Max-Planck-Institut
für Mathematik
in den Naturwissenschaften
Leipzig

HLIBpro C Language Interface

(revised version: February 2008)

by

Ronald Kriemann

Technical Report no.: 10 2008

$\mathcal{H}$-Lib^pro

v0.13.2

C Language Interface

by
Ronald Kriemann

Contents

1 Preface **1**

2 Introductory Examples **3**
2.1 Integral Equation . . . 3
2.2 Sparse Linear Equation System . . . 9

3 General Functions and Data types **13**
3.1 Initialisation and Finalisation . . . 13
3.2 Error Handling . . . 13
3.3 Data types . . . 16
3.4 Reference Counting . . . 17
3.5 Parallel Computing . . . 17

4 Coordinates and Cluster Trees **19**
4.1 Coordinates . . . 19
4.1.1 Coordinate Management Functions . . . 20
4.1.2 Coordinate I/O . . . 21
4.2 Cluster Trees . . . 21
4.2.1 Cluster Tree Management Functions . . . 21
4.2.2 Cluster Tree Construction . . . 22
4.2.3 Cluster Tree I/O . . . 25
4.3 Block Cluster Trees . . . 26
4.3.1 Block Cluster Construction . . . 26
4.3.2 Block Cluster Tree Management Functions . . . 27
4.3.3 Block Cluster Tree I/O . . . 27

5 Vectors and Matrices **29**
5.1 Vectors . . . 29
5.1.1 Creating and Accessing Vectors . . . 29
5.1.2 Vector Management Functions . . . 31
5.1.3 Algebraic Vector Functions . . . 32
5.1.4 Vector I/O . . . 33
5.2 Matrices . . . 34
5.2.1 Importing Matrices from Data structures . . . 34
5.2.2 Building $\mathcal{H}$-Matrices . . . 37
5.2.3 Matrix Management . . . 42
5.2.4 Matrix Norms . . . 44
5.2.5 Matrix I/O . . . 45

6 Algebra **49**

6.1 Matrix Vector Multiplication . . . 49
6.2 Matrix Addition . . . 50
6.3 Matrix Multiplication . . . 50
6.4 Matrix Inversion . . . 51
6.5 Solving Linear Systems . . . 52
6.6 Changing Algebra behaviour . . . 55

7 Miscellaneous Functions **57**
7.1 Measuring Time . . . 57
7.2 Progress Meter . . . 57
7.3 Quadrature Rules . . . 58
7.3.1 Gaussian Quadrature . . . 58
7.3.2 Quadrature Rules for Triangles . . . 58

Bibliography **61**

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 standard 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 come 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.

This document describes the C language interface to the $\mathcal{H}$-Lib^{pro} library. Since $\mathcal{H}$-Lib^{pro} is programmed in C++, the internal functions and classes have to be mapped to C data structures. Furthermore, the functionality of the C interface is limited compared to the C++ interface, but nevertheless provides most techniques used for standard problem solving. Furthermore, due to this simplification, the C interface involves a less steep learning curve to get familiar with $\mathcal{H}$-matrices and $\mathcal{H}$-Lib^{pro}.

For an introduction into $\mathcal{H}$-matrices, please refer to the $\mathcal{H}$-Lib^{pro} user manual ([Kri]). This also applies to the documentation of the installation process.

Content

The documentation of the interface to the C programming language begins with two typical examples in Chapter 2. General functions and data types are described in Chapter 3. Coordinates, cluster trees and block cluster trees as the basic building blocks for $\mathcal{H}$-matrices are the topic of Chapter 4, whereas vectors and matrices will be discussed in Chapter 5. Functions for the $\mathcal{H}$-matrix algebra are introduced in Chapter 6. Finally, descriptions of auxiliary functions can be found in Chapter 7.

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 Introductory Examples

Before the individual functions are described in detail, two typical examples of their usage are used for an introduction into the C interface to $\mathcal{H}$-Lib^{pro}.

2.1 Integral Equation

In this example an integral equation is to be solved by using $\mathcal{H}$-matrices. For this, the system matrix has to be represented in the $\mathcal{H}$-format and a preconditioner shall be computed in with the $\mathcal{H}$-arithmetics. Since the matrices in question are dense, the following procedures apply also to this type of matrices in general.

Remark

Source code for the complete example with additional output statements and timing of each function can be found in `examples/bem1d.c`.

Problem Definition

The following integral equation is considered:

$$\int_0^1 \log|x-y| u(y) dy = f(x), \quad x \in [0,1]$$

Here, the function $u : [0,1] \to \mathbb{R}$ is sought for a given right hand side $f : [0,1] \to \mathbb{R}$. The Galerkin discretisation uses constant ansatz functions $\varphi_i, 0 \leq i < n$

$$\varphi_i(x) = \begin{cases} 1 & x \in [\frac{i}{n}, \frac{i+1}{n}] \\ 0 & \text{otherwise} \end{cases}$$

This leads to a linear equations system with the matrix A defined by

$$\begin{aligned} a_{ij} &= \int_0^1 \int_0^1 \varphi_i(x) \log|x-y| \varphi_j(y) dy dx \qquad (2.1) \\ &= \int_{\frac{i}{n}}^{\frac{i+1}{n}} \int_{\frac{j}{n}}^{\frac{j+1}{n}} \log|x-y| dy dx \end{aligned}$$

Initialisation

Before we can call any $\mathcal{H}$-Lib^{pro} functions, the library is initialised with `hlib_init` in line 6. To increase the output of $\mathcal{H}$-Lib^{pro}, the default verbosity is increased to level 2 in line 7.

```
1 #include "hlib-c.h"
2
3 int
```

```
4  main ( int argc, char ** argv ) {
5    int  info;
6
7    hlib_init( & argc, & argv, & info );                          CHECK_INFO;
8    hlib_set_verbosity( 2 );
```

Error checking is performed by the macro `CHECK_INFO`, which looks at the value of the variable `info` after each function call to $\mathcal{H}$-Lib^{pro} and tests whether an error occured. The definition of `CHECK_INFO` is as follows:

```
#define CHECK_INFO  { if ( info != HLIB_NO_ERROR ) \
                      { char buf[1024]; hlib_error_desc( buf, 1024 ); \
                        printf( "\n%s\n\n", buf ); exit(1); } }
```

There, the complete error message is copied into the string `buf` by the function `hlib_error_desc` and printed to the standard output. Afterwards, the program is aborted.

Coordinates and Cluster Trees

In order to represent the system matrix in the $\mathcal{H}$-matrix format, one has to create a cluster tree and a block cluster tree. For the cluster tree, coordinate informations are necessary for each index.

The dimension of the problem is defined by the n, which also defines the stepsize h of the discretisation. The coordinates for the indices are chosen as the midpoints of the i'th interval $\left[\frac{i}{n}, \frac{i+1}{n}\right]$. After allocating and initialising the data for the index positions, it is imported into $\mathcal{H}$-Lib^{pro} by `hlib_coord_import` at line 18. Using the coordinate data, the cluster tree can be created with `hlib_ct_build_bsp` as it is done at line 20. The resulting tree is afterwards printed to the file `ct.ps` in PostScript format. Finally, one can construct the block cluster tree using `hlib_bct_build`.

```
9    int       n = 1024;
10   double    h = 1.0 / ((double) n);
11   double ** vertices = (double**) malloc( n * sizeof(double*) );
12
13   for ( int i = 0; i < n; i++ ) {
14     vertices[i]    = (double*) malloc( sizeof(double) );
15     vertices[i][0] = h * ((double) i) + (h / 2.0);
16   }
17
18   hlib_coord_t   coo = hlib_coord_import( n, 1, vertices,
19                                            NULL, & info );       CHECK_INFO;
20   hlib_cluster_t ct  = hlib_ct_build_bsp( coord, & info );       CHECK_INFO;
21   hlib_ct_print_ps( ct, "ct.ps", & info );                       CHECK_INFO;
22
23   hlib_blockcluster_t  bct = hlib_bct_build( ct, ct, & info );  CHECK_INFO;
24   hlib_bct_print_ps( bct, "bct.ps", & info );                    CHECK_INFO;
```

Matrix Construction

After the partitioning of the block indexset in the form of the block cluster tree is computed, the actual $\mathcal{H}$-matrix can be build. For this, adaptive cross approximation (or ACA, see Section 5.2.2) is used in this example. ACA needs a matrix coefficient function, which computes

the matrix entries a_{ij} for given index pairs $(i, j) \in I \times I$. In our case, this function is given by (2.1), which, after evaluating the integral, translates into the following code:

```
void coeff_fn ( const int n, const int * rowidx,
                const int m, const int * colidx,
                double * matrix, void * arg ) {
  int     rowi, colj;
  double  h = *((double*) arg);

  for ( colj = 0; colj < m; colj++ ) {
    const int idx1 = colidx[colj];

    for ( rowi = 0; rowi < n; rowi++ ) {
      const int  idx0 = rowidx[rowi];
      double     value;

      if ( idx0 == idx1 )
        value = -1.5*h*h + h*h*log(h);
      else {
        const double  dist = h * ( fabs( (double) ( idx0-idx1 ) ) - 1.0 );
        const double  t1   = dist+1.0*h;
        const double  t2   = dist+2.0*h;

        value = ( - 1.5*h*h + 0.5*t2*t2*log(t2) - t1*t1*log(t1) );

        if ( fabs(dist) > 1e-8 )
          value += 0.5*dist*dist*log(dist);
      }

      matrix[(colj*n) + rowi] = -value;
} } }
```

The arguments `n`, `rowidx`, `m` and `colc` define a submatrix of dimension `n`×`m` with row and column indices stored in `rowidx` and `colidx`. The coefficients must be stored in *column wise ordering* as in Fortran instead of row wise ordering as in C in the array `matrix`. The additional argument `arg` points to user supplied data and contains the h stepwidth in our example (see below).

Equipped with the coefficient function, the code for constructing an $\mathcal{H}$-matrix looks like:

```
25   hlib_matrix_t A = NULL;
26   A = hlib_matrix_build_coeff( bct, coeff_fn, & h, HLIB_LRAPX_ACAPLUS,
27                                1e-4, 1, & info );                CHECK_INFO;
28
29   hlib_matrix_print_ps( A, "A.ps", HLIB_MATIO_SVD, & info );     CHECK_INFO;
30
31   long bytesize = hlib_matrix_bytesize( A, & info );             CHECK_INFO;
32
33   printf( "  compression ratio = %.2f%% (%.2f MB compared to %.2f MB)\n",
34           100.0 * ((double) bytesize) /
35           (((double) n) * ((double) n) * ((double) sizeof(double))),
36           ((double) bytesize) / (1024.0 * 1024.0),
```

```
37             (((double) n) * ((double) n) * ((double) sizeof(double))) /
38             (1024.0 * 1024.0) );
```

Here, ACA is chosen by the option HLIB_LRAPX_ACAPLUS, which defines the advanced and preferable version of ACA. The coefficient function together with the stepsize h form the second and third parameter to hlib_matrix_build_coeff, whereby h is the user supplied data of arbitrary format.

Since the $\mathcal{H}$-matrix usually does not represent the exact dense matrix but only an approximation, the accuracy of this approximation is defined by the fifth parameter to the function hlib_matrix_build_coeff and is set to 10^{-4} in this example. The accuracy applies per matrix block, e.g. for each matrix block the approximation is performed with an error of 10^{-4}. Unfortunately, the error for the whole matrix can not be controlled that easily.

