Max-Planck-Institut
für Mathematik
in den Naturwissenschaften
Leipzig

HLIBpro User Manual

(revised version: February 2008)

by

Ronald Kriemann

Technical Report no.: 9 2008

$\mathcal{H}$-Lib$^{\text{pro}}$

v0.13.2

User Manual

by
Ronald Kriemann

Contents

1 Preface **1**

2 Introduction **3**

2.1 Cluster Trees and Block Cluster Trees 4

2.2 $\mathcal{H}$-Matrices 7

2.3 HLIBpro 8

2.3.1 Revision History 9

3 Installation **13**

3.1 Prerequisites 13

3.1.1 Operating System and Compiler 13

3.1.2 LAPACK 13

3.1.3 Misc. Tools 13

3.2 Configuration 14

3.3 Compilation 15

4 Tutorials **17**

4.1 Integral Equation 17

4.2 Sparse Matrices 17

Bibliography **21**

1 Preface

$\mathcal{H}$-Lib^{pro} is a software library implementing hierarchical matrices or $\mathcal{H}$-matrices for short. This type of matrices, first introduced in [Hac99], provides a technique to represent various full matrices in a data-sparse format and furthermore, allows usual matrix arithmetic, e.g. matrix-vector multiplication, matrix multiplication and inversion, with almost linear complexity. Examples of matrices, which can be represented by $\mathcal{H}$-matrices stem from the area of partial differential or integral equations.

Beside the standard arithmetic mentioned above, $\mathcal{H}$-Lib^{pro} also provides additional algorithms for decomposing matrices, e.g. LU-factorisation and for solving linear equation systems with various direct or iterative methods. Furthermore, it contains methods for directly converting a dense operator into an $\mathcal{H}$-matrix without constructing the corresponding dense matrix. See Section 2.3 for a detailed list of the functionality of $\mathcal{H}$-Lib^{pro}.

Content

Chapter 2 contains an introduction to $\mathcal{H}$-matrices in which basic concepts needed to understand the $\mathcal{H}$-Lib^{pro} functions are explained. In particular, it describes the path from a given matrix or equation system to a $\mathcal{H}$-matrix representing the corresponding data. Experienced users might want to skip this chapter.

In Chapter 3 the configuration and installation process of $\mathcal{H}$-Lib^{pro} is discussed.

The description of the functions and classes in $\mathcal{H}$-Lib^{pro} starts with two tutorials describing typical usage scenarios in Chapter 4.

Conventions

The following typographic conventions are used in this documentation:

`CODE`	For functions and other forms of source code appearing in the document.
`TYPES`	For data types, e.g. structures, pointers and classes.
`FILES`	For files, programs and command line arguments.

Furthermore, several boxes signal different kind of information. A remark to the corresponding subject is indicated by

Remark

This is a remark

Important information regarding crucial aspects of the topic are displayed as

Attention

This is important.

Examples for a specific function or algorithm are enclosed by

```
void example ();
```

2 Introduction

In this chapter, the basic concepts in the context of $\mathcal{H}$-matrices are introduced as it is needed to understand the functions discussed later. It shall not give a complete overview over the field of hierarchical matrices. For this, please refer to [Hac99], [Gra01] and [GH03].

$\mathcal{H}$-matrices are a special data-sparse storage format for dense matrices, e.g. each coefficient of the matrix can be different from zero. Even so, for a dense $n \times n$ matrix the total storage complexity is of order $n \log n$. Due to the special structure of $\mathcal{H}$-matrices, one is also capable of doing matrix arithmetic, e.g. matrix multiplication and inversion, with a similar complexity of $n \log^2 n$. This allows an efficient handling of many problems in mathematics, physics or engineering.

As an example consider the linear equation system

$$Ax = y$$

with the $n \times n$ matrix A, which can be solved directly by computing

$$x = A^{-1}y$$

in almost linear time.

Unfortunately, this is *not* possible for all types of matrices and the computed vector x is only an approximation of the true solution of the above equation. But, since most numerical algorithms involve some kind of approximation, e.g. the transition from continuous to discrete models, the additional error due to the solution phase does not change the overall approximation quality.

The question, whether a given problem can be handled by $\mathcal{H}$-matrices usually comes down to the question, whether parts of a matrix can be approximated by *low-rank* matrices. Consider for example the $n \times m$ matrix M. If the actual rank of M is $k \ll \min\{n, m\}$, it is possible to represent M as the product

$$M = A \cdot B^T$$

with the $n \times k$ matrix A and the $m \times k$ matrix B. The storage required to store M is now

$$\boldsymbol{k \cdot (n + m)} \quad \text{instead of} \quad \boldsymbol{n \cdot m}$$

which is significantly smaller and of order $\max\{n, m\}$.

If M is not of rank k but a rank k matrix $\tilde{M} = A'B'^T$ exists, which approximates M up to a given accuracy $\varepsilon \geq 0$, e.g.

$$\|M - \tilde{M}\| \leq \varepsilon$$

in some norm $\|\cdot\|$, one could again replace M by a low-rank matrix if ε is small enough and reduce the storage requirements.

Now we come back to our linear equation system $Ax = y$. Usually it is not possible to represent or approximate A directly by a low-rank matrix. But if parts of A, e.g. matrix blocks, can be represented, this would also reduces the needed memory to store A (see Figure 2.1).

