SOLVING LARGE SCALE EIGENVALUE PROBLEMS IN AMORPHOUS MATERIALS

GIUSEPPE ACCAPUTO
Computational Science and Engineering
ETH Zürich

Master Thesis

SUPERVISORS:

Prof. Dr. Peter Arbenz
Dr. Peter Derlet

The journey of a thousand miles begins with a single step.

— Lao Tzu

Dedicated to my parents, Teresa and Paolo, and my sister, Alessia, for all their love, patience, kindness, and support throughout my studies at the ETH Zürich.

ABSTRACT

Amorphous solids, like metallic glasses exhibit an excess of states in the low frequency regime very close to the Boson peak, and the precise nature of these low frequency vibrations remains unclear.

The purpose of this thesis is to investigate the use of a polynomial filtered eigensolver for the computation and study of low frequency eigenmodes of a Hessian matrix located in a specific interval close to the Boson peak regime.

A distributed-memory parallel implementation of a polynomial filtered eigensolver is presented. Our implementation is then applied to Hessian matrices of different atomistic bulk metallic glass structures derived from molecular dynamics simulations for the computation of low frequency eigenmodes close to the Boson peak region. In addition, we demonstrate the parallel scalability of our implementation on multicore nodes.

Our resulting calculations successfully concur with previous results [5], and anomalous behavior of the particles in the region close to the Boson peak can be observed from the data.

ACKNOWLEDGMENTS

I want to thank Professor Arbenz and Peter Derlet for the possibility to work on this project, for their knowledge, patience and for their time. Further, I want to thank Dmitry Ozerov for his extensive, speedy and kind help with setting everything up and running computations on the Ra cluster at PSI. Finally, I want to thank my girlfriend Mirjam for all her love, support, and patience throughout this exciting period of time in my life.

CONTENTS

1 INTRODUCTION 1

2 BACKGROUND 3

2.1 Polynomial Filtering 3

2.1.1 Least-Squares Polynomial Filters 3

2.1.2 Choosing the Degree of the Filter 5

2.1.3 Balancing the Filter 5

2.2 The Thick-Restart Lanczos Algorithm 8

3 IMPLEMENTATION 13

3.1 Trilinos 13

3.1.1 Epetra 14

3.1.2 Anasazi 14

3.1.3 Teuchos 14

3.2 Parallelization 14

3.3 Parallel Matrix Import 17

3.4 Estimation of the Extremal Eigenvalues of $\mathbf{A}$ 17

3.5 Estimation of the Eigenvalue Count in a Specified Interval 18

3.6 Computation of the Polynomial Filter $\hat{\rho}_k$ 20

3.7 Combining the Eigensolver and Polynomial Filtering 23

3.8 Computation of the Polynomial Filter Operator $\hat{\rho}_k(\hat{\mathbf{A}})$ 24

4 EXPERIMENTAL RESULTS 27

4.1 Eigenmode Computations 27

4.1.1 Experimental Setup 27

4.1.2 Participation Ratio and Displacement Visualization 27

4.2 Performance Measurements 31

4.2.1 Experimental Setup 31

4.2.2 Polynomial Filter Operator $\hat{\rho}_k(\hat{\mathbf{H}})$ 31

5 CONCLUSION AND FUTURE DEVELOPMENTS 35

A APPENDIX A 37

A.1 User Manual: BosonPeak Utility 37

A.1.1 Deploying the BosonPeak Utility on the Euler (ETH Zürich) and/or Ra (PSI) Cluster 37

A.1.2 Code Compilation 37

A.1.3 Configuring the BosonPeak Utility to Run Different Calculations 38

A.1.4 Building and Running the BosonPeak Unit Tests 41

A.2 Using the Polynomial Filter MATLAB Implementation 41

A.3 Converting MM Files Storing Large Symmetric Matrices to Trilinos Compatible HDF5 Files 45

BIBLIOGRAPHY 47

LIST OF FIGURES

Figure 1 Polynomial filters $\hat{\rho}_k$ of degree $k = 11$ for the interval $[-0.2, 0.2]$, using three different smoothing approaches. 5

Figure 2 Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$, using three different thresholds ϕ and Jackson smoothing. The dotted lines connect $\hat{\rho}_k(\hat{\xi})$ to $\hat{\rho}_k(\hat{\eta})$ for each polynomial filter, clearly showing $\hat{\rho}_k(\hat{\xi}) \neq \hat{\rho}_k(\hat{\eta})$. 6

Figure 3 Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$, using three different thresholds ϕ and Jackson smoothing. Each polynomial filter has been balanced such that $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta}) = \tau$ (dotted lines). 8

Figure 4 Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [0.75, 1]$, using three different thresholds ϕ and Jackson smoothing. Each polynomial filter has been balanced such that $\hat{\rho}_k(\hat{\xi}) = \tau$ (dotted lines). 9

Figure 5 Row-wise distribution pattern of a matrix $\mathbf{A} \in R^{6\times6}$ using 3 MPI ranks. 15

Figure 6 Handling end interval cases by using different degrees for the polynomial filter $\hat{\rho}_k$. 21

Figure 7 Eigenvalues $\lambda \in [0.1, 2.0]$ of the different samples $\mathbf{H}_i, i = 1, 2, \ldots, 8$ plotted against the participation ratios. 29

Figure 8 Displacement plots for different eigenmodes of $\mathbf{H}_1$. The plots show a cut along the $x - y$ plane ($\delta z = 0.15$ Å) containing the particle with the largest displacement. The arrows are proportional to the displacements of the particles, and the displacements have been magnified $\times 300$ for visibility reasons. 30

Figure 9 Measured speedup and parallel efficiency of the polynomial operator implementation with $k = 500$ and $b = 32$, with the time measurements being averaged over 10 runs. 32

Figure 10 Measured speedup and parallel efficiency of the matrix-multivector multiplication in Eq. (57) with $k = 151$ and $b = 128$, with the time measurements being averaged over 10 runs. 33

Figure 11 Weak scaling measurements of the matrix-multivector multiplication in Eq. (57) with increasing block size b (left) and increasing polynomial degree k (right). 33

Figure 12 Strong scaling measurements of the matrix-multivector multiplication in Eq. (57). The blue line results from a linear regression using the measured timings. 34

Figure 13 Polynomial filter $\hat{\rho}_k$ (Eq. (5)) computed and plotted by using the MATLAB code displayed in Listing 6. 45

LIST OF TABLES

Table 1 Tabular overview of properties of the Hessian matrices $\mathbf{H}_1, \dots, \mathbf{H}_8$ of a system with $N = 256000$ atoms. The second column contains the Number of Nonzeros (NNZ) of $\mathbf{H}_i$, the third column shows the total number of eigenvalues $\lambda_j \in [0.1, 2]$, and in the third column the maximal residual norm over all computed residual norms $\|r_j\|$ for a given $\mathbf{H}_i$ is displayed, where the residual vector is defined as $\mathbf{r}_j = \mathbf{H}_i \mathbf{u}_j - \lambda_j \mathbf{u}_j$. Finally, in the last column the Time to Solution (Time to Solution (TTS)) for the eigensolver part of the utility (Line 7 in Algorithm 3) is shown; the TTS is based on computations with 48 cores on Euler. 28

LISTINGS

Listing 1 Configuration of the extremal eigenvalues to be used for the transformation of the matrix as defined in Eq. (6). 38

Listing 2 Extended configuration for the approximation of the extremal eigenvalues. 39

Listing 3 Configuration for the estimation of the eigenvalue count 39

Listing 4 Complete configuration for the computation of eigenpairs in a specific interval of the spectrum. 40

Listing 5 Example code showing the usage of the MATLAB implementation of the polynomial filter for the computation of eigenvalues in a specific interval of interest. We are using a simple diagonal matrix $\mathbf{A}$ with spectrum $\sigma = \{1, 2, 3, 4, \dots, 20\}$ and compute all the eigenvalues λ_i in the interval $[11.5, 14.2]$. Before running this code it is important to first load all the necessary environment variables by running the `setup-<euler|psi>.sh` script as described in Section A.1.1. 41

Listing 6 Example code showing the usage of the MATLAB implementation of the polynomial filter for the visualization of the resulting filter. Fig. 13 shows the resulting plot from this code. Before running this code it is important to first load all the necessary environment variables by running the `setup-<euler|psi>.sh` script as described in Section A.1.1. 44

ACRONYMS

MPI	Message Passing Interface
CRS	Compressed Row Storage
RCP	Reference Counting Pointer
BKSM	a block version of the Krylov-Schur method
PFO	Polynomial Filter Operator
HDF5	Hierarchical Data Format 5
MM	MatrixMarket
SPMMM	sparse Matrix-Multivector Multiplication
I.I.D.	identically independently distributed
TTS	Time to Solution
NNZ	Number of Nonzeros
DOS	Density of States
BPU	`BosonPeak` Utility

1 INTRODUCTION

In condensed matter physics, an amorphous or disordered solid is a solid that lacks the long-range order in the position of the atoms characteristic of a crystal.

For a crystal, vibrational excitations are understood in terms of quantized plane waves, the phonons, and in the low frequency regime the vibrational modes are the acoustic phonons. In the case of disordered solids, such as metallic glasses, it has been shown that the majority of the vibrations are not plane waves. Propagative modes, including plane waves are restricted to the low frequency regime, and localized modes occupy the high frequency tail of the spectrum.

For various amorphous solids, an excess of low frequency vibrations as compared to the Debye prediction has been observed. The origin of these modes in excess is called the Boson peak, which historically has been measured via Raman spectroscopy. At low frequencies and long wave-lengths, acoustic plane waves do not interact with disorder, and thus can still propagate in disordered solids. Once the frequency is increased beyond the Ioffe-Regel limit, the acoustic plane waves interact with the disorder and are significantly scattered. This strong scattering regime occurs around the Boson peak position, and the exact origin of this phenomenon and its connection to the Boson peak is still debated.

Molecular dynamics simulations are able to produce metallic glasses. To decide on the nature of a mode, the Hessian matrix — a square matrix of second-order partial derivatives of the potential energy V — of such a generated system has to be diagonalized. An analysis based on the computed eigenfrequencies and eigenvectors would tell whether a mode is a propagating plane wave or not.

In our case, the matrix of interest is the Hessian matrix $\mathbf{H}$ of atomistic bulk metallic glass structures derived from molecular dynamics simulations using a binary Lennard Jones pair potential [5]. For a system consisting of N atoms, the vibrational modes of interest can be calculated by solving the real symmetric eigenvalue problem defined by

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i, \quad \mathbf{H} \in \mathbb{R}^{3N\times 3N},\, \mathbf{u}_i \in \mathbb{R}^{3N}\,,\ i = 1, \dots, 3N\,, \tag{1}$$

where $\lambda_i = \omega_i^2$ with ω_i being the fundamental frequency and $\mathbf{u}_i$ is the eigenmode or eigenvector representing the displacement of the particles in the system. Further, the eigenvalues λ_i are arranged in ascending order, i.e. $\lambda_{\min} = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_{3N} = \lambda_{\max}$.

We are interested in obtaining the significant portion of the long-wavelength (low-frequency) vibrational eigenmodes close to the Boson peak, which will be analyzed to investigate the relationship between the atomic-scale structure of the metallic glass and the Boson peak regime. This task requires the computation of all eigenvalues that are located in an interval close to the Boson peak region, including the associated eigenvectors.

Extreme eigenvalue problems, where the interval of interest $[\xi, \eta]$ is located at the end of the spectrum, i.e., when $\xi = \lambda_{\min}$ or $\eta = \lambda_{\max}$ are handled rather well by most methods. The situation when $[\xi, \eta]$ lies within the boundaries of the spectrum is harder to solve in general and is called an *interior* eigenvalue problem.

The interval of interest close to the Boson peak region is located in an interior region of the spectrum of the Hessian matrix $\mathbf{H}$, and thus the problem stated in Eq. (1) is an interior eigenvalue problem.

For a general diagonalizable square matrix $\mathbf{A}$ of dimensions $n \times n$, a frequent approach to obtain part of the spectrum in an interior interval is to apply the Lanczos algorithm to a transformed matrix $\mathbf{B} = \rho(\mathbf{A})$, where ρ is either a rational function or a polynomial. Nonetheless, the best known approach for interior eigenvalue problems is based on a shift-and-invert approach, where the Lanczos algorithm is applied to $\mathbf{B} = (\mathbf{A} - \sigma\mathbf{I})^{-1}$. σ is called the *shift* and is selected to point to the eigenvalues in the interval of interest. With the shift-and-invert transformation, the eigenvalues of $\mathbf{A}$ closest to the shift σ are then mapped to the extreme ones of $\mathbf{B}$. Since we are working with an inverse matrix, shift-and-invert requires a factorization of the matrix $\mathbf{A} - \sigma\mathbf{I}$ which is not always readily available and can be rather expensive to compute in some cases. Polynomial filtering instead replaces $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ by a polynomial $\rho(\mathbf{A})$ such that all eigenvalues of $\mathbf{A}$ in $[\xi, \eta]$ are transformed into dominant eigenvalues of $\rho(\mathbf{A})$.

The goal of this thesis is to investigate the usage of the polynomial filters presented in [12] combined with the thick-restart Lanczos algorithm to compute eigenvalues and corresponding eigenvectors in an interior interval of the spectrum of the Hessian matrix $\mathbf{H}$ close to the Boson peak regime.

The thesis is organized as follows: Chapter 2 introduces the concept of polynomial filters and explains the idea behind the thick-restart Lanczos algorithm. Chapter 3 provides details about our implementation and the general parallelization of our code. Next, in Chapter 4 results from interval-specific eigenpair computations with Hessian matrices of large systems and various performance measurements are discussed. Finally, the thesis ends with concluding remarks about our work and results in Chapter 5, and an outlook on possible enhancements of our implementation based on the physical problem at hand is given.

2 BACKGROUND

In the first part of this chapter we present the concept of least-squares polynomial filters as presented in [12], and how it can be applied to filter unwanted eigenvalues outside of an interval of interest.

In the second part we introduce the thick-restart version of the Lanczos algorithm, which later in this thesis will be combined with the polynomial filter for the computation of interval-specific eigenpairs.

2.1 POLYNOMIAL FILTERING

Polynomial filtering is a process by which a $n \times n$ real symmetric (or complex Hermitian) matrix $\mathbf{A}$ is replaced by a function $\rho(\mathbf{A})$, where the filter function ρ has the property of filtering out unwanted eigenvalues.

Suppose we want to compute eigenvalues in the interval of interest $[\xi, \eta] \subset [a, b]$, where the interval $[a, b]$ contains the complete spectrum of $\mathbf{A}$. The polynomial $\rho(\lambda)$ is chosen in such a way that all eigenvalues of $\mathbf{A}$ in $[\xi, \eta]$ are transformed into dominant eigenvalues of $\rho(\mathbf{A})$. Therefore, due to the nature of the Lanczos algorithm the dominant eigenvalues of $\rho(\mathbf{A})$ will be approximated first.

The polynomials ρ we are using in this work are least-squares approximations to the Dirac-delta function using a series expansion based on Chebyshev polynomials of the first kind [12]. In the following sections we summarize the concepts presented by Li et al in [12].

2.1.1 *Least-Squares Polynomial Filters*

The Dirac-delta distribution is a generalized function that was introduced by Paul Dirac for the representation of an idealized point object, such as a point mass or point charge. It is a function that is equal to zero everywhere except for $t = \gamma$, where it represents a spike that is infinitely high and the integral over the line is equal to 1. In the approach presented in [12], the filter is a least-squares approximation to the Dirac-delta function.

The approximate polynomial $\rho_k(t)$ of the Dirac-delta δ_γ centered at γ is realized by expanding δ_γ as a degree k Chebyshev polynomial series defined as

$$\rho_k(t) = \sum_{j=0}^{k} \mu_j T_j(t) \tag{2}$$

with

$$\mu_j = \begin{cases} \frac{1}{2} & \text{if} \quad j = 0 \\ \cos(j \arccos(\gamma)) & \text{otherwise} \; , \end{cases} \tag{3}$$

where k is the degree of the polynomial and the Chebyshev polynomial T_j of the first kind of order j is defined as

$$T_j(t) = \cos(n \arccos(t)), \quad t \in [-1,1], \quad j = 0,1,2,\dots \tag{4}$$

Since the Dirac-delta function is a distribution — and thus expanding it using Eq. (2) may not be mathematically rigorous or permissible — the authors state with [12, Proposition 3.1] that the polynomial defined as

$$\hat{\rho}_k(t) = \rho_k(t)/\rho_k(\gamma) \tag{5}$$

is the polynomial that minimizes $\|r(t)\|_w$ over all polynomials r of degree $\leq k$, such that $r(\gamma) = 1$, with $\|\cdot\|_w$ being the Chebyshev L^2-norm.

If we want to apply the polynomial $\hat{\rho}_k$ in Eq. (5) to the matrix $\mathbf{A}$ using the Chebyshev polynomials as defined in Eq. (4), we need to map the eigenvalues of $\mathbf{A}$ to the reference interval $[-1,1]$, which can be accomplished by using the following linear map from $[\lambda_{\text{min}}, \lambda_{\text{max}}]$ to $[-1,1]$:

$$\hat{\mathbf{A}} = \frac{\mathbf{A} - c\mathbf{I}}{d} \quad \text{with} \quad c = \frac{\lambda_{\text{max}} + \lambda_{\text{min}}}{2}, \; d = \frac{\lambda_{\text{max}} - \lambda_{\text{min}}}{2} \; . \tag{6}$$

The extremal eigenvalues λ_{min} and λ_{max} of $\mathbf{A}$ used in Eq. (6) can be approximated by lower and upper bounds obtained by adequate perturbations of the largest and smallest eigenvalue, which can be computed by running a small number of iterations of the Lanczos algorithm.