Finally, the second last parameter indicates the symmetry of the matrix.

The matrix is afterwards printed in PostScript format to a file with hlib_matrix_print_ps in line 29. For each matrix block of the $\mathcal{H}$-matrix, a singular value decomposition is computed and the singular values are printed in a logarithmic scale. This allows a control of the approximation properties for the $\mathcal{H}$-matrix, e.g. if the singular values decrease rapidly in a block, this part of the matrix can be approximated well (see also Section 5.2.5.5).

To determine the efficiency of the $\mathcal{H}$-matrix approximation in terms of memory usage, the consumption of the matrix is calculated at line 31 and compared to a dense matrix at line 33.

One can also compare the quality of the approximation due to ACA and the coarse approximation by eps $= 10^{-4}$ with the best approximation of the given matrix with respect to machine precision. For the best approximation, the singular value decomposition is the method of choice. Unfortunately, it has a complexity of $\mathcal{O}\left(n^3\right)$ and is therefore only practical for small problem sizes. The machine precision is assumed to be 10^{-16}, which is roughly valid for most systems and the double precision floating point format.

```
39   if ( n < 2000 ) {
40     hlib_matrix_t  B = NULL;
41     B = hlib_matrix_build_coeff( bct, coeff_fn, & h, HLIB_LRAPX_SVD,
42                                 1e-16, 1, & info );              CHECK_INFO;
43     hlib_matrix_print_ps( B, "A_svd.ps", HLIB_MATIO_SVD, & info );CHECK_INFO;
44     printf( "  |A-~A|_F/|A|_F = %.4e\n" ,
45             hlib_matrix_norm_spectral_diff( A, B, & info ) );    CHECK_INFO;
46     hlib_matrix_free( B, & info );                              CHECK_INFO;
47   }
```

Using hlib_matrix_norm_spectral_diff, the relative difference with respect to the spectral norm is computed and printed at line 44. Since it is no longer used, matrix B is released at line 46 with hlib_matrix_free.

Remark

Instead of comparing A with another $\mathcal{H}$-matrix, one could also compute the exact matrix A_{exact} and determine $\|A - A_{exact}\|_F$ (see ???). But remember, that this requires $\mathcal{O}\left(n^2\right)$ memory whereas the above procedure usually consumes $\mathcal{O}\left(n \log n\right)$ memory, albeit with a large constant.

Before the equation system can be solved, the right hand side also has to be discretised and represented by a corresponding vector. The above described ansatz leads to the following

function for the right hand side $b_i = \int_0^1 \varphi_i(x) f(x) dx$, where f is chosen such that for the solution $u \equiv 1$ holds:

```
double rhs ( const int i, const int n ) {
    const double  a     = ((double)i)     / ((double) n);
    const double  b     = ((double)i+1.0) / ((double) n);
    double        value = -1.5 * (b - a);

    if ( fabs( b )       > 1e-8 ) value += 0.5*b*b*log(b);
    if ( fabs( a )       > 1e-8 ) value -= 0.5*a*a*log(a);
    if ( fabs( 1.0 - b ) > 1e-8 ) value -= 0.5*(1.0-b)*(1.0-b)*log(1.0-b);
    if ( fabs( 1.0 - a ) > 1e-8 ) value += 0.5*(1.0-a)*(1.0-a)*log(1.0-a);

    return value;
}
```

The actual vectors for b and the solution, stored in x, are constructed out of C arrays of length n:

```
48  double * x_arr = (double *) malloc( n * sizeof(double) );
49  double * b_arr = (double *) malloc( n * sizeof(double) );
50
51  hlib_vector_t x = hlib_vector_import_array( x_arr, n, & info );CHECK_INFO;
52  hlib_vector_t b = hlib_vector_import_array( b_arr, n, & info );CHECK_INFO;
53
54  for ( i = 0; i < n; i++ ) b_arr[i] = rhs( i, n );
```

Please note, that the access to the elements of the vector b at line 54 is done via the array b_arr, which is faster then the equivilant call to the corresponding $\mathcal{H}$-Lib^{pro} function hlib_vector_entry_set.

Solving the System

The solution x is computed using:

```
55  hlib_solve( A, x, b, NULL, & info );                          CHECK_INFO;
```

Since the solution is known, one can compute the error $\|x - \mathbb{1}\|_2$:

```
56  hlib_vector_t one = hlib_vector_copy( x, & info );            CHECK_INFO;
57  hlib_vector_fill( one, 1.0, & info );                         CHECK_INFO;
58
59  hlib_vector_axpy( x, 1.0, one, & info );                      CHECK_INFO;
60  double error = hlib_vector_norm2( x, & info );                CHECK_INFO;
61
62  printf( "  error of solution = \%.4e\n", error );
```

Here, the linear algebra functions for vectors in $\mathcal{H}$-Lib^{pro} are used. The vector one contains the exact solution.

The unpreconditioned iteration is usually inefficient. Therefore, matrix inversion is used to speed up the process. Since only matrix-vector multiplications with the inverse are needed for all implemented iteration techniques, LU factorisation is the method of choice.

To obtain a LU decomposition, the matrix A is copied into LU using hlib_matrix_copy_eps. Since only a preconditioner is needed, this copy is not exact but with a reduced blockwise

accuracy of 10^{-2}. The same accuracy is then used for the LU factorisation performed by hlib_matrix_inv_lu at line 64. This function overwrites the given matrix by its LU factors. The quality of the resulting preconditioner is tested at line 68. There, the largest eigenvalue of $\|I - A(LU)^{-1}\|$ is computed. Solving the preconditioned system and computing the resulting error is analogous to the unpreconditioned case.

```
63  hlib_matrix_t LU = hlib_matrix_copy_eps( A, 1e-2, & info );   CHECK_INFO;
64  hlib_matrix_inv_lu( LU, 1e-2, & info );                        CHECK_INFO;
65
66  hlib_matrix_print_ps( LU, "LU.ps", HLIB_MATIO_SVD, & info );   CHECK_INFO;
67
68  printf( "  inversion error = \%.4e\n",
69          hlib_matrix_norm_inv_approx( A, LU, & info ) );        CHECK_INFO;
70
71  hlib_solve_precond( A, LU, x, b, NULL, & info );               CHECK_INFO;
72
73  hlib_vector_axpy( x, 1.0, one, & info );                       CHECK_INFO;
74  error = hlib_vector_norm2( x, & info );                        CHECK_INFO;
75
76  printf( "  error of solution = \%.4e\n", error );
77
78  hlib_matrix_free( LU, & info );                                CHECK_INFO;
```

Remark

In example/bem1d.c, a preconditioner based on Gaussian elimination is computed in addition to the one due to LU factorisation presented here.

Finalisation

Finally, one has to release all resources allocated in the example and finish $\mathcal{H}$-Lib^{pro}:

```
79  hlib_vector_free( x, & info );                                 CHECK_INFO;
80  hlib_vector_free( b, & info );                                 CHECK_INFO;
81  hlib_vector_free( one, & info );                               CHECK_INFO;
82  hlib_matrix_free( A, & info );                                 CHECK_INFO;
83  hlib_bct_free( bct, & info );                                  CHECK_INFO;
84  hlib_ct_free( ct, & info );                                    CHECK_INFO;
85
86  free( x_arr ); free( b_arr );
87
88  for ( i = 0; i < n; i++ ) free( vertices[i] );
89  free( vertices );
90
91  hlib_done( & info );                                           CHECK_INFO;
```

Please note the manual freeing of the C arrays x_arr and b_arr which are associated with the vectors x and b. This has to be done by the user, since $\mathcal{H}$-Lib^{pro} will not release user allocated memory.

2.2 Sparse Linear Equation System

In this example, a linear equation system

$$Sx = b$$

with a sparse matrix S and a right hand side b is considered. Such a system usually occures in the context of finite difference or finite element discretisations. The linear system itself is provided in the form of a SAMG dataset with the basename "samg_matrix" (see Section 5.1.4.2 and Section 5.2.5.2). For this specific example, we assume that no geometrical information about the position of the degrees of freedom is available.

Remark

The complete example with additional output statements and timing of each function is available in `examples/crsalg.c`. A similar example with geometrical clustering can be found in `examples/crsgeom.c`.

Initialisation and Data Import

Again, before any $\mathcal{H}$-Lib^{pro} function can be used, the library has to be initialised. Afterwards, the data from the files containing the matrix and the right-hand side is imported to $\mathcal{H}$-Lib^{pro} using the functions `hlib_load_matrix` and `hlib_load_vector` respectively. Here, the I/O functions are used, which automatically detect the format of the corresponding files. After the import, the matrix is printed to the file `S.ps` in PostScript format (see Section 5.2.5.5). As the option `HLIB_MATIO_PATTERN` is supplied, only the pattern of non-zero matrix elements is printed.

The vector for the unknown x is constructed via `hlib_matrix_col_vector` such that it is compatible for a right multiplication with the matrix S (see Section 5.2.3).

Evaluating the content of `info` is again done by the macro `CHECK_INFO` which is defined as in the previous section.

```
#include "hlib-c.h"

int
main ( int argc, char ** argv ) {
  hlib_matrix_t   S;
  hlib_vector_t   b, x;
  int             n, info;
  char            mtx_file[] = "samg_matrix.amg";
  char            rhs_file[] = "samg_matrix.rhs";

  hlib_init( & argc, & argv, & info );                        CHECK_INFO;

  S = hlib_load_matrix( mtx_file, & info );                   CHECK_INFO;
  hlib_matrix_print_ps( S, "S.ps", HLIB_MATIO_PATTERN, & info );CHECK_INFO;
  b = hlib_load_vector( rhs_file, & info );                   CHECK_INFO;
  x = hlib_matrix_col_vector( S, & info );                    CHECK_INFO;
```

Since the complete data for the equation system is now available, one can solve it via `hlib_solve` and the default solver of $\mathcal{H}$-Lib^{pro}:

```
17   hlib_solve( S, x, b, NULL, & info );                           CHECK_INFO;
```

In this particular case, no object for obtaining information about the solution process was supplied.

LU Factorisation for Preconditioning

As in the previous example, the standard iteration process is usually far to costly. Therefore, a suitable preconditioner based on the $\mathcal{H}$-matrix technique shall be constructed to speed up the iteration. At first, this is accomplished by using LU factorisation in combination with nested dissection (see Section 4.2.2 and Section 6.4).

But before the matrix can be factorised, it has to be converted to an $\mathcal{H}$-matrix. For this, one needs a cluster tree and a block cluster tree. Since no geometrical data is available in this example, both objects are constructed algebraically with the functions `hlib_ct_build_alg_nd` and `hlib_bct_build`:

```
18   hlib_cluster_t ct = hlib_ct_build_alg_nd( S, & info );         CHECK_INFO;
19   hlib_ct_print_ps( ct, "ct_nd.ps", & info );                    CHECK_INFO;
20
21   hlib_blockcluster_t bct = hlib_bct_build( ct, ct, & info );    CHECK_INFO;
22   hlib_bct_print_ps( bct, "bct_nd.ps", & info );                 CHECK_INFO;
```

At line numbers 19 and 22, the two trees are printed to the files `ct_nd.ps` and `bct_nd.ps`, respectively.

Now the sparse matrix `S` can be converted to an $\mathcal{H}$-matrix, which then is factorised into LU factors with a blockwise precision of 10^{-4}.

