HAMLS Eigensolver

Eigensovler Module for HLIBpro's Hierarchical Matrices and for Elliptic PDE Eigenvalue Problems

Peter Gerds

1. Introduction

This software implements the so called $\mathcal{H}$-AMLS method which has been introduced in (Gerds 2017; Gerds and Grasedyck 2015). $\mathcal{H}$-AMLS is a very efficient method for the solution of elliptic PDE eigenvalue problems. The method is combining a recursive version of the automated multi-level substructuring (short AMLS) method – which is a domain decomposition technique for the solution of elliptic PDE eigenvalue problems – with the concept of hierarchical matrices (short $\mathcal{H}$-matrices). $\mathcal{H}$-AMLS allows the efficient solution of large scale eigenvalue problems with millions degrees of freedom on today's workstations. In particular, the method is well suited for problems posed in three dimensions and allows the computation of a large amount of eigenpair approximations in optimal complexity.

Besides the implementation of the $\mathcal{H}$-AMLS method, this software provides classical iterative eigensolvers combined with the fast $\mathcal{H}$-matrix arithmetic. The software allows to solve the generalized eigenvalue problem $Kx = \lambda Mx$ where K and M are symmetric sparse or $\mathcal{H}$-matrices, and where M is positive definite.

This software requires an installed HLIBpro library, and in order to use all provided iterative eigensolvers an installed ARPACK library is required as well. As this software is based on the HLIBpro library it is recommended to be familiar with the basic operation of HLIBpro.

 For questions or needed support contact peter_gerds@gmx.de.

2. Installation and Integration into own Project

To install the HAMLS software a working distribution of the HLIBpro is needed (this software has been installed using HLIBpro v2.7). HLIBpro is a software library implementing algorithms for hierarchical matrices and it is freely available for academic purposes. See HLIBpro's webpage for download and installation information. Recommended, but not absolutely necessary (see the remark below), is also an installed ARPACK library. Furthermore, like HLIBpro, the HAMLS software requires for the internal make system the software library SCons. However, to include HAMLS into your own projects you do not have to use SCons.

After downloading and extracting the HAMLS package, adjust the following line

```
HLIBPRO_PATH = '/home/user/hlibpro/main'
```

in the file **SConstruct** where you specify the location of your installed HLIBpro library. Then, open your terminal and execute the command

```
$scons
```

within the HAMLS installation directory. This starts the installation of the HAMLS library and the example **eigensolver.cc** contained in the folder **example/** gets compiled. Own programs can be added in the example folder as well. For compilation of your programs the **SConstruct** file has to be adjusted by adopting the instructions – which are applied for the **eigensolver.cc** example – correspondingly for your program. See https://www.scons.org/ for further instructions.

You can integrate the HAMLS library as well to your own project. For this purpose we make use of the **hlib-config** shell script which is contained in your HLIBpro installation directory. Executing

```
$<path_to_HLIBpro>/bin/hlib-config --cflags
```

and

```
$<path_to_HLIBpro>/bin/hlib-config --lflags
```

in the terminal prints out HLIBpro's compilation and linking options. Note that, to make this work, the installation path of HLIBpro has to be specified in the file **bin/hlib-config**. So, open in your HLIBpro folder the file **bin/hlib-config** and adjust the path to the HLIBpro installation in the definition of *prefix* by the corresponding path. For more details see the installation instruction (https://www.hlibpro.com/doc/) of your HLIBpro version. Using the **hlib-config** script the compilation (assuming a Unix system) of the program, for example, **eigensolver.cc** can be performed as follows

```
$g++ -o eigensolver example/eigensolver.cc `<path_to_HLIBpro>/bin/hlib-config --cflags --lflags` -I/<pa
```

Remark ARPACK Library

Note that the installation instruction described above assumes an installed ARPACK library, which is needed to use a special iterative eigensolver that is provided by the HAMLS module. If an installed ARPACK library cannot be provided, adjust the **SConstruct** file by removing the corresponding build instructions made for the files **TEigenArpack.cc** and **TEigenArpack.hh**.

3. Problem Description and some Theory

This eigensolver module focuses on the efficient solution of the continuous eigenvalue problem

$$\begin{cases} Lu = \lambda u & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

where Ω is a bounded d-dimensional domain (d=2,3) with a Lipschitz boundary $\partial\Omega$ and L is a uniformly elliptic second order partial differential operator in divergency form

$$Lu = -\mathrm{div}\big(A\nabla u\big) + cu = -\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}\left(a_{ij}\frac{\partial}{\partial x_j}u\right) + cu$$

with $L^\infty(\Omega)$-functions a_{ij} and c. Furthermore, it holds $A := (a_{ij})_{i,j=1}^d$ and $c \geq 0$, and in the equation above the eigenvalues $\lambda \in \mathbb{C}$ and the associated eigenfunctions $u \neq 0$ are sought. To recap, the operator L is called uniformly elliptic if for all $x \in \Omega$ the matrix $A(x) := \big(a_{ij}(x)\big)_{i,j=1}^d$ is symmetric and the eigenvalues of $A(x)$ are uniformly bounded from below by a positive constant. This form of partial differential equation (short PDE) eigenvalue problems arises in many fields of physical and engineering application, and their efficient solution is of high importance, especially for three-dimensional real-world problems which are typically costly to solve.