2.1.1.1 *Oscillations and Damping*

The expansion of discontinuous functions will exhibit oscillations near the discontinuities, which are known as *Gibbs oscillations*. To alleviate this behavior it is thus usual to add smoothing coefficients g_j^k so that Eq. (2) is replaced by

$$\rho_k(t) = \sum_{j=0}^{k} g_j^k \, \mu_j T_j(t) \; . \tag{7}$$

These coefficients can be calculated by using different smoothing approaches. Jackson smoothing [9, 14] is one of the best known approaches, and computes the smoothing factors g_j^k by using the formula

$$g_j^k = \frac{\sin(j+1)\alpha_k}{(k+2)\sin\alpha_k} + \left(1 - \frac{j+1}{k+2}\right)\cos(j\,\alpha_k) \quad \text{with} \quad \alpha_k = \frac{\pi}{k+2} \; . \tag{8}$$

Another smoothing approach proposed by Lanczos [11, Chapter 4] is called σ-smoothing and uses simpler smoothing coefficients:

$$\sigma_0^k = 1, \quad \sigma_j^k = \frac{\sin(j\theta_k)}{j\theta_k}, j = 1,\dots,k, \quad \text{with} \quad \theta_k = \frac{\pi}{k+1} \; . \tag{9}$$

Fig. 1 shows three polynomial filters $\hat{\rho}_k$ of degree $k = 11$ for the interval $[-0.2, 0.2]$, one of which is without smoothing and the other two with Jackson damping or the Lanczos σ-damping, respectively.

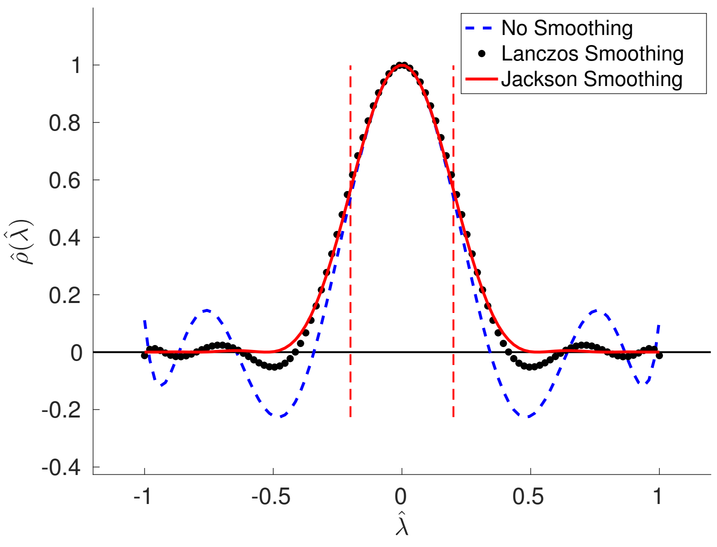


Figure 1: Polynomial filters $\hat{\rho}_k$ of degree $k = 11$ for the interval $[-0.2, 0.2]$, using three different smoothing approaches.

2.1.2 *Choosing the Degree of the Filter*

The degree of the polynomial filter $\hat{\rho}_k$ is automatically computed based on the defined interval of interest $[\xi, \eta] \subset [\lambda_{\min}, \lambda_{\max}]$ and on the type of smoothing coefficients to be used. The interval boundaries ξ and η are first transformed using Eq. (6) to $\hat{\xi}$ and $\hat{\eta}$, respectively:

$$\hat{\xi} = (\xi - c)/d \;, \tag{10}$$

$$\hat{\eta} = (\eta - c)/d \;. \tag{11}$$

These transformations guarantee that $\hat{\xi}, \hat{\eta} \in [-1, 1]$. The procedure then starts with a low degree polynomial and increases k until the values of $\hat{\rho}_k(\hat{\xi})$ and $\hat{\rho}_k(\hat{\eta})$ both fall below a certain threshold ϕ. Once this happens, we set $\tau = \min\{\hat{\rho}_k(\hat{\xi}), \hat{\rho}_k(\hat{\eta})\}$, with $\tau < \phi$ being a *bar value* that will be used in Section 2.1.3 to filter out unwanted eigenvalues.

In Fig. 2 three scaled polynomial filters $\hat{\rho}_k$ with different thresholds ϕ for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$ are shown. The dotted lines in Fig. 2 connect the point $\hat{\rho}_k(\hat{\xi})$ to $\hat{\rho}_k(\hat{\eta})$. As we can recognize from the slope of the dotted line connecting the two points, we have $\hat{\rho}_k(\hat{\xi}) \neq \hat{\rho}_k(\hat{\eta})$ which is not very helpful in selecting the desired eigenvalues in the interval $[\xi, \eta]$. For this reason, in the next section a *balancing* step is introduced, such that $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta})$ is guaranteed.

2.1.3 *Balancing the Filter*

For the selection of eigenvalues λ in the interval of interest $[\xi, \eta]$ it is desirable to have a polynomial filter $\hat{\rho}_k$ whose values at the transformed boundaries $\hat{\xi}$ and $\hat{\eta}$ are the same. This is accomplished by moving the center γ of the Dirac-delta function δ_γ away from the midpoint of the interval $[\xi, \eta]$. Once we have adjusted

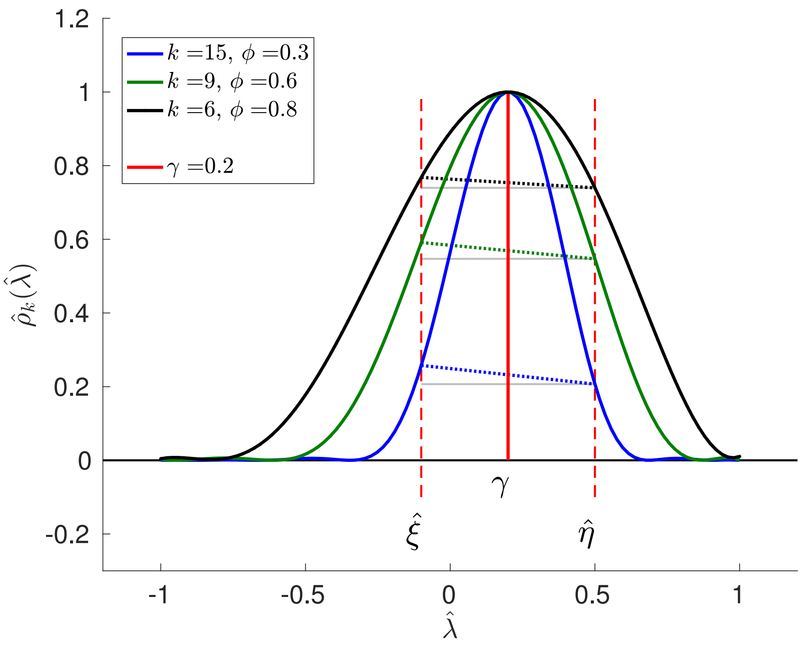


Figure 2: Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$, using three different thresholds ϕ and Jackson smoothing. The dotted lines connect $\hat{\rho}_k(\hat{\xi})$ to $\hat{\rho}_k(\hat{\eta})$ for each polynomial filter, clearly showing $\hat{\rho}_k(\hat{\xi}) \neq \hat{\rho}_k(\hat{\eta})$.

the center, and thus $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta})$, determining if a computed eigenvalue θ_j of $\hat{\rho}_k(\hat{\mathbf{A}})$ corresponds to an eigenvalue λ_j in $[\xi, \eta]$ becomes a simple task, namely: If $\tau \equiv \hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta})$, then

$$\lambda_j \in [\xi, \eta] \quad \Longleftrightarrow \quad \theta_j \geq \tau \ . \tag{12}$$

Hence, finding all eigenvalues $\lambda_j \in [\xi, \eta]$ can be accomplished by finding all eigenvalues θ_j of $\hat{\rho}_k(\hat{\mathbf{A}})$ that are greater than or equal to τ. It is important to mention that this step is only used as a preselection tool. The matrix $\mathbf{A}$ and $\hat{\rho}_k(\hat{\mathbf{A}})$ share the same eigenvectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ [7]. Thus, once all eigenpairs $(\theta_j, \mathbf{u}_j)$ with $\theta_j \geq \tau$ have been computed, we can use the corresponding eigenvectors $\mathbf{u}_j$ to extract the eigenvalues of $\mathbf{A}$ in $[\xi, \eta]$ by first evaluating the Rayleigh quotient

$$\tilde{\lambda}_j = \mathbf{u}_j^T \mathbf{A} \mathbf{u}_j \ , \tag{13}$$

and then check if $\tilde{\lambda}_j \in [\xi, \eta]$.

To adjust the center γ such that $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta})$, the polynomial ρ_k from Eq. (7) is first written in terms of the variable $\theta_t = \arccos(t)$, i.e.

$$\rho_k(\cos(\theta_t)) = \sum_{j=0}^{k} g_j^k \cos(j\theta_\gamma) \cos(j\theta_t) \ . \tag{14}$$

Newton's method is then used to solve the equation

$$\rho_k(\cos(\theta_{\hat{\xi}})) - \rho_k(\cos(\theta_{\hat{\eta}})) = 0 \tag{15}$$

with respect to θ_γ. Since $\cos(j\theta_\gamma) = \mu_j$ in Eq. (14), with μ_j being defined in Eq. (3), it is important to note that the first smoothing coefficient g_0^k is multiplied by $1/2$

to simplify notation, meaning that the first term with $j = 0$ is not 1 but rather $1/2$. Using this notation and Eq. (14), Eq. (15) can be rewritten as

$$\begin{aligned} f(\theta_\gamma) &\equiv \rho_k(\cos(\theta_{\hat{\xi}})) - \rho_k(\cos(\theta_{\hat{\eta}})) \\ &= \sum_{j=0}^{k} g_j^k \cos(j\theta_\gamma))[\cos(j\theta_{\hat{\xi}}) - \cos(j\theta_{\hat{\eta}}))] = 0 \; . \end{aligned} \tag{16}$$

Next, Newton's method requires the first derivative of f with respect to θ_γ, which is given by

$$f'(\theta_\gamma) = -\sum_{j=0}^{k} g_j^k j \sin(j\theta_\gamma)[\cos(j\theta_{\hat{\xi}}) - \cos(j\theta_{\hat{\eta}})] \; . \tag{17}$$

Further, the authors of [12] provide a good initial guess with a mid-angle defined as

$$\theta_c = \frac{1}{2}(\theta_{\hat{\xi}} + \theta_{\hat{\eta}}) \; , \tag{18}$$

which often yields convergence of the Newton iteration in one or two steps. Still, in some cases (e.g. for low degree polynomials) Newton's method may fail to converge in two steps, and in these cases the roots of Eq. (16) are computed exactly by solving the eigenvalue problem given by

$$\frac{1}{2}\mathbf{C}\mathbf{t} = \gamma \mathbf{t} \; , \tag{19}$$

where

$$\mathbf{C} = \begin{pmatrix} 0 & 2 & & & & \\ 1 & 0 & 1 & & & \\ & 1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 1 \\ -\beta_0 & -\beta_1 & \dots & \dots & 1-\beta_{k-2} & -\beta_{k-1} \end{pmatrix} \tag{20}$$

is a Hessenberg matrix in $\mathbb{R}^{k\times k}$, β_j is defined as

$$\beta_j = \frac{g_j^k[\cos(j\theta_{\hat{\xi}} - \cos(j\theta_{\hat{\eta}}))]}{g_k^k[\cos(k\theta_{\hat{\xi}} - \cos(k\theta_{\hat{\eta}}))]} \; , \tag{21}$$

and $\mathbf{t}$ has components $t_j = \cos(j\theta_\gamma) = T_j(\gamma)$ (see [12, Appendix A] for a detailed derivation). The new center γ_{new} is then taken to be the eigenvalue of $\mathbf{C}/2$ that is closest to the value $\cos(\theta_c)$.

Fig. 3 shows three balanced polynomial filters $\hat{\rho}_k$ for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$. In Fig. 2 the same interval has been used to compute the polynomials $\hat{\rho}_k$ without the balancing step. By comparing the dotted lines in Fig. 3 to the ones in Fig. 2, we can see that for all balanced polynomial filters shown in Fig. 3 $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta}) = \tau$ holds. Further, by comparing the value of each γ in both Figs. 2 and 3, respectively, we can see that in Fig. 3 the new midpoint γ of each approximated Dirac-delta function has been moved away from the previous midpoint, thus yielding balanced polynomials.

In Fig. 4 the polynomial filters for the end interval $[\hat{\xi}, \hat{\eta}] = [0.75, 1]$ are shown. The Dirac-delta function is now centered at $\gamma = 1$. Since $[\hat{\xi}, \hat{\eta}]$ is a right-end interval of $[-1, 1]$, only the bar value $\tau = \hat{\rho}_k(\hat{\xi})$ is needed to discard unwanted eigenvalues.

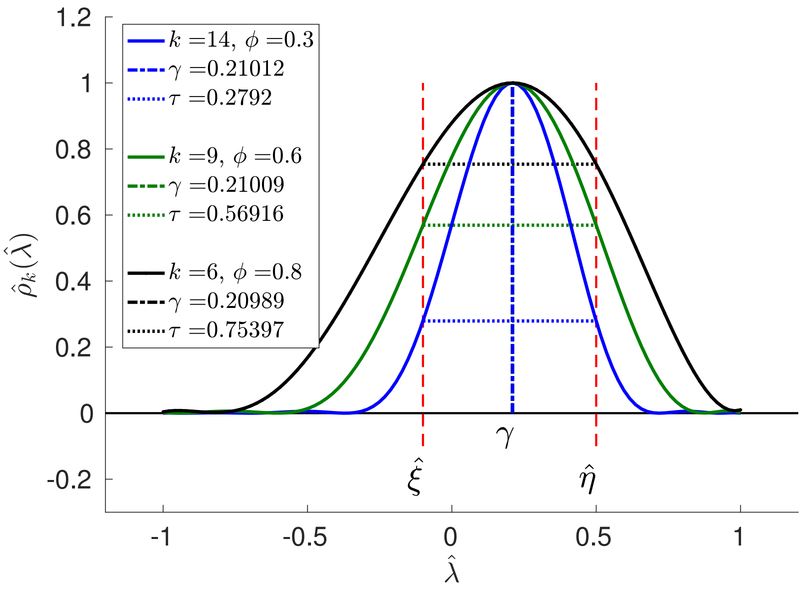


Figure 3: Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [-0.1, 0.5]$, using three different thresholds ϕ and Jackson smoothing. Each polynomial filter has been balanced such that $\hat{\rho}_k(\hat{\xi}) = \hat{\rho}_k(\hat{\eta}) = \tau$ (dotted lines).

2.2 THE THICK-RESTART LANCZOS ALGORITHM

We present a general description of the thick-restart version of the Lanczos algorithm for the computation of extremal eigenvalues [18]. Although the eigensolver is going to be applied to the matrix $\hat{\rho}_k(\hat{\mathbf{A}})$ in our work, throughout this section we are going to work with a Hermitian matrix denoted by $\mathbf{B}$ to keep the description of the algorithm as general as possible.

The Lanczos algorithm [10] is an iterative algorithm applicable to the eigenvalue problem

$$\mathbf{B}\mathbf{x} = \lambda \mathbf{x} \,, \tag{22}$$

where $\mathbf{B}$ is Hermitian, or in the real case a symmetric matrix operator.

The algorithm starts with a unit vector $\mathbf{q}_1$ and builds a sequence of vectors $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_m \in \mathbb{C}^n$ which form an orthonormal basis of the Krylov subspace

$$\mathcal{K}_m(\mathbf{B}, \mathbf{q}_1) = \text{span}\left\{\mathbf{q}_1, \mathbf{B}\,\mathbf{q}_1, \dots, \mathbf{B}^{m-1}\,\mathbf{q}_1\right\} \,. \tag{23}$$

At each step i, the vector $\mathbf{B}\mathbf{q}_i$ is orthogonalized against $\mathbf{q}_i$ and (when $i > 1$) against $\mathbf{q}_{i-1}$ by a Gram-Schmidt process:

$$\beta_{i+1}\mathbf{q}_{i+1} = \mathbf{B}q_i - \alpha_i \mathbf{q}_i - \beta_i \mathbf{q}_{i-1} \,. \tag{24}$$

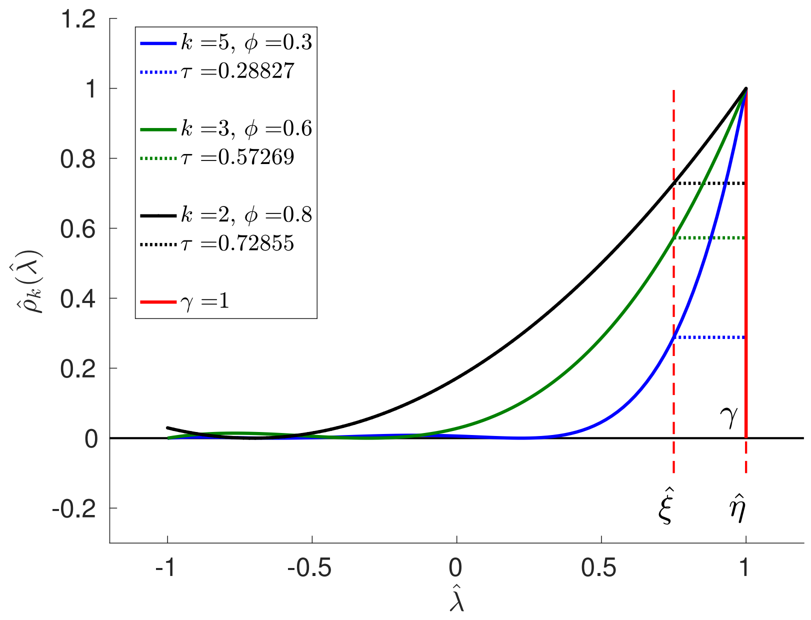


Figure 4: Polynomial filters $\hat{\rho}_k$ (Eq. (5)) of different degrees k for the interval $[\hat{\xi}, \hat{\eta}] = [0.75, 1]$, using three different thresholds ϕ and Jackson smoothing. Each polynomial filter has been balanced such that $\hat{\rho}_k(\hat{\xi}) = \tau$ (dotted lines).

