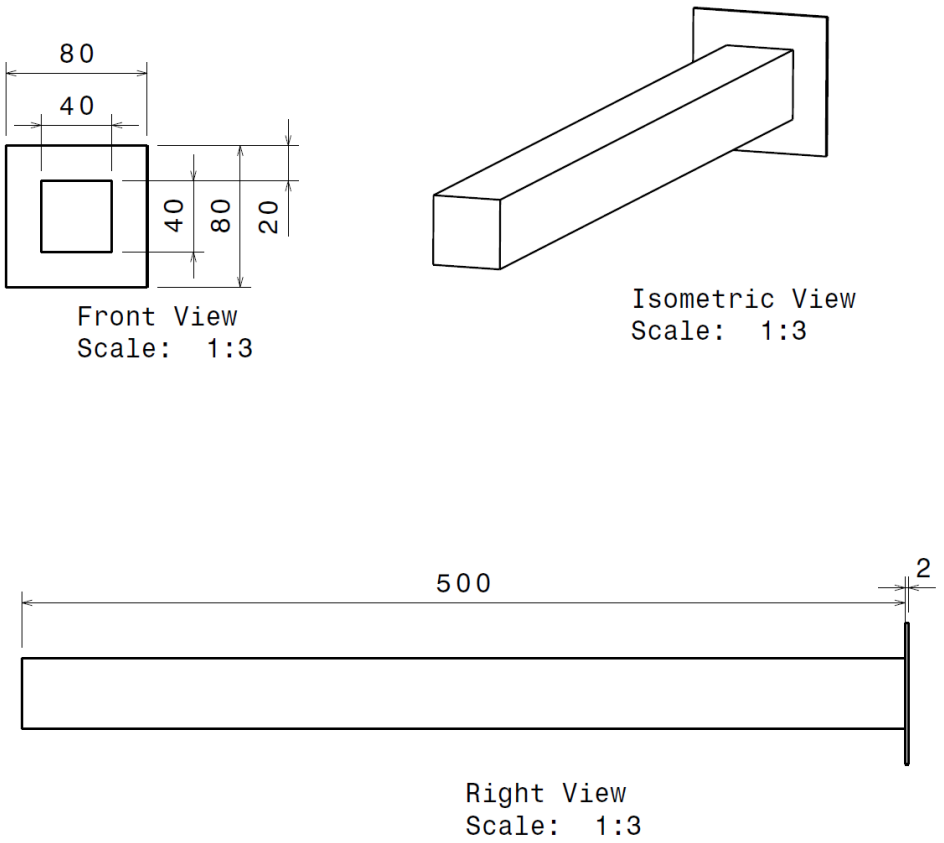


Figure 10: CF-Beam used in section 6.1

B.2. Tuning Fork

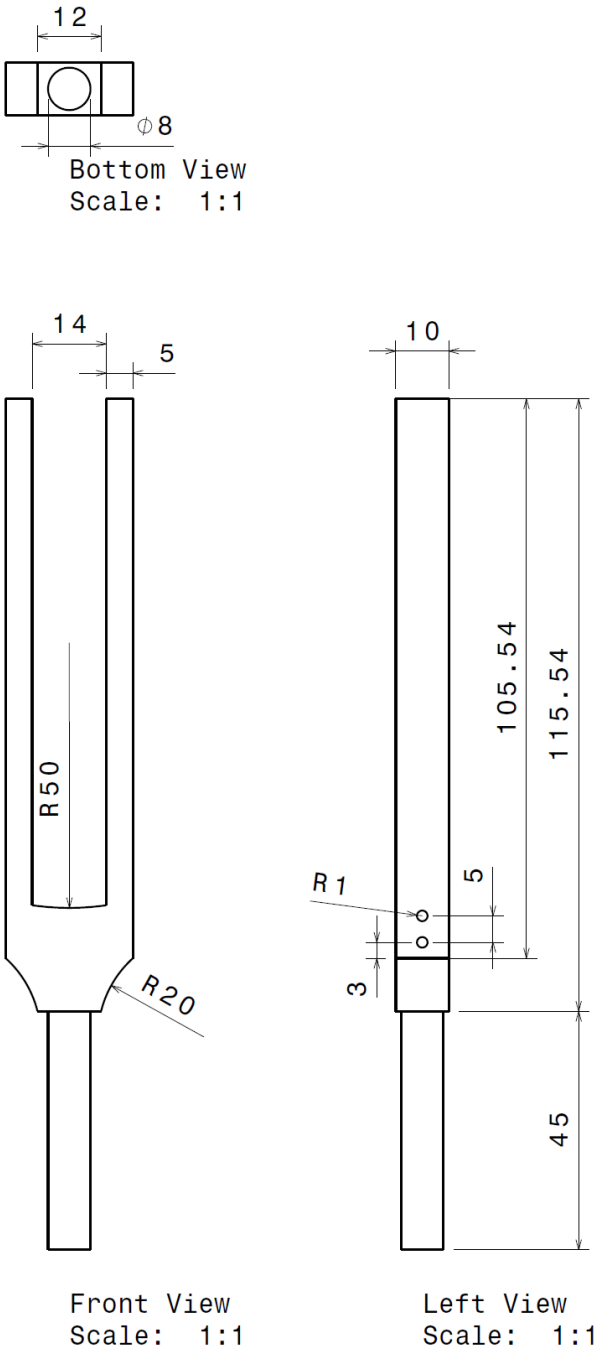


Figure 11: Tuning fork used in section 6.2

References

Blümlinger, Martin (2019). *Analysis 3*. URL: https://www.asc.tuwien.ac.at/~blue/ (visited on 12/03/2020) (cit. on p. 4).