Weak Formulation

To solve the elliptic PDE eigenvalue problem above the corresponding weak formulation

$$\begin{cases} \text{find } (\lambda, u) \in \mathbb{R} \times H_0^1(\Omega) \setminus \{0\} \text{ such that} \\ a(u,v) \;=\; \lambda\,(u,v)_0 \qquad \forall\, v \in H_0^1(\Omega) \end{cases}$$

is derived where $a(u,v) := \int_\Omega \nabla u^T A \nabla v + cuv \;\mathrm{d}x$ is a symmetric elliptic bilinear form and $(u,v)_0 := \int_\Omega uv \;\mathrm{d}x$ is the inner product of $L^2(\Omega)$. Using the Fredholm-Riesz-Schauder theory it can be shown (see, e.g., Hackbusch 1992) that the weak formulation of the elliptic PDE eigenvalue problem possesses a countable family of eigensolutions

$$\big(\lambda_j, u_j\big)_{j=1}^{\infty} \in \mathbb{R}_{>0} \times H_0^1(\Omega) \qquad \text{with } \lambda_j \le \lambda_{j+1},$$

where all eigenvalues are positive real and where the eigensolutions can be arranged according to the size of the eigenvalues.

Discretisation

In the most cases, the eigensolutions of the weak formulation above cannot be computed analytically, they have to be computed numerically. In practice typically the finite element discretisation is used for this purpose: Using a conform finite element space $V_h \subset H_0^1(\Omega)$ of mesh width h, spanned by its nodal basis functions $\big(\varphi_i^{(h)}\big)_{i=1}^{N_h}$ and with dimension N_h, the weak formulation of the elliptic PDE eigenvalue problem is discretised by

$$\begin{cases} \text{find } (\lambda^{(h)}, x^{(h)}) \in \mathbb{R} \times \mathbb{R}^{N_h} \setminus \{0\} \text{ with} \\ K^{(h)}\, x^{(h)} \;=\; \lambda^{(h)}\, M^{(h)}\, x^{(h)} \end{cases}$$

where the stiffness and mass matrix

$$K^{(h)} := \Big(a(\varphi_j^{(h)}, \varphi_i^{(h)})\Big)_{i,j=1}^{N_h} \in \mathbb{R}^{N_h \times N_h} \quad \text{and} \quad M^{(h)} := \Big((\varphi_j^{(h)}, \varphi_i^{(h)})_0\Big)_{i,j=1}^{N_h} \in \mathbb{R}^{N_h \times N_h}$$

are both sparse, symmetric and positive definite. The eigenvalues of the discrete problem are positive real, and the corresponding eigenpairs are given in arranged order by

$$\big(\lambda_j^{(h)}, x_j^{(h)}\big)_{j=1}^{N_h} \in \mathbb{R}_{>0} \times \mathbb{R}^{N_h} \setminus \{0\} \qquad \text{with } \lambda_j^{(h)} \le \lambda_{j+1}^{(h)}.$$

The eigensolutions $(\lambda_j^{(h)}, x_j^{(h)})$ of the discrete problem $(K^{(h)}, M^{(h)})$ provide approximations of the sought continuous eigensolutions (λ_j, u_j) of the weak formulation when $h \to 0$. More specifically, the following approximation result (see, e.g., Hackbusch 1992) holds:

- The discrete eigenvalues are approximating the continuous ones, i.e., it holds

$$\lambda_j^{(h)} \xrightarrow{\;h \to 0\;} \lambda_j \qquad \text{for all } j \in \mathbb{N}.$$

- Consider for some fixed $j \in \mathbb{N}$ the discrete eigenfunction $u_j^{(h)} := \sum_{i=1}^{N_h} (x_j^{(h)})_i \;\varphi_i^{(h)} \in V_h$ associated to the eigenvector $x_j^{(h)} = \big((x_j^{(h)})_1, \ldots, (x_j^{(h)})_{N_h}\big)^T \in \mathbb{R}^{N_h}$. Assuming that it holds $(u_j^{(h)}, u_j^{(h)})_0 = 1$, then there exists for each sequence $\big(u_j^{(h_k)}\big)_{k \in \mathbb{N}}$ with $h_k \to 0$ for $k \to \infty$ a subsequence which converges in $H_0^1(\Omega)$ to an eigenfunction $\widetilde{u}_j \in E(\lambda_j)$, where $E(\lambda_j) \subset H_0^1(\Omega)$ is the eigenspace of eigenvalue λ_j which is defined by

$$E(\lambda_j) := \operatorname{span}\Big\{ u \in H_0^1(\Omega) \;:\; a(u,v) = \lambda_j (u,v)_0 \quad \forall\, v \in H_0^1(\Omega) \Big\}.$$