In theory, i.e. with exact arithmetic, this three-term recurrence computes an orthonormal basis $\{\mathbf{q}_1, \ldots, \mathbf{q}_m\}$ of $\mathcal{K}_m(\mathbf{B}, \mathbf{q})$, but in the presence of rounding, orthogonality between the $\mathbf{q}_i$'s is lost soon after at least one eigenvector starts converging. A remedy to this problem is to reorthogonalize the vectors when needed, such that the orthogonality among the $\mathbf{q}_i$'s to working precision is enforced. The m-step Lanczos algorithm is shown in Algorithm 1, in which $\mathbf{Q}_j \equiv [\mathbf{q}_1, \ldots, \mathbf{q}_j]$ contains the basis constructed up to step j as its column vectors and a reorthogonalization step is included on Line 7 in Algorithm 1.

In the new orthonormal basis $\mathbf{Q}_m$ the operator $\mathbf{B}$ is represented by the real symmetric tridiagonal matrix

$$\mathbf{T}_m = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{m-1} \\ & & \beta_{m-1} & \alpha_m \end{pmatrix}, \tag{25}$$

where the scalars α_i, β_i are those produced by the Lanczos algorithm. Eq. (24) can be rewritten in the form

$$\mathbf{B}\mathbf{Q}_m = \mathbf{Q}_m \mathbf{T}_m + \beta_{m+1} \mathbf{q}_{m+1} \mathbf{e}_m^H, \tag{26}$$

where $\mathbf{e}_m$ is the mth column of the canonical basis and $\mathbf{q}_{m+1}$ is the last vector computed by the Lanczos algorithm at step m.

Let $(\theta_i^{(m)}, \mathbf{y}_i^{(m)})$ be the eigenpair of $\mathbf{T}_m$ at the mth step of the process. The eigenvalues $\theta_i^{(m)}$ are called Ritz values and will approximate some of the eigenvalues of $\mathbf{B}$ as m increases. The vectors $\mathbf{u}_i^{(m)} = \mathbf{Q}_m \mathbf{y}_i^{(m)}$, known as Ritz vectors, will ap-

Algorithm 1 The m-step Lanczos algorithm

```
1:  Input: A Hermitian matrix $\mathbf{B} \in \mathbb{C}^{n\times n}$, and an initial unit vector $\mathbf{q}_1 \in \mathbb{C}^n$.
2:  $\mathbf{q}_0 = 0, \beta_1 = 0$
3:  for $i = 1, 2, \ldots, m$ do
4:      $\mathbf{w} = \mathbf{B}\mathbf{q}_i - \beta_i \mathbf{q}_{i-1}$
5:      $\alpha_i = \mathbf{q}_i^H \mathbf{w}$
6:      $\mathbf{w} = \mathbf{w} - \alpha_i \mathbf{q}_i$
7:      Reorthogonalize: $\mathbf{w} = \mathbf{w} - \mathbf{Q}_i(\mathbf{Q}_i^H \mathbf{w})$
8:      $\beta_{i+1} = \|\mathbf{w}\|_2$
9:      if $\beta_{i+1} = 0$ then
10:         $\mathbf{q}_{i+1}$ = a random vector of unit norm that is orthogonal to $\mathbf{q}_1, \ldots, \mathbf{q}_i$
11:     else
12:         $\mathbf{q}_{i+1} = \mathbf{w}/\beta_{i+1}$
```

proximate the related eigenvectors of $\mathbf{B}$. The Ritz pair $(\theta_i^{(m)}, \mathbf{u}_i^{(m)})$ will be a good approximation to an eigenpair of $\mathbf{B}$ if the residual norm

$$\|\mathbf{r}\| = \|\mathbf{B}\mathbf{u}_i - \theta_i \mathbf{u}_i\| \tag{27}$$

is less than a prescribed threshold. Further, the Lanczos algorithm yields good approximations to extreme eigenvalues of $\mathbf{B}$ rather fast, whereas convergence to eigenvalues located deep inside the spectrum is much slower.

A disadvantage of the Lanczos algorithm is the fact that there is no way to determine in advance how many steps will be needed. In many cases, convergence to the eigenvalues of interest within a specified accuracy will not occur until the number of iterations m gets very large. Maintaining the orthogonality of such large bases becomes expensive and even intractable for large m.

To help maintain orthogonality and thus minimize the costs caused by reorthogonalizing the bases, the dimension of the search space is limited and *restarting schemes* are introduced. Restarting means that the starting vector $\mathbf{q}_1$ is replaced with an improved vector $\mathbf{q}_1^*$ (a Ritz vector computed during the previous iterations) and a new Lanczos decomposition with the new vector is computed. In the case of a thick restart, the algorithm is restarted not with one but with multiple Ritz vectors.

In the following we recall the steps from [12] for extending the standard Lanczos method shown in Algorithm 1 to the thick-restart Lanczos method as defined by Wu and Simon [18].

Suppose that after m steps of the Lanczos algorithm we have l Ritz vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_l$. We wish to use these Ritz vectors along with the last vector $\mathbf{q}_{m+1}$ as restarting vectors. From [15, Proposition 6.8] we know that each Ritz vector $\mathbf{u}_i$ has a residual in the direction of $\mathbf{q}_{m+1}$, i.e.

$$(\mathbf{B} - \theta_i \mathbf{I})\mathbf{u}_i = (\beta_{m+1}\mathbf{e}_m^H \mathbf{y}_i)\mathbf{q}_{m+1} \equiv s_i \mathbf{q}_{m+1} \quad \text{for} \quad i = 1, \ldots, l \; . \tag{28}$$

By defining $\hat{\mathbf{q}}_i = \mathbf{u}_i$ and $\hat{\mathbf{q}}_{l+1} = \mathbf{q}_{m+1}$ and rewriting Eq. (28) in matrix form, it follows that

$$\mathbf{B}\hat{\mathbf{Q}}_{\mathbf{l}} = \hat{\mathbf{Q}}_{\mathbf{l}}\mathbf{\Theta}_l + \hat{\mathbf{q}}_{\mathbf{l+1}}\mathbf{s}^H \ , \tag{29}$$

where $\hat{\mathbf{Q}}_{\mathbf{l}} = [\hat{\mathbf{q}}_{\mathbf{1}}, \ldots, \hat{\mathbf{q}}_{\mathbf{l}}]$, $\hat{\mathbf{\Theta}}_{\boldsymbol{l}} = \text{diag}(\theta_1, \ldots, \theta_l)$, and $\mathbf{s}^T = [s_1, \ldots, s_l]$.

To compute the $(l+2)$th basis vector $\hat{\mathbf{q}}_{\mathbf{l+2}}$, we compute $\mathbf{B}\hat{\mathbf{q}}_{\mathbf{l+1}}$ and orthonormalize it against the previous $l+1$ basis vectors:

$$\beta_{l+2}\hat{\mathbf{q}}_{\mathbf{l+2}} = \mathbf{B}\hat{\mathbf{q}}_{\mathbf{l+1}} - \sum_{i=1}^{l} s_i\hat{\mathbf{q}}_{\mathbf{i}} - \alpha_{l+1}\hat{\mathbf{q}}_{\mathbf{l+1}} \ , \tag{30}$$

where $\alpha_{l+1} = \hat{\mathbf{q}}_{\mathbf{l+1}}{}^H\mathbf{B}\hat{\mathbf{q}}_{\mathbf{l+1}}$.

After completing the first step after the restart, the following equation holds:

$$\mathbf{B}\hat{\mathbf{Q}}_{\mathbf{l+1}} = \hat{\mathbf{Q}}_{\mathbf{l+1}}\hat{\mathbf{T}}_{\mathbf{l+1}} + \beta_{l+2}\hat{\mathbf{q}}_{\mathbf{l+2}}\mathbf{e}_{l+1}^H \quad \text{with} \quad \hat{\mathbf{T}}_{\mathbf{l+1}} = \begin{pmatrix} \hat{\mathbf{\Theta}}_{\boldsymbol{l}} & \mathbf{s} \\ \mathbf{s}^H & \alpha_{l+1} \end{pmatrix} . \tag{31}$$

The algorithm now proceeds just like the Lanczos algorithm, i.e., $\hat{\mathbf{q}}_{\mathbf{k}}$ for $k \geq l+3$ is computed until the dimension reaches m. Selecting which Ritz pairs $(\theta_j, \mathbf{u}_j = \mathbf{Q}_m\mathbf{y}_j)$ the algorithm should keep for the restart can be based on different heuristic schemes; we refer to [18, 19] for a few examples and further details.

The thick-restart Lanczos algorithm is sketched in Algorithm 2. A detailed description of the thick-restart Lanczos method can be found in Wu and Simon [18].

Algorithm 2 Thick-restart Lanczos algorithm

```
 1: Input: A Hermitian matrix $\mathbf{B} \in \mathbb{C}^{n\times n}$ and an initial unit vector $\mathbf{q}_1 \in \mathbb{C}^n$.
 2: $\mathbf{q}_0 = 0, \beta_1 = 0, Its = 0, lock = 0, U = [\ ]$
 3: while $Its \leq MaxIts$ do
 4:     if $l > 0$ then
 5:         Perform thick restart step (30), which results in $\hat{\mathbf{Q}}_{l+2}$ and $\hat{\mathbf{T}}_{l+1}$ in (31)
 6:     for $i = l+1, \dots, m$ do
 7:         $\mathbf{w} = \mathbf{B}\mathbf{q}_i - \beta_i \mathbf{q}_{i-1}$
 8:         $\alpha_i = \mathbf{q}_i^H \mathbf{w}$
 9:         $\mathbf{w} = \mathbf{w} - \alpha_i \mathbf{q}_i$
10:         Reorthogonalize: $\mathbf{w} = \mathbf{w} - \mathbf{Q}_i(\mathbf{Q}_i^H \mathbf{w})$
11:         $\beta_{i+1} = \|\mathbf{w}\|_2$
12:         if $\beta_{i+1} = 0$ then
13:             $\mathbf{q}_{i+1}$ = a random vector of unit norm that is orthogonal to $\mathbf{q}_1, \dots, \mathbf{q}_i$
14:         else
15:             $\mathbf{q}_{i+1} = \mathbf{w}/\beta_{i+1}$
16:         Set $Its = Its + 1$
17:     Results: $\mathbf{Q}_m \in \mathbb{C}^{n\times m}$ and $\mathbf{T}_m \in \mathbb{R}^{m\times m}$.
18:     Compute all eigenpairs $(\theta_j, \mathbf{y}_j)$ of $\mathbf{T}_m$ and the norm of the corresponding
        residuals defined in Eq. (28)
19:     if the norm of all residuals is smaller than a prescribed threshold then
20:         return the Ritz pairs $(\theta_j, \mathbf{u}_j = \mathbf{Q}_m \mathbf{y}_j)$ of interest (smallest or largest magni-
            tude of $\theta_j$)
21:     else
22:         Select which vectors of the current basis should be used at the next restart
            based on a heuristic scheme, e.g. [18, 19]
```

3 IMPLEMENTATION

In this chapter we combine the methods from Chapter 2 and a few other useful tools into a utility that can be used to compute the eigenpairs of a $n \times n$ real symmetric (or complex Hermitian) matrix $\mathbf{A}$ within a specified interval of interest $[\xi, \eta]$ in parallel by simply providing an XML configuration file (see A.1). The outline of the utility can be found in Algorithm 3.

In Section 3.1 we first introduce Trilinos, the workhorse behind the numerical work done by the utility. Next, in Section 3.2 we give some details about the distribution pattern used throughout our implementation for the distributed-memory parallel computations. Finally, in the remaining sections we explain the ideas and implementation details behind most of the steps in Algorithm 3.

Algorithm 3 The BosonPeak Utility

1. Import user-specified configuration via XML file (see A.1).
2. Import the matrix $\mathbf{A}$.
3. *If requested*, estimate the extremal eigenvalues $\lambda_{\min}, \lambda_{\max}$ of $\mathbf{A}$ using a small number of Lanczos steps.
4. Transform the matrix $\mathbf{A}$ to $\hat{\mathbf{A}}$ based on Eq. (6).
5. *If requested*, estimate the number of eigenvalues in the specified interval $[\xi, \eta]$.
6. Compute the polynomial filter $\hat{\rho}_k$ using Algorithm 4.
7. Compute the eigenpairs $(\tilde{\lambda}_j, \mathbf{u}_j)$ of the matrix $\mathbf{A}$ with $\tilde{\lambda}_j \in [\xi, \eta]$ and residual norms $r_j = \|\mathbf{A}\mathbf{u}_j - \tilde{\lambda}_j \mathbf{u}_j\|$ using Algorithm 6
8. *If requested*, write the eigenvalues $\tilde{\lambda}_j$, eigenvectors $\mathbf{u}_j$ and residual norms r_j in the MatrixMarket (MM) format to the hard disk.

3.1 TRILINOS

The utility is written in C++11 and uses Trilinos[1] extensively, a collection of open-source software libraries, called *packages*, for the development of scientific applications.

1 https://trilinos.org/

3.1.1 *Epetra*

The Epetra[2] package provides classes for the construction and use of serial and distributed parallel linear algebra objects, and many of the Trilinos solver packages work with Epetra objects. The most used linear algebra objects in our implementation are sparse, Compressed Row Storage (CRS) matrices, and collections of dense vectors called *multivectors*. An `Epetra_CrsMatrix` object stores a sparse matrix in the CRS format with real-valued double-precision entries. An `Epetra_MultiVector` object instead stores each vector in a multivector as a contiguous array of double-precision numbers. Both objects are extensively used for sparse matrix-vector multiplications in the various Trilinos solver packages.

3.1.2 *Anasazi*

Anasazi[3] is a package that offers a collection of algorithms for solving large-scale eigenvalue problems. As part of the package it provides solver managers to implement a strategy for solving an eigenvalue problem.

3.1.3 *Teuchos*

The Teuchos package is a collection of common tools used throughout Trilinos. Among other things, it provides templated access to BLAS and LAPACK interfaces, parameter lists that allow to specify parameters for different packages, and memory management tools for aiding in correct allocation and deletion of memory.

Part of the memory management tools is an implementation of a Reference Counting Pointer (RCP) class, which for an object tracks a count of the number of references to it held by other objects. Once the counter reaches zero, the object can be destroyed. The advantage of a RCP is that the possibility of memory leaks in a program can be reduced, which is especially important when working with rather large objects, e.g. an `Epetra_CrsMatrix` object storing over 10^9 nonzero entries. RCP objects are heavily used throughout our implementation to manage large objects, especially large temporary objects that are only needed during a fraction of the whole computation.

3.2 PARALLELIZATION

Trilinos supports distributed-memory parallel computations through the Message Passing Interface (MPI). Both the `Epetra_CrsMatrix` and the `Epetra_Multivector` objects can be used in a distributed memory environment by defining data distribution patterns using `Epetra_Map` objects.

The entries of a distributed object (such as rows or columns of a `Epetra_CrsMatrix` or the rows of a `Epetra_Multivector`) are represented by *global indices* uniquely over the entire object. A map essentially assigns global indices to available MPI ranks (further referred to as only *ranks*), which in our case a single rank corresponds —

2 https://trilinos.org/packages/epetra/
3 https://trilinos.org/packages/anasazi/

but in general is not limited — to a single core on a processor. For example, if the map assigns the global row index i of a sparse matrix to a rank p, we say that the rank p *owns* the global row index i. Within a rank, we refer to a global index using a *local index*. In the specific example of a global row index i, this means that a rank p can access the local data of the global row i it owns by using the local index $l(i)$, where the function l maps the global index i to a local index for the specific rank.

For the addressing, local and global indices in Epetra use by default a 32-bit `int` type. Since our implementation is based on the C++11 language standard and we want to allow computations with large matrices, we explicitely use 64-bit global indices of type `long long` when working with distributed linear algebra objects.

An `Epetra_Map` object enapsulates the details of distributing data over MPI ranks. In our implementation, we use *contiguous* and *one-to-one* maps for the distribution of the rows of `Epetra_CrsMatrix` and `Epetra_MultiVector` objects. *Contiguous* means that the list of global indices on each MPI rank forms an interval and is strictly increasing. A *one-to-one* map instead allows a global index only to be owned by a single rank. For the columns, the distribution pattern we are using distributes the complete set of global column indices for a given global row, meaning that if a rank p owns the global row index i, it also owns all global column indices j on that row, thus having local access to the global entry (i, j). The map used for the distribution of the columns is thus not a one-to-one map, since a global column index can be owned by multiple ranks.

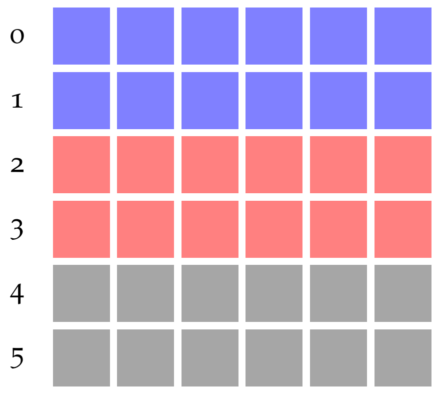


Figure 5: Row-wise distribution pattern of a matrix $\mathbf{A} \in R^{6\times6}$ using 3 MPI ranks.