Remark

Usually, when converting a sparse matrix, the precision parameter does not apply since the admissibility condition ensures that admissible matrix blocks do not contain non-zero matrix coefficients. Albeit, in some cases it is still possible that these matrix blocks are non-empty and a low-rank approximation will be computed.

The matrix `A` is overwritten with its LU factorisation, or, to be precise, with the inverse of its LU factorisation, although the inverse itself is not computed. Both matrices are printed in PostScript format at the lines 24 and 28. In the case of the factorised matrix, the singular value decomposition of the matrix is printed as chosen by the parameter `HLIB_MATIO_SVD`. The size of the LU factors is determined by `hlib_matrix_bytesize` and printed at line 30.

```
23   hlib_matrix_t A = hlib_matrix_build_sparse( bct, S, & info ); CHECK_INFO;
24   hlib_matrix_print_ps( A, "A_nd.ps", HLIB_MATIO_PATTERN,
25                         & info );                                 CHECK_INFO;
26
27   hlib_matrix_inv_lu( A, 1e-4, & info );                         CHECK_INFO;
28   hlib_matrix_print_ps( A, "LU_nd.ps", HLIB_MATIO_SVD, & info );CHECK_INFO;
29
30   printf( "  size of LU factor = \%.2f MB\n",
31        ((double) hlib_matrix_bytesize( A, & info )) / (1024.0 *1024.0) );
```

The above equation system can now be solved with the inverse of the LU factorisation of `A` as a preconditioner:

```
32  hlib_solve_precond( S, A, x, b, NULL, & info );               CHECK_INFO;
```

Again, no object for returning informations about the solution process was supplied.

Finally, the locally created objects, e.g. the cluster tree, the block cluster tree and the $\mathcal{H}$-matrix can be deleted:

```
33  hlib_matrix_free( A, & info );                                 CHECK_INFO;
34  hlib_bct_free( bct, & info );                                  CHECK_INFO;
35  hlib_ct_free( ct, & info );                                    CHECK_INFO;
```

Matrix Inversion for Preconditioning

An alternative procedure for the fast solution of the above system is by using matrix inversion . Again, before the matrix can be inverted, a cluster tree and a block cluster tree have to be constructed. Nested dissection, as it was used for LU decomposition, is not a suitable technique for matrix inversion (see Section 4.2.2). Hence, standard algebraic partitioning functions employing bisection techniques are used with the function `hlib_ct_build_alg`:

```
36  hlib_cluster_t ct = hlib_ct_build_alg( S, & info );            CHECK_INFO;
37  hlib_ct_print_ps( ct, "ct.ps", & info );                       CHECK_INFO;
38
39  hlib_blockcluster_t bct = hlib_bct_build( ct, ct, & info );    CHECK_INFO;
40  hlib_bct_print_ps( bct, "bct.ps", & info );                    CHECK_INFO;
```

Converting the sparse matrix into an $\mathcal{H}$-matrix is done by the same function as before. Only the matrix inversion is now performed by `hlib_matrix_inv`:

```
41  hlib_matrix_t A = hlib_matrix_build_sparse( bct, S, & info ); CHECK_INFO;
42  hlib_matrix_print_ps( A, "A.ps", HLIB_MATIO_PATTERN, & info );CHECK_INFO;
43
44  hlib_matrix_inv( A, 1e-4, & info );                            CHECK_INFO;
45  hlib_matrix_print_ps( A, "I.ps", HLIB_MATIO_SVD, & info );     CHECK_INFO;
46
47  printf( "  size of Inverse = \%.2f MB\n",
48       ((double) hlib_matrix_bytesize( A, & info )) / (1024.0 *1024.0) );
```

Again, `A` is overwritten by its inverse matrix. Both matrices are also printed in PostScript format, whereby for the inverse matrix the singular value decomposition of each matrix block is written.

The linear equation system is then solved as in the previous case:

```
49  hlib_solve_precond( S, A, x, b, NULL, & info );               CHECK_INFO;
```

Again, all locally created objects are released when they are no longer needed:

```
50  hlib_matrix_free( A, & info );                                 CHECK_INFO;
51  hlib_bct_free( bct, & info );                                  CHECK_INFO;
52  hlib_ct_free( ct, & info );                                    CHECK_INFO;
```

Finalisation

Finally, all global objects which where created at the beginning should be freed. Furthermore, $\mathcal{H}$-Lib^{pro} is finished.

```
53    hlib_vector_free( x, & info );                                CHECK_INFO;
54    hlib_vector_free( b, & info );                                CHECK_INFO;
55    hlib_matrix_free( S, & info );                                CHECK_INFO;
56
57    hlib_done( & info );                                          CHECK_INFO;
```

3 General Functions and Data types

3.1 Initialisation and Finalisation

Before using any functions, $\mathcal{H}$-Lib^{pro} has to be initialised to set up internal data. In the same way, when $\mathcal{H}$-Lib^{pro} is no longer needed, it should be finished. Both is done by the following functions:

Syntax

```
void hlib_init ( int * argc, char *** argv, int * info );
void hlib_done ( int * info );
```

Arguments

`argc`
: Number of command line arguments.

`argv`
: array of strings containing the command line arguments.

`info`
: to return error status

The function `hlib_init` expects the command line parameters `argc` and `argv` for the program as arguments and initialises $\mathcal{H}$-Lib^{pro}. The values of `argc` and `argv` might be modified by `hlib_init`. Accordingly, `hlib_done` finishes $\mathcal{H}$-Lib^{pro}. The meaning of the argument `info` will be discussed in the next section.

Remark

The license management is also performed by `hlib_init` and `hlib_done`, e.g. a license is acquired during initialisation and released during finalisation. Without this, most functions in $\mathcal{H}$-Lib^{pro} will not work.

Normally, $\mathcal{H}$-Lib^{pro} does not produce any output at the console. This behaviour can be changed with

Syntax

```
void hlib_set_verbosity ( const unsigned int verb );
```

Here, larger values correspond to more output, e.g. error messages, timing information, algorithmic details.

3.2 Error Handling

Almost all functions in the C interface of $\mathcal{H}$-Lib^{pro} expect a pointer to an integer, usually named `info`, which is used to indicate the status of the function, i.e. whether an error occurred and

what kind of error this was. If info points to NULL, it will not be accessed and no information about errors will be delivered to the user.

Alternatively, the user can specify an error function which will be called. See below on how to use this feature.

The following lists contain all error codes of $\mathcal{H}$-Lib^{pro}:

General Errors	
HLIB_NO_ERROR	no error occured
HLIB_ERR_INIT	not initialised
HLIB_ERR_LICENSE	invalid license
HLIB_ERR_NOT_IMPL	functionality not implemented
HLIB_ERR_CONSISTENCY	general consistency error
HLIB_ERR_COMM	communication error
HLIB_ERR_PERM	permission denied
Numerical Errors	
HLIB_ERR_REAL	data is real valued
HLIB_ERR_NREAL	data is not real valued
HLIB_ERR_COMPLEX	data is complex valued
HLIB_ERR_NCOMPLEX	data is not complex valued
HLIB_ERR_DIV_ZERO	division by zero
HLIB_ERR_NEG_SQRT	sqrt of negative number
HLIB_ERR_INF	infinity occured
HLIB_ERR_NAN	not-a-number occured
HLIB_ERR_NCONVERGED	iteration did not converge
Memory, Datasize and Argument Errors	
HLIB_ERR_ARG	error with argument
HLIB_ERR_MEM	insufficient memory available
HLIB_ERR_NULL	unexpected null pointer encountered
HLIB_ERR_SIZE	size of data incorrect
HLIB_ERR_DIM	invalid or incompatible dimension
HLIB_ERR_ARR_BOUND	out-of-bound error in array
HLIB_ERR_DIAG_ENTRY	entry is not on diagonal
Coordinate Errors	
HLIB_ERR_COORD_INVALID	invalid coordinates
Cluster Tree Errors	
HLIB_ERR_CT_INVALID	invalid cluster tree
HLIB_ERR_CT_TYPE	wrong type of cluster tree
HLIB_ERR_CT_STRUCT	invalid structure of cluster tree
HLIB_ERR_CT_INCOMP	given cluster trees are incompatible
HLIB_ERR_CT_SPARSE	missing sparse matrix for given cluster tree
Block Cluster Tree Errors	
HLIB_ERR_BCT_INVALID	invalid block cluster tree
HLIB_ERR_BCT_STRUCT	invalid block cluster tree structure
Vector Errors	
HLIB_ERR_VEC_INVALID	invalid vector
HLIB_ERR_VEC_TYPE	wrong vector type
HLIB_ERR_VEC_STRUCT	invalid vector structure
HLIB_ERR_VEC_SIZE	invalid size of vector
HLIB_ERR_VEC_INCOMP	vector with incompatible dimension
HLIB_ERR_VEC_NSCALAR	vector is not a scalar vector
Matrix Errors	

HLIB_ERR_MAT_TYPE	invalid matrix type
HLIB_ERR_MAT_STRUCT	invalid structure of matrix
HLIB_ERR_MAT_SIZE	invalid size of matrix
HLIB_ERR_MAT_SINGULAR	singular matrix detected
HLIB_ERR_MAT_NSPARSE	matrix not a sparse matrix
HLIB_ERR_MAT_NDENSE	matrix not a dense matrix
HLIB_ERR_MAT_NHMAT	matrix not an H-matrix
HLIB_ERR_MAT_INCOMP_TYPE	matrices with incompatible type
HLIB_ERR_MAT_INCOMP_CT	matrices with incompatible cluster tree
HLIB_ERR_MAT_INVALID	invalid matrix
HLIB_ERR_MAT_NSYM,	matrix not symmetric
HLIB_ERR_MAT_NHERM,	matrix not hermitian
HLIB_ERR_MAT_NPOSDEF,	matrix not positiv definite
File Format Errors	
HLIB_ERR_FMT_UNKNOWN	detected unkown file format
HLIB_ERR_FMT_HFORMAT	error while parsing HLIBpro format
HLIB_ERR_FMT_SAMG	error while parsing SAMG format
HLIB_ERR_FMT_MATLAB	error while parsing Matlab format
HLIB_ERR_FMT_PLTMG	error while parsing PLTMG format
HLIB_ERR_FMT_HB	error while parsing Harwell Boeing format
HLIB_ERR_FMT_MTX	error while parsing MatrixMarket format
HLIB_ERR_FMT_PLY	error while parsing Ply format
HLIB_ERR_GRID_FORMAT	invalid format of grid file
HLIB_ERR_GRID_DATA	invalid data in grid file
File I/O Errors	
HLIB_ERR_FOPEN	could not open file
HLIB_ERR_FCLOSE	could not close file
HLIB_ERR_FWRITE	could not write to file
HLIB_ERR_FREAD	could not read from file
HLIB_ERR_FSEEK	could not seek in file
HLIB_ERR_FNEXISTS	file does not exists
Bytestream Errors	
HLIB_ERR_BS_SIZE	size of bytestream too small
HLIB_ERR_BS_WRITE	error while writing to bytestream
HLIB_ERR_BS_READ	error while reading from bytestream
HLIB_ERR_BS_TYPE	type error in bytestream
HLIB_ERR_BS_DATA	general data error in bytestream
Library Support Errors	
HLIB_ERR_NOZLIB	no zlib support compiled in
HLIB_ERR_ZLIB_UNZIP	error during zlib uncompression
HLIB_ERR_NOMETIS	no METIS support compiled in
HLIB_ERR_NOSCOTCH	no Scotch support compiled in
HLIB_ERR_SCOTCH	error in call to Scotch function
HLIB_ERR_NOCHACO	no Chaco support compiled in
HLIB_ERR_NOFFTW3	no FFTW3 support compiled in
Data Errors	
HLIB_ERR_SOLVER_INVALID	invalid solver
HLIB_ERR_LRAPX_INVALID	invalid low-rank approximation type
HLIB_ERR_GRID_INVALID	invalid grid
HLIB_ERR_FNSPACE_INVALID	invalid function space

To get detailed information about the error, e.g. where it occurred inside $\mathcal{H}$-Lib^{pro}, you can use the following function:

Syntax

```
void hlib_error_desc ( char * desc, const unsigned int size );
```

Arguments

`desc`
  Character array to copy description to

`size`
  Size of character array `desc` in bytes.

which returns a corresponding string.