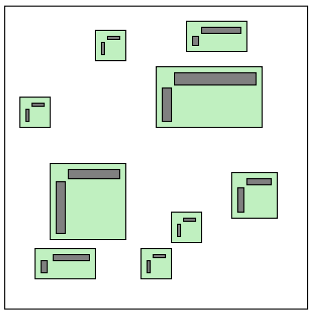


Figure 2.1: Parts of a Matrix represented by low-rank Matrices

The question is, which blocks of the matrices can be approximated? Here one has to take a closer look at the considered problem, e.g. the partial differential or the integral equation. A sub block M of the matrix A corresponds to two indexsets I, J where I and J are associated with the rows and the columns of M, e.g. $M \in \mathbb{R}^{I \times J}$. Furthermore, I and J usually also correspond to parts of the geometry associated with the given problem (see Figure 2.2). This leads us to the following informal rule for many differential and integral operators:

> If the parts of the geometry associated with I and J are well separated, e.g. have a positive distance, then the matrix block $M \in \mathbb{R}^{I \times J}$ can be approximated by a low-rank matrix.

We will come back to the exact mathematical definition of this rule in the next section.

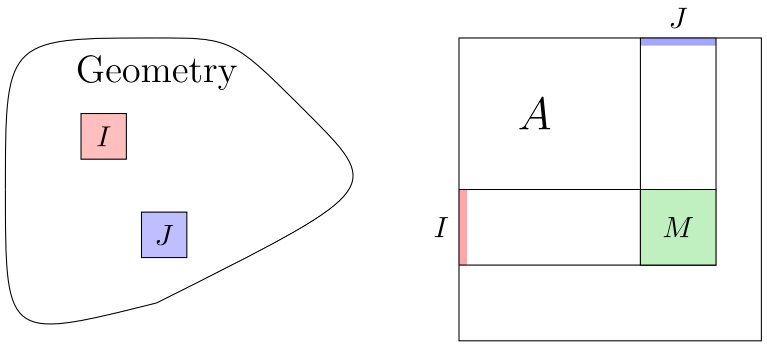


Figure 2.2: Correspondence of Geometry with Indexsets

Unfortunately, this rule only tells us, that these sub blocks in the matrix exists. We still have to choose them by our own and since one has 2^{2n} different possibilities, the right strategy is crucial for efficient algorithms. This brings us to *cluster trees* and *block cluster trees*.

2.1 Cluster Trees and Block Cluster Trees

A *cluster* is a subset of the indexset, e.g. the set I, whereas a *block cluster* is the product of two clusters, e.g. $I \times J$, and therefore corresponds to matrix blocks. To restrict the choice of sub matrices, where a low-rank approximation can be constructed, one introduces a hierarchical decomposition of a clusters.

The main idea is to decompose the indexset I into two (or more) disjoint subsets I_1, I_2 which are again subdivided until only a single index per set remains. If one identifies each of the constructed sets by a node and introduces edges between indexsets according to the decomposition hierarchy, e.g. between an indexset and it's direct subsets, one ends up with a *cluster tree* (see Figure 2.3).

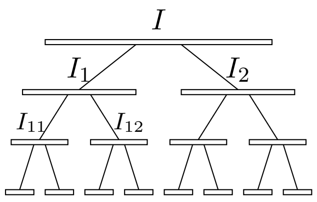


Figure 2.3: Cluster tree construction

Several algorithms are available for subdividing a cluster depending on the specific problem, e.g.

- *cardinality balanced*: decompose each indexset into two sub clusters with an equal number of indices or
- *geometrically balanced*: decompose each indexset into two sub clusters of equal volume.

It is also possible to decompose a cluster without geometrical information for sparse matrices by examining the connectivity defined by the matrix structure (see [GKB05]).

We now restrict our search for matrix blocks, which we want to approximate by low-rank matrices, to products of clusters in a given cluster tree. Furthermore, the products shall contain only clusters of equal depth in the tree. The possible set of such block clusters is defined by the product of two cluster trees: the *block cluster tree* . In Figure 2.4 the first three levels of the product of the above cluster tree for I is shown.

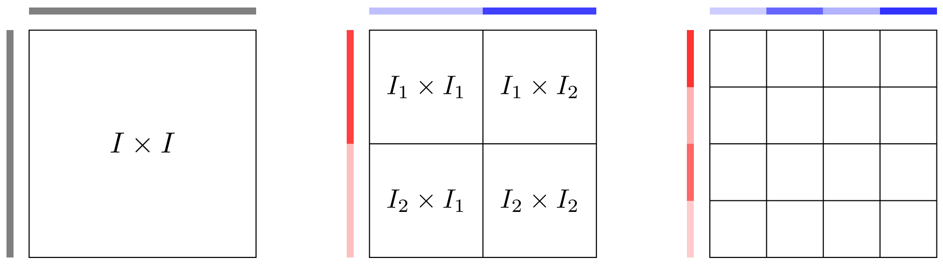


Figure 2.4: Product of two Cluster trees

Unfortunately, multiplying two cluster trees completely results in $\#I^2$ leafs in the block cluster tree. Therefore, one has to apply further restrictions on the set of nodes in such a tree. Here, the above mentioned rule comes into play and is now called *admissibility condition* defined by:

A block cluster $t \times s$ with clusters t, s is called *admissible*, if the following condition holds:

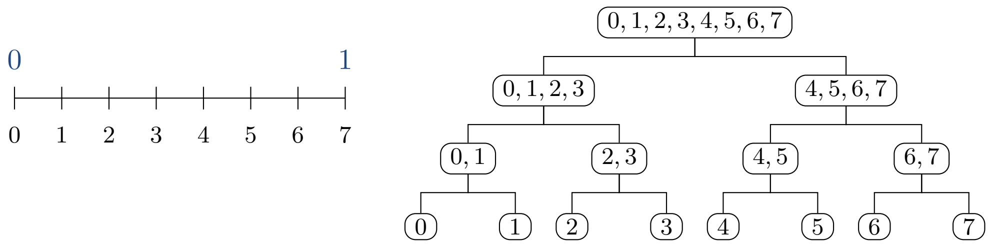
$$\max\{\operatorname{diam}(t), \operatorname{diam}(s)\} \leq \eta \operatorname{dist}(t, s)$$

with $\operatorname{diam}(t)$ being the diameter of the cluster t, $\operatorname{dist}(t, s)$ the distance between the cluster t and s and $\eta \geq 0$.