As an illustrative example we distribute the following 8×8 sparse matrix $\mathbf{A}$ over 4 ranks using the previously specified distribution pattern for the rows and columns, respectively:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 0 & 0 & 5 & 0 & 0 & 7 \\ 0 & 0 & 0 & 4 & 0 & 2 & 3 & 1 \\ 10 & 0 & 11 & 0 & 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 120 & 13 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 & 4 \\ 77 & 0 & 18 & 0 & 0 & 0 & 0 & 6 \\ 10 & 20 & 55 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 11 & 153 & 131 & 0 \end{pmatrix} \tag{32}$$

The distribution of the matrix **A** over the 4 ranks looks as follows:

- Rank 0:
  - Global row indices owned: $\{0, 1\}$
  - Global column indices owned: $\{0, 1, 3, 4, 5, 6, 7\}$
  - Global entries locally accessible:
    $\{(0,0), (0,1), (0,4), (0,7), (1,3), (1,5), (1,6), (1,7)\}$
- Rank 1:
  - Global row indices owned: $\{2, 3\}$
  - Global column indices owned: $\{0, 2, 3, 4, 5\}$
  - Global entries locally accessible:
    $\{(2,0), (2,2), (2,5), (3,3), (3,4)\}$
- Rank 2:
  - Global row indices owned: $\{4, 5\}$
  - Global column indices owned: $\{0, 2, 5, 6, 7\}$
  - Global entries locally accessible:
    $\{(4,5), (4,6), (4,7), (5,0), (5,2), (5,7)\}$
- Rank 3:
  - Global row indices owned: $\{6, 7\}$
  - Global column indices owned: $\{0, 1, 2, 4, 5, 6\}$
  - Global entries locally accessible:
    $\{(6,0), (6,1), (6,2), (7,4), (7,5), (7,6)\}$

Now that the data is distributed, each rank can start to work on the owned global matrix entries. For this, the ranks need the local indices to the owned rows and columns. This is accomplished by using a mapping function $l(i)$ that maps a global index i to a local one. A specific rank can then use the local indices to locally access the owned global entries. An example of such a mapping is shown for the global indices on rank 3:

- Rank 3:

- Local row indices: $l(4) = 0,\ l(5) = 1$
- Local column indices:
  $l(0) = 0,\ l(2) = 1,\ l(5) = 2,\ l(6) = 3,\ l(7) = 4$

Rank 3 can now use the local indices to access the data. For example, by locally setting $\mathbf{A}(l(5), l(7)) = 49$, the global entry $\mathbf{A}(5,7)$ gets the value 49 set. These changes to the structure of an `Epetra_CrsMatrix` object have to be committed via a call to the `FillComplete` function, which for one helps construct communication patterns to support distributed sparse matrix-vector multiplications.

3.3 PARALLEL MATRIX IMPORT

The matrix import implemented in the utility allows to efficiently import large matrices stored in a Hierarchical Data Format 5 (HDF5)[4] file directly to a `Epetra_CrsMatrix` object.

HDF5 is a data model, library, and file format for storing and managing data collections of all sizes and complexity. One of the advantages of using the HDF5 format to store and import large matrices is the possibility to use MPI to read the HDF5 files in parallel. For this reason Trilinos provides the `EpetraExt::HDF5` class for importing a matrix stored in a HDF5 file to a `Epetra_CrsMatrix`.

Since the `EpetraExt::HDF5` class currently does not provide an import function for matrices with 64-bit global indices of type `long long`, we extended the class by a function suitable for working with 64-bit data. Further, because some of the matrices may be delivered in the MM format, in a preprocessing step a Python script can be used to convert the matrices stored in the MM format to a HDF5 file suitable for the import (see Appendix A.3).

3.4 ESTIMATION OF THE EXTREMAL EIGENVALUES OF $\mathbf{A}$

For the transformation of $\mathbf{A}$ to $\hat{\mathbf{A}}$, as shown in Eq. (6), we need to specify a value for the extremal eigenvalues λ_{min} and λ_{max} of $\mathbf{A}$. In case we want to approximate the extremal eigenvalues, we simply run a small number of Lanczos iterations (e.g. 10 iterations in our case) for the approximation of the required extremal eigenvalues denoted by $\tilde{\lambda}_{\text{min}}$ and $\tilde{\lambda}_{\text{max}}$, respectively.

In case the Lanczos method manages to return converged eigenvalues by only running a few iterations, we use $\lambda_{\text{min}} = \tilde{\lambda}_{\text{min}}$ and $\lambda_{\text{max}} = \tilde{\lambda}_{\text{max}}$. In situations where the computation of the extremal eigenvalues does not converge, we simply subtract (when approximating λ_{min}) or add (when approximating λ_{max}) a small perturbation from the approximated eigenvalues by using

$$\lambda_{\text{min}} = \tilde{\lambda}_{\text{min}} - p \cdot |\tilde{\lambda}_{\text{min}}| \ , \tag{33}$$
$$\lambda_{\text{max}} = \tilde{\lambda}_{\text{max}} + p \cdot |\tilde{\lambda}_{\text{max}}| \ , \tag{34}$$

where $p \ll 1$ is a small perturbation factor.

4 https://support.hdfgroup.org/HDF5/

3.5 ESTIMATION OF THE EIGENVALUE COUNT IN A SPECIFIED INTERVAL

The estimation of the number of eigenvalues located in a given interval is computed by approximating the trace of an eigenprojector [6]. This estimation will be later needed (on Line 7 of Algorithm 3) to specify how many of the largest eigenvalues of $\hat{\rho}_k(\hat{\mathbf{A}})$ the eigensolver specified in Algorithm 6 should compute and return. In the following we give a summary of the technique as presented by Di Napoli et al in [6].

Let $\lambda_j, j = 1, \ldots, n$ be the eigenvalues and $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ the associated orthonormal eigenvectors of our real, symmetric (or complex Hermitian) matrix $\mathbf{A}$. For a specified interval $[\xi, \eta]$, with $\lambda_{\min} \leq \xi < \eta \leq \lambda_{\max}$, our aim is to count the number of eigenvalues λ_i in the interval $[\xi, \eta]$. We estimate the eigenvalue count by seeking an approximation of the trace of the eigenprojector

$$\mathbf{P} = \sum_{\lambda_i \in [\xi, \eta]} \mathbf{u}_i \mathbf{u}_i^T \ . \tag{35}$$

Since the eigenvalues of a projector are either zero or one, the trace of $\mathbf{P}$ in Eq. (35) is equal to the number of eigenvalues in $[\xi, \eta]$. Thus, the number of eigenvalues $\mu_{[\xi,\eta]}$ located in the interval $[\xi, \eta]$ can be calculated by evaluating the trace of the projector in Eq. (35):

$$\mu_{[\xi,\eta]} = \operatorname{tr}(\mathbf{P}) \ . \tag{36}$$

Considering that the projector $\mathbf{P}$ is typically not available, we approximate it in the form of a polynomial function of $\mathbf{A}$. For this, we can interpret $\mathbf{P}$ as a characteristic function of the interval $[\xi, \eta]$:

$$\mathbf{P} = h(\mathbf{A}) \quad \text{where} \quad h(t) = \begin{cases} 1 & \text{if} \quad t \in [\xi, \eta] \ , \\ 0 & \text{otherwise} \end{cases} \tag{37}$$

In our case, we approximate $h(t)$ with a finite sum $\psi(t)$ of Chebyshev polynomials, namely $\mathbf{P} \approx \psi(\mathbf{A})$. Further, we estimate the trace of $\mathbf{P}$ using Hutchinson's unbiased estimator [8]. Hutchinson showed that given a general matrix $\mathbf{A}$ and a randomly generated vector $\mathbf{v}$ with identically independently distributed (i.i.d.) random variables as entries, the equation $\mathrm{E}(\mathbf{v}^T \mathbf{A} \mathbf{v}) = \operatorname{tr}(\mathbf{A})$ holds. Thus, the idea behind the estimator is to compute an estimate $\mathcal{T}_{n_v}$ of the trace $\operatorname{tr}(\mathbf{A})$ by generating n_v samples of random vectors $\mathbf{v}_k, k = 1, \ldots, n_v$ and then computing the average of $\mathbf{v}_k^T \mathbf{A} \mathbf{v}$ over these samples:

$$\operatorname{tr}(\mathbf{A}) \approx \mathcal{T}_{n_v} = \frac{1}{n_v} \sum_{k=1}^{n_v} \mathbf{v}_k^T \mathbf{A} \mathbf{v} \ . \tag{38}$$

Hutchinson originally used i.i.d. Rademacher random variables, whereby each entry of $\mathbf{v}$ assumes the values -1 or 1 with probability $1/2$. In general, any sequence of random vectors v_k whose entries are i.i.d. random variables can be used, as long as the mean of their entries is zero [2]. In our case, we used a Gaussian estimator to compute $\mathcal{T}_{n_v}$ in Eq. (38) by using normally distributed variables for the entries of the random vectors $\mathbf{v}_k$. Despite the fact that the Gaussian estimator has a larger variance than that of Hutchinson, it shows better convergence in terms of the number of sample vectors n_v [1].

The trace of $\mathbf{P}$ can now be computed as

$$\mu_{[\xi,\eta]} = \mathrm{tr}(\mathbf{P}) \approx \frac{n}{n_v} \sum_{k=1}^{n_v} \mathbf{v}_k^T \psi(\mathbf{A}) \mathbf{v}_k \ , \tag{39}$$

where the sample vectors $\mathbf{v}_k \in \mathbb{R}^n$ are normalized to one $\|\mathbf{v}_k\| = 1$ and the factor n is introduced by this constraint.

Using the polynomial filtering approach, the step function $h(t)$ in Eq. (37) is expanded into a degree p Chebyshev polynomial series:

$$h(t) \approx \psi_p(t) = \sum_{j=0}^{p} \gamma_j T_j(t) \ , \tag{40}$$

where T_j are the j-degree Chebyshev polynomials of the first kind, and the coefficients γ_j are the expansion coefficients of the step function h, which are defined as

$$\gamma_j = \begin{cases} \frac{1}{\pi}(\arccos(a) - \arccos(b)) & \text{if} \quad j = 0 \ , \\ \frac{2}{\pi}\left(\frac{\sin(j \arccos(a)) - \sin(j \arccos(b))}{j}\right) & \text{if} \quad j > 0 \ . \end{cases} \tag{41}$$

Following the definition of h in Eq. (40) and by using the transformed matrix $\hat{\mathbf{A}}$ from Eq. (6), we obtain an expansion of the projector $\mathbf{P}$ by defining

$$\mathbf{P} \approx \psi_p(\hat{\mathbf{A}}) = \sum_{j=0}^{p} \gamma_j T_j(\hat{\mathbf{A}}) \ , \tag{42}$$

where the transformation to $\hat{\mathbf{A}}$ is needed to ensure that the eigenvalues of $\hat{\mathbf{A}}$ lie in the reference interval $[-1, 1]$ on which the Chebyshev polynomials are defined. Next, we need to take care of Gibbs oscillations at the boundaries, since we are trying to approximate the discontinuos function h in Eq. (37). For this reason the Jackson damping coefficients g_j^p are introduced using Eq. (8), and Eq. (42) is replaced by

$$\mathbf{P} \approx \psi_p(\hat{\mathbf{A}}) = \sum_{j=0}^{p} g_j^p \gamma_j T_j(\hat{\mathbf{A}}) \ . \tag{43}$$

Combining Eqs. (42) and (39) finally yields the following estimate

$$\mu_{[\xi,\eta]} = \mathrm{tr}(\mathbf{P}) \approx \frac{n}{n_v} \sum_{k=1}^{n_v} \left[\sum_{j=0}^{p} \mathbf{g}_j^p \, \gamma_j \, v_k^T T_j(\hat{\mathbf{A}}) \mathbf{v}_k \right] \ . \tag{44}$$

The advantage of this approach is that it requires only matrix-vector products. Further, the vectors $\mathbf{w}_j = T_j(\hat{\mathbf{A}})\mathbf{v}$ for a given $\mathbf{v}$ can be computed using the three-term recurrence relation of Chebyshev polynomials

$$T_j(t) = 2\,t\,T_{j-1}(t) - T_{j-2}(t) \quad \text{with} \quad T_0(t) = 1 \quad \text{and} \quad T_1(t) = t \ , \tag{45}$$

which results in

$$\mathbf{w}_j = 2\,\hat{\mathbf{A}}\,\mathbf{w}_{j-1} - \mathbf{w}_{j-2} \quad \text{with} \quad \mathbf{w}_0 = T_0(\hat{\mathbf{A}})\,\mathbf{v} = \mathbf{v} \quad \text{and} \quad \mathbf{w}_1 = T_1(\hat{\mathbf{A}})\,\mathbf{v} = \hat{\mathbf{A}}\mathbf{v} \; . \tag{46}$$

In our implementation, by default we use $n_v = 40$ and a polynomial degree $p = 100$ for the estimation of the eigenvalue count $\mu_{[\xi,\eta]}$ with Eq. (44).

3.6 COMPUTATION OF THE POLYNOMIAL FILTER $\hat{\rho}_k$

Algorithm 4 contains the needed elements for the computation of the polynomial filter $\hat{\rho}_k$ as defined by Eqs. (5) and (7). The algorithm incorporates the ideas presented in Section 2.1 and is mostly based (with some minor adaptions) on the EigenValues Slicing Library [16] developed by Saad et al.

Simply put, Algorithm 4 tries to compute a suitable polynomial filter $\hat{\rho}_k$ for the interval $[\xi, \eta]$ by iterating through the degrees $k = k_{\min}, k_{\min}+1, \ldots, k_{\max}$ and checking if the polynomial $\hat{\rho}_k$ evaluated at the transformed boundaries $\hat{\xi}, \hat{\eta}$ falls below the specified threshold ϕ. Once this is the case, the algorithm returns the optimal degree k of $\hat{\rho}_k$, the center γ such that $\hat{\rho}_k(\gamma) = 1$, and the set of normalized expansion coefficients $\mathcal{N} = \{\hat{v}_0^k, \hat{v}_1^k, \ldots, \hat{v}_k^k\}$ with $\hat{v}_j^k$ being the normalized expansion coefficient defined as

$$\hat{v}_j^k = g_j^k\,\mu_j / \rho_k(\gamma) \; . \tag{47}$$

These parameters can then be used to construct the optimal polynomial filter $\hat{\rho}_k$:

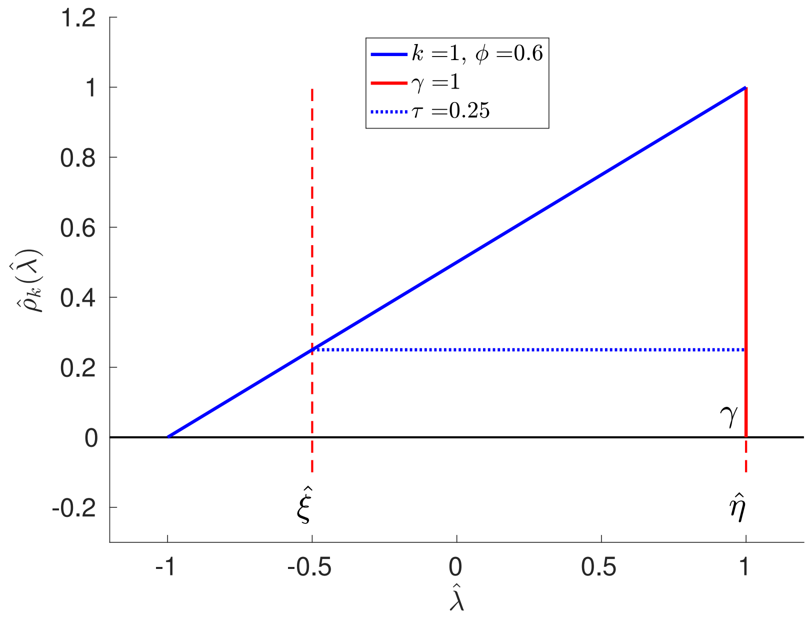
$$\hat{\rho}_k(t) = \sum_{j=0}^{k} \hat{v}_j^k T_j(t) \; . \tag{48}$$

Further, the bar value $\tau = \min(\hat{\rho}_k(\hat{\xi}),\ \hat{\rho}_k(\hat{\eta}))$ with $\tau < \phi$ is returned, which will be needed to filter out unwanted eigenvalues θ_j of $\hat{\rho}_k(\hat{\mathbf{A}})$ by simply checking if $\theta_j < \tau$. For the threshold ϕ, in our implementation we use by default $\phi = 0.9$ for interior intervals and $\phi = 0.6$ for end intervals.