Number of Well Approximable Eigensolutions

Using the finite element discretisation for the solution of the elliptic PDE eigenvalue problem, it has to be noted that only the smaller eigenvalues λ_j and their corresponding eigenfunctions u_j can be well approximated by the finite element space V_h because the approximation error increases with increasing size of the eigenvalue. In particular, the eigenvalues λ_j have to be small enough so that necessary conditions on the discretisation mesh width h of the finite element space get fulfilled, and corresponding convergence rates become valid, see (Banjai, Börm, and Sauter 2008; Sauter 2010) for details. Although knowing that only the discrete eigensolutions $(\lambda_j^{(h)}, u_j^{(h)})$ associated to the smallest $\lambda_j^{(h)}$ provide good approximations for the sought continuous eigensolutions, it is hard to answer how many continuous eigensolutions (λ_j, u_j) are well approximable by a given finite element space V_h. Based on certain smoothness assumptions on the underlying data a first asymptotic quantification on the number of well approximable eigensolutions could be derived in (Gerds 2017). The term "well approximable" is understood in (Gerds 2017) in the sense that certain convergence rates become valid for the eigenvalue/eigenfunction approximation. For example, it is shown in (Gerds 2017) that for three-dimensional problems, under certain smoothness assumptions on the data, only the first $C_1 N_h^{2/5}$ eigenvalues and only the first $C_2 N_h^{1/4}$ eigenfunctions can be well approximated by the finite element discretisation (with constants $C_1, C_2 > 0$) when using the finite element spaces V_h of piece-wise affine functions with uniform mesh refinement.

Correspondingly, in the following we are only interested in computing a portion of the smallest eigenpairs of the discrete problem $(K^{(h)}, M^{(h)})$, e.g., computing not more than the first

$$n_{\text{es}} = C N_h^{\beta} \in \mathbb{N}$$

eigenpairs with some constant $C > 0$ and $n_{\text{es}} \leq N_h$, and where the exponent $\beta \in (0, 1)$ depends on the dimension d of the problem and on the polynomial degree of the underlying finite element space (see Gerds 2017 for details). The computation of the remaining eigenpairs of $(K^{(h)}, M^{(h)})$ associated to larger eigenvalues is not reasonable because typically they do not provide useful approximations of the continuous eigensolutions.

Solving the Problem using a Classical Approach with $\mathcal{H}$-matrix Arithmetic

The solution of the discrete eigenvalue problem $(K^{(h)}, M^{(h)})$ is typically performed by a classical approach, i.e., by some iterative algebraic eigensolver such as the Lanczos method (see, e.g., Bai et al. 2000; Grimes, Lewis, and Simon 1994). Classical iterative approaches are particularly well suited if the number of sought eigensolutions n_{es} is rather small, e.g., if $n_{\text{es}} = 10$. Beside matrix-vector multiplications classical iterative approaches (like the Lanczos method) involve typically the costly computation of a preconditioner for the matrix $K^{(h)}$ in order to solve the problem $(K^{(h)}, M^{(h)})$. Using the fast $\mathcal{H}$-matrix arithmetic such a precondtioner can be computed with almost linear complexity.

The HAMLS eigensolver module provides two implementations that combine a classical approach with the fast $\mathcal{H}$-matrix arithmetic so that the eigenvalue problem $(K^{(h)}, M^{(h)})$ can be solved very efficiently, especially, when n_{es} is rather small:

- The $\mathcal{H}$-ARPACK eigensolver is based on the well-proven and well-tested FORTRAN 77 library *ARPACK* (Lehoucq, Yang, and Sorensen 1998) and combines it with the fast $\mathcal{H}$-matrix arithmetic. The ARPACK library implements the so-called *Implicitly Restarted Lanczos Method* which is a combination of the Lanczos process with the implicitly shifted QR technique. The $\mathcal{H}$-ARPACK eigensolver is implemented in the class `TEigenArpack`, and in total it is fast and very robust. To apply this eigensolver, however, the user has to provide an installation of the ARPACK library.
- The class `TEigenArnoldi` provides a basic implementation of the Arnoldi method in combination with the fast $\mathcal{H}$-matrix arithmetic. This eigensolver is fast, however, not as robust as the $\mathcal{H}$-ARPACK eigensolver and computes in some cases eigenpair approximations with a lower accuracy.

Both eigensolvers have been parallelized for shared memory systems and can be executed with multiple threads. Furthermore, both solver support the solution of shifted problems, so that eigenpairs of $(K^{(h)}, M^{(h)})$ can be computed whose eigenvalues are located around a given shift $\mu \geq 0$.

Note that the eigensolvers implemented in `TEigenArpack` and `TEigenArnoldi` compute (in the most cases) numerically exact eigenpairs of the discrete problem $(K^{(h)}, M^{(h)})$.

Solving the Problem using the $\mathcal{H}$-AMLS method

To solve the discretised elliptic PDE eigenvalue problem and to compute many (e.g., all well approximable) eigenvalues/eigenfunctions, a new method called $\mathcal{H}$-AMLS has been introduced in (Gerds 2017; Gerds and Grasedyck 2015) which combines a recursive version of the *automated multi-level substructuring* (short AMLS) with the concept of $\mathcal{H}$-matrices.

The AMLS method (see, e.g., Bennighof 1993; Bennighof and Lehoucq 2004) itself is a domain decomposition technique for the solution of elliptic PDE eigenvalue problems where, after some transformation, a reduced eigenvalue problem is derived whose eigensolutions deliver approximations of the sought eigensolutions of the original problem. Whereas the classical AMLS method is very efficient for PDE eigenvalue problems posed in two dimensions, it is getting very expensive for three-dimensional problems, due to the fact that it computes the reduced eigenvalue problem via dense matrix operations. Using the fast $\mathcal{H}$-matrix arithmetic for the computation of the reduced eigenvalue problem, this problem is resolved. Furthermore, applying AMLS recursively the size of the reduced eigenvalue problem is bounded which significantly reduces the computational costs further. Altogether this leads to the new $\mathcal{H}$-AMLS method which has the following features (see Gerds 2017 for details):

- $\mathcal{H}$-AMLS is well suited for the solution of problems posed in three dimensions.
- $\mathcal{H}$-AMLS enables the computation of a large amount of eigenpair approximations in optimal complexity.
- $\mathcal{H}$-AMLS outperforms classical AMLS.
- $\mathcal{H}$-AMLS outperforms classical approaches when are larger number of eigenpair approximations is sought.
- $\mathcal{H}$-AMLS has been parallelized for shared memory systems and can be used with multiple threads.