Just like the previous informal description, the admissibility condition is true if two clusters are small compared to their distance. The distance is scaled by the parameter η which is problem dependent.

All admissible blocks in the block cluster tree will not be further refined, e.g. the multiplication of the two cluster trees stops at such a node.

Let us examine this condition on a simple example in $\mathbb{R}^1$, namely the interval $[0, 1]$ with equally distant indices and a cardinality balanced subdivision:



Starting the construction of the block cluster tree, one notes, that diagonal blocks can never be admissible. Hence, one level of refinement is always necessary and yields Figure 2.5 (left). On the next level, for the block cluster $\{0, 1, 2, 3\} \times \{4, 5, 6, 7\}$ one has to check the following admissibility condition:

$$\begin{aligned} \max\left\{\operatorname{diam}(\{0,1,2,3\}), \operatorname{diam}(\{4,5,6,7\})\right\} &= \max\left\{\frac{3}{7}, \frac{3}{7}\right\} \\ &\leq \\ \eta \operatorname{dist}\left\{\{0,1,2,3\}), \operatorname{diam}(\{4,5,6,7\}\right\} &= \eta\,\frac{1}{7} \end{aligned}$$

which requires $\eta \geq 3$ to be true. We now assume $\eta = 0.5$ and hence the previous block to be inadmissible. For the corresponding block cluster tree this means further refinement.

On the next level, we will check the admissibility for all products of the cluster $\{0, 1\}$:

- $\{0, 1\} \times \{2, 3\}$: $\max\left\{\frac{1}{7}, \frac{1}{7}\right\} \nleq \eta\,\frac{1}{7}$,
- $\{0, 1\} \times \{4, 5\}$: $\max\left\{\frac{1}{7}, \frac{1}{7}\right\} \leq \eta\,\frac{3}{7}$,
- $\{0, 1\} \times \{6, 7\}$: $\max\left\{\frac{1}{7}, \frac{1}{7}\right\} \leq \eta\,\frac{5}{7}$.

Similar results are obtained for the other clusters. The resulting decomposition of the block indexset is shown in Figure 2.5 (middle), where all admissible blocks are marked green. The remaining non-admissible blocks are further refined yielding the final decomposition as can be seen in Figure 2.5 (right).

Other admissibility condition can be defined, e.g. by using the minimum of the diameter instead of the maximum. The exact definition of the diameter of a cluster is also dependent on the problem.

In either case, one now has decomposed the initial block indexset $I \times I$ into $\mathcal{O}(\#I)$ sub sets and has identified suitable blocks for approximating the original matrix.

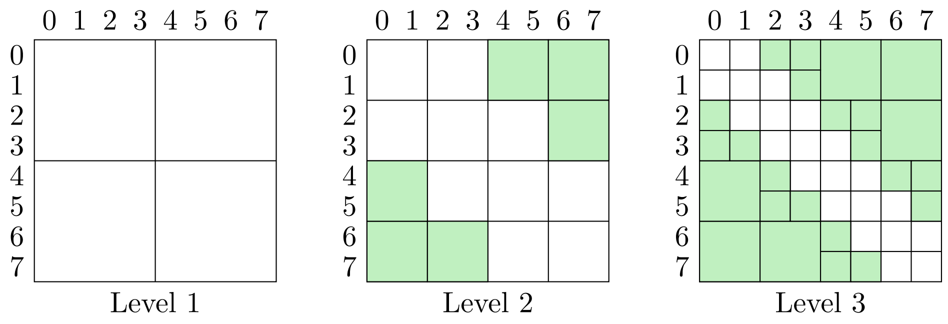


Figure 2.5: Construction of a block cluster tree

2.2 $\mathcal{H}$-Matrices

Having a block cluster tree and the knowledge, that each matrix block corresponding to an admissible node (or block cluster) can be approximated by a low-rank matrix, we can build an $\mathcal{H}$-matrix.

Still open is the method to use for computing the low-rank approximation. This is highly dependent on the problem under consideration. For partial differential equations, one usually has the property, that *all* admissible blocks have rank 0 because matrix entries are non-zero only for direct neighbours in the grid and since admissible matrix blocks are associated with well separated areas of the grid, no direct connection exists. Here one should also note, that the storage requirements of $\mathcal{H}$-matrices are usually higher than for sparse matrix formats and hence the conversion of such matrices in $\mathcal{H}$-matrices is only advised, if further operations, e.g. inversion or LU factorisations, are performed.

For integral equations the situation is different and several approximation algorithms exist which compute a low-rank matrix for a dense block, e.g. adaptive cross approximation (see [Beb00]) or hybrid cross approximation (see [BG05]).

Examples for real $\mathcal{H}$-matrices are shown in Figure 2.6. There, the green blocks again correspond to admissible blocks whereas red blocks are inadmissible.