By default, only the error codes will be returned by functions in $\mathcal{H}$-Lib^{pro}. To also produce a corresponding message at the current terminal, the verbosity level has to be increased to at least 2.

Instead of supplying an integer, the user can also define an error function of type

```
typedef void * hlib_errorfn_t ( const int    errcode,
                                const char * errmsg );
```

The arguments `errcode` contains the corresponding code of the error as defined above, whereas `errmsg` is a string with the complete error message. The user supplied error function is defined by

Syntax

```
void hlib_set_error_fn ( const hlib_errorfn_t errorfn );
```

The user can also define an error function and supply an `info` pointer to each function.

3.3 Data types

Most data types in $\mathcal{H}$-Lib^{pro} are defined as *handles*, implemented by pointers, to the actual data. This applies to cluster trees (Section 4.2), block cluster trees (Section 4.3), matrices (Section 5.2). Although they are pointers, a user must not use standard C functions like `malloc` or `free` to allocate or deallocate the associated memory (see also Section 3.4).

$\mathcal{H}$-Lib^{pro} supports single and double precision in the $\mathcal{H}$-arithmetics, albeit not both at the same time. The choice between these has to be done during configuration (see [Kri, Section 2]). All functions in the C interface requiring real valued data are defined using

```
typedef double  hlib_real_t;
```

as the corresponding data type. Exceptions to this rule are for instance doordinate informations which are always supplied by double precision data.

Furthermore, since $\mathcal{H}$-Lib^{pro} is also capable of handling complex arithmetic, a special data type

```
typedef struct { hlib_real_t re, im; } hlib_complex_t;
```

is introduced to allow the exchange of information between an application and $\mathcal{H}$-Lib^{pro}. Here, `re` represents the real part of a complex number whereas `im` corresponds to the imaginary part.

3.4 Reference Counting

Most objects in $\mathcal{H}$-Lib[pro], e.g. cluster trees, block cluster trees and matrices, might be referenced by more than one variable. As an example, a block cluster tree always stores the defining row and column cluster trees (see Section 4.3).

To efficiently handle these references, inside $\mathcal{H}$-Lib[pro] *reference counting* is used, i.e. each object stores the number of references to it. Due to this, a copy operation is done by just increasing this reference counter without any further overhead. This also means, that you must use the functions provided by $\mathcal{H}$-Lib[pro] to free objects.

Attention

To repeat it again: never directly release an object, e.g. via `free(void *)`, since it might be shared by other objects leading to an undefined behaviour of the program.

The usage of the $\mathcal{H}$-Lib[pro]-routines also has the advantage, that some further checks are performed to test whether an object was already released or not. This means that instead of an undefined program behaviour an error is generated if you want to access an object previously freed.

3.5 Parallel Computing

$\mathcal{H}$-Lib[pro] supports parallel computing on shared memory systems with multiple threads[1]. For this, $\mathcal{H}$-Lib[pro] creates a pool of threads during initialisation which are later used for concurrent tasks, e.g. matrix multiplication or inversion. The user can define the number of threads to use by setting the environment variable `HLIB_NTHREADS` or by the function

Syntax

```
void hlib_set_nthreads ( const unsigned int p );
```

Arguments

`p`
Number of threads to use during $\mathcal{H}$-matrix arithmetics.

Remark

The value supplied by `hlib_set_nthreads` overrides the value defined by `HLIB_NTHREADS`.

[1]Distributed memory computations via MPI are not yet available for the C interface.

4 Coordinates and Cluster Trees

$\mathcal{H}$-matrices are based on two basic building blocks: *cluster trees* and *block cluster trees*. A cluster tree defines a hierarchical decomposition of an indexset, whereas a block cluster tree represents a decomposition of a block indexset. Both objects have to be created before building an $\mathcal{H}$-matrix.

Although not necessary, the typical way to construct cluster trees and block cluster trees involves geometrical information about indices which are stored in *coordinates*.

4.1 Coordinates

Most applications using $\mathcal{H}$-matrices will have geometrical data associated with unknowns. This data forms the basis for the clustering routines described in Section 4.2. To let $\mathcal{H}$-Lib^{pro} know about the coordinates, they have to be imported into an internal data type:

```
typedef struct hlib_coord_s * hlib_coord_t;
```

The fundamental import function is `hlib_coord_import` :

Syntax

```
hlib_coord_t
hlib_coord_import ( const unsigned int n,
                    const unsigned int dim,
                    double **          coord,
                    const double *     period,
                    int *              info );
```

Arguments

`n`
: Number of coordinates.

`dim`
: Spatial dimension each coordinate.

`coord`
: Array of size `n` pointers to coordinates of dimension `dim`. Coordinate i, e.g. `coord[`i`]` has to correspond to the i'th unknown.

`period`
: Vector of dimension `dim` defining the periodicity of the coordinates, or `NULL`.

In the following example, four coordinates for a 2-dimensional problem with indices at the corners of the unit square are imported to $\mathcal{H}$-Lib^{pro}:

```
double         pos[4][2] = { { 0, 0 }, { 1, 0 }, { 1, 1 }, { 0, 1 } };
double *       coord[4]  = { pos[0], pos[1], pos[2], pos[3] };
hlib_coord_t   hcoord    = NULL;

hcoord = hlib_coord_import( 4, 2, coord, NULL, & info );
```

Remark

The data in the coordinate array is not copied by $\mathcal{H}$-Lib[pro]. Therefore, any changes to this array will also affect the behaviour of the corresponding $\mathcal{H}$-Lib[pro] functions.

One can also specify a periodicity for the coordinates. By supplying a non-NULL array period, the coordinates are assumed to repeat every period[i] step in the i'th spatial direction. If period[i] = 0, no periodicity is given. The above example can be modified to have x-periodicity by

```
double  period[2] = { 1, 0 };

hcoord = hlib_coord_import( 4, 2, coord, & period, & info );
```

To access a specific coordinate in a $\mathcal{H}$-Lib[pro] coordinate variable the following functions is available:

Syntax

```
double * hlib_coord_get ( const hlib_coord_t  coord,
                          const unsigned int  i,
                          int *               info );
```

Furthermore, the two functions below return the number of coordinates and their spatial dimension

Syntax

```
unsigned int hlib_coord_size ( const hlib_coord_t coord, int * info );
unsigned int hlib_coord_dim  ( const hlib_coord_t coord, int * info );
```

4.1.1 Coordinate Management Functions

Releasing coordinate variables is done via hlib_coord_free.

Syntax

```
void hlib_coord_free ( hlib_coord_t coord, int * info );
```

If the coordinate data was provided by the user, these arrays are not freed from memory. Only if $\mathcal{H}$-Lib[pro] has created the coordinates, e.g. via I/O (see below), memory deallocation is performed.

The memory consumption of a coordinate field is obtained by

Syntax

```
unsigned long hlib_coord_bytesize ( const hlib_coord_t coord, int * info );
```

4.1.2 Coordinate I/O

Coordinates can be read and written in the $\mathcal{H}$-Lib^{pro}-format by using

Syntax
```
hlib_coord_t
hlib_hformat_load_coord ( const char *         filename,
                          int *                info );

void
hlib_hformat_save_coord ( const hlib_coord_t  coord,
                          const char *         filename,
                          int *                info );
```

Furthermore, coordinates stored in the SAMG format can be imported to $\mathcal{H}$-Lib^{pro}. The corresponding function is

Syntax
```
hlib_coord_t hlib_samg_load_coord ( const char * filename, int * info );
```

Also, a general function is provided which tries to autodetect the file format used to hold the coordinates:

Syntax
```
hlib_coord_t hlib_load_coord ( const char * filename, int * info );
```

4.2 Cluster Trees

Inside $\mathcal{H}$-Lib^{pro} cluster trees are represented by objects of type

```
typedef struct hlib_cluster_s * hlib_cluster_t;
```

and can be created in various ways according to the type of data supplied to $\mathcal{H}$-Lib^{pro}.

4.2.1 Cluster Tree Management Functions

To safely free all resources coupled with a cluster tree the following function can be used.

Syntax
```
void hlib_ct_free ( hlib_cluster_t ct, int * info );
```

The object `ct` and all coupled resources are freed from memory unless the cluster tree is used by another object.

Of interest is also the amount of memory used by the cluster tree. This information can be obtained by the function

Syntax
```
unsigned long hlib_ct_bytesize ( const hlib_cluster_t ct, int * info );
```

This function returns the size of the memory footprint in bytes.

4.2.2 Cluster Tree Construction

Two different methods are available to build a cluster tree. The first algorithm is based on geometrical data associated with each index, e.g. the position of the unknown, and uses *binary space partitioning* to decompose the indexset. If no geometry information is available, the connectivity information between indices defined by a sparse matrix can be used in a purely algebraic method. Furthermore, both algorithms can be combined with *nested dissection*, which introduces another level of separation between neighbouring indexsets and is especially suited for LU decomposition methods (see Section 6.4).

Functions for Geometrical Clustering

Geometrical clustering is based on binary space partitioning which is either performed with respect to the cardinality or the geometrical size of the resulting sub-clusters. The detection of a separating interface between two neighbouring indexsets by the nested dissection technique is accomplished by the connectivity information defined by a sparse matrix and therefore does not need geometrical information.

Remark

For the geometrical clustering, only the positions of the indices are needed, e.g. no grid or other management data.

The following two functions perform the geometrical clustering with or without nested dissection:

Syntax

```
hlib_cluster_t hlib_ct_build_bsp    ( const hlib_coord_t   coord,
                                      int *                info );

hlib_cluster_t hlib_ct_build_bsp_nd ( const hlib_coord_t   coord,
                                      const hlib_matrix_t  S,
                                      int *                info );
```

Arguments

`coord`
: Coordinates for each index in the indexset.

`S`
: Sparse matrix defining connectivity of the indices.

The type of binary space partitioning can be changed by

Syntax

```
void hlib_set_bsp_type ( const hlib_bsp_t bsp );
```

Arguments

`bsp`
: Defines the strategy used by the binary space partitioning algorithm and can be one of:

HLIB_BSP_AUTO	Automatically decide suitable strategy. This is the default.
HLIB_BSP_GEOM	Partition such that sub clusters have an equal geometrical size.
HLIB_BSP_REGULAR	Use geometrical partitioning with periodically changed splitting axis instead of longest axis.
HLIB_BSP_CARD	Partition such that sub clusters have an (almost) equally sized indexset.