The $\mathcal{H}$-AMLS eigensolver is implemented in the class `THAMLS`.

Note that when AMLS or $\mathcal{H}$-AMLS are applied to a discrete eigenvalue problem it computes only approximations of the discrete eigenpairs whereas classical approaches, like the Lanczos method, compute these eigenpairs almost numerically exact. This seems to be disadvantageous, but since in our setting discrete eigenvalue problems result always from a finite element discretisation of a continuous problem, all eigenpairs of the discrete problem are related to a discretisation error. Hence, as long as the projection/approximation error caused by $\mathcal{H}$-AMLS is of the same order as the discretisation error, the computed eigenpair approximations of $\mathcal{H}$-AMLS are of comparable quality as the eigenpairs computed by some classical approach. The parameter setting used in $\mathcal{H}$-AMLS to control the projection/approximation error – in order to compute eigenpair approximations with satisfactional accuracy – is described in the following. Details can be found in (Gerds 2017; Gerds and Grasedyck 2015).

4. Importing Input Data

The program starts with including the needed HAMLS and HLIBpro header files (the HLIBpro types for real and complex values are imported as well), and with the standard initialization of the HLIBpro.

```
#include <iostream>
#include <hlib.hh>

#include "hamls/TEigenArnoldi.hh"
#include "hamls/TEigenArpack.hh"
#include "hamls/THAMLS.hh"

using namespace HLIB;
```

```
using namespace HAMLS;
using namespace std;

using  real_t    = HLIB::real;
using  complex_t = HLIB::complex;

int main ( int argc, char ** argv )
{
  try
  {
    INIT();
```

Using this initialization the program is executed in parallel with all available threads by default. After initialization, the data of the finite element discretisation of the elliptic PDE eigenvalue problem has to made accessible. In detail the sparse matrices $K^{(h)}$ and $M^{(h)}$ are needed, and geometric representative points $(\xi_i^{(h)})_{i=1}^{N_h}$ that are associated to the triangulation used in the underlying finite element space V_h. These representatives only have to fulfil

$$\xi_i^{(h)} \in \operatorname{supp}\big(\varphi_i^{(h)}\big) \quad \text{for } i = 1, \ldots, N_h$$

and can be chosen, for example, as the nodal points of the finite element basis functions $\varphi_i^{(h)}$ or as the geometric centres of the corresponding supports. In this particular example, this data is imported from memory as follows

```
    THLibMatrixIO  matrix_io;

    const string  path_to_data = "./";

    unique_ptr< TMatrix >   K_temp( matrix_io.read( path_to_data + "input_K" ) );
    unique_ptr< TMatrix >   M_temp( matrix_io.read(  path_to_data +  "input_M" ) );

    unique_ptr< TSparseMatrix >  K( ptrcast( K_temp.release(), TSparseMatrix ) );
    unique_ptr< TSparseMatrix >  M( ptrcast( M_temp.release(), TSparseMatrix ) );

    TAutoCoordIO    coord_io;
    unique_ptr< TCoordinate >  coord( coord_io.read (  path_to_data + "input_coord") );
```

The data files **input_K** and **input_M** have to be the sparse matrices $K^{(h)}$ and $M^{(h)}$ saved in the corresponding HLIBpro data format, and the file **input_coord** contains the geometric representatives $(\xi_i^{(h)})_{i=1}^{N_h}$ saved as well in the corresponding HLIBpro data format. In this example, the three files are stored in the working directory where the program is executed. For details about HLIBpro's data formats see the correspodning documentation (https://www.hlibpro.com/doc/).

5. Construction of the $\mathcal{H}$-matrix Representation

In the next step of the program the $\mathcal{H}$-matrix representations of the sparse matrices $K^{(h)}$ and $M^{(h)}$ have to be derived. For this purpose, based on the geometric information contained in `coord`, the associated cluster tree and block cluster tree is constructed. The construction of the cluster tree is performed as follows

```
    const uint  nmin = 40;

    // cluster partitioning is based on an adaptive bisection
    TGeomBSPPartStrat  part_strat( adaptive_split_axis );

    TBSPNDCTBuilder  ct_builder( M.get(), & part_strat, nmin );
```

```
unique_ptr< TClusterTree > ct( ct_builder.build( coord.get() ) );
```

where geometric nested dissection is applied for cluster tree construction by using the class `TBSPNDCTBuilder`. The parameter `nmin` is controlling the minimal size of the generated clusters and has been set to 40 which is a standard value for this parameter, and which lead to good computational performance in numerical tests. When creating the instance `ct_builder` of class `TBSPNDCTBuilder` a pointer to the sparse matrix `M` has been used as input parameter of the constructor in this particular example. It is important to note that exactly that sparse matrix has to be used as input parameter that has everywhere there a non-zero entry where the matrices K or M have a non-zero entry. In case of doubt, use the sparse matrix $|K| + |M|$ as input, i.e., that sparse matrix which is obtained by adding entry-wise the absolute values of the matrix entries of K and M.