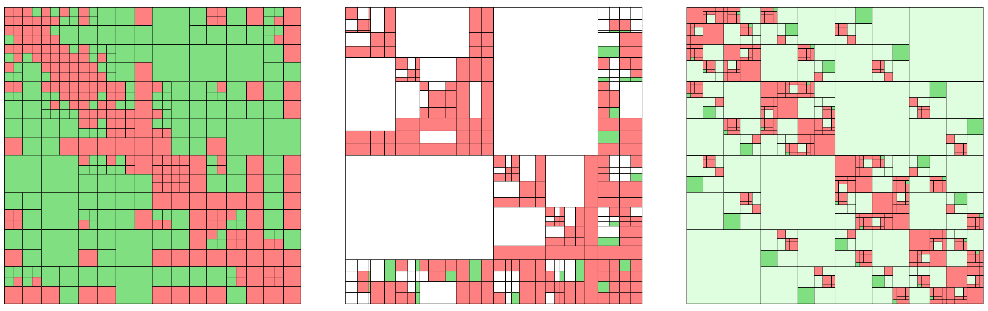


Figure 2.6: $\mathcal{H}$-matrix examples

Once a $\mathcal{H}$-matrix is available, one can perform almost all matrix algebra with it. Furthermore, all of these algebra operations are performed in linear-logarithmic time. The following table lists the basic arithmetic functions available for $\mathcal{H}$-matrices together with their computations complexity:

Operation	Complexity
Matrix Vector Product	$\mathcal{O}(n \log n)$
Matrix Addition	$\mathcal{O}(n \log n)$
Matrix Multiplication	$\mathcal{O}\left(n \log^2 n\right)$
Matrix Inversion	$\mathcal{O}\left(n \log^2 n\right)$
LU Factorisation	$\mathcal{O}\left(n \log^2 n\right)$

Due to the availability of the matrix arithmetic, $\mathcal{H}$-matrices can be used for many more problems than just linear equations. One example would be matrix Riccati equations (see [GHK03]). Or the direct treatment of Schur complement systems (see [Hac03]) as the arise in various problems, e.g. domain decomposition.

2.3 $\mathcal{H}$-Lib$^{\text{pro}}$

$\mathcal{H}$-Lib$^{\text{pro}}$ is a software library which implements all of the above described algorithms. Due to the much more complex data structures and more involved methods in the context of $\mathcal{H}$-matrices compared to sparse and dense matrices, one major goal for $\mathcal{H}$-Lib$^{\text{pro}}$ was to simplify the access to these functions and therefore make hierarchical matrices available to a broader audience.

The development of $\mathcal{H}$-Lib$^{\text{pro}}$ started in 1999 and was done in parallel to HLib (see [BG]). Some algorithms which were first tested in HLib later found their way into $\mathcal{H}$-Lib$^{\text{pro}}$. Other techniques, especially the parallel $\mathcal{H}$-matrix arithmetics is only available in $\mathcal{H}$-Lib$^{\text{pro}}$. Furthermore, $\mathcal{H}$-Lib$^{\text{pro}}$ is also capable of doing complex valued arithmetic for $\mathcal{H}$-matrices. In addition to this, the implementation of $\mathcal{H}$-Lib$^{\text{pro}}$ focuses on more robust algorithms in terms of numerics and software technology, e.g. error handling.

To use $\mathcal{H}$-matrices in $\mathcal{H}$-Lib$^{\text{pro}}$, one has to follow the general path which was laid out in this chapter:

1. decompose the indexset in the form of a cluster tree,
2. identify approximable subsets of the block indexset by building the block cluster tree and
3. build low-rank approximations for admissible blocks and put them together in an $\mathcal{H}$-matrix.

By default, for each of these steps only minimal information and user interaction is necessary, while $\mathcal{H}$-Lib$^{\text{pro}}$ uses parameters best suited for the given problem.

For the last step, the $\mathcal{H}$-matrix construction, $\mathcal{H}$-Lib$^{\text{pro}}$ implements the following approximation algorithms:

- Singular Value Decomposition,
- Adaptive Cross Approximation (classical, advanced and full) and
- Hybrid Cross Approximation.

Beside this $\mathcal{H}$-matrix specific functionality, $\mathcal{H}$-Lib$^{\text{pro}}$ contains many more methods, e.g.

- Matrix norm computation ($\|A\|, \|A^{-1}\|, \|A - B\|$) for

  Frobenius norm ($\|\cdot\|_F$) and Spectral norm ($\|\cdot\|_2$)

- Iterative Solvers for the solution of linear equations:

  Richardson iteration, CG iteration, BiCG-Stab iteration, MINRES iteration and GMRES iteration

- Advanced Triangle Quadrature Rules (see [SS04]),
- Input/Output in various formats:

  SAMG ([Fra]), Matlab ([Mat]), Harwell-Boeing ([Natb]), Matrix-Market ([Nata]) and $\mathcal{H}$-Lib^{pro} format

Examples for the usage of the functions in $\mathcal{H}$-Lib^{pro} can be found in Chapter 4, in which a dense and a sparse linear equation system are solved using $\mathcal{H}$-matrices and LU factorisation as well as inversion.

2.3.1 Revision History

v0.13.2 (2008-02-29)

- replaced threads and mutices by OpenMP (thread start only, no scheduling)
- included log file support in addition to stdout
- added parallel LDL^T factorisation (DD and blockdiag only)
- added parallel blockdiag LU factorisation
- added zero approximation during matrix construction (for nearfield only)
- fixed bug in algebraic nested dissection clustering (wrong path length in interface)

v0.13.1 (2008-02-04)

- reduced memory consumption/fragmentation in ACA generated matrices with large rank
- added Fiduccia/Mattheyses bisection optimisation for BFS clustering
- added FFT for vectors by implementing support for FFTW3 (optional)
- fixed bug in TBSPPartCTBuilder when using more than two partitions
- fixed potential issues in sorting algorithms
- fixed type issues with `*_bytesize` functions in C interface
- fixed bug in PostScript visualisation of matrices if matrix norm is zero
- fixed issues with GCC-4.3
- fixed bug in command line parsing of configuration system
- minor modifications to SCons system to increase userfriendliness

v0.13 (2007-12-19)

- general Algorithmic Changes
  - support for single precision arithmetic; has to be decided *before* compiling $\mathcal{H}$-Lib^{pro}
  - made complete C++ functions and classes visible from outside instead of just C interface functions
  - rewrote complex arithmetic to distinguish between symmetric and hermitian matrices; added LDL^H and LL^H factorisations
  - inversion now based on LU, thereby reducing memory consumption (roughly halved)
  - added computation of the diagonal of the inverse without computing the inverse
  - added evaluation of LU, LDL^T factorisations (instead of just solving)