Consider the following example of a 1d problem over the interval $[0, 1]$ with 8 vertices:

0 1 2 3 4 5 6 7

0 **1**

The cluster tree for this example is now build using standard binary space partitioning. For this, the coordinates of the indices have to be defined, before the corresponding function is called:

```
double          pos[8][1] = { { 0./7. }, { 1./7. }, { 2./7. }, { 3./7. },
                              { 4./7. }, { 5./7. }, { 6./7. }, { 7./7. } };
double *        coord[8]  = { pos[0], pos[1], pos[2], pos[3],
                              pos[4], pos[5], pos[6], pos[7] };
hlib_coord_t    hcoord    = hlib_coord_import( 8, 1, coord, NULL, & info );
hlib_cluster_t  ct1       = hlib_ct_build_bsp( hcoord, & info );
```

The resulting tree would then look as follows:

$0, 1, 2, 3, 4, 5, 6, 7$

$0, 1, 2, 3$ $4, 5, 6, 7$

$0, 1$ $2, 3$ $4, 5$ $6, 7$

0 1 2 3 4 5 6 7

Now the cluster tree shall be computed with binary space partitioning and nested dissection. Here, beside the coordinates, the connectivity of the indices is also needed. For this, we assume that geometrically neighboured indices are also algebraically connected, which would result in the following sparsity pattern of a corresponding matrix:

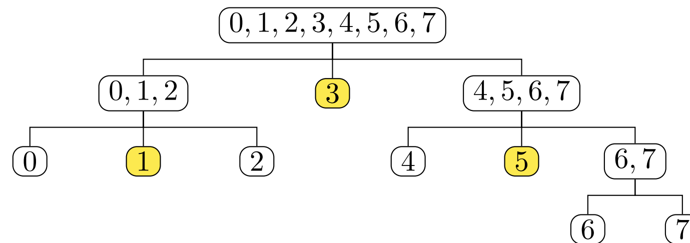
$$\begin{pmatrix}
* & * & & & & & & \\
* & * & * & & & & & \\
& * & * & * & & & & \\
& & * & * & * & & & \\
& & & * & * & * & & \\
& & & & * & * & * & \\
& & & & & * & * & * \\
& & & & & & * & *
\end{pmatrix}$$

For the definition of such a matrix please refer to Section 5.2.1. The corresponding source for this example is:

```
double          pos[8][1] = { { 0./7. }, { 1./7. }, { 2./7. }, { 3./7. },
                              { 4./7. }, { 5./7. }, { 6./7. }, { 7./7. } };
double *        coord[8]  = { pos[0], pos[1], pos[2], pos[3],
                              pos[4], pos[5], pos[6], pos[7] };
hlib_coord_t    hcoord    = hlib_coord_import( 8, 1, coord, NULL, & info );

hlib_matrix_t   S         = /* see chapters below */;
hlib_cluster_t  ct2       = hlib_ct_build_bsp_nd( coord, S, & info );
```

Due to the interface nodes chosen by the nested dissection part of the algorithm, the resulting cluster tree has (mostly) a ternary structure. In the following tree, the coloured nodes correspond to the interface vertices.



Functions for Algebraic Clustering

If no geometrical data is available, an algebraic algorithm can be used, which is based only on the connectivity described by a sparse matrix as it results from finite difference or finite element methods. Since this does not always reflect the real data dependency in a grid, the resulting clustering usually leads to a less efficient matrix arithmetic than the geometrical approach. As in the previous case, the algebraic method can be combined with nested dissection.

Syntax

```
hlib_cluster_t hlib_ct_build_alg    ( const hlib_matrix_t  S, int * info );
hlib_cluster_t hlib_ct_build_alg_nd ( const hlib_matrix_t  S, int * info );
```

Arguments

`S`
: Sparse matrix defining connectivity between indices.

Due to the simple structure of the previous examples the resulting cluster trees using algebraic clustering would be identical.

Userdefined Partitions

In some applications some particular connection exists between special indices, e.g. in coupled systems or when adding special conditions to a matrix, and these indices have to be treated separately. Since $\mathcal{H}$-Lib^{pro} is in general not capable of detecting this kind of grouping between indices, the user is able to give this information when building cluster trees.

For this, the user has to sort the corresponding indices into separate groups by building an array holding the indices for the groups for each index. This array is afterwards supplied to one of the cluster tree functions

Syntax

```
hlib_cluster_t
hlib_ct_build_bsp_part ( const hlib_coord_t   coord,
                         const unsigned int * partition,
                         int *                info );

hlib_cluster_t
hlib_ct_build_alg_part ( const hlib_matrix_t  S,
                         const unsigned int * partition,
                         int *                info );
```

Arguments

`coord,S`
: Coordinates or sparse matrix used for standard geometrical or algebraical clustering.

`partition`
: Array of length n, where n corresponds to the number of coordinates in `coord` or the dimension of the matrix `S`, with `partition[i]` holding the group of index i. The groups have to be numbered consecutively starting from 0.

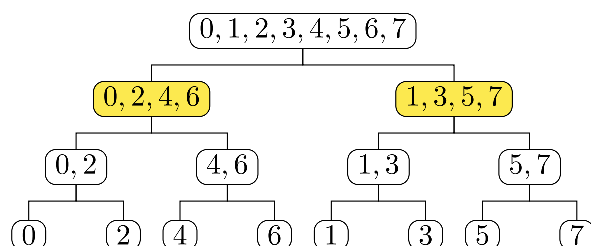
Both functions first sort the indices according to the group information stored in `partition` and will create the first level of the cluster tree by constructing a son node for each group. Afterwards, the indices are further clustered by the standard clustering algorithms, e.g. BSP or algebraically.

As an example, for the index set $I = \{0, \ldots, 7\}$ with corresponding coordinates $i/7$ for index $i \in I$, the indices shall first be separated into odd and even indices before standard BSP clustering is applied.

```
double       pos[8][1]    = { { 0./7. }, { 1./7. }, { 2./7. }, { 3./7. },
                              { 4./7. }, { 5./7. }, { 6./7. }, { 7./7. } };
double *     coord[8]     = { pos[0], pos[1], pos[2], pos[3],
                              pos[4], pos[5], pos[6], pos[7] };
hlib_coord_t hcoord       = hlib_coord_import( 8, 1, coord, NULL, & info );

unsigned int    partition[8] = { 0, 1, 0, 1, 0, 1, 0, 1 };
hlib_cluster_t  ct           = hlib_ct_build_bsp_part( coord, partition,
                                                       & info );
```

The resulting cluster tree `ct` then is:



4.2.3 Cluster Tree I/O

The tree structure of cluster trees can be exported in the PostScript format to a file by

Syntax

```
void hlib_ct_print_ps ( const hlib_cluster_t  ct,
                        const char *          filename,
                        int *                 info );
```

Arguments

`ct`
: Cluster tree to be printed.

`filename`
: Name of the PostScript file to which `ct` shall be printed to.

4.3 Block Cluster Trees

The next building block for $\mathcal{H}$-matrices are *block cluster trees* which represent a hierarchical partitioning of the block indexset over which matrices are defined. They are constructed by multiplying two cluster trees and choosing *admissible* nodes, e.g. nodes where the associated block indexset allows a low-rank approximation in the matrix.

Block cluster trees are represented in $\mathcal{H}$-Lib^{pro} by objects of type

```
typedef hlib_blockcluster_s * hlib_blockcluster_t;
```

4.3.1 Block Cluster Construction

Due to the definition of a block cluster tree two cluster trees are needed for the construction. The actual building is performed by the routine

Syntax

```
hlib_blockcluster_t hlib_bct_build ( const hlib_cluster_t rowct,
                                     const hlib_cluster_t colct,
                                     int *                info );
```

Arguments

`rowct`
: Cluster tree representing the row indexset of the block cluster tree.

`colct`
: Cluster tree representing the column indexset of the block cluster tree.

The cluster trees given to `hlib_bct_build` can be identical.

Usually, the admissibility condition responsible for detecting admissible nodes in the block cluster tree is chosen automatically based on the given cluster trees. The strategy can be changed by the user with the function

Syntax

```
void hlib_set_admissibility ( const hlib_adm_t adm, const hlib_real_t eta );
```

Arguments

adm
: Define admissibility condition to be either:

HLIB_ADM_AUTO	automatic choice,
HLIB_ADM_STD_MIN	standard admissibility with minimal cluster diameter,
HLIB_ADM_STD_MAX	standard admissibility with maximal cluster diameter or
HLIB_ADM_WEAK	weak admissibility.

eta
: Scaling parameter for the distance between clusters in the admissibility condition.

Attention

Since changes to the admissibility condition can lead to a failure of the $\mathcal{H}$-matrix approximation or to a reduced computational efficiency of the $\mathcal{H}$-matrix arithmetic, only change the default behaviour if you really know what you are doing.

To access the row and column cluster trees of a given block cluster tree, one can use the functions

Syntax

```
hlib_cluster_t hlib_bct_row_ct    ( const hlib_blockcluster_t bct, int * info );
hlib_cluster_t hlib_bct_column_ct ( const hlib_blockcluster_t bct, int * info );
```

which will return a copy of the corresponding cluster tree objects.

4.3.2 Block Cluster Tree Management Functions

A block cluster tree object is released by the function

Syntax

```
void hlib_bct_free ( hlib_blockcluster_t bct, int * info );
```

which frees all resources associated with it. This includes the row and column cluster trees if no other object uses them.

The memory footprint of an object of type block cluster tree can be determined by

Syntax

```
unsigned long hlib_bct_bytesize ( const hlib_blockcluster_t bct, int * info );
```

which returns the size in bytes.

4.3.3 Block Cluster Tree I/O

The partitioning of the block index set over which a block cluster tree lives can be written in PostScript format by

Syntax

```
void hlib_bct_print_ps ( const hlib_blockcluster_t  bct,
                         const char *               filename,
                         int *                      info );
```

Arguments

bct
: Block cluster tree to be printed.

filename
: Name of the PostScript file bct will be printed to.

5 Vectors and Matrices

5.1 Vectors

Instead of representing vectors by standard C arrays, $\mathcal{H}$-Lib^pro uses a special data type

```
typedef struct hlib_vector_s * hlib_vector_t;
```

One reason for this is the usage of special vector types in parallel environments. Furthermore, this data type gives $\mathcal{H}$-Lib^pro additional information to check the correctness of vectors, e.g. the size.

5.1.1 Creating and Accessing Vectors

Although vectors in $\mathcal{H}$-Lib^pro are not equal to arrays, they can be defined by such data:

Syntax

```
hlib_vector_t hlib_vector_import_array  ( const hlib_real_t *    arr,
                                          const unsigned int     size,
                                          int *                  info );
hlib_vector_t hlib_vector_import_carray ( const hlib_complex_t * arr,
                                          const unsigned int     size,
                                          int *                  info );
```

Arguments

`arr`
: Array of size `size`.

`size`
: Size of the array.

The arrays are directly used by the vector data type, i.e. changing the content of the arrays also changes the content of the vector. This gives the possibility to efficiently access the vector coefficients as it is shown in the following example:

```
unsigned int   n   = 1024;
hlib_real_t *  arr = (hlib_real_t *) malloc( sizeof(hlib_real_t) * n );
hlib_vector_t  x   = hlib_vector_import_array( arr, n, & info );

for ( i = 0; i < n; i++ )
  arr[i] = i+1;
```

The coefficient i of the vector `x` would also equal $i + 1$ as does the array element `arr[i+1]`.