Evans, Lawrence C. (2010). *Partial differential equations*. Providence, R.I.: American Mathematical Society (cit. on p. 4).

Feng, Y. T. and D. R. J. Owen (1996). "Conjugate gradient methods for solving the smallest eigenpair of large symmetric eigenvalue problems". In: *Internat. J. Numer. Methods Eng.* 39, pp. 2209–2229 (cit. on p. 17).

Han, S., H. Benaroya, and Timothy K. J. Wei (1999). "DYNAMICS OF TRANSVERSELY VIBRATING BEAMS USING FOUR ENGINEERING THEORIES". In: *Journal of Sound and Vibration* 225, pp. 935–988 (cit. on p. 33).

Hirnschall, Sebastian and Rafael Dorigo (2020). *CG-Verfahren für dünnbesetzte Matrizen*. URL: https://blog.hirnschall.net/downloads/download.php?file=numerik2.pdf (visited on 11/24/2020) (cit. on p. 17).

Knyazev, Andrew (2000). "Toward The Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method". In: *SIAM Journal on Scientific Computing* 23. DOI: 10.1137/S1064827500366124 (cit. on pp. 17 sqq., 26).

Knyazev, Andrew V. and Klaus Neymeyr (2003). "A geometric theory for preconditioned inverse iteration III: A short and sharp convergence estimate for generalized eigenvalue problems". In: *Linear Algebra and its Applications* 358.1, pp. 95–114. DOI: https://doi.org/10.1016/S0024-3795(01)00461-X. URL: http://www.sciencedirect.com/science/article/pii/S002437950100461X (cit. on pp. 7, 9, 11 sq., 15, 26).

Knyazev, Andrew, M. Argentati, Ilya Lashuk, and Evgueni Ovtchinnikov (2007). "Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in hypre and PETSc". In: *SIAM Journal on Scientific Computing* 29. DOI: 10.1137/060661624 (cit. on pp. 2, 21, 23).

Meerbergen, Karl (2007). *Incomplete factorization*. URL: https://people.cs.kuleuven.be/~karl.meerbergen/didactiek/h03g1a/ilu.pdf (visited on 01/25/2021) (cit. on p. 28).

Nannen, Lothar (2019). *Numerische Mathematik A*. URL: https://tiss.tuwien.ac.at/education/course/documents.xhtml?dswid=3351&dsrid=39&courseNr=101313&semester=2019W# (visited on 12/03/2019) (cit. on pp. 6, 8, 26, 28 sq.).

Neymeyr, Klaus (2001a). "A geometric theory for preconditioned inverse iteration I: Extrema of the Rayleigh quotient". In: *Linear Algebra and its Applications* 322.1, pp. 61–85. DOI: https://doi.org/10.1016/S0024-3795(00)00239-1. URL: http://www.sciencedirect.com/science/article/pii/S0024379500002391 (cit. on pp. 12 sq.).

Neymeyr, Klaus (2001b). "A geometric theory for preconditioned inverse iteration II: Convergence estimates". In: *Linear Algebra and its Applications* 322.1, pp. 87–104. DOI: https://doi.org/10.1016/S0024-3795(00)00236-6. URL: http://www.sciencedirect.com/science/article/pii/S0024379500002366 (cit. on p. 11).

Schöberl, Joachim (2011). *Numerische Methoden in der Kontinuumsmechanik*. URL: http://www.asc.tuwien.ac.at/~schoeberl/wiki/lva/nummech/nummech.pdf (visited on 11/06/2020) (cit. on p. 33).

Schofield, Grady, James R. Chelikowsky, and Yousef Saad (2012). "A spectrum slicing method for the Kohn–Sham problem". In: *Computer Physics Communications* 183.3, pp. 497–505. DOI: https://doi.org/10.1016/j.cpc.2011.11.005. URL: http://www.sciencedirect.com/science/article/pii/S0010465511003675 (cit. on p. 9).

Strang, Gilbert (2006). *Multigrid Methods*. URL: https://ocw.mit.edu/courses/mathematics/18-086-mathematical-methods-for-engineers-ii-spring-2006/readings/am63.pdf (visited on 01/18/2021) (cit. on p. 30).

T.-Z. Huang, Y. Zhang and L. Li (2018). *MODIFIED INCOMPLETE CHOLESKY-FACTORIZATION FOR SOLVING ELECTROMAGNETICSCATTERING PROBLEMS*. URL: http://www.jpier.org/PIERB/pierb13/03.08112407.pdf (visited on 01/23/2021) (cit. on p. 28).

Tatebe, Osamu (2018). *The Multigrid Preconditioned Conjugate Gradient Method*. URL: http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/research/paper/CM93-tatebe.pdf (visited on 01/18/2021) (cit. on p. 30).

Taylor, Jason (2007). *Implementation of ILU(P) Factorization of Sparse Matrices*. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.330.2112&rep=rep1&type=pdf (visited on 01/18/2021) (cit. on pp. 27 sq.).

Listings

List of Figures

List of Tables