- removed point-wise LU and LDL^T factorisation (only blocked) to improve robustness with zeroes on diagonal
- added (optional) check and fix for singular sub matrices during inversion and factorisation
- added complex valued HCA
- new version of ACA+
- multiplication $C = A \cdot D \cdot B$ with diagonal D implemented
- implemented bilinear forms for Helmholtz single and double layer potential
- implemented bilinear form for acoustic scattering
- rewrote algebraic clustering for sparse matrices; added support for Scotch and CHACO
- added support for periodic coordinates in clustering
- added clustering with user defined index partition on first level in cluster tree
- added standard admissibility for algebraic clustering
- added maximal level in clustering to prevent infinite recursion
- modified solvers to handle complex valued data
- added permutation of dense matrices without temporary storage (needed in IO)

▶ parallel Arithmetic

- added thread parallel algorithms for matrix construction, matrix multiplication, inversion and LU factorisation
- redesigned thread pool, thereby fixing race conditions
- added support for Windows threads
- fixed several issues with thread safety

▶ Input and Output

- added general I/O functions with autodetection of file format
- added output of matrices in Harwell/Boeing format
- added MatrixMarket format
- added support for Ply and surface mesh format (NetGen) for Grid I/O
- fixed format errors in SAMG output
- conversion of arbitrary matrices to sparse format when writing in SAMG or Harwell/Boeing format
- fixed support for symmetric matrices in Harwell/Boeing format

▶ C interface

- prefixed *all* functions, types and constants with `hlib` (or `HLIB`) to prevent collisions with other definitions (OS or libraries)
- added support for C99 complex types (if available)
- added `hlib_set_coarsening` to activate/deactivate coarsening during matrix construction (default: on) and matrix arithmetic (default: off)
- added `hlib_matrix_inv_diag` to return diagonal of inverse
- added `hlib_matrix_is_complex` to test for real or complex valued matrices
- added `hlib_set_nthreads` to set number of threads
- added `hlib_coord_t` as special type for coordinates
- separated stop criterion and solver in solver interface

▶ Miscellaneous

- updated CPUflags and Rmalloc
- fixed optimisation issues (leading to infinite loops) in enclosed CLAPACK

v0.12 (2006-11-01)

▶ Algorithmic Changes

- added (blocked) LDL^T factorisation (now default for symmetric matrices)
- no longer need extra matrix in matrix inversion
- using ACAFull in HCA instead of SVD to reduce runtime
- adaptively choosing quadrature and interpolation order in ACA and HCA
- rewrote matrix addition to support general cases, e.g. low-rank to blocked
- rewrote low-rank truncation handling
- support for METIS in algebraic clustering routines
- added basic support for "dense" sparse matrices, e.g. with highly coupled indices
- added SSE2 based HCA algorithm
- added infinity norm for vectors
- using norm of preconditioned residual for all solvers if preconditioner is present
- added MINRES iteration
- using `ADM_AUTO` as default admissibility
- finally removed all asserts and replaced by internal error checking

▶ Input and Output

- VRML97 support
- added Matlab compression (Matlab v7) and *structs* support
- support for Harwell-Boeing matrix format (read-only)
- modified PostScript output of block-wise SVD; now scaled w.r.t. 2-norm of matrix

▶ OS and Library support

- MS Windows support
- shared libraries for Linux and Windows
- added internal `xerbla` to handle LAPACK errors directly
- changed configure system to better handle MS Windows environment

▶ C interface

- automatic choice of matrix building in `hlib_matrix_build_bem_grid`
- introduced `vector_t` as type to vectors (no more C arrays)
- added Gauss and Sauter triangle quadrature rules
- added functions to access matrix and vector entries
- added `copyto` and `copyto_eps` functions
- added `hlib_matrix_build_dense` to build $\mathcal{H}$-matrix from dense matrix
- changed solver management

▶ Miscellaneous

- several improvements and bug fixes
- cleaned up error codes
- updated CPUflags and Rmalloc

3 Installation

In this section the installation procedure of $\mathcal{H}$-Lib^{pro} is discussed. This includes the various tools and libraries need for $\mathcal{H}$-Lib^{pro} to compile and work, the configuration system of $\mathcal{H}$-Lib^{pro} and the usage of the supplied tools for compiling programs with $\mathcal{H}$-Lib^{pro}.

3.1 Prerequisites

3.1.1 Operating System and Compiler

$\mathcal{H}$-Lib^{pro} was tested on a variety of operating systems and is known to work on the following environments

Linux, Solaris, AIX, HP-UX, Darwin, Tru64, FreeBSD and Windows

In particular, each POSIX conforming system should be fine. The more crucial part plays the compiler. $\mathcal{H}$-Lib^{pro} demands a C++ compiler which closely follows the C++ standard. The following compiler versions are known to work

GCC	Version 3.4 and above, including 4.x
Intel Compiler	Version 5 and above
Portland Compiler	Version 2.x
Sun Forte/Studio	Version 5.3 and above
IBM VisualAge	Version 6
HP C++–Compiler	Version 3.31 and above
Compaq C++	Version 6.5 and above
Microsoft VisualC++	Version 8 and above

3.1.2 LAPACK

Internally, $\mathcal{H}$-Lib^{pro} uses *LAPACK* (see [ABB$^+$99]) for most arithmetic operations. Therefore, an implementation of this library is needed for $\mathcal{H}$-Lib^{pro}. Most operating systems provide a LAPACK library optimised for the corresponding processor, e.g. the Intel Math Kernel Library or the Sun Performance Library. If no such implementation is available, $\mathcal{H}$-Lib^{pro} contains a modified version of CLAPACK. But it should be noted that this might result in a reduced performance of $\mathcal{H}$-Lib^{pro}.