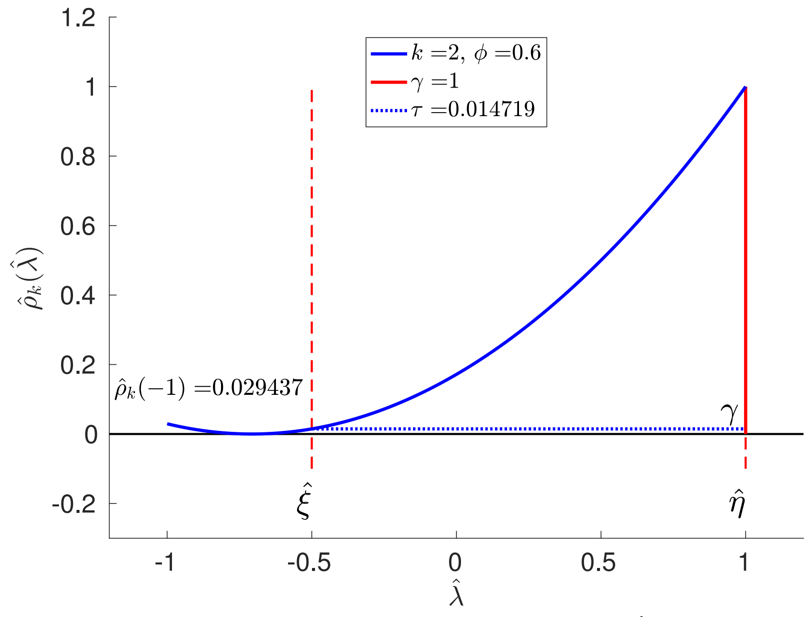
Although the smoothing approach can be specified as an input parameter in Algorithm 4, in case $[\xi, \eta]$ is an end interval we explicitly use only Jackson-smoothed polynomial filters. Additionally, we first try to use a degree-one Jackson-smoothed polynomial filter $\hat{\rho}_1(t) = \rho_1(t)/\rho_1(\gamma)$, with $\hat{\rho}_1$ defined as

$$\hat{\rho}_1(t) = g_0^k \mu_0 T_0(t) + g_1^k \mu_1 T_1(t) \; . \tag{49}$$

In cases where the end interval is rather large, $\hat{\rho}_1$ suffices to filter out the unwanted eigenvalues. Further, using $\hat{\rho}_1$ in such a situation helps to better split the region containing the wanted eigenvalues from the region containing the unwanted eigenvalues. Fig. 6a shows a degree-one polynomial filter $\hat{\rho}_1$ for the end interval $[\hat{\xi}, \hat{\eta}] = [-0.5, 1]$. In Fig. 6b, a degree-two polynomial is shown for the same end interval. The approach shown in Fig. 6b is problematic for discarding eigenvalues using the bar value τ during the preselection step, since at the left boundary of the interval $[-1, 1]$ we can see that $\hat{\rho}_k(-1) > \tau$, despite $\hat{\lambda} = -1$ being outside of the interval of interest. Still, even if we would use the approach shown in Fig. 6b, once we compute the eigenpair $(\hat{\rho}_k(-1),\ \mathbf{u}_l)$ of $\hat{\rho}_k(\hat{\mathbf{A}})$ using the Lanczos method, in a next step the eigenvalue $\tilde{\lambda}_l = \mathbf{u}_l^T \mathbf{A} \mathbf{u}_l$ would be rejected due to $\tilde{\lambda}_l \notin [\xi, \eta]$. Although feasible, the approach in Fig. 6b is rather unfavorable compared to Fig. 6a, since the computation of a much larger number of eigenvalues of $\hat{\rho}_k(\hat{\mathbf{A}})$ would be needed to capture all eigenvalues $\hat{\lambda} \in [\xi, \eta]$ during the preselection, thus requiring the eigensolver to construct a much larger Krylov basis as actually needed.



(a) Polynomial filter $\hat{\rho}_k$ with $k = 1$ for the end interval $[\hat{\xi}, \hat{\eta}] = [-0.5, 1]$ using a threshold $\phi = 0.6$ and Jackson smoothing.



(b) Polynomial filter $\hat{\rho}_k$ with $k = 2$ for the end interval $[\hat{\xi}, \hat{\eta}] = [-0.5, 1]$ using a threshold $\phi = 0.6$ and Jackson smoothing. This approach is unfavorable in this case, since $\hat{\rho}_k(-1) > \tau$ for $\hat{\lambda} = -1$ outside of the interval $[\hat{\xi}, \hat{\eta}]$.

Figure 6: Handling end interval cases by using different degrees for the polynomial filter $\hat{\rho}_k$.

Algorithm 4 Computation of the polynomial filter $\hat{\rho}_k$ from Eqs. (5) and (7)

```
1:  Input: Boundaries $\xi$ and $\eta$ of the interval of interest $[\xi, \eta] \subseteq [\lambda_{\min}, \lambda_{\max}]$, the extremal eigenvalues (or approximations) $\lambda_{\min}$ and $\lambda_{\max}$ of $\mathbf{A}$, a minimum starting degree $k_{\min}$ and a maximum degree $k_{\max}$, the smoothing approach to compute the coefficients $g_j^k$ (no smoothing, Jackson [Eq. (8)], or Lanczos [Eq. (9)]), a threshold $\phi_{\text{interior}}$ for interior intervals and a threshold $\phi_{\text{end}}$ for end intervals.
2:  Output: An optimal degree $k$, the center $\gamma$, a bar value $\tau$ and the set $\mathcal{N} = \{\hat{v}_0^k, \hat{v}_1^k, \ldots, \hat{v}_k^k\}$ containing the normalized expansion coefficients $\hat{v}_j^k = \mu_j g_j^k / \rho_k(\gamma)$ with $j = 0, \ldots, k$. The polynomial filter $\hat{\rho}_k$ can then be constructed using Eq. (48). Further, $\tau$ can be used to filter out unwanted eigenvalues $\theta_j$ of $\hat{\rho}_k(\hat{\mathbf{A}})$ by checking $\theta_j < \tau$.
3:  $c = (\lambda_{\max} + \lambda_{\min})/2$                          ▷ See Eq. (6)
4:  $d = (\lambda_{\max} - \lambda_{\min})/2$
5:  $\hat{\xi} = (\xi - c)/d$                  ▷ Transform the interval boundaries of $[\xi, \eta]$
6:  $\hat{\eta} = (\eta - c)/d$
7:  if $[\hat{\xi}, \hat{\eta}]$ is an end interval then
8:      Compute the degree-one polynomial approximation $\rho_1(t) = g_0^k \mu_0 T_0(t) + g_1^k \mu_1 T_1(t)$, where $g_0^k, g_1^k$ are Jackson smoothing coefficients (Eq. (8)).
9:      $\hat{\rho}_1(\hat{\xi}) = \rho_1(\hat{\xi}) / \rho_1(\gamma)$
10:     $\hat{\rho}_1(\hat{\eta}) = \rho_1(\hat{\eta}) / \rho_1(\gamma)$
11:     if $\hat{\rho}_1(\hat{\xi}) < \phi$ or $\hat{\rho}_1(\hat{\eta}) < \phi$ then
12:         $k = 1$
13:         $\mathcal{N} = \{\nu_0 / \rho_1(\gamma), \nu_1 / \rho_1(\gamma)\}$
14:         $\tau = \min(\hat{\rho}_1(\hat{\xi}), \hat{\rho}_1(\hat{\eta}))$
15:         return $\gamma, k, \mathcal{N}$, and $\tau$          ▷ $\hat{\rho}_1$ is a suitable polynomial filter
16: if $\hat{\rho}_1$ is not a suitable polynomial filter then
17:     $\gamma = 0$.
18:     for degree $k = k_{\min}, k_{\min} + 1, \ldots, k_{\max}$ do
19:         if $[\hat{\xi}, \hat{\eta}]$ is an interior interval then
20:             Compute $\gamma_{\text{balanced}}$ by first running two steps of the Newton iteration to compute the root $\theta_{\text{root}}$ of Eq. (16) using $\theta_c$ from Eq. (18) as initial guess.
21:             if Newton iteration converged then
22:                 $\gamma_{\text{balanced}} = \cos(\theta_{\text{root}})$
23:             else
24:                 Solve the eigenvalue problem in Eq. (19), resulting in $k$ eigenvalues $\tilde{\gamma}_j$.
25:                 $\gamma_{\text{balanced}} = \min_{j=1,\ldots,k} |\tilde{\gamma}_j - \cos(\theta_c)|$
26:             Set $\gamma = \gamma_{\text{balanced}}$
27:             Set $\phi = \phi_{\text{interior}}$
28:             Compute $g_j^k$ by using the specified smoothing approach (no smoothing, Jackson [Eq. (8)], or Lanczos [Eq. (9)]).
```

Algorithm 5 Computation of the polynomial filter $\hat{\rho}_k$ from Eqs. (7) and (5) (continued)

```
29:         else
30:             if $[\hat{\xi}, \hat{\eta}]$ is an interval on the left end of $[-1, 1]$ then
31:                 Set $\gamma = -1$.        ▷ Center the Dirac-delta function at the left boundary of $[-1, 1]$
32:             else if $[\hat{\xi}, \hat{\eta}]$ is an interval on the right end of $[-1, 1]$ then
33:                 Set $\gamma = 1$.         ▷ Center the Dirac-delta function at the right boundary of $[-1, 1]$
34:             $\phi = \phi_{\text{end}}$
35:             Compute $g_j^k$ by explicitly using the Jackson smoothing approach (Eq. (8)).
36:         Compute the expansion coefficients $\nu_j = \mu_j g_j^k$ with $j = 0, \ldots, k$, where $\mu_j$ is computed using Eq. (3)
37:         Compute $\rho_k(\hat{\eta})$ and $\rho_k(\gamma)$, where $\rho_k(t) = \sum_{j=0}^{k} \nu_j T_j(t)$.
38:         if $\rho_k(\hat{\eta}) / \rho_k(\gamma) < \phi$ then
39:             $\mathcal{N} = \{\nu_0/\rho_k(\gamma), \ldots, \nu_k/\rho_k(\gamma)\}$
40:             $\tau = \rho_k(\eta)/\rho_k(\gamma)$
41:             break                     ▷ $\hat{\rho}_k$ is a suitable polynomial filter
42:     if a suitable polynomial filter $\hat{\rho}_k$ has been found then
43:         return $\gamma, k, \mathcal{N}$, and $\tau$
44:     else
45:         Require a larger value for the maximum degree $k_{\max}$.
```

3.7 COMBINING THE EIGENSOLVER AND POLYNOMIAL FILTERING

The thick-restart Lanczos algorithm (Algorithm 2) will be used to compute a prescribed number n_{ev} of the largest eigenvalues of the matrix $\hat{\rho}_k(\hat{\mathbf{A}})$, including the corresponding eigenvectors. In this case, on Line 7 of Algorithm 2 the matrix-vector product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{q}_i$ is computed instead of $\mathbf{A}\mathbf{q}_i$.

Let $\theta_{n-n_{\text{ev}}+1} \leq \theta_{n-n_{\text{ev}}+2} \leq \cdots \leq \theta_n$ denote the n_{ev} largest eigenvalues of $\hat{\rho}_k(\hat{\mathbf{A}})$. In a first step, from these extremal eigenvalues the eigenpairs $(\theta_i, \mathbf{u}_i)$ with $\theta_i < \tau$ are discarded. Next, for the remaining eigenpairs with $\theta_i \geq \tau$ the Rayleigh quotients relative to the matrix $\mathbf{A}$ are evaluated by computing $\tilde{\lambda}_i = \mathbf{u}_i^T \mathbf{A} \mathbf{u}_i$. Finally, it is checked if $\tilde{\lambda}_i \in [\xi, \eta]$.

Trilinos offers a parallel implementation of a block version of the Krylov-Schur method (BKSM) [17] as part of the Anasazi package, which is provided by the `Anasazi::BlockKrylovSchurSolMgr` class. As mentioned in [17], the Krylov-Schur method applied to a real, symmetric (or complex Hermitian) matrix is identical to the thick restart Lanczos algorithm of Wu and Simon [18]. Thus, on Line 3 of Algorithm 6 we use the `Anasazi::BlockKrylovSchurSolMgr` class to efficiently compute the n_{ev}

largest eigenvalues of $\hat{\rho}_k(\hat{\mathbf{A}})$. In this case, BKSM forms the orthonormal basis of the Krylov subspace

$$\mathcal{K}_m(\hat{\rho}_k(\hat{\mathbf{A}}), \mathbf{Q}_1) = \operatorname{span}\{\mathbf{Q}_1,\ \hat{\rho}_k(\hat{\mathbf{A}})\,\mathbf{Q}_1,\ \dots,\ \hat{\rho}_k(\hat{\mathbf{A}})^{m-1}\,\mathbf{Q}_1\}\ , \tag{50}$$

where $\mathbf{Q}_1 \in \mathbf{R}^{n\times b}$ and b is the block size. Hence, the product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{q}_i$ is further replaced with the matrix-multivector product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{Q}_i$.

Algorithm 6 Eigensolver with polynomial filtering

```
1: **Input:** Matrix $\hat{\rho}_k(\hat{\mathbf{A}})$, the bar value $\tau$ (Algorithm 4) and the number $n_{\text{ev}}$ of
   largest eigenvalues of $\hat{\rho}_k(\hat{\mathbf{A}})$ the eigensolver should compute and return
2: **Output:** All eigenpairs $(\tilde{\lambda}_j, \mathbf{u}_j)$ with $\tilde{\lambda}_j \in [\xi, \eta]$ and the corresponding residual
   norm $r_j = \|\mathbf{A}\mathbf{u}_j - \tilde{\lambda}_j \mathbf{u}_j\|$
3: Compute the $n_{\text{ev}}$ eigenpairs $(\theta_j, \mathbf{u}_j)$ with the largest magnitude of $\theta_j$ by ap-
   plying the thick-restart Lanczos algorithm [12] (or the Krylov-Schur algorithm
   [17]) to the matrix $\hat{\rho}_k(\hat{\mathbf{A}})$.
4: **for** each resulting eigenpair $(\theta_j, \mathbf{u}_j)$ **do**
5:     **if** $\theta_j < \tau$ **then**
6:         Ignore this pair.
7:     **else**
8:         Compute $\tilde{\lambda}_j = \mathbf{u}_j^H \mathbf{A}\mathbf{u}_j$.
9:         **if** $\tilde{\lambda}_j \notin [\xi, \eta]$ **then**
10:            Ignore this pair.
11:        **else**
12:            Keep the pair $(\tilde{\lambda}_j, \mathbf{u}_j)$.

13: **return** all eigenpairs $(\tilde{\lambda}_j, \mathbf{u}_j)$ with $\tilde{\lambda} \in [\xi, \eta]$ and the corresponding residual
    norm $r_j = \|\mathbf{A}\mathbf{u}_j - \tilde{\lambda}_j \mathbf{u}_j\|$
```

3.8 COMPUTATION OF THE POLYNOMIAL FILTER OPERATOR $\hat{\rho}_k(\hat{\mathbf{A}})$

In Anasazi, the `Anasazi::Eigenproblem` class encapsulates the information necessary to define an eigenvalue problem and stores the solutions computed by an eigensolver. Further, it also stores the operators associated with an eigenproblem.

In our case, we want to specifically compute the n_{ev} largest eigenpairs of $\hat{\rho}_k(\hat{\mathbf{A}})$ on Line 3 of Algorithm 6. We thus need to provide a suitable implementation of the Polynomial Filter Operator (PFO) $\hat{\rho}_k(\hat{\mathbf{A}})$, which by using Eqs. (2) and (5) is defined as

$$\hat{\rho}_k(\hat{\mathbf{A}}) = \sum_{j=0}^{k} \hat{v}_j^k\, T_j(\hat{\mathbf{A}})\ , \tag{51}$$

with $\hat{v}_j^k$ being defined in Eq. (47).

The PFO in Eq. (51) is implemented as an `Epetra_Operator` object, the latter being an interface for the implementation of real-valued double-precision operators. The `Epetra_Operator` interface requires a deriving class to implement an `Apply` function, which for the given PFO $\hat{\rho}_k(\hat{\mathbf{A}})$ computes the matrix-multivector product

$$\mathbf{Y} = \hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X} \,, \tag{52}$$

where $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n \times b}$ are multivectors, with b being the block size[5].

Using Eq. (7) we can write the product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X}$ as

$$\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X} = \sum_{j=0}^{k} \hat{v}_j^k \, T_j(\hat{\mathbf{A}}) \, \mathbf{X} \,. \tag{53}$$

Further, we can define $\mathbf{W}_j = T_j(\hat{\mathbf{A}})\,\mathbf{X}$ and use the three-term recurrence of Chebyshev polynomials shown in Eq. (45) to get

$$\mathbf{W}_j = 2\,\hat{\mathbf{A}}\,\mathbf{W}_{j-1} - \mathbf{W}_{j-2} \quad \text{with} \quad \mathbf{W}_0 = T_0(\hat{\mathbf{A}})\,\mathbf{X} = \mathbf{X} \quad \text{and} \quad \mathbf{W}_1 = T_1(\hat{\mathbf{A}})\,\mathbf{X} = \hat{\mathbf{A}}\mathbf{X} \,, \tag{54}$$

thus replacing Eq. (53) with

$$\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X} = \sum_{j=0}^{k} \hat{v}_j^k \mathbf{W}_j \,. \tag{55}$$

As explained in Section 3.2, the sparse matrices and multivectors are distributed row-wise. Hence, we exploit the locality of the row-data on each rank by constructing the product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X}$ row-wise. Algorithm 7 shows an outline of the distributed computation of the product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X}$ as defined in Eq. (55).

5 The matrix-multivector product in Eq. (52) is used by the BKSM implementation in Trilinos to compute the next multivector of the orthonormal basis spanning the Krylov subspace shown in Eq. (50) after completion.