Such kind of vectors, e.g. scalar vectors, can also be created directly by $\mathcal{H}$-Lib^pro:

Syntax

```
hlib_vector_t hlib_vector_alloc_scalar  ( const unsigned int size, int * info );
hlib_vector_t hlib_vector_alloc_cscalar ( const unsigned int size, int * info );
```

Arguments

`size`
Size of the scalar vector.

Since no external C array is available to access the elements of the vectors, this is accomplished by $\mathcal{H}$-Lib^{pro} functions. To get a specific element of a vector, the following two functions can be used:

Syntax

```
hlib_real_t    hlib_vector_entry_get  ( const hlib_vector_t  x,
                                        const unsigned int   i,
                                        int *                info );
hlib_complex_t hlib_vector_centry_get ( const hlib_vector_t  x,
                                        const unsigned int   i,
                                        int *                info );
```

Arguments

`x`
Vector to get element from.

`i`
Position of the element in the vector.

Similarly, the setting of a vector element is defined:

Syntax

```
void hlib_vector_entry_set  ( const hlib_vector_t   x,
                              unsigned int          i,
                              const hlib_real_t     f,
                              int *                 info );
void hlib_vector_centry_set ( const hlib_vector_t   x,
                              unsigned int          i,
                              const hlib_complex_t  f,
                              int *                 info );
```

Arguments

`x`
Vector to modify element in.

`i`
Position of the element to modify.

`f`
New value of the `i`'th element in `x`.

Two more functions are available to change a complete vector. First, all elements can be set to a given constant value with

Syntax

```
void hlib_vector_fill  ( hlib_vector_t x, const hlib_real_t    f, int * info );
void hlib_vector_cfill ( hlib_vector_t x, const hlib_complex_t f, int * info );
```

Arguments

`x`
: Vector to be filled with constant value.

`f`
: Value to be assigned to all elements of the vector.

Furthermore, the vector can be initialised to random values with

Syntax

```
void hlib_vector_fill_rand ( hlib_vector_t x, int * info );
```

5.1.2 Vector Management Functions

A copy of a vector is constructed by using the function

Syntax

```
hlib_vector_t hlib_vector_copy ( const hlib_vector_t x, int * info );
```

which returns a new vector object with it's own data. If the original vector `v` corresponds to a C array, the newly created vector does not represent this array but uses a new array.

The size of a vector can be determined by the function

Syntax

```
unsigned int hlib_vector_size ( const hlib_vector_t x, int * info );
```

Similarly, the memory size of a vector in bytes is obtained by

Syntax

```
unsigned long hlib_vector_bytesize ( const hlib_vector_t x, int * info );
```

Finally, vectors are released by using

Syntax

```
void hlib_vector_free ( hlib_vector_t x, int * info );
```

which frees all local memory of a vector. This does not apply to associated C arrays, e.g. if the vector was constructed with `hlib_vector_import_array`. There, the array has to be deleted by the user.

5.1.3 Algebraic Vector Functions

A complete set of function for standard algebraic vector operations is available in $\mathcal{H}$-Lib^{pro}.

In contrast to the vector copy function above, the following routine does not create a new vector but copies the content of x to the vector y. For this, both vectors have to be of the same type.

Syntax

```
void hlib_vector_assign ( hlib_vector_t y, const hlib_vector_t x, int * info );
```

Arguments

y
: Destination vector of the assignment.

x
: Source vector of the assignment.

Scaling a vector, e.g. the multiplication of each element with a constant is performed by

Syntax

```
void hlib_vector_scale  ( hlib_vector_t x, const hlib_real_t    f, int * info );
void hlib_vector_cscale ( hlib_vector_t x, const hlib_complex_t f, int * info );
```

Summing up to vectors is implemented in the more general form

$$y := y + \alpha x$$

with vectors x and y and the constant α. This operation is performed by the functions

Syntax

```
void hlib_vector_axpy  ( hlib_vector_t        y,
                         const hlib_real_t    alpha,
                         const hlib_vector_t  x,
                         int *                info );
void hlib_vector_caxpy ( hlib_vector_t        y,
                         const hlib_complex_t alpha,
                         const hlib_vector_t  x,
                         int *                info );
```

Real and complex valued dot-products can be computed with

Syntax

```
hlib_complex_t hlib_vector_dot ( const hlib_vector_t x,
                                 const hlib_vector_t y,
                                 int *               info );
```

And the euclidean and the infinity norm of a vector are returned by the functions

Syntax

```
hlib_real_t hlib_vector_norm2    ( const hlib_vector_t x, int * info );
hlib_real_t hlib_vector_norm_inf ( const hlib_vector_t x, int * info );
```

If support for FFTW3 was compiled in $\mathcal{H}$-Lib^{pro}, forward and backward FFT for vectors is available with the functions

Syntax

```
void hlib_vector_fft  ( const hlib_vector_t v, int * info );
void hlib_vector_ifft ( const hlib_vector_t v, int * info );
```

5.1.4 Vector I/O

In this section, functions for reading and saving vectors from/to files are discussed. Beside it's own format, $\mathcal{H}$-Lib^{pro} supports several other vector formats.

5.1.4.1 $\mathcal{H}$-Lib^{pro}

Vectors can be saved and reloaded in a special $\mathcal{H}$-Lib^{pro} format. There, the specific type of vector, e.g. whether it is distributed among processors, is correctly handled by $\mathcal{H}$-Lib^{pro}.

Syntax

```
hlib_vector_t hlib_hformat_load_vector ( const char *        filename,
                                         int *               info );
void          hlib_hformat_save_vector ( const hlib_vector_t v,
                                         const char *        filename,
                                         int *               info );
```

Arguments

`v`
: Vector to be saved in $\mathcal{H}$-Lib^{pro} format.

`filename`
: Name of the file containing a vector or where the vector shall be written to.

For storing vectors, $\mathcal{H}$-Lib^{pro} uses a binary format but takes care of different computer architectures, e.g. little and big endianess.

5.1.4.2 SAMG

Vectors stored in the SAMG format (see [Fra]) are also supported by $\mathcal{H}$-Lib^{pro}. Since the SAMG format distributes the description of data to several files, beside the actual file, the corresponding *format* file also has to be present in the same directory and with the same *basename*, e.g. without the suffix, as the vector file. Otherwise, the I/O will fail.

To read and store vectors in the SAMG format, the following functions are available:

Syntax

```
hlib_vector_t hlib_samg_load_vector ( const char * filename, int * info );

void          hlib_samg_save_vector ( const hlib_vector_t   x,
                                      const char *          filename,
                                      int *                 info );
```

Due to the restrictions of the SAMG format, the vectors have to be of a scalar type.

5.1.4.3 Matlab

$\mathcal{H}$-Lib^pro supports the Matlab V7 file format (see [Mat]) for dense and sparse vectors. The corresponding vectors can be part of a Matlab *structure*. All other Matlab data types, e.g. cells, are not supported. If *zlib* support was enabled during $\mathcal{H}$-Lib^pro compilation (see [Kri, Section 2]), compressed fields in Matlab files are also supported.

All types of vectors will be converted to scalar vector types upon reading, e.g. sparse vectors become dense. Conversely, only scalar vectors can be saved in the Matlab format.

Since a vector in the Matlab file format is associated with a name, this name has to be supplied to the corresponding I/O functions.

Syntax

```
hlib_vector_t hlib_matlab_load_vector ( const char *        filename,
                                        const char *        vecname,
                                        int *               info );
void          hlib_matlab_save_vector ( const hlib_vector_t v,
                                        const char *        filename,
                                        const char *        vecname,
                                        int *               info );
```

Arguments

`v`
: Scalar vector to be saved in Matlab format.

`filename`
: Name of Matlab file containing the vector.

`vecname`
: Name of the vector in the Matlab file.

5.2 Matrices

All matrices in $\mathcal{H}$-Lib^pro are represented by the type

```
typedef struct hlib_matrix_s * hlib_matrix_t;
```

No difference is made between special matrix types, e.g. sparse matrices, dense matrices or $\mathcal{H}$-matrices. Of course, inside $\mathcal{H}$-Lib^pro this distinction is done and the appropriate or expected type is checked in each function.

To use matrices in $\mathcal{H}$-Lib^pro three different ways are possible: import a matrix given by some data structures, build a matrix or load a matrix from a file. The first method usually applies to sparse and dense matrices whereas $\mathcal{H}$-matrices are normally build by $\mathcal{H}$-Lib^pro. These different methods will be discussed in the following sections.

5.2.1 Importing Matrices from Data structures

Sparse Matrices

Before using sparse matrices in $\mathcal{H}$-Lib^pro, they have to be imported to the internal representation. For this, the sparse matrix is expected to be stored in *compressed row storage* or *CRS* format.

The CRS format consists of three arrays: `colind`, `coeffs` and `rowptr`. The array `colind` holds the column indices of each entry in the sparse matrix ordered according to the row, e.g. at first all indices for the first row, then all indices for the second row and so forth. Here the

indices itself are numbered beginning from 0. In the same way, the array `coeffs` holds the coefficients of the corresponding entries in the same order. Both array have dimension `nnz`, i.e. the number of non-zero entries in the matrix. The last array, `rowptr`, has dimension $n+1$, where n is the dimension of the matrix. It stores at position $i-1$ the index to the `colind` and `coeffs` array for the i'th row, e.g. the entries for the i'th row have the column indices `colind[`$i-1$`]` ... `colind[`i`]-1` and the coefficients `coeffs[`$i-1$`]` ... `coeffs[`i`]-1`. The value `rowptr[`n`]` holds the number of non-zero entries.

As an example, the matrix

$$S = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 2 \end{pmatrix}$$

of dimension 4×4 with 10 non-zero entries would result in the following arrays

```
int          rowptr[5]   = { 0, 2, 5, 8, 10 };
int          colind[10] = { 0,  1,   0, 1,  2,   1, 2,  3,   2, 3 };
hlib_real_t coeff[10]   = { 2, -1,  -1, 2, -1,  -1, 2, -1,  -1, 2 };
```

To import a sparse matrix in CRS format into $\mathcal{H}$-Lib^{pro}, the following two functions can be used:

Syntax

```
hlib_matrix_t hlib_matrix_import_crs  ( const int            rows,
                                        const int            cols,
                                        const int            nnz,
                                        const int          * rowptr,
                                        const int          * colind,
                                        const hlib_real_t  * coeffs,
                                        const int            sym,
                                        int                * info );
hlib_matrix_t hlib_matrix_import_ccrs ( const int            rows,
                                        const int            cols,
                                        const int            nnz,
                                        const int          * rowptr,
                                        const int          * colind,
                                        const hlib_complex_t * coeffs,
                                        const int            sym,
                                        int                * info );
```

Arguments

`rows,cols`
Number of rows and columns of the sparse matrix.

`nnz`
Number of non-zero entries in the sparse matrix.

`rowptr, colind, coeffs`
Arrays containing the sparse matrix in CRS format.

`sym`
If `sym` is non-zero, the sparse matrix is assumed to be symmetric.

The two routines only differ by the coefficient type of the sparse matrix which can either be real or complex valued.