3.1.3 Misc. Tools

To use the configuration system, a Python interpreter is needed. Furthermore, the build system of $\mathcal{H}$-Lib^{pro} is using SCons, which again is based on Python.

3.2 Configuration

If the right operation mode is chosen, the configuration system of $\mathcal{H}$-Lib$^{\text{pro}}$ can be used to create all the neccessary files to construct the package. For this, simply type

```
./configure
```

which uses default settings for your operating system and compiler and chooses appropriate options for the compilation, e.g. include directives, libraries etc..

All available options for the configuration system can be printed by

```
./configure --help
```

which will result in the following list:

--prefix=`dir`
: Set the prefix for installing $\mathcal{H}$-Lib$^{\text{pro}}$ to the directory `dir`. By default, the local directory is chosen.

--cc=`CC`
--cxx=`CXX`
: Use `CC` and `CXX` as the C- and C++-compiler. By default appropriate settings for the considered operating system are used, e.g. `gcc` and `g++` for *Linux*.

--cflags=`FLAGS`
--cxxflags=`FLAGS`
: Define the compiler flags for C and C++ files.

--enable-single
--enable-double
: Define precision of $\mathcal{H}$-arithmetics. By default double precision is used.

--with-cpuflags[=`PATH`]
: Enable the usage of cpuflags which will determine appropriate compiler flags for the combination of compiler, operating system and processor. The optional argument `PATH` defines the correct path to `cpuflags`. By default, the supplied copy of `cpuflags` will be used.

--lapack[=`FLAGS`]
: Defines the linking flags for the LAPACK implementation needed by $\mathcal{H}$-Lib$^{\text{pro}}$. By default, a modified copy of CLAPACK is used, which can also be defined by specifying "CLAPACK".

--with-threads[=`type`]
: Enable support for multiple threads in $\mathcal{H}$-Lib$^{\text{pro}}$, whereby `type` is one of `auto`, `pthread` or `win32`.

--with[**out**]**-zlib**[=`DIR`]
--zlib-cflags=`FLAGS`
--zlib-lflags=`FLAGS`
: Turns on/off *zlib* support in $\mathcal{H}$-Lib$^{\text{pro}}$. The optional argument `DIR` specifies the location of the zlib environment, e.g. includes and libraries. Alternatively, you can define the compilation flags for zlib directly.

--with[**out**]**-metis**[=`DIR`]
--metis-lflags=`FLAGS`
: Turns on/off METIS support in $\mathcal{H}$-Lib$^{\text{pro}}$. The optional argument `DIR` specifies the location of the METIS library. Alternatively, you can define the linking flags for METIS directly.

--with[**out**]**-scotch**[=`DIR`]
--scotch-cflags=`FLAGS`
--scotch-lflags=`FLAGS`
: Turns on/off Scotch support in $\mathcal{H}$-Lib$^{\text{pro}}$. The optional argument `DIR` specifies the location of the Scotch library. Alternatively, you can define the compilation and linking flags for Scotch directly.

--with[out]-chaco[=DIR]
--chaco-lflags=FLAGS
: Turns on/off CHACO support in $\mathcal{H}$-Lib^{pro}. The optional argument DIR specifies the location of the CHACO library. Alternatively, you can define the linking flags for CHACO directly.

--with[out]-fftw3
--fftw3-cflags=FLAGS
--fftw3-lflags=FLAGS
: Turns on/off FFTW3 support in $\mathcal{H}$-Lib^{pro}. By default, standard flags for compiling and linking are assumed. Alternatively, you can define the compilation and linking flags for FFTW3 directly.

--with-rmalloc
: Enable the Rmalloc memory allocator provided by $\mathcal{H}$-Lib^{pro}.

A typical example for the usage of the configuration system might be

```
./configure --with-cpuflags --with-zlib
```

which enables the usage of `cpuflags` to get the correct compiler flags and support for compressed files through the zlib library.

Beside these $\mathcal{H}$-Lib^{pro} related options, the following parameters can be used to change the behaviour of the configuration system itself:

-c, --check
: Turns on checking of used tools and libraries, e.g. if the C++ compiler is capable of producing object files.

-n, --no-check
: Turns off tool and library checking.

-h, --help
: Prints a detailed description of all options for the configuration system.

-q, --quiet
: Do not print any information during configuration.

-s, --show
: Just print the current options of the configuration system without creating construction files.

-v, --verbose
: Print additional information during configuration.

All settings which where changed by the user will be written to the file `config.cache`, where one can optionally edit the parameters with a text-editor.

3.3 Compilation

After $\mathcal{H}$-Lib^{pro} was configured, you can compile it by

```
scons
```

Parallel compilation, e.g. via `-j`, is also supported.

By default, a version of $\mathcal{H}$-Lib^{pro} with debugging symbols will be created. This can be changed by arguments to `scons`:

debug=[0|1]
: Turn on/off debugging support during compilation.

optimise=[0|1]
: Turn on/off optimisation support during compilation.

profile=[0|1]
: Turn on/off profiling support during compilation.

shared=[0|1]
: Enable/disable building of shared libraries.

static=[0|1]
: Enable/disable building of static libraries.

fullmsg=[0|1]
: Enable/disable printing of complete compilation commandline.

The current status of these options is printed by

```
scons options
```

After compilation, a library should reside in the `lib/` directory and examples for the usage of $\mathcal{H}$-Lib$^{\text{pro}}$ should have been generated in the `example/` subdirectory. If CLAPACK was chosen as the LAPACK implementation, a corresponding library can be found in the `lib/` directory.