Algorithm 7 Distributed computation of the product $\hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X}$ from Eq. (53)

```
1:  Input: Transformed matrix $\hat{\mathbf{A}} \in \mathbb{R}^{n \times n}$ (Eq. (6)), multivector $\mathbf{X} \in \mathbb{R}^{n \times b}$ and the
    normalized expansion coefficients $\hat{v}_j^k = \mu_j g_j^k / \rho_k(\gamma)$ with $j = 0, \ldots, k$.
2:  Output: Multivector $\mathbf{Y} \in \mathbb{R}^{n \times b}$, with $\mathbf{Y} = \hat{\rho}_k(\hat{\mathbf{A}})\mathbf{X}$.
3:  for each rank $p$ do
4:      for each owned row $i$ do
5:          $(\mathbf{W}_0)_{i,:} = \mathbf{X}_{i,:}$   ▷ The subscript in $\mathbf{X}_{i,:}$ denotes the complete $i$th row of $\mathbf{X}$
6:      $\mathbf{W}_1 = \hat{\mathbf{A}}\mathbf{X}$   ▷ Distributed sparse matrix-multivector multiplication
7:      for each owned row $i$ do
8:          $\mathbf{Y}_{i,:} = \hat{v}_0^k (\mathbf{W}_0)_{i,:} + \hat{v}_1^k (\mathbf{W}_1)_{i,:}$   ▷ Initialize first two terms of Eq. (55)
9:      $\mathbf{W}_{j-2} = \mathbf{W}_0$
10:     $\mathbf{W}_{j-1} = \mathbf{W}_1$
11:     $\mathbf{W}_j = [\ ],\ \mathbf{W}_{\text{tmp}} = [\ ]$
12:     for each polynomial degree $j = 2, \ldots, k$ do
13:         $\mathbf{W}_{\text{tmp}} = \hat{\mathbf{A}}\mathbf{W}_{j-1}$   ▷ Distributed sparse matrix-multivector multiplication
14:         for each owned row $i$ do
15:             $(\mathbf{W}_j)_{i,:} = 2\,(\mathbf{W}_{\text{tmp}})_{i,:} - (\mathbf{W}_{j-2})_{i,:}$
16:             $\mathbf{Y}_{i,:} = \mathbf{Y}_{i,:} + \hat{v}_j^k\,(\mathbf{W}_j)_{i,:}$
17:         $\mathbf{W}_{j-2} = \mathbf{W}_{j-1}$
18:         $\mathbf{W}_{j-1} = \mathbf{W}_j$
```

4 EXPERIMENTAL RESULTS

In this chapter we present the experimental results. In Section 4.1 the results of interval-specific eigenmode computations close to the Boson peak regime involving multiple Hessian matrices are presented and analyzed. Section 4.2 shows the performance results of our implementation of the PFO $\hat{\rho}_k$, where the scaling behavior and parallel performance is examined. For both the computation and performance results, the respective experimental setup and data set used are presented.

4.1 EIGENMODE COMPUTATIONS

4.1.1 *Experimental Setup*

The eigenmode computations were carried out on the Euler cluster of the ETH Zürich [1]. Euler stands for *Erweiterbarer, Umweltfreundlicher, Leistungsfähiger ETH-Rechner* and consists of Euler I (448 nodes, each equipped with two 12-core Intel Xeon E5-2697v2 processors processors), Euler II (768 nodes, each equipped with two 12-core Intel Xeon E5-2680v3 processors), and Euler III (1215 nodes, each equipped with a quad-core Intel Xeon E3-1285Lv5 processor).

In the specified environment, our implementation worked with OpenMPI 1.65, HDF5 1.8.12, Boost 1.57.0, and Trilinos 12.2.1. Further, the code has been compiled with GCC 4.8.2 and the following optimization flags:

```
-ftree-vectorize -march=corei7-avx -mavx -std=c++11 -O3
```

4.1.2 *Participation Ratio and Displacement Visualization*

Using the utility from Chapter 3, 8 Hessian matrices $\mathbf{H}_1, \ldots, \mathbf{H}_8 \in \mathbb{R}^{768000 \times 768000}$ of a system with $N = 256000$ atoms have been partially diagonalized to capture all eigenvalues $\lambda_i = \omega_i^2$ and corresponding eigenmodes $\mathbf{u}_i$ in the interval of interest $[0.1, 2]$. The Hessian matrices $\mathbf{H}_i$ are derived from computer generated three-dimensional model binary Lennard-Jones glasses computed by Derlet et al [5].

To quantify the amount of particles moving together in the vibrational eigenmodes we use the participation ratio defined for each eigenmode $\mathbf{u}_i$ as

$$PR(\lambda_i) = \frac{1}{N} \frac{\left(\sum_{j=1}^{N} \|\mathbf{u}_i(j)\|_2^2\right)^2}{\sum_{j=1}^{N} \|\mathbf{u}_i(j)\|_2^4}, \tag{56}$$

where $\mathbf{u}_i(j) \in \mathbb{R}^3$ denotes the displacement vector of particle j for the eigenmode i. $PR = 1$ denotes the involvement of all particles in the vibration of this mode, and $PR = 1/N$ means only an isolated particle is vibrating [3].

1 https://scicomp.ethz.ch/wiki/Euler

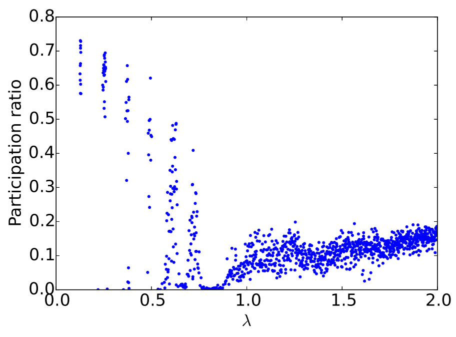
Table 1: Tabular overview of properties of the Hessian matrices $\mathbf{H}_1, \ldots, \mathbf{H}_8$ of a system with $N = 256000$ atoms. The second column contains the NNZ of $\mathbf{H}_i$, the third column shows the total number of eigenvalues $\lambda_j \in [0.1, 2]$, and in the third column the maximal residual norm over all computed residual norms $\|r_j\|$ for a given $\mathbf{H}_i$ is displayed, where the residual vector is defined as $\mathbf{r}_j = \mathbf{H}_i \mathbf{u}_j - \lambda_j \mathbf{u}_j$. Finally, in the last column the Time to Solution (TTS) for the eigensolver part of the utility (Line 7 in Algorithm 3) is shown; the TTS is based on computations with 48 cores on Euler.

Matrix	NNZ	Num. EV in $[0.1, 2]$	Max. $\|r_j\|$	TTS
$\mathbf{H}_1$	191 893 806	1 068	1.1303×10^{-9}	5 304.0621 s
$\mathbf{H}_2$	191 883 888	1 030	7.2249×10^{-9}	4 850.0367 s
$\mathbf{H}_3$	191 903 166	1 050	3.5170×10^{-8}	5 813.8738 s
$\mathbf{H}_4$	191 851 848	1 075	7.0602×10^{-9}	5 863.0400 s
$\mathbf{H}_5$	191 832 588	1 077	9.1160×10^{-9}	5 310.6390 s
$\mathbf{H}_6$	191 859 012	1 079	4.0634×10^{-9}	5 407.2488 s
$\mathbf{H}_7$	191 887 542	1 058	1.9001×10^{-9}	5 398.9464 s
$\mathbf{H}_8$	191 853 378	1 051	8.0480×10^{-9}	5 371.8900 s

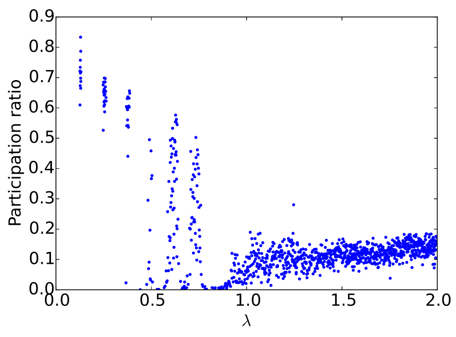
Fig. 7 displays the participation ratios resulting from the partial diagonalization of the Hessian matrices $\mathbf{H}_i, i = 1, \ldots, 8$. As we can observe from the data, each plot shows a rather interesting behavior for participation ratios in the region of $\lambda \in [0.7, 1.0]$. This region, where the participation ratio reaches a minimum, contains the Boson peak [5]. Further, we can recognize two regimes from the data. The peak structure in the regime with eigenvalues $\lambda < 0.7$ actually corresponds to sound due to the rather ordered structure denoted by large values for the participation ratios. For the regime $\lambda \geq 0.7$ the participation ratio instead is very small, and we observe disordered vibrations resulting in the breakup of sound.

In Fig. 8 the resulting displacements in the $x - y$ plane of some interesting eigenmodes of $\mathbf{H}_1$ are visualized. The eigenmodes corresponding to smaller eigenvalues shown in Figs. 8a and 8b show a plane-wave-like character due to having peak participation ratios. For eigenmodes belonging to larger eigenvalues in $[0.1, 2]$ as displayed in Figs. 8c and 8d, we can see that the shape of the eigenmodes becomes more complex, since for one the plane-wave-like character is not present anymore and patches with large displacements are formed.

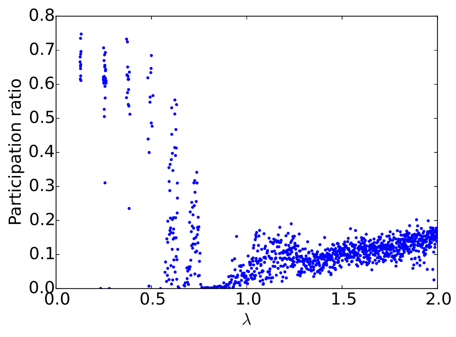
Finally, Fig. 8e shows the displacements for an eigenmode corresponding to a high frequency (large eigenvalue). In the top right corner we recognize rather large vibrations caused by a small number of particles. Although the participation ratio is rather small, further analysis using larger systems would be needed to actually classify the vibrations as really localized in such cases.



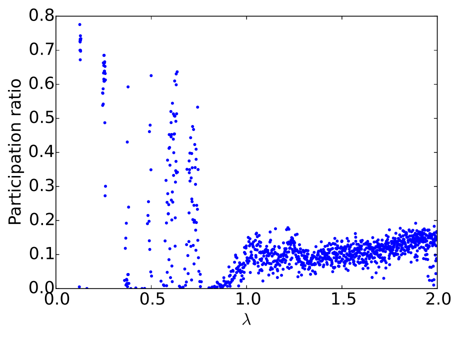
(a) $\mathbf{H}_1$



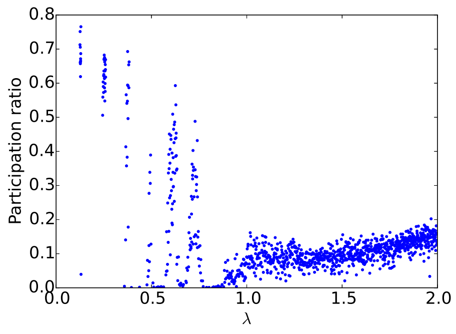
(b) $\mathbf{H}_2$



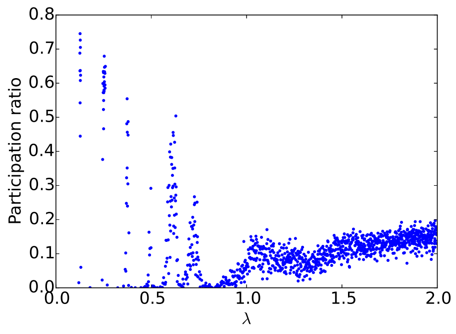
(c) $\mathbf{H}_3$



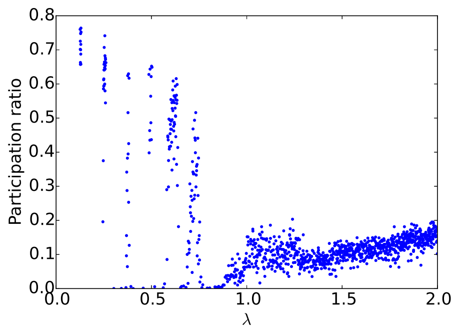
(d) $\mathbf{H}_4$



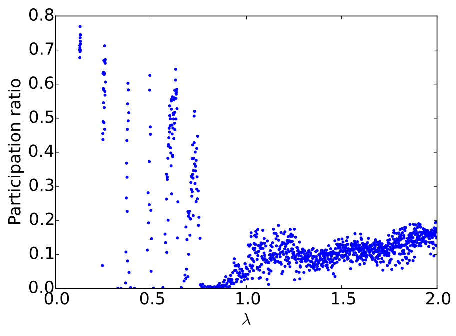
(e) $\mathbf{H}_5$



(f) $\mathbf{H}_6$



(g) $\mathbf{H}_7$



(h) $\mathbf{H}_8$

Figure 7: Eigenvalues $\lambda \in [0.1, 2.0]$ of the different samples $\mathbf{H}_i, i = 1, 2, \ldots, 8$ plotted against the participation ratios.

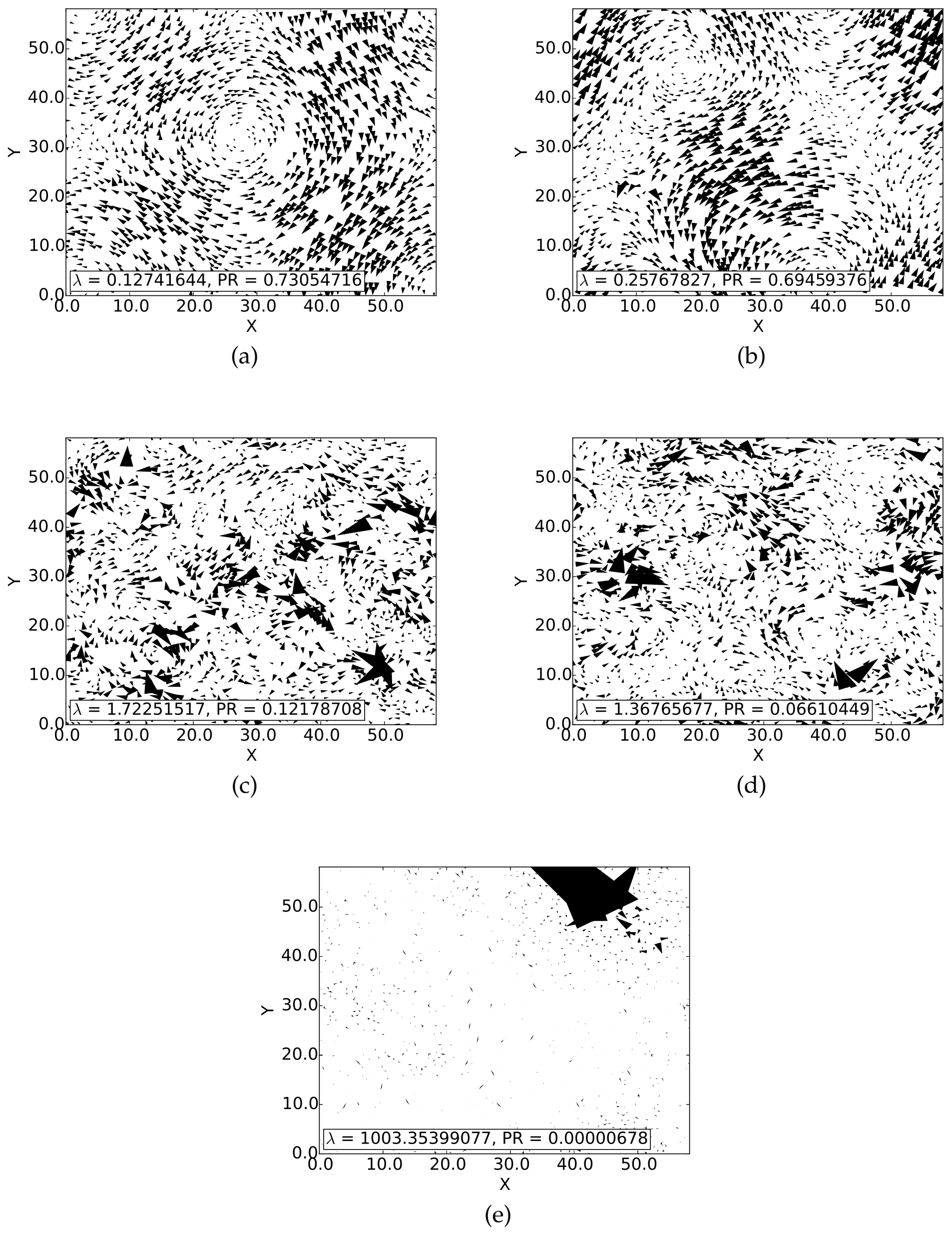


Figure 8: Displacement plots for different eigenmodes of $\mathbf{H}_1$. The plots show a cut along the $x - y$ plane ($\delta z = 0.15$ Å) containing the particle with the largest displacement. The arrows are proportional to the displacements of the particles, and the displacements have been magnified $\times 300$ for visibility reasons.

4.2 PERFORMANCE MEASUREMENTS

4.2.1 *Experimental Setup*

All the performance measurements were carried out on the Ra cluster[2] at PSI. The cluster itself consists of 32 nodes, of which 16 are equipped with 2 Intel Xeon E5-2690v3 (2.60 GHz) processors, each providing 12 cores, for a total of 24 cores per node. The other 16 nodes are equipped with 2 Intel Xeon E5-2697Av4 (2.60 GHz) processors, each processor providing 16 cores, resulting in 32 cores per node. Our performance measurements were run on four nodes that are part of the latter 16 nodes, thus allowing to perform measurements with up to 128 cores.

Our performance tests have been compiled with GCC 6.2.0 and the following optimization flags:

```
-ftree-vectorize -march=core-avx2 -mavx2 -std=c++11 -O3
```

Further, the following libraries were used: OpenMPI 1.10.4, HDF5 1.8.18, Boost 1.62.0, and Trilinos 12.10.1.

The performance measurements are all based on computations with a Hessian matrix $\mathbf{H}_1$ of dimension $n_1 = 96\,000$ with $nnz = 23\,985\,126$ non-zero elements.

The data shown in the speedup, parallel efficiency, strong scaling and weak scaling plots for our implementations are based on the average time over 10 runs of a given computation on a given number of cores. Further, 2 warm-up computations are run before the beginning of the actual measurements. This procedure avoids computations on an empty ("cold") cache, ensuring that after the first warm-up runs the cache is filled with data, resulting in fewer cache-misses during the first performance measurement compared to a cold-cache run.

Finally, all performance measurements presented in this section were carried out on $N_p = 1, 2, 4, \ldots, 128$ cores.