To finish the above described example by creating a matrix object from the constructed arrays, one has to add a call to the corresponding function:

```
int             rowptr[5]  = { 0, 2, 5, 8, 10 };
int             colind[10] = { 0,  1,   0, 1,  2,   1, 2,  3,   2, 3 };
hlib_real_t     coeff[10]  = { 2, -1,  -1, 2, -1,  -1, 2, -1,  -1, 2 };
hlib_matrix_t   S;

S = hlib_import_crs( 4, 10, rowptr, colind, coeffs );
```

Dense Matrices

Although, due to their high memory and computational overhead, dense matrices do not represent the preferred format for matrix storage. Nevertheless, $\mathcal{H}$-Lib^{pro} is also capable of handling dense matrices. As for sparse matrices, they have to be imported for further usage.

When using dense matrices, the user has to keep in mind a very important aspect of their storage:

Attention

In contrast to the standard way in which C addresses dense matrices, $\mathcal{H}$-Lib^{pro} expects them to be in *column major format*, e.g. stored column wise. The matrix coefficient a_{ij} of a $n \times m$ matrix A is therefore at position $j \cdot n + i$ of a corresponding array containing A.

For example, the matrix

$$D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

has to be stored in an array as follows:

```
hlib_real_t  D[4] = { 1, 3, 2, 4 };
```

The reason for this is the usage of LAPACK in $\mathcal{H}$-Lib^{pro}. LAPACK is originally written in Fortran and therefore uses column major format for all matrices.

To import a dense matrix into $\mathcal{H}$-Lib^{pro}, the following two functions for real and complex valued matrices are available:

Syntax

```
hlib_matrix_t hlib_matrix_import_dense  ( const int                rows,
                                          const int                cols,
                                          const hlib_real_t *      D,
                                          const int                sym,
                                          int *                    info );
hlib_matrix_t hlib_matrix_import_cdense ( const int                rows,
                                          const int                cols,
                                          const hlib_complex_t *   D,
                                          const int                sym,
                                          int *                    info );
```

Arguments

`rows, cols`
The number of rows and columns of the dense matrix.

`D`
Array of dimension `rows`·`cols` in column major format containing the matrix coefficients.

`sym`
If `sym` is non-zero, the matrix is assumed to be symmetric.

Importing the above defined matrix D is accomplished by

```
hlib_real_t    D[4] = { 1, 3, 2, 4 };
hlib_matrix_t  M    = hlib_matrix_import_dense( 2, 2, D, 0, & info );
```

5.2.2 Building $\mathcal{H}$-Matrices

Before any $\mathcal{H}$-matrix can be built, a corresponding block cluster tree has to be available, which describes the partitioning of the block indexset of the matrix and defines admissible matrix blocks. Please refer to Section 4.3 on how to get a suitable block cluster tree object.

5.2.2.1 Sparse Matrices

Sparse matrices are converted into $\mathcal{H}$-matrices by using

Syntax

```
hlib_matrix_t hlib_matrix_build_sparse ( const hlib_blockcluster_t bct,
                                         const hlib_matrix_t       S,
                                         const hlib_real_t         eps,
                                         int *                     info );
```

Arguments

`bct`
Block cluster tree defining partitioning of the block indexset of the sparse matrix.

`S`
Sparse matrix to be converted to an $\mathcal{H}$-matrix.

`eps`
Block-wise approximation accuracy of the $\mathcal{H}$-matrix w.r.t. the given sparse matrix.

Usually, the resulting $\mathcal{H}$-matrix is an exact copy of the given sparse matrix and therefore, the parameter `eps` is not used. This usually holds, if the discretisation of the underlying operator of the sparse matrix maintains strong locality conditions, e.g. basis functions with

small support. If this does not the case, e.g. global connectivity between indices, the sparse matrix is *approximated* by the $\mathcal{H}$-matrix. The accuracy of this approximation for each subblock is specified by `eps` (see also next section).

5.2.2.2 Dense Matrices

Since dense matrices involve an unacceptable memory overhead, an $\mathcal{H}$-matrix approximation should not be built out of a given dense matrix but by constructing the data sparse approximation directly. This is accomplished by various algorithms.

Attention

The algorithms for computing a $\mathcal{H}$-matrix approximation of a given dense matrix only work for certain classes of matrices, e.g. coming from integral equations with specific smoothness properties (see ???). They might not work for general matrices. Be sure to check the applicability of the algorithms before using a certain routine. Otherwise quadratic complexity for the storage and cubic complexity for the computation might occur.

Adaptive Cross Approximation

Adaptive cross approximation or *ACA* (see [Beb00]) is a technique which constructs an approximation to a dense matrix by successively adding rank-1 matrices to the final approximation. For this, only the matrix coefficients of the dense matrix are needed. These coefficients are given by the user in terms of a coefficient function which evaluates certain parts of the global dense matrix. The definition of these functions is as follows:

Syntax

```
typedef void (* hlib_coeff_fn_t)  ( int n, int * rowidx, int m, int * colidx,
                                    hlib_real_t * matrix, void * arg );
typedef void (* hlib_ccoeff_fn_t) ( int n, int * rowidx, int m, int * colidx,
                                    hlib_complex_t * matrix, void * arg );
```

Arguments

`n, rowidx`
: Number of row indices and an array containing the row indices at which the matrix shall be evaluated.

`m, colidx`
: Number of column indices and an array containing the column indices at which the matrix shall be evaluated.

`matrix`
: Array of dimension `n·m` at which the computed coefficients at the positions defined by `rowidx` and `colidx` shall be stored in column major format, e.g. coefficient (`rowidx`[i],`colidx`[j]) at position `matrix`[j*n + i].

`arg`
: Optional argument to the coefficient function (see below).

Instead of block coefficient functions of type `hlib_coeff_fn_t` or `hlib_ccoeff_fn_t`, one could also implement a function returning the single matrix coefficient A_{ij}:

```
hlib_real_t coeff ( int i, int j, void * arg ) {
    /* implement computation of A_ij */
```

```
}

void bcoeff ( int n, int * rowidx, int m, int * colidx,
              hlib_real_t * matrix, void * arg ) {
    int  i, j;
    for ( j = 0; j < m; j++ )
        for ( i = 0; i < n; i++ )
            matrix[ j*n + i ] = coeff( rowidx[i], colidx[j], arg );
}
```

The block variant was chosen in $\mathcal{H}$-Lib^{pro}, to allow certain optimisations, e.g. reusage of auxiliary data for the computation of matrix coefficients.

Remark

The given coeffiecient array `matrix` is by default initialised to 0. If additional checks shall be performed, `matrix` is initialised with *NaN* and checked after calling the callback function, whether all entries have been set (see ???).

Different variants of ACA are available in $\mathcal{H}$-Lib^{pro}, each of them with it's own advantages and disadvantages. Interesting, from a computational point of view, are the original formulation and *advanced ACA* (see [BG05]), ACA+ for short. These two have a linear complexity in the dimension of the matrix and a quadratic complexity in the rank of the approximation. This reduced complexity is possible since only a minor part of all coefficients is used inside the algorithms. Unfortunately, this sometimes leads to errors in the approximation and hence, both methods represent a *heuristic* approach. In practise however, at least ACA+ works quite well.

To guarantee a certain approximation, one has to look at all coefficients of the matrix, which then leads to a quadratic complexity in the size of the matrix block. This algorithm is called *ACAFull* (see [BGH03]). Although the approximation can be guaranteed, the resulting rank due to ACAFull might not be minimal, leading to an increase in the memory usage of the resulting $\mathcal{H}$-matrix.

Since an non-optimal rank might also happen with ACA or ACA+, each approximation is truncated afterwards to ensure minimal memory overhead. This truncation procedure has a computational complexity linear in the size of the matrix block and quadratic in the rank.

An optimal rank right from the beginning can be achieved with *singular value decomposition* or *SVD*. Although this algorithm is not directly related to adaptive cross approximation, it is included in $\mathcal{H}$-Lib^{pro}. Unfortunately, SVD has a cubic complexity in the size of the matrix block and is therefore only applicable for small matrices.

After defining a coefficient function, the following two routines can be used to construct a $\mathcal{H}$-matrix approximation to the corresponding dense matrix:

Syntax

```
hlib_matrix_t hlib_matrix_build_coeff  ( const hlib_blockcluster_t bct,
                                         const hlib_coeff_fn_t     f,
                                         void *                    arg,
                                         const hlib_lrapx_t        lrapx,
                                         const hlib_real_t         eps,
                                         const int                 sym,
                                         int *                     info );

hlib_matrix_t hlib_matrix_build_ccoeff ( const hlib_blockcluster_t bct,
                                         const hlib_ccoeff_fn_t    f,
                                         void *                    arg,
                                         const hlib_lrapx_t        lrapx,
                                         const hlib_real_t         eps,
                                         const int                 sym,
                                         int *                     info );
```

Arguments

`bct`
Block cluster tree over which the $\mathcal{H}$-matrix shall be built.

`f`
Coefficient function defining dense matrix.

`arg`
Optional argument which will be passed to the coefficient function.

`lrapx`
Defines the type of low-rank approximation used in each admissible matrix block and can be one of

HLIB_LRAPX_SVD	use singular value decomposition
HLIB_LRAPX_ACA	use adaptive cross approximation
HLIB_LRAPX_ACAPLUS	use advanced adaptive cross approximation
HLIB_LRAPX_ACAFULL	use adaptive cross approximation with full pivot search
HLIB_LRAPX_ZERO	approximate low-rank blocks by zero

`eps`
Block-wise approximation accuracy of the $\mathcal{H}$-matrix w.r.t. the given dense matrix.

Remark

`HLIB_LRAPX_ZERO` can be used to build only the nearfield part of the matrix, since all farfield blocks are left empty.

Since an $\mathcal{H}$-matrix is usually not an exact representation of the dense matrix but only an approximation, the accuracy for this approximation has to be specified. In $\mathcal{H}$-Lib^{pro}, this is always done in a *block-wise* fashion. That means, that for a given dense matrix D and the corresponding $\mathcal{H}$-matrix A build out of D with an accuracy of $\varepsilon \geq 0$ this accuracy only holds for all block indexsets defined by leaves $t \times s$ in the block cluster tree:

$$\|D|_{t\times s} - A|_{t\times s}\| \leq \varepsilon$$

but not necessarily for the matrices itself. This is a general property for all matrix operations in the context of $\mathcal{H}$-matrices.

As an example for building an $\mathcal{H}$-matrix with adaptive cross approximation, we consider the integral equation

$$\int_0^1 \log|x-y|u(y)dy = f(x), \quad x \in [0,1]$$

with a suitable right hand side $f : [0,1] \to \mathbb{R}$. We are looking for the solution $u : [0,1] \to \mathbb{R}$. A standard Galerkin discretisation with constant ansatz functions $\varphi_i, 0 \le i < n$,

$$\varphi_i(x) = \begin{cases} 1 & x \in \left[\frac{i}{n}, \frac{i+1}{n}\right] \\ 0 & \text{otherwise} \end{cases}$$

leads to a linear equation system with the matrix coefficients

$$\begin{aligned} a_{ij} &= \int_0^1 \int_0^1 \varphi_i(x) \log|x-y| \varphi_j(y) dy dx \\ &= \int_{\frac{i}{n}}^{\frac{i+1}{n}} \int_{\frac{j}{n}}^{\frac{j+1}{n}} \log|x-y| dy dx \\ &=: \texttt{integrate_a}(i,j,n), \end{aligned}$$

where integrate_a denotes a function which evaluates the integral.