4 Tutorials

Before all of the functionality of $\mathcal{H}$-Lib$^{\text{pro}}$ will be discussed in it's full detail, we will begin with an introduction based on typical situations in which $\mathcal{H}$-matrices will be used: representing integral equation operators and working with sparse matrices.

4.1 Integral Equation

4.2 Sparse Matrices

In this example, a linear equation system with a sparse matrix S shall be solved by the use of $\mathcal{H}$-matrices. For this, we will first convert the sparse matrix into an $\mathcal{H}$-matrix, compute a $\mathcal{H}$-LU factorisation of the $\mathcal{H}$-matrix and finally use this LU factorisation as a preconditioner for solving the equation system.

To access $\mathcal{H}$-Lib$^{\text{pro}}$ functions and classes, we have to include the corresponding header files. Although each file can be included individually, a general header file is provided:

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

Remark

Since with `hlib.hh` *all* header files in $\mathcal{H}$-Lib$^{\text{pro}}$ are included in the current source, compilation times can be increased significantly. Therefore, this is only advised for small examples.

But before we can start, $\mathcal{H}$-Lib$^{\text{pro}}$ has to be initialised:

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

Please note the `HLIB` namespace, which applies to all functions in $\mathcal{H}$-Lib$^{\text{pro}}$[1].

Cluster Tree and Block Cluster Tree

As was discussed in Section 2.1, the index sets over which the matrix S is defined have to be decomposed into cluster trees before $\mathcal{H}$-arithmetics can be used. For this, two different methods are available depending on whether or not geometrical information is available for each index or not. For this example, we will assume the presence of such geometrical data, which has to be imported into $\mathcal{H}$-Lib$^{\text{pro}}$ before further usage:

```
  HLIB::TAutoCoordIO                      cooio;
  HLIB::autoptr< HLIB::TCoordinate >  coord;

  coord = cooio.read( "coordfile.coo" );
```

[1]Accept functions in the C language interface (see [Kri]).

Here, we have read the coordinate information from the file `coordfile.coo` by using file format autodetection implemented in `TAutoCoordIO`.

The usage of `autoptr` (see ???) is advised, and extensively used in $\mathcal{H}$-Lib$^{\text{pro}}$, since they decrease the possibility of memory leaks by automatically deleting the objects upon destruction of the autopointer variable, e.g. when leaving the local block.

Once the coordinates for each index are available, the index set can be clustered. For this, standard geometrical bissection is used, implemented in `TBSPCTBuilder` (see ???).

```
HLIB::TBSPCTBuilder                     ct_builder( HLIB::BSP_AUTO, 20 );
HLIB::autoptr< HLIB::TClusterTree >   ct;

ct = ct_builder.build( coord.get() );
```

The two parameters to the constructor of `ct_builder` define the partitioning technique, which is automatically chosen, and the upper limit of index sets not decomposed any further (see ???) .

After construction the cluster tree, the block index set has to be partitioned into a block cluster tree . At this point, blocks in the yet to be constructed $\mathcal{H}$-matrix are identified by the so-called *admissibility condition* , which will be represented in low-rank format. Since geometrical data is used to built the cluster tree, the admissibility condition is also based on the geometry of clusters and compares diameters with distances (see ???).

```
HLIB::TStdAdmCond                         adm_cond( 2.0 );
HLIB::TBCBuilder                          bct_builder;
HLIB::autoptr< HLIB::TBlockClusterTree > bct;

bct = bct_builder.build( ct.get(), ct.get(), & adm_cond );

std::cout << "sparsity constant = " << bct->compute_c_sp() << std::endl;
```

Since an important characteristic of the block cluster tree and involved in many complexity estimates in the context of $\mathcal{H}$-matrices, the *sparsity constant* c_{sp} is computed for the constructed block cluster tree.

$\mathcal{H}$-Matrix Construction

The next step is the construction of the $\mathcal{H}$-matrix itself. But before doing so, the sparse matrix has to be made available. Again, format autodetection is used to read a file containing the sparse matrix. For convenience, the general pointer to the matrix is represented as a sparse matrix of type `TSparseMatrix`. Note, that this is a standard pointer instead of an autopointer since the matrix is already managed by the autopointer `M`.

```
HLIB::TAutoMatrixIO                  mio;
HLIB::autoptr< HLIB::TMatrix >      M;
HLIB::TSparseMatrix *               S = NULL;

M = mio.read( "matrix.mat" );
S = dynamic_cast< HLIB::TSparseMatrix * >( M.get() );

if ( S == NULL )
  throw HLIB::Error( HLIB::ERR_MAT_NSPARSE );
```

In case that `M` does not contain a sparse matrix, an exception is thrown with the corresponding error code (see ???).

Finally, we can build the $\mathcal{H}$-matrix with the help of `TSparseMBuilder`, which expects the sparse matrix and permutations of the index set. The latter are a direct result of the clustering procedure above, since that involves a reordering of the indices. To access to correct coefficients in `S`, this ordering has to be reverted.

Since `S` is sparse, usually all low-rank blocks have rank zero. Therefore, an exact approximation can be done by setting the relative and the absolute error in `acc` to 0.

```
27   HLIB::TSparseMBuilder              h_builder( S, ct->perm_i2e(),
28                                                     ct->perm_e2i() );
29   HLIB::TTruncAcc                    acc( 0.0, 0.0 );
30   HLIB::autoptr< HLIB::TMatrix >     A;
31
32   A = h_builder.build( bct.get(), S->form(), acc, progress.get() ) );
33
34   std::cout << "size of H-matrix = "
35             << HLIB::Sys::Mem::to_string( A->byte_size() ) << std::endl;
```

The size of the created $\mathcal{H}$-matrix is printed in line 34 by making use of the memory format function in `Sys::Mem`.