4.2.2 *Polynomial Filter Operator $\hat{\rho}_k(\hat{\mathbf{H}})$*

For these performance tests we measured the average wall-clock time it takes to compute the matrix-multivector product

$$\hat{\rho}_k(\hat{\mathbf{H}}_1)\,\mathbf{X} \tag{57}$$

using our distributed implementation of the PFO $\hat{\rho}_k(\hat{\mathbf{H}})$ shown in Algorithm 7, where $\mathbf{X} \in \mathbb{R}^{n_1 \times b}$, b is the block size, and $\hat{\mathbf{H}}_1$ is the transformed matrix $\mathbf{H}_1$.

4.2.2.1 *Speedup and Parallel Efficiency*

The *speedup* S_p is defined to be the ratio

$$S_p = \frac{t_1}{t_p}\,, \tag{58}$$

2 https://www.psi.ch/photon-science-data-services/offline-computing-facility-for-sls-and-swissfel-data-analysis

where t_1 is the time to execute the workload on one core and t_p is the time to execute the workload on p cores. The speedup can be also written as

$$S_p = \frac{t_1}{t_p} = \frac{1}{f_s + f_p/p}\,, \tag{59}$$

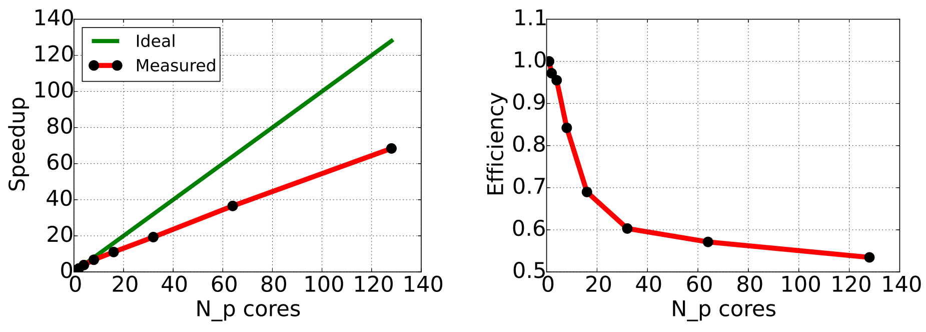
where f_s is the serial fraction of a routine, f_p the parallel fraction of the same routine and p the number of cores.

The *parallel efficiency* is defined as

$$E_p = \frac{t_1}{p\,t_p} = \frac{S_p}{p}\,, \tag{60}$$

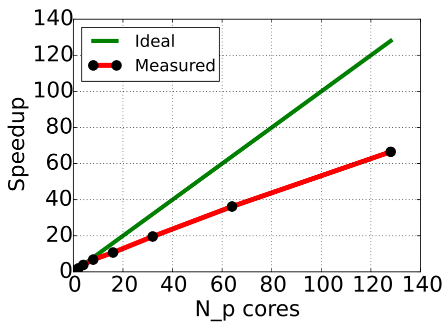
and depicts the speedup per core.

Fig. 9 shows the speedup and parallel efficiency of our implementation of Eq. (57) for an experiment with a large degree $k = 500$ and block size $b = 32$, both parameters being held fixed during the measurements. The green line depicts the ideal linear speedup. From the plot shown in Fig. 9 we can infer that the speedup of our implementation is linear, but still less than the ideal speedup. Fig. 10 shows the speedup and parallel efficiency for the same implementation with $k = 151$ and $b = 128$, which shows asymptotically a similar behavior to the previous experiment with a larger degree k and smaller block size b. Reason for these results is for one the communication overhead caused by the distributed sparse Matrix-Multivector Multiplication (spMMM) in Algorithm 7. The distributed spMMM used is the default implemented spMMM in Trilinos based on the row-wise distribution of the data as defined in Section 3.2.

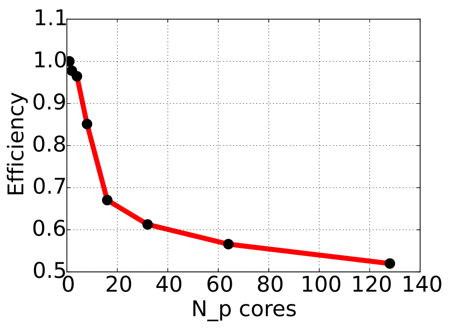


(a) Speedup (b) Parallel efficiency

Figure 9: Measured speedup and parallel efficiency of the polynomial operator implementation with $k = 500$ and $b = 32$, with the time measurements being averaged over 10 runs.



(a) Speedup



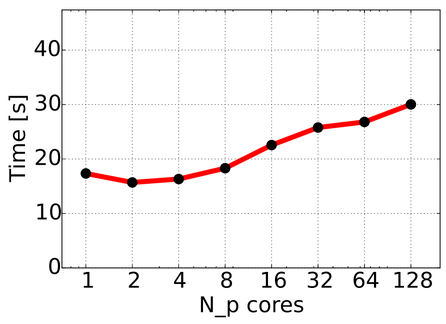
(b) Parallel efficiency

Figure 10: Measured speedup and parallel efficiency of the matrix-multivector multiplication in Eq. (57) with $k = 151$ and $b = 128$, with the time measurements being averaged over 10 runs.

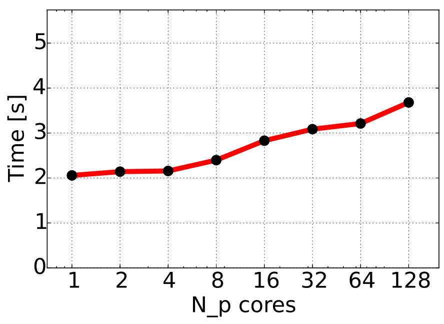
4.2.2.2 *Strong and Weak Scaling*

For the weak scaling measurements of Algorithm 7 the problem size assigned to each core stays constant and additional cores are used to solve a larger total problem. In our case, the problem size is depicted either by the block size b of $\mathbf{X}$ or by the polynomial degree k of $\rho_k(\hat{\mathbf{H}})$ in Eq. (53), and is scaled linearly with the number of cores.

Fig. 11a shows the weak scaling of the matrix-multivector multiplication with increasing block size. Fig. 11b instead shows the weak scaling with increasing polynomial degree. As can be observed from the plots, the weak scaling measurements show quite an increase in the time to solution, caused mostly by the previously mentioned communication overhead from the spMMM with the row-based distribution pattern.



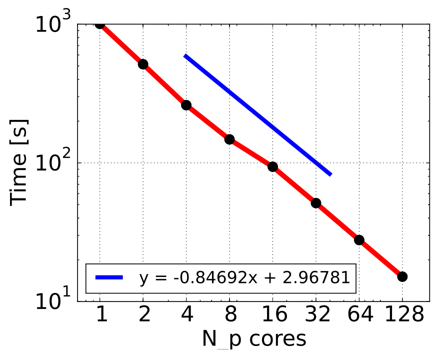
(a) $b = 2$ per core and overall fixed $k = 151$.



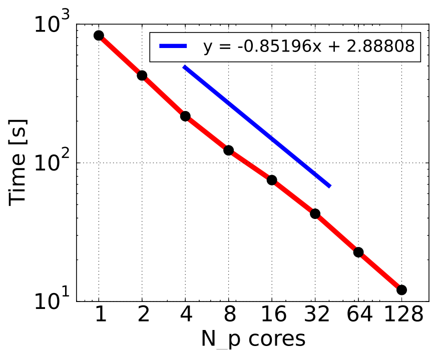
(b) $k = 5$ per core and overall fixed $b = 8$.

Figure 11: Weak scaling measurements of the matrix-multivector multiplication in Eq. (57) with increasing block size b (left) and increasing polynomial degree k (right).

The strong scaling measurements were carried out by keeping the total block size b of $\mathbf{X}$ or total polynomial degree k of $\rho_k(\hat{\mathbf{H}})$ constant and increasing the number of cores. Both strong scaling plots shown in Fig. 12a and Fig. 12b respectively display linear behavior with increasing number of cores.



(a) $b = 32$ and $k = 500$.



(b) $b = 128$ and $k = 151$.

Figure 12: Strong scaling measurements of the matrix-multivector multiplication in Eq. (57). The blue line results from a linear regression using the measured timings.

5 CONCLUSION AND FUTURE DEVELOPMENTS

The goal of this thesis was to compute the eigenpairs of a Hessian matrix $\mathbf{H}$ in a given interior interval of the spectrum close to the Boson peak regime by using a polynomial filtered eigensolver.

We were able to successfully run multiple interval-specific eigenpair computations close to the Boson peak regime involving Hessian matrices $\mathbf{H} \in \mathbb{R}^{768000 \times 768000}$ of systems with $N = 256000$ atoms. Our calculations concur with the previous results presented by Derlet et al [5].

With the current implementation of the PFO we managed to measure a maximal speedup of 68 with 128 cores, which demonstrates that our implementation scales up with increasing number of cores. Although the speedup behavior is satisfactory, our performance investigations show that there is still room for improvement. Since most of the work is done by the spMMM, in a future endeavor it would be advantageous to actually try different parallel distribution patterns and implement the spMMM based on these patterns. One example of such a distribution pattern is the two-dimensional block data distribution as shown in e.g. [4].

To actually further study the anomalous behavior around the Boson peak region of the spectrum, the partial diagonalization of Hessian matrices of systems with millions of atoms is required, and our current implementation should be able to handle computations with larger matrices. Nonetheless, with increasing matrix dimensions the number of eigenpairs to be computed within an interval increases also, and thus it would be favorable to further distribute the computational work on a given interval to different ranks based on smaller subintervals. A good spectrum slicing strategy is to divide the interval based on the distribution of the eigenvalues, which can be accomplished by computing estimations of the Density of States (DOS) [12, 13]. The methods for spectral density estimation shown in [13] have the common characteristics that they all use a stochastic and averaging technique to obtain an approximate DOS. A very similar technique is used by our parallel implementation of the estimation of the eigenvalue count [6]. Thus, extending our implementation with methods for the estimation of the spectral density could be accomplished in rather reasonable time.

Computations at the end of the spectrum of a Hessian matrix are also of some interest for the analysis of the general behavior of particles with increasing system sizes. In this case, an improved approximation of the extremal eigenvalues would be needed to actually deliver reliable results. The approach presented by Zhou and Li [20] provides tight upper bounds with low cost, and could be very suitable for the reliable computation of eigenpairs in the extremal regions of the spectrum.

Finally, detailed studies of the distribution of eigenvalues in intervals close to the Boson peak regime could be beneficial to actually tell more about the accuracy of the computed eigenmodes. Also, depending on the distribution of eigenvalues, more suitable strategies for the estimation of the spectral density could be employed, possibly resulting in an improved distribution of interval-specific computations for a given interval if needed.

APPENDIX A

A

A.1 USER MANUAL: BOSONPEAK UTILITY

In this section we are going to show the steps to first deploy, then compile, and finally run the BosonPeak Utility (BPU).

A.1.1 *Deploying the BosonPeak Utility on the Euler (ETH Zürich) and/or Ra (PSI) Cluster*

In a first step, we need to load all the necessary environment variables and program libraries. In the folder `BosonPeak/scripts/cluster_setup` two bash scripts can be found that actually accomplish this task. The script `setup-euler.sh` actually can be used to deploy the BPU on the Euler cluster, whereas `setup-psi.sh` can be used to do the same on the Ra cluster at PSI.

The scripts can be run with the following command (to be issued on the console):

```
> source setup-<euler|psi>.sh
```

where `<euler|psi>` can be replaced with either `euler` or `psi`, i.e., in this case either run `source setup-euler.sh` or `source setup-psi.sh`

The `source` command is important in this scenario, since it will actually load all the environment variables into the global variable space for the current session. Without the `source` command the environment variables won't be loaded correctly, and the compilation later will fail, since it depends on these variables.

A.1.2 *Code Compilation*

Once the steps in Section A.1.1 have been successfully done, we can move on to compiling the BPU source code.

First, change into the source code directory:

```
> cd BosonPeak/src/bosonpeak_utility
```

The folder contains the following files:

```
- BosonPeak_Utility.cpp
- CMakeLists.txt
- do-configure-euler.sh
- do-configure-psi.sh
```

The first file consists of the C++ code of the utility, which we are going to compile later. `CMakeLists.txt` contains the project configuration and will be used by CMake to actually build the project. Finally, the `do-configure-*.sh` files actually, when run, issue the `CMake` command and load the necessary Trilinos environment variables.

To actually build all the necessary files for the compilation of the source code we only need to run the corresponding bash script:

```
> ./do-configure-<euler|psi>.sh
```

After this, we can simply compile the source code using multiple cores with

```
> make -j<nr-cores>
```

where `<nr-cores>` can be replaced with the number of cores to use for compiling the code. If the compilation was successful, you can find the resulting binary in the `bin` folder of the same directory (or simply as an absolute path: `BosonPeak/src/bosonpeak_utility/bin/bosonpeak_utility`). This is the executable we are going to use later to actually run different computations.

A.1.3 *Configuring the BosonPeak Utility to Run Different Calculations*

The XML configuration file contains all the necessary configuration parameters to run different computations, be it an approximation of the extremal eigenvalues, the estimation of the eigenvalue count in an interval, or the computation of a specified number of eigenpairs in an interval. The BPU can then be launched with the absolute path to the XML configuration file as a parameter:

```
> cd BosonPeak/src/bosonpeak_utility
> ./bin/bosonpeak_utlity --cf=/path/to/config.xml
```

A.1.3.1 *Estimation of the Extremal Eigenvalues of* $\mathbf{A}$

For the estimation of the extremal eigenvalues λ_{min} and λ_{max} of the matrix $\mathbf{A}$ we first need to specify the path to the HDF5 file containing the matrix entries and which of the extremal eigenvalues we want to estimate. If we actually already know the extremal eigenvalues, we can simply specify them with the parameter list `Spectrum Boundaries Approximation` as shown in Listing 1.

Listing 1: Configuration of the extremal eigenvalues to be used for the transformation of the matrix as defined in Eq. (6).

```
<ParameterList name="BosonPeak Configuration">
    <Parameter name="Verbose" type="bool" value="1" />

    <ParameterList name="Matrix Import">
        <Parameter name="Path to Matrix HDF5 File" type="string" value="/
            cluster/matrix.h5"/>
    </ParameterList>
    <ParameterList name="Spectrum Boundaries Approximation">
        <Parameter name="LambdaMin" type="double" value="0.0" />
        <Parameter name="LambdaMax" type="double" value="3.0" />
    </ParameterList>
</ParameterList>
```

If we want to estimate both or just a single extremal eigenvalue, we can specify the `Spectrum Boundaries Approximation` parameter list (Listing 1) and leave out the eigenvalues we want to estimate. The utility will then estimate the extremal eigenvalues based on the missing specifications in the `Spectrum Boundaries Approximation` parameter list.

By default we run 10 iterations of the Lanczos algorithm for the estimation of the required extremal eigenvalues. If we wish to use a larger number of Lanczos iterations, define a larger convergence tolerance, or simply change the perturbation factor, we can do so by adding the `Eigensolver` parameter list to the `Spectrum Boundaries Approximation` parameter list as shown in Listing 2. In the example shown in Listing 2, `LambdaMax` is left out, thus letting the utility compute an approximation for $\lambda_{\max}$. Further, the Lanczos algorithm used for the approximation is run with the configuration parameters specified in the `Eigensolver` parameter list. In this scenario, BPU will print out the approximated extremal eigenvalues to the console once the computation is finished.

Listing 2: Extended configuration for the approximation of the extremal eigenvalues.

```
<ParameterList name="BosonPeak Configuration">
    <Parameter name="Verbose" type="bool" value="1" />

    <ParameterList name="Matrix Import">
        <Parameter name="Path to Matrix HDF5 File" type="string" value="/
            cluster/matrix.h5"/>
    </ParameterList>
    <ParameterList name="Spectrum Boundaries Approximation">
        <Parameter name="LambdaMin" type="double" value="0.0" />
        <ParameterList name="Eigensolver">
            <Parameter name="Block Size" type="int" value="1" />
            <Parameter name="Num Blocks" type="int" value="20" />
            <Parameter name="Maximum Restarts" type="int" value="10" />
            <Parameter name="Convergence Tolerance" type="double" value="1e-6"
                 />
            <Parameter name="Perturbation Factor" type="double" value="0.15" /
                >
        </ParameterList>
    </ParameterList>
</ParameterList>
```

A.1.3.2 *Estimation of the Eigenvalue Count in a Specified Interval*

In case the eigenvalue count for a given interval of interest is not known beforehand (as happens most of the times), BPU can be configured to run an eigenvalue count estimation. To configure different values for both parameters n_v and p as defined in Section 3.5, we can specify a `Eigenvalue Count Estimation` parameter list as shown in Listing 3. `Num Random Vectors` is used to specify n_v and `Degree` to specify p for the estimation of the eigenvalue count with Eq. (44). Additionally, for the estimation of the eigenvalue count it is required to specify the interval of interest by adding the `Interval of Interest` parameter list. With the configuration shown in Listing 3, BPU will print out the estimated eigenvalue count for the specified interval to the console once the computation is finished.

Listing 3: Configuration for the estimation of the eigenvalue count

```
<ParameterList name="BosonPeak Configuration">
    <Parameter name="Verbose" type="bool" value="1" />

    <ParameterList name="Matrix Import">
        <Parameter name="Path to Matrix HDF5 File" type="string" value="/
            cluster/matrix.h5"/>
    </ParameterList>

    <ParameterList name="Spectrum Boundaries Approximation">
        <Parameter name="LambdaMin" type="double" value="0.0" />
        <Parameter name="LambdaMax" type="double" value="3.0" />
    </ParameterList>

    <ParameterList name="Interval of Interest">
        <Parameter name="Interval Start" type="double" value="0.1" />
        <Parameter name="Interval End" type="double" value="1" />
    </ParameterList>

    <ParameterList name="Eigenvalue Count Estimation">
        <Parameter name="Num Random Vectors" type="int" value="40"/>
        <Parameter name="Degree" type="int" value="100"/>
    </ParameterList>
</ParameterList>
```

A.1.3.3 *Eigensolver Configuration*

Once we have estimations for the extremal eigenvalues and an estimated number of eigenvalues in the interval of interest, we are ready to run eigenpair computations in the specific interval. An example of such a configuration is shown in Listing 4. Once the computation has finished, files containing the computed eigenvalues, eigenvectors and norms of all residuals will be written to the location specified by the `Path to Output Folder` parameter. The files are all stored in the MM file format.