The construction of the block cluster is performed by

```
const real_t  eta = 50.0;

TStdGeomAdmCond  adm_cond( eta );

TBCBuilder                          bct_builder;
unique_ptr< TBlockClusterTree >  bct( bct_builder.build( ct.get(), ct.get(), & adm_cond ) );
```

where the standard (geometric) admissibility condition has been used. The parameter `eta` is controlling the number of admissible subblocks in the block cluster tree and is typically set to 1 in many applications. For this application, however, in numerical tests better results have been obtained according to the computational time using larger `eta` and correspondingly having many admissible subblocks (a similar observation has been made, e.g., in (Hackbusch, Khoromskij, and Kriemann 2004)). In numerical tests `eta=50` has been a good choice.

Finally, performing

```
TSparseMBuilder  h_builder_K( K.get(), ct->perm_i2e(), ct->perm_e2i() );
TSparseMBuilder  h_builder_M( M.get(), ct->perm_i2e(), ct->perm_e2i() );

TTruncAcc  acc_exact( real_t(0) );

unique_ptr< TMatrix >  KH( h_builder_K.build( bct.get(), K->form(), acc_exact ) );
unique_ptr< TMatrix >  MH( h_builder_M.build( bct.get(), M->form(), acc_exact ) );
```

the $\mathcal{H}$-matrix representations `KH` and `MH` of the sparse matrices `K` and `M` are constructed using the block cluster tree `bct` as input. In particular, the representation of the sparse matrices `K` and `M` in the corresponding $\mathcal{H}$-matrix format is exact, i.e., no low-rank approximation is needed to be applied.

6. Solving the Eigenvalue Problem

Using the $\mathcal{H}$-matrix representations `KH`, `MH` of `K` and `M`, we can compute in the next step the eigensolution of the discrete eigenvalue problem (K, M). For reasons of convenience, the index h used for $K^{(h)}$, $M^{(h)}$, N_h indicating the underlying mesh width h of the finite element discretisation is left out in the following. As described above, the HAMLS eigensolver module provides the classes `THAMLS`, `TEigenArnoldi` and `TEigenArpack` for solving the eigenvalue problem (K, M). The theoretical background of these eigensolvers has already been outlined in the sections above. In the following the usage of each solver is described, its properties and in which cases they are best used.

Usage of the Eigensolvers `TEigenArpack` and `TEigenArnoldi`

Given the number of sought eigensolutions $n_{\text{es}} \in \mathbb{N}$ the classes `TEigenArnoldi` and `TEigenArpack` compute

the numerically exact partial eigensolution

$$K\,S \;=\; M\,S\,D \qquad \text{with} \quad S^T M\,S \;=\; \mathrm{Id}$$

where $D \in \mathbb{R}^{n_{\text{es}} \times n_{\text{es}}}$ is a diagonal matrix containing the n_{es} smallest eigenvalues and $S \in \mathbb{R}^{N \times n_{\text{es}}}$ the matrix containing column-wise the corresponding eigenvectors.

The eigensolver `TEigenArpack` works as follows

```
unique_ptr< TDenseMatrix >  D ( new TDenseMatrix() );
unique_ptr< TDenseMatrix >  S ( new TDenseMatrix() );

const uint  n_es = 10;

TEigenArpack arpack_solver;

// adjust the eigensolver
arpack_solver.set_n_ev_searched ( n_es  );
arpack_solver.set_print_info    ( true );
arpack_solver.set_shift         ( 0.0 );

// solve the eigenvalue problem
arpack_solver.comp_decomp( KH.get(), MH.get(), D.get(), S.get() );
```

After creating an instance of the class `TEigenArpack` basic parameters of the eigensolver are adjusted:

- The number of sought eigensolutions n_{es} is adjusted by applying the method `set_n_ev_searched(n_es)`.
- If the argument of the method `set_print_info( )` is set `true` summarized information of the eigensolver's execution is printed to the HLIBpro log-file. By default no information is printed.
- The eigensolver supports the solution of shifted problems. Using `set_shift(mu)` the eigensolver computes these `n_es` eigenpairs of $(K^{(h)}, M^{(h)})$ whose eigenvalues are located closest around the shift `mu`≥ 0. By default the parameter is set to `mu=0.0`, such that the eigensolver computes the `n_es` smallest eigenpairs of $(K^{(h)}, M^{(h)})$.

By executing the method `comp_decomp(...)` the (almost) numerically exact factorization $KS = MSD$ described above is computed. The result of the factorization is written into the dense matrices `D` and `S` which have to be instantiated before execution. The eigensolver `TEigenArnoldi` is applied in the analogous way

```
unique_ptr< TDenseMatrix >  D ( new TDenseMatrix() );
unique_ptr< TDenseMatrix >  S ( new TDenseMatrix() );

const uint  n_es = 10;

TEigenArnoldi arnoldi_solver;

// adjust the eigensolver
arnoldi_solver.set_n_ev_searched ( n_es  );
arnoldi_solver.set_print_info    ( true );
arnoldi_solver.set_shift         ( 0.0 );

// solve the eigenvalue problem
arnoldi_solver.comp_decomp( KH.get(), MH.get(), D.get(), S.get() );
```

Eigensolver `TEigenArpack` is based on the well-proven and well-tested ARPACK library. It is fast and very robust. However, it requires that the ARPACK library is provided by the user. Eigensolver `TEigenArnoldi` is a basic implementation of the Arnoldi solver. It is fast but computes in some cases eigenpair approximations with lower accuracy. Both solver are recommended when the number of eigenpairs sought is rather small.