Solving the Linear Equation System

The goal of this example was the solution of a given linear system of equations by the $\mathcal{H}$-matrix technique. What is still missing to solve $Sx = b$ is a good preconditioner for S. This is computed by an $\mathcal{H}$-LU factorisation of A, which allows the evaluation of the inverse of A, and therefore of S.

```
36   HLIB::TLU  lu;
37
38   lu.factorise( 1, A.get(), HLIB::TTruncAcc( 1e-4, 0.0 ) );
39
40   std::cout << "size of LU factor = "
41             << HLIB::Sys::Mem::to_string( A->byte_size() ) << std::endl;
```

Here, A is factorised with a block-wise accuracy of 10^{-4}, i.e. all low-rank blocks are truncated to the best approximation w.r.t. the Frobenius norm with an error not larger than 10^{-4}. The first parameter to the `factorise` function defines the number of threads to use during LU factorisation.

Remark

For symmetric matrices, LDL^T factorisation is done automatically instead of LU factorisation.

Unfortunately, the error bound of 10^{-4} only applys to each block individually. The approximation of the inverse of S by the $\mathcal{H}$-LU factorisation of A usually differs from this value. Important for solving the equation system is the norm of the iteration matrix in a linear iteration $\|I - (A)^{-1}S\|_2$ which can be computed by:

```
42   HLIB::TSpectralNorm  mnorm;
```

```
43   HLIB::TLUInvMatrix   A_inv( A.get() );
44
45   std::cout << " inversion error   = "
46             << mnorm.inv_approx( S, & A_inv ) << std::endl;
```

Here, `A` is wrapped by `A_inv`, which behaves like $A^{-1} = (LU)^{-1}$ instead of $A = LU$ and therefore, allows matrix vector multiplication with the inverse of A.

Equipped with an approximation to the inverse of S and therefore with a reasonable preconditioner, the equation system can be solved:

```
47   HLIB::TSolver::TInfo            solve_info;
48   HLIB::TAutoVectorIO             vio;
49   HLIB::autoptr< HLIB::TVector >  b, x;
50
51   b = vio.read( "rhs.vec" );
52   x = S->col_vector();
53
54   HLIB::solve( S, x.get(), b.get(), & A_inv, & solve_info );
55
56   if ( solve_info.converged() )
57     std::cout << "  converged with " << solve_info.n_iter() << " steps "
58               << "  with rate " << solve_info.conv_rate()
59               << ", |r| = " << solve_info.res_norm() << std::endl;
```

The right-hande side `b` is read from file `rhs.vec`, whereas the solution vector `x` is built by asking `S` for a compatible vector w.r.t. to the column index set of `S`.

In line 54, the system is solved by $\mathcal{H}$-Lib$^{\text{pro}}$ by automatically choosing a suitable iteration technique based on `S` and `A_inv` (see ???). In `solve_info` information about the solution process, e.g. the number of iterations steps or the convergence rate, is stored.

Finalisation

The final step in this example program is the finalisation of $\mathcal{H}$-Lib$^{\text{pro}}$ by

```
60   HLIB::done();
61
62   return 0;
63 }
```

Please note, that all objects, e.g. matrices, vectors, cluster trees, are automatically freed since they are managed by autopointers. Therefore, no explicit release of these objects is performed.

Bibliography

[ABB$^+$99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[Beb00] M. Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86:565–589, 2000.

[BG] S. Börm and L. Grasedyck. HLib. http://www.hlib.org.

[BG05] S. Börm and L. Grasedyck. Hybrid cross approximation of integral operators. *Numerische Mathematik*, 2:221 – 249, 2005.

[Fra] Fraunhofer SCAI, http://www.scai.fraunhofer.de/. *SAMG file format specification.*

[GH03] L. Grasedyck and W. Hackbusch. Construction and Arithmetics of $\mathcal{H}$-Matrices. *Computing*, 70:295–334, 2003.

[GHK03] L. Grasedyck, W. Hackbusch, and B.N. Khoromskij. Solution of large scale algebraic matrix Riccati equations by use of hierarchical matrices. *Computing*, 70:121–165, 2003.

[GKB05] L. Grasedyck, R. Kriemann, and S. Le Borne. Parallel black box domain decomposition based h-lu preconditioning. Preprint nr. 115/2003, MPI Leipzig, 2005.

[Gra01] L. Grasedyck. *Theorie und Anwendungen Hierarchischer Matrizen.* Dissertation, University of Kiel, 2001.

[Hac99] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. I. Introduction to $\mathcal{H}$-matrices. *Computing*, 62(2):89–108, 1999.

[Hac03] W. Hackbusch. Direct Domain Decomposition using the Hierarchical Matrix Technique. In *Proceedings of the 14th International Conference on Domain Decomposition Methods*, pages 38–50, Cocoyoc, Mexico, June 2003.

[Kri] Ronald Kriemann. *HLIBpro C language interface.* Max-Planck-Institute for Mathematics in the Sciences, Leipzig.

[Mat] The MathWorks, http://www.mathworks.com/. *MAT-File Format Version 7.*

[Nata] National Institute of Standards and Technology, http://math.nist.gov/MatrixMarket/formats.html. *The Matrix Market Exchange Formats: Initial Design.*

[Natb] National Institute of Standards and Technology, http://math.nist.gov/MatrixMarket/collections/hb.html. *User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release I).*

[SS04] S. Sauter and C. Schwab. *Randelementmethoden: Analysen, Numerik und Implementierung schneller Algorithmen.* Teubner, Stuttgart, 2004.

Index

A

admissibility condition 5, 18

autopointer 18

B

bissection 18

block cluster 4

block cluster tree 5, 18

C

cluster 4

cluster tree 5

coordinates 18

c_{sp} 18

H

header files 17

N

n_{min} 18

P

Python 13

S

SCons 13

sparse matrix 17

sparsity constant 18

Function and Datatype Index

A
autoptr 18