Listing 4: Complete configuration for the computation of eigenpairs in a specific interval of the spectrum.

```
<ParameterList name="BosonPeak Configuration">
    <Parameter name="Verbose" type="bool" value="1" />

    <ParameterList name="Matrix Import">
        <Parameter name="Path to Matrix HDF5 File" type="string" value="/
            cluster/matrix.h5"/>
    </ParameterList>

    <ParameterList name="Spectrum Boundaries Approximation">
        <Parameter name="LambdaMin" type="double" value="0.0" />
        <Parameter name="LambdaMax" type="double" value="1919.756431988085296"
             />
    </ParameterList>

    <ParameterList name="Interval of Interest">
        <Parameter name="Interval Start" type="double" value="0.0" />
        <Parameter name="Interval End" type="double" value="3.0" />
    </ParameterList>

    <ParameterList name="Filter Polynomial Computation">
        <Parameter name="Min Degree" type="int" value="2"/>
        <Parameter name="Max Degree" type="int" value="1000"/>
        <Parameter name="Smoother" type="string" value="Jackson"/>
        <Parameter name="Threshold End Interval" type="double" value="0.6"/>
        <Parameter name="Threshold Interior Interval" type="double" value="0.9
            "/>
    </ParameterList>

    <ParameterList name="Interval-Specific Eigensolver">
        <Parameter name="Block Size" type="int" value="4" />
        <Parameter name="Maximum Restarts" type="int" value="500" />
        <Parameter name="Num Eigenvalues" type="int" value="300" />
        <Parameter name="Num Blocks" type="int" value="225" />
        <Parameter name="Convergence Tolerance" type="double" value="1e-8" />
        <Parameter name="Path to Output Folder" type="string" value="/cluster/
            output/" />
    </ParameterList>
</ParameterList>
```

A.1.4 *Building and Running the BosonPeak Unit Tests*

The `BosonPeak` source code comes with a test suite, which is based on data-driven testing and has been implemented to support a test-driven development approach.

To actually run the test suite, we first need to generate the test data. For this, a test data generator written in MATLAB is provided. The test data generator will create all the necessary test data files needed to run all the unit and integration tests. To accomplish this, simply point MATLAB to the `BosonPeak/test/testdata_generators/matlab` directory and run the `TestDataGenerator_Main.m` script from there. Once all the test data has been successfully generated, change into the `BosonPeak/test` folder, run the corresponding setup file and compile everything with `make`:

```
> cd BosonPeak/test
> ./do-configure-<euler|psi>.sh
> make -j<nr-cores>
```

After a successful build, simply run all tests with the following command:

```
> ./bin/bosonpeak_tests
```

A.2 USING THE POLYNOMIAL FILTER MATLAB IMPLEMENTATION

A complete MATLAB implementation of the least-squares polynomial filter as described in Section 2.1 is provided in the `BosonPeak/src/matlab` folder. This code has been mainly implemented for unit and integration testing purposes and is thus extensively used by the test data generator mentioned in Section A.1.4. Nonetheless, the provided implementation can also be used to run computations on smaller matrices or for visualization purposes. Examples of such computations are shown in Listings 5 and 6.

Listing 5: Example code showing the usage of the MATLAB implementation of the polynomial filter for the computation of eigenvalues in a specific interval of interest. We are using a simple diagonal matrix $\mathbf{A}$ with spectrum $\sigma = \{1, 2, 3, 4, \ldots, 20\}$ and compute all the eigenvalues λ_i in the interval $[11.5, 14.2]$. Before running this code it is important to first load all the necessary environment variables by running the `setup-<euler|psi>.sh` script as described in Section A.1.1.

```
BOSONPEAK_MATLAB_SOURCE_DIR = 'BOSONPEAK_MATLAB_SOURCE_DIR';
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/'));
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/constants'));
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/types'));

% Specify some example spectrum boundaries
lambda_min = 1;
lambda_max = 20;

% Define a simple diagonal matrix
matrix_A = diag((lambda_min:lambda_max));

% Polynomial filter configuration
min_degree = 2;
max_degree = 300;
smoother = BosonPeak_SmootherType.JACKSON;
threshold_interior_interval = 0.6;
threshold_end_interval = 0.3;
% We want to compute all the eigenpairs in the
% interval of interest [11.5, 17.2]
interval_start = 11.5;
interval_end = 14.2;

% Compute the polynomial filter
[gamma_, bar_value, optimal_degree, expansion_coefficients] =
    computeFilterPolynomial(lambda_min, lambda_max, interval_start,
    interval_end, min_degree, max_degree, smoother,
    threshold_interior_interval, threshold_end_interval);

fprintf('\n=========================================\n');
fprintf('Configuration of polynomial filter\n');
fprintf('-----------------------------------------\n');
fprintf('gamma_  = %.15f\n', gamma_);
fprintf('bar_value = %.15f\n', bar_value);
fprintf('optimal_degree = %d\n', optimal_degree);
fprintf('=========================================\n');

% Compute the scaling factors and transform the matrix H
[c,d] = computeScalingFactors(lambda_min, lambda_max);
matrix_A_hat = transformMatrix(matrix_A, c, d);

% Compute the matrix rho_k(A_hat) by using the
% Chebyshev three-term recurrence and the computed
% polynomial filter parameters
T_jm2 = eye(size(matrix_A_hat));
T_jm1 = matrix_A_hat;
rho_matrix_A_hat_0 = expansion_coefficients(1) * T_jm2;
```

```
rho_matrix_A_hat = zeros(size(matrix_A));

if(optimal_degree == 0)
    rho_matrix_A_hat = rho_matrix_A_hat_0;
elseif (optimal_degree == 1)
    rho_matrix_A_hat = rho_matrix_A_hat_0 + expansion_coefficients(2) * T_jm1;
else
    rho_matrix_A_hat = rho_matrix_A_hat_0 + expansion_coefficients(2) * T_jm1;
    for j=3:optimal_degree+1
        T_j = 2 * matrix_A_hat * T_jm1 - T_jm2;
        T_jm2 = T_jm1;
        T_jm1 = T_j;
        rho_matrix_A_hat = rho_matrix_A_hat + expansion_coefficients(j) * T_j;
    end
end

% Compute all eigenvalues of rho_k(A_hat),
% since it's only a small matrix
[eigvecs_rho, eigvals_matrix_rho] = eig(full(rho_matrix_A_hat));
eigvals_rho = diag(eigvals_matrix_rho);

n_eigvecs = size(eigvecs_rho, 2);

fprintf('\n========================================\n');
fprintf('Eigenvalues in interval of interest\n');
fprintf('----------------------------------------\n');
for i=1:n_eigvecs
    % Eigenvalues smaller than the bar value are discarded
    if(eigvals_rho(i) >= bar_value)
        % Compute Rayleigh quotient
        eigval = eigvecs_rho(:,i)' * matrix_A * eigvecs_rho(:,i);
        if((eigval >= interval_start) && (eigval <= interval_end))
            fprintf('Eigval %.3f is in [%.3f, %.3f]\n', eigval, interval_start
                , interval_end);
        end
    end
end
fprintf('========================================\n');

% Output on MATLAB console:

%     =========================================
%     Configuration of polynomial filter
%     -----------------------------------------
%     gamma_ = 0.250076644878696
%     bar_value = 0.599538469253713
%     optimal_degree = 20
%     =========================================

%     =========================================
%     Eigenvalues in interval of interest
%     -----------------------------------------
%     Eigval 14.000 is in [11.500, 14.200]
%     Eigval 12.000 is in [11.500, 14.200]
%     Eigval 13.000 is in [11.500, 14.200]
%     =========================================
```

Listing 6: Example code showing the usage of the MATLAB implementation of the polynomial filter for the visualization of the resulting filter. Fig. 13 shows the resulting plot from this code. Before running this code it is important to first load all the necessary environment variables by running the `setup-<euler|psi>.sh` script as described in Section A.1.1.

```
BOSONPEAK_MATLAB_SOURCE_DIR = 'BOSONPEAK_MATLAB_SOURCE_DIR';
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/'));
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/constants'));
addpath(strcat(getenv(BOSONPEAK_MATLAB_SOURCE_DIR), '/types'));

% Specify some example spectrum boundaries
lambda_min = 1;
lambda_max = 20;

% Polynomial filter configuration
min_degree = 2;
max_degree = 300;
smoother = BosonPeak_SmootherType.JACKSON;
threshold_interior_interval = 0.6;
threshold_end_interval = 0.3;

% We want to compute all the eigenpairs in the
% interval of interest [11.5, 17.2]
interval_start = 11.5;
interval_end = 14.2;

% Compute the polynomial filter
[gamma_, bar_value, optimal_degree, expansion_coefficients] =
    computeFilterPolynomial(lambda_min, lambda_max, interval_start,
    interval_end, min_degree, max_degree, smoother,
    threshold_interior_interval, threshold_end_interval);

% Plotting setup
n_x = 100;
% Define the data points where the polynomial filter should be
% evaluated
x = linspace(BosonPeak_Constants.REFERENCE_INTERVAL_LEFT_ENDPOINT_VALUE,
    BosonPeak_Constants.REFERENCE_INTERVAL_RIGHT_ENDPOINT_VALUE, n_x);

% Vector containing the values of the polynomial filter evaluated at
% each point contained in x
rho_x = zeros(1, n_x);
for i=1:n_x
    rho_x(i) = evalRho(x(i), optimal_degree, expansion_coefficients);
end

% Compute the scaling factors and transform the boundaries of the
% interval of interest
[c,d] = computeScalingFactors(lambda_min, lambda_max);
```

```
interval_start_hat = (interval_start - c) / d;
interval_end_hat = (interval_end - c) / d;

polynomial_filter_plot = figure;
set(gca,'fontsize',16)
hold on;

endpoint_offset = 0.2;
min_rho_x = 0;
max_rho_x = 1;
xlim([BosonPeak_Constants.REFERENCE_INTERVAL_LEFT_ENDPOINT_VALUE -
    endpoint_offset, BosonPeak_Constants.
    REFERENCE_INTERVAL_RIGHT_ENDPOINT_VALUE + endpoint_offset]);
ylim([min_rho_x - endpoint_offset, max_rho_x + endpoint_offset]);

xlabel('$\hat\lambda$','Interpreter','LaTex');
ylabel('$\rho(\hat\lambda)$','Interpreter','LaTex');

line([interval_start_hat interval_start_hat], [min_rho_x, max_rho_x], 'Color',
    'red','LineStyle','--', 'LineWidth', 2);
line([interval_end_hat interval_end_hat], [min_rho_x, max_rho_x], 'Color','red
    ','LineStyle','--', 'LineWidth', 2);
plot(x, rho_x, '-', 'Color', 'blue', 'LineWidth', 2);
```

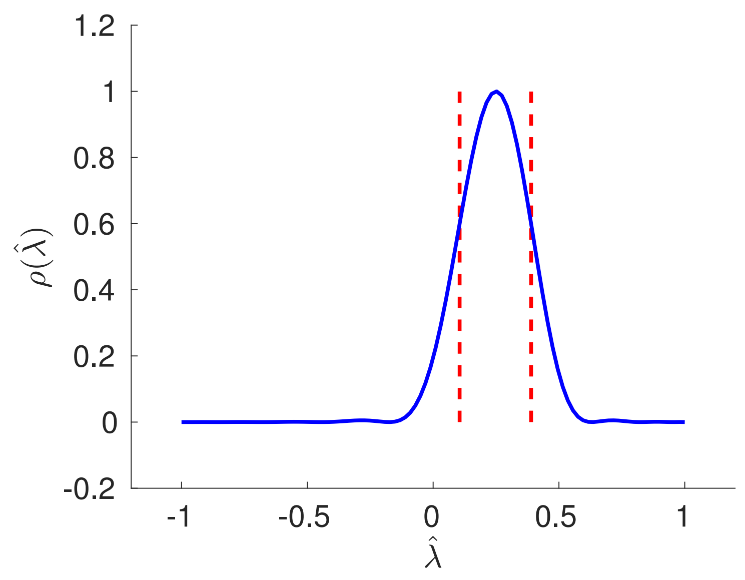


Figure 13: Polynomial filter $\hat\rho_k$ (Eq. (5)) computed and plotted by using the MATLAB code displayed in Listing 6.

A.3 CONVERTING MM FILES STORING LARGE SYMMETRIC MATRICES TO TRILINOS COMPATIBLE HDF5 FILES

As part of this project, the Hessian matrices were provided in the symmetric MM format. To actually import such a matrix via BPU we first have to convert the

MM file to a Trilinos compatible HDF5 file. This is accomplished by using the `convert_mm_matrix_to_hdf5.py` script located in the `BosonPeak/scripts/matrix_converter` directory. Just specify the path to the source file containing the matrix entries in the symmetric MM format and the path to the destination HDF5 file:

```
> cd BosonPeak/scripts/matrix_converter
> python convert_mm_matrix_to_hdf5.py /source/matrix.mm /destination/matrix.
    hdf5
```

Since we want to be able to convert very large files (up to hundreds of Gigabytes), we implemented a buffered conversion approach to avoid overflowing the main memory, meaning that we only load and convert small data chunks at a time. The size of the data chunks can be specified by supplying the Python script with a third parameter, namely the size of the data chunks in Bytes:

```
> python convert_mm_matrix_to_hdf5.py /source/matrix.mm /destination/matrix.
    hdf5 1000000
```

In the above example, the script will be loading and converting 1 Megabyte of data at a time.

BIBLIOGRAPHY

[1] Haim Avron and Sivan Toledo. "Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix." In: *Journal of the ACM* 58.2 (2011), p. 8.

[2] Costas Bekas, Effrosyni Kokiopoulou, and Yousef Saad. "An estimator for the diagonal of a matrix." In: *Applied numerical mathematics* 57.11 (2007), pp. 1214–1229.

[3] YM Beltukov, C Fusco, DA Parshin, and A Tanguy. "Boson peak and Ioffe-Regel criterion in amorphous siliconlike materials: The effect of bond directionality." In: *Physical Review E* 93.2 (2016), p. 023006.

[4] Aydin Buluç and John R Gilbert. "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments." In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C170–C191.

[5] PM Derlet, R Maaß, and JF Löffler. "The Boson peak of model glass systems and its relation to atomic structure." In: *The European Physical Journal B-Condensed Matter and Complex Systems* 85.5 (2012), pp. 1–20.

[6] Edoardo Di Napoli, Eric Polizzi, and Yousef Saad. "Efficient estimation of eigenvalue counts in an interval." In: *Numerical Linear Algebra with Applications* 23.4 (2016), pp. 674–692.

[7] Haw-ren Fang and Yousef Saad. "A filtered Lanczos procedure for extreme and interior eigenvalue problems." In: *SIAM Journal on Scientific Computing* 34.4 (2012), A2220–A2246.

[8] Michael F Hutchinson. "A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines." In: *Communications in Statistics-Simulation and Computation* 19.2 (1990), pp. 433–450.

[9] Laurent O Jay, Hanchul Kim, Yousef Saad, and James R Chelikowsky. "Electronic structure calculations for plane-wave codes without diagonalization." In: *Computer physics communications* 118.1 (1999), pp. 21–30.

[10] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[11] Cornelius Lanczos. *Applied analysis*. Dover Publications, 1988.

[12] Ruipeng Li, Yuanzhe Xi, Eugene Vecharynski, Chao Yang, and Yousef Saad. "A Thick-Restart Lanczos algorithm with polynomial filtering for Hermitian eigenvalue problems." In: *SIAM Journal on Scientific Computing* 38.4 (2016), A2512–A2534.

[13] Lin Lin, Yousef Saad, and Chao Yang. "Approximating spectral densities of large matrices." In: *SIAM Review* 58.1 (2016), pp. 34–65.

[14] Theodore J Rivlin. *An introduction to the approximation of functions*. Blaisdell Publishing Company, 1969.

[15] Yousef Saad. *Numerical methods for large eigenvalue problems*. Manchester University Press, 1992.

[16] Yousef Saad, Ruipeng Li, Yuanzhe Xi, Luke Erlandson, Eugene Vecharinsky, and Chao Yang. *EVSL: EigenValues Slicing Library*. `http://www-users.cs.umn.edu/~saad/software/EVSL/`. 2016 – 2017.

[17] Gilbert W Stewart. "A Krylov–Schur algorithm for large eigenproblems." In: *SIAM Journal on Matrix Analysis and Applications* 23.3 (2002), pp. 601–614.

[18] Kesheng Wu and Horst Simon. "Thick-restart Lanczos method for large symmetric eigenvalue problems." In: *SIAM Journal on Matrix Analysis and Applications* 22.2 (2000), pp. 602–616.

[19] Ichitaro Yamazaki, Zhaojun Bai, Horst Simon, Lin-Wang Wang, and Kesheng Wu. "Adaptive projection subspace dimension for the thick-restart Lanczos method." In: *ACM Transactions on Mathematical Software (TOMS)* 37.3 (2010), p. 27.

[20] Yunkai Zhou and Ren-Cang Li. "Bounding the spectrum of large Hermitian matrices." In: *Linear Algebra and its Applications* 435.3 (2011), pp. 480–493.