Usage of the Eigensolver `THAMLS`

Given the number of sought eigensolutions $n_{\text{es}} \in \mathbb{N}$ the class `THAMLS` computes the approximative partial eigensolution

$$K\,S \;\approx\; M\,S\,D \qquad \text{with} \quad S^T M\,S \;=\; \text{Id}$$

where $D \in \mathbb{R}^{n_{\text{es}} \times n_{\text{es}}}$ is a diagonal matrix containing approximations of the n_{es} smallest eigenvalues and $S \in \mathbb{R}^{N \times n_{\text{es}}}$ the matrix containing column-wise the corresponding eigenvector approximations.

The accuracy of the computed approximative eigensolutions is controlled by the following two types of parameters:

- **Accuracy of $\mathcal{H}$-matrix arithmetic:** The accuracy of the computed eigensolutions is influenced by the chosen accuracy $\varepsilon \geq 0$ of the approximative $\mathcal{H}$-matrix arithmetic that is applied in $\mathcal{H}$-AMLS. The smaller the value ε, the higher is the accuracy of the approximative $\mathcal{H}$-matrix arithmetic, and with it the accuracy of the computed eigenpair approximations. On the other hand, with a higher accuracy for the $\mathcal{H}$-matrix arithmetic the computational costs of $\mathcal{H}$-AMLS increase.
- **Modal Truncation:** $\mathcal{H}$-AMLS is a domain decomposition method where the domain $\Omega \subset \mathbb{R}^d$ of the PDE problem is automatically subdivided into subdomains that are separated by interfaces. On the subdomains and on the interfaces suitable eigenvalue problems are defined which, however, do not solve the global problem but whose eigenfunctions are well suited to approximate the sought eigensolutions of the global problem. In particular eigenfunctions belonging to the smallest eigenvalues are selected from each subproblem to form a suitable subspace that is used for a Ritz-Galerkin approximation of the global problem. More precisely, in $\mathcal{H}$-AMLS we consider $m > 0$ subproblems, where each subproblem is associated to $N_i > 0$ degrees of freedom with $\sum_{i=1}^{m} N_i = N$, and where from each subproblem eigenvectors associated to the $k_i \leq N_i$ smallest eigenvalues are selected to form a subspace for the Ritz-Galerkin discretisation. How large k_i has to be chosen is not easy to answer and is part of ongoing research, see (Gerds 2017) for details. The larger k_i is selected, the more spectral information from each subproblem is kept and the better the accuracy of the computed eigenpair approximations of $\mathcal{H}$-AMLS will be. On the other hand, with larger k_i the computational costs of $\mathcal{H}$-AMLS increases. In (Gerds 2017) a new mode selection strategy is proposed to compute all discrete eigenvectors of the subproblems which still lead to reasonable approximations of the corresponding continuous eigenfunctions. According to (Gerds 2017) the number of selected eigenvectors should be of the form $k_i \in \mathcal{O}(N_i^{\beta})$ where the constant $\beta \in (0, 1)$ is depending on the polynomial degree of the used finite element space, the underlying dimension d and the type of subproblem. For example, when computing with $\mathcal{H}$-AMLS the $n_{\text{es}} = \mathcal{O}(N^{1/3})$ smallest eigenvalues and associated eigenvectors of a three-dimensional problem – when using piece-wise affine functions for finite element discretisation – the number of selected eigenpairs k_i should be of the order $\mathcal{O}(N_i^{1/3})$ for subdomain problems and of the order $\mathcal{O}(N_i^{1/2})$ for interface problems.

As described above the parameter ε of the approximative $\mathcal{H}$-matrix arithmetic and the parameter k_i of the modal truncation have to be balanced, in order to obtain with $\mathcal{H}$-AMLS eigenpair approximations with high accuracy while the computational costs of $\mathcal{H}$-AMLS are reduced as much as possible. Since in our setting discrete eigenvalue problems (K, M) result always from a finite element discretisation of a continuous problem, all eigenpairs of the discrete problem are related to a discretisation error. Hence, as long as the approximation error of $\mathcal{H}$-AMLS is of the same order as the discretisation error, the computed eigenpair approximations of $\mathcal{H}$-AMLS are of comparable quality as the eigenpairs computed by some classical approach.

Using the parameters described above the eigensolver `THAMLS` works as follows

```
unique_ptr< TDenseMatrix >  D ( new TDenseMatrix() );
unique_ptr< TDenseMatrix >  S ( new TDenseMatrix() );

const uint  n_es = 300;

THAMLS hamls_solver;
```

```
    // adjust basic options of HAMLS
    hamls_solver.set_n_ev_searched ( n_es );
    hamls_solver.set_print_info    ( true );

    // set the accurcy of the H-matrix arithmetic used in HAMLS
    const real_t  eps = 1e-3;
    hamls_solver.set_arithmetic_acc( eps );

    // set options for mode selection applied in HAMLS
    const real_t  C_sub = 1.5;
    const real_t  C_int = 1.0;
    const real_t  beta_sub = 1.0/3.0;
    const real_t  beta_int = 1.0/2.0;

    hamls_solver.set_factor_subdomain  ( C_sub  );
    hamls_solver.set_factor_interface  ( C_int  );
    hamls_solver.set_exponent_subdomain( beta_sub );
    hamls_solver.set_exponent_interface( beta_int );

    // solve the problem
    hamls_solver.comp_decomp ( ct.get(), KH.get(), MH.get(), D.get(), S.get(), K.get(), M.get() );
```

After creating an instance of class `TEigenArpack` basic parameters of the eigensolver are adjusted:

- The number of sought eigensolutions n_{es} is adjusted by applying the method `set_n_ev_searched(n_es)`.
- If the argument of the method `set_print_info( )` is set `true` summarized information of the eigensolver's execution is printed to the HLIBpro log-file. By default no information is printed.
- Using `set_arithmetic_acc( eps )` the relative accuracy of the applied $\mathcal{H}$-matrix arithmetic is set to the value `eps`≥ 0.
- In the example for the modal truncation the number of selected eigenpairs k_i is set to $C_{\text{sub}} N_i^{\beta_{\text{sub}}}$ if the subproblem is associated to a subdomain problem and to $C_{\text{int}} N_i^{\beta_{\text{int}}}$ if it is associated to an interface problem.

By executing the method `comp_decomp(...)` the approximative factorisation $KS \approx MSD$ described above is computed. The result of the factorisation is written into the dense matrices `D` and `S` which have to be instantiated before execution.

The `THAMLS` eigensolver is very fast especially when many eigenpairs are sought. This solver is highly recommended when a large portion of the eigenpairs is sought and when numerical exact discrete eigenpairs are not needed, e.g., when only the eigensolutions of the underlying continuous problem are of interest.

7. Finalization

The example finishes with the standard finalization used for HLIBpro programs and the catch block:

```
    DONE();
  }
  catch ( Error & e )
  {
    std::cout << e.to_string() << std::endl;
  }
  return 0;
}
```

8. The Plain Program

The complete program, including some evaluation output, is given in the following. Note that for program execution the data files **input_K**, **input_M** and **input_coord** have to be available in the program's working directory. As mentioned above, these files contain the sparse matrices K, M and the geometric representatives $(\xi_i)_{i=1}^N$ of the used finite element discretisation saved in the corresponding HLIBpro data format.

```
#include <iostream>
#include <hlib.hh>

#include "hamls/TEigenArnoldi.hh"
#include "hamls/TEigenArpack.hh"
#include "hamls/THAMLS.hh"

using namespace HLIB;
using namespace HAMLS;
using namespace std;

using  real_t    = HLIB::real;
using  complex_t = HLIB::complex;

int main ( int argc, char ** argv )
{
  try
  {
    INIT();

    //------------------------------
    // Importing Input Data
    //------------------------------
    THLibMatrixIO  matrix_io;

    const string  path_to_data = "./";

    unique_ptr< TMatrix >   K_temp( matrix_io.read( path_to_data + "input_K" ) );
    unique_ptr< TMatrix >   M_temp( matrix_io.read(  path_to_data +  "input_M" ) );

    unique_ptr< TSparseMatrix >  K( ptrcast( K_temp.release(), TSparseMatrix ) );
    unique_ptr< TSparseMatrix >  M( ptrcast( M_temp.release(), TSparseMatrix ) );

    TAutoCoordIO    coord_io;
    unique_ptr< TCoordinate >  coord( coord_io.read (  path_to_data + "input_coord") );

    //-------------------------------------------
    // Construction of the H-matrix Representation
    //-------------------------------------------
    const uint  nmin = 40;

    // cluster partitioning is based on an adaptive bisection
    TGeomBSPPartStrat  part_strat( adaptive_split_axis );

    TBSPNDCTBuilder  ct_builder( M.get(), & part_strat, nmin );

    unique_ptr< TClusterTree > ct( ct_builder.build( coord.get() ) );
```

```
const real_t  eta = 50.0;

TStdGeomAdmCond  adm_cond( eta );

TBCBuilder                        bct_builder;
unique_ptr< TBlockClusterTree >  bct( bct_builder.build( ct.get(), ct.get(), & adm_cond ) );

TSparseMBuilder  h_builder_K( K.get(), ct->perm_i2e(), ct->perm_e2i() );
TSparseMBuilder  h_builder_M( M.get(), ct->perm_i2e(), ct->perm_e2i() );

TTruncAcc  acc_exact( real_t(0) );

unique_ptr< TMatrix >  KH( h_builder_K.build( bct.get(), K->form(), acc_exact ) );
unique_ptr< TMatrix >  MH( h_builder_M.build( bct.get(), M->form(), acc_exact ) );

//-----------------------------------------
// Solving Eigenvalue Problem with TEigenArpack
//-----------------------------------------
unique_ptr< TDenseMatrix >  D ( new TDenseMatrix() );
unique_ptr< TDenseMatrix >  S ( new TDenseMatrix() );

const uint  n_es = 100;

TEigenArpack arpack_solver;

// adjust the eigensolver
arpack_solver.set_n_ev_searched ( n_es  );
arpack_solver.set_print_info    ( true );
arpack_solver.set_shift         ( 0.0 );

// solve the eigenvalue problem
arpack_solver.comp_decomp( KH.get(), MH.get(), D.get(), S.get() );

// analyse the computed eigensolution
TEigenAnalysis  eigen_analysis;
eigen_analysis.set_verbosity( 4 );
eigen_analysis.analyse_vector_residual ( K.get(), M.get(), D.get(), S.get(), ct.get() );

//-----------------------------------------------
// Solving Eigenvalue Problem with TEigenArnoldi
//-----------------------------------------------
TEigenArnoldi arnoldi_solver;

// adjust the eigensolver
arnoldi_solver.set_n_ev_searched ( n_es  );
arnoldi_solver.set_print_info    ( true );
arnoldi_solver.set_shift         ( 0.0 );

// solve the eigenvalue problem
arnoldi_solver.comp_decomp( KH.get(), MH.get(), D.get(), S.get() );

// analyse the computed eigensolution
eigen_analysis.analyse_vector_residual ( K.get(), M.get(), D.get(), S.get(), ct.get() );
```

```
    //-----------------------------------------------
    // Solving Eigenvalue Problem with THAMLS
    //-----------------------------------------------
    THAMLS hamls_solver;

    // adjust basic options of HAMLS
    hamls_solver.set_n_ev_searched ( n_es );
    hamls_solver.set_print_info    ( true );

    // set the accurcy of the H-matrix arithmetic used in HAMLS
    const real_t  eps = 1e-3;
    hamls_solver.set_arithmetic_acc( eps );

    // set options for mode selection applied in HAMLS
    const real_t  C_sub = 1.5;
    const real_t  C_int = 1.0;
    const real_t  beta_sub = 1.0/3.0;
    const real_t  beta_int = 1.0/2.0;

    hamls_solver.set_factor_subdomain  ( C_sub  );
    hamls_solver.set_factor_interface  ( C_int  );
    hamls_solver.set_exponent_subdomain( beta_sub );
    hamls_solver.set_exponent_interface( beta_int );

    // solve the problem
    hamls_solver.comp_decomp ( ct.get(), KH.get(), MH.get(), D.get(), S.get(), K.get(), M.get() );

    // analyse the computed eigensolution
    eigen_analysis.analyse_vector_residual ( K.get(), M.get(), D.get(), S.get(), ct.get() );

    DONE();
  }
  catch ( Error & e )
  {
    std::cout << e.to_string() << std::endl;
  }
  return 0;
}
```

9. References

Bai, Zhaojun, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, eds. 2000. *Templates for the Solution of Algebraic Eigenvalue Problems*. Vol. 11. Computer software, Environments, and Tools. Philadelphia, PA: Society for Industrial; Applied Mathematics (SIAM). https://doi.org/10.1137/1.9780898719581.

Banjai, L., S. Börm, and S. Sauter. 2008. "FEM for Elliptic Eigenvalue Problems: How Coarse Can the Coarsest Mesh Be Chosen? An Experimental Study." *Comput. Vis. Sci.* 11 (4-6): 363–72. https://doi.org/10.1007/s00791-008-0101-5.

Bennighof, Jeffrey K., and R. B. Lehoucq. 2004. "An Automated Multilevel Substructuring Method for Eigenspace Computation in Linear Elastodynamics." *SIAM J. Sci. Comput.* 25 (6): 2084–2106 (electronic). https://doi.org/10.1137/S1064827502400650.

Bennighof, J. K. 1993. "Adaptive Multi-Level Substructuring Method for Acoustic Radiation and Scattering from Complex Structures." *Computational Methods for Fluid/Structure Interaction* 178: 25–38.

Gerds, Peter. 2017. "Solving an elliptic PDE eigenvalue problem via automated multi-level substructuring and hierarchical matrices." Dissertation, RWTH Aachen University; Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University. https://doi.org/10.18154/RWTH-2017-10520.

Gerds, Peter, and Lars Grasedyck. 2015. "Solving an Elliptic Pde Eigenvalue Problem via Automated Multi-Level Substructuring and Hierarchical Matrices." *Computing and Visualization in Science* 16 (6): 283–302. https://doi.org/10.1007/s00791-015-0239-x.

Grimes, Roger G., John G. Lewis, and Horst D. Simon. 1994. "A Shifted Block Lanczos Algorithm for Solving Sparse Symmetric Generalized Eigenproblems." *SIAM J. Matrix Anal. Appl.* 15 (1): 228–72. https://doi.org/10.1137/S0895479888151111.

Hackbusch, Wolfgang. 1992. *Elliptic differential equations: theory and numerical treatment.* Vol. 18. Springer Series in Computational Mathematics. Berlin: Max Planck Institute for Mathematics in the Sciences; Springer.

Hackbusch, Wolfgang, Boris N. Khoromskij, and Ronald Kriemann. 2004. "Hierarchical matrices based on a weak admissibility criterion." *Computing* 73 (3): 207–43. https://doi.org/10.1007/s00607-004-0080-4.

Lehoucq, Richard B., Chao-Chih Yang, and Danny C. Sorensen. 1998. *ARPACK Users' Guide : Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Philadelphia: SIAM. http://opac.inria.fr/record=b1104502.

Sauter, S. 2010. "Hp-Finite Elements for Elliptic Eigenvalue Problems: Error Estimates Which Are Explicit with Respect to Lambda, H, and P." *SIAM J. Numerical Analysis* 48 (1): 95–108. https://doi.org/http://dx.doi.org/10.1137/070702515.