All of this is put together in the following example. There the coordinates of the indices are set at the centre of the support of each basis function. The function coeff_fn evaluates integrate_a at the given indices and writes the result into the given dense matrix. Please note the access to D in column major form. The actual $\mathcal{H}$-matrix is built using ACA+, which is the recommended variant of adaptive cross approximation.

```
void coeff_fn ( int n, int * rowidx, int m, int * colidx,
                hlib_real_t * D, void * arg ) {
  int   i, j;
  int * n = (int *) arg;

  for ( i = 0; i < n; i++ )
    for ( j = 0; j < m; j++ )
      D[ j*n + i ] = integrate_a( rowidx[i], colidx[j], *n );
}

void build_matrix () {
  int                    i, info;
  int                    n    = 1024;
  double **              pos = (double**) malloc( sizeof(double*) * n );
  hlib_coord_t           coord;
  hlib_cluster_t         ct;
  hlib_blockcluster_t    bct;
  hlib_matrix_t          A;

  for ( i = 0; i < n; i++ ) {
    pos[i]    = (double*) malloc( sizeof(double) );
    pos[i][0] = (((double) i) + 0.5) / ((double) n);
  }
```

```
  coord = hlib_coord_import( n, 1, pos, NULL, & info );
  ct    = hlib_ct_build_bsp( coord, & info );
  bct   = hlib_bct_build( ct, ct, & info );
  A     = hlib_matrix_build_coeff( bct, coeff_fn, & n, HLIB_LRAPX_ACAPLUS,
                                   1e-4, 1, & info );
}
```

Remark

The above example is also implemented in the file `examples/bem1d.c`. There, also source code for the function `integrate_a` is available.

Hybrid Cross Approximation

To be done.

5.2.2.3 Matrix Coarsening

By default, $\mathcal{H}$-matrix will be *coarsend* during construction, i.e. submatrices of a decomposed matrix block will be agglomerated either in a low-rank or a dense matrix. The goal of this technique is the reduction of the memory usage since the newly created matrix block will be deleted if it consumes more memory than the sum of the submatrices and vice versa.

This coarsening strategy involves conversion and truncation and is therefore time consuming. Depending on the restrictions (memory or time) on the problem to compute, the user can either enable or disable this feature by

Syntax

```
void hlib_set_coarsening ( const int build, const int arith );
```

Arguments

`build`
: Activate coarsening during $\mathcal{H}$-matrix construction if `build` is non-zero and deactivate otherwise.

`arith`
: Activate coarsening during $\mathcal{H}$-matrix algebra if `arith` is non-zero and deactivate otherwise (see Section 6.6).

5.2.3 Matrix Management

Using the following two function one can get the number of rows and columns of a specific matrix.

Syntax

```
unsigned int hlib_matrix_rows ( const hlib_matrix_t A, int * info );
unsigned int hlib_matrix_cols ( const hlib_matrix_t A, int * info );
```

The number of rows and columns can be used to constructs vectors of the right dimension for matrix computations, e.g. matrix vector multiplication. An easier and also safer way, since vector do not have to be of scalar type in $\mathcal{H}$-Lib^{pro}, is the usage of the following two functions, which return a vector of compatible format and size either for the row or the column cluster tree of the given matrix `A`:

Syntax

```
hlib_vector_t hlib_matrix_row_vector ( const hlib_matrix_t A, int * info );
hlib_vector_t hlib_matrix_col_vector ( const hlib_matrix_t A, int * info );
```

To obtain a copy of a block cluster tree associated with a matrix, the function

Syntax

```
hlib_blockcluster_t hlib_matrix_bct ( const hlib_matrix_t A, int * info );
```

is available.

In contrast to the copy operations so far, e.g. for cluster trees and block cluster trees, copying a matrix always creates a new matrix. The copy can be either exact or up to a given accuracy. Both operations are done with the functions:

Syntax

```
hlib_matrix_t hlib_matrix_copy     ( const hlib_matrix_t A, int * info );
hlib_matrix_t hlib_matrix_copy_eps ( const hlib_matrix_t A, hlib_real_t eps,
                                     int * info);
```

Arguments

`eps`
    Block-wise accuracy of the copy compared to `A`.

Alternatively, the copy operation can be restricted to the blockdiagonal part of the matrix, e.g. all off-diagonal blocks are omitted. This form of a matrix copy is particularly interesting for preconditioning, if the preconditioner itself has to be only a rough approximation of the inverse but needs to be computed very fast (see also Section 6.4). To control the size of the remaining diagonal blocks, only off-diagonal blocks on the first `lvl` levels are omitted, where `lvl` is defined by the user. The corresponding functions are:

Syntax

```
hlib_matrix_t hlib_matrix_copy_blockdiag     ( const hlib_matrix_t A,
                                               const unsigned int  lvl,
                                               int *               info );
hlib_matrix_t hlib_matrix_copy_blockdiag_eps ( const hlib_matrix_t A,
                                               const unsigned int  lvl,
                                               const hlib_real_t   eps,
                                               int *               info);
```

Arguments

`lvl`
    Number of levels on which off-diagonal blocks shall be omitted from the copy operation.

The methods above will produce new matrix objects. If the content of a matrix shall be copied to an existing matrix, one of the two following functions can be used:

Syntax

```
void hlib_matrix_copyto     ( const hlib_matrix_t A, hlib_matrix_t B, int * info );
void hlib_matrix_copyto_eps ( const hlib_matrix_t A, hlib_matrix_t B,
                              const hlib_real_t eps, int * info );
```

Arguments

`A,B`
Source and destination matrix for the copy operation.

`eps`
Block-wise accuracy of the copy compared to `A`.

In order for the copy operation to be successfull, the format of the matrices `A` and `B` has to be compatible, e.g. $\mathcal{H}$-matrices over the same block cluster tree. An example of an illegal copy would be a dense `A` and a sparse `B`.

Deallocating a matrix is accomplished with

Syntax

```
void hlib_matrix_free ( hlib_matrix_t A, int * info );
```

Again, please remember to only use this function and not `free` to prevent undefined behaviour.

The memory usage of a specific function can be obtained by

Syntax

```
unsigned long hlib_matrix_bytesize ( const hlib_matrix_t A, int * info );
```

In some situations it might be necessary to access single matrix coefficients. For this the following functions are available to return the entry A_{ij}:

Syntax

```
hlib_real_t    hlib_matrix_entry_get  ( const hlib_matrix_t  A,
                                        const unsigned int   i,
                                        const unsigned int   j,
                                        int *                info );
hlib_complex_t hlib_matrix_centry_get ( const hlib_matrix_t  A,
                                        const unsigned int   i,
                                        const unsigned int   j,
                                        int *                info );
```

Remark

For $\mathcal{H}$-matrices obtaining a single coefficient has the complexity $\mathcal{O}(k \log n)$, where k is the maximal rank in the matrix and n the number of rows/columns of A. It should therefore only be used when absolutely necessary.

5.2.4 Matrix Norms

$\mathcal{H}$-Lib^{pro} supports two basic norms for matrices: the *Frobenius* and the *spectral* norm. The Frobenius norm plays a crucial role in the approximation of each matrix block, where the local accuracy is always meant with respect to the Frobenius norm of the local matrix. The spectral

norm, e.g. the largest eigenvalue, gives a better overview of the global approximation, e.g. how good the computed, approximate inverse compares to the exact inverse.

In the case of the spectral norm, the largest eigenvalue is computed by using the *Power iteration* (see [GL96]). Since this is also only an approximate method and due to efficiency the computational effort for computing the norm is restricted in $\mathcal{H}$-Lib^{pro}, the result of this procedure does not necessarily represent the exact spectral norm of the matrix. Although in practise, the convergence behaviour for most matrices is quite well.

Computing the Frobenius and the spectral norm for a single matrix is done by the following two functions:

Syntax

```
hlib_real_t hlib_matrix_norm_frobenius ( const hlib_matrix_t  A, int * info );
hlib_real_t hlib_matrix_norm_spectral  ( const hlib_matrix_t  A, int * info );
```

Furthermore, if the matrix A is not ill-conditioned the spectral norm of A^{-1} can be obtained with

Syntax

```
hlib_real_t hlib_matrix_norm_spectral_inv ( const hlib_matrix_t  A, int * info );
```

To compute the norm of the difference $\|A - B\|$ between two matrices A and B in the Frobenius or the spectral norm, the functions

Syntax

```
hlib_real_t hlib_matrix_norm_frobenius_diff ( const hlib_matrix_t   A,
                                              const hlib_matrix_t   B,
                                              int *                 info );
hlib_real_t hlib_matrix_norm_spectral_diff  ( const hlib_matrix_t   A,
                                              const hlib_matrix_t   B,
                                              int *                 info );
```

are available. Here, in the case of the Frobenius norm, both matrices have to be of the same type and, if they are $\mathcal{H}$-matrices, defined over the same block cluster tree. This does not apply to the spectral norm since it only relies on matrix vector multiplication.

Finally, the approximation of the inverse of a matrix A by another matrix B, e.g. the norm

$$\|I - AB\|_2$$

can be computed by the function

Syntax

```
hlib_real_t hlib_matrix_norm_inv_approx ( const hlib_matrix_t   A,
                                          const hlib_matrix_t   B,
                                          int *                 info );
```

5.2.5 Matrix I/O

Various matrix file formats are supported by $\mathcal{H}$-Lib^{pro} which will be discussed in this section.

5.2.5.1 $\mathcal{H}$-Lib^{pro}

Since $\mathcal{H}$-matrices can not be stored in an efficient way in previously described matrix formats, $\mathcal{H}$-Lib^{pro} defines it's own file format for matrices. This format also supports sparse and dense matrices.

The corresponding functions to load and save matrices from/to files are:

Syntax

```
hlib_matrix_t hlib_hformat_load_matrix ( const char *          filename,
                                         int *                 info );

void          hlib_hformat_save_matrix ( const hlib_matrix_t  A,
                                         const char *          filename,
                                         int *                 info );
```

Arguments

`A`
Matrix to save to `filename`

`filename`
Name of file to load/save matrix from/to.

5.2.5.2 SAMG

The SAMG package (see [Fra]) defines a matrix file format for sparse matrices and vectors in a linear equations system, namely the solution and the right-hand-side. Im- and exporting these objects is supported by $\mathcal{H}$-Lib^{pro}.

The SAMG defines several files together describing the format and data of matrices. The SAMG I/O functions in $\mathcal{H}$-Lib^{pro} only expect the actual data files but presume the format file with the same *basename*, e.g. the filename without a suffix, to be present in the same directory.

Syntax

```
hlib_matrix_t hlib_samg_load_matrix ( const char *          filename,
                                      int *                 info );

void          hlib_samg_save_matrix ( const hlib_matrix_t  S,
                                      const char *          filenamename,
                                      int *                 info );
```

Arguments

`S`
Sparse matrix to store in SAMG format.

`filename`
Name of the matrix file.

5.2.5.3 Matlab

The same restriction, which apply to reading vectors from Matlab files are also valid for matrices, i.e. only dense and sparse matrices either as a single data element or as part of a structure are supported. Compressed fields can only be read, when *zlib* support was chosen during compilation (see [Kri, Section 2]). Other data types, e.g. cells, are not supported and will be skipped. Furthermore, due to the different storage format, $\mathcal{H}$-matrices can not be saved in the Matlab format.

Since a matrix in the Matlab file format is associated with a name, this name has to be supplied to the corresponding I/O functions.

Syntax

```
hlib_matrix_t hlib_matlab_load_matrix ( const char *         filename,
                                        const char *         matname,
                                        int *                info );

void          hlib_matlab_save_matrix ( const hlib_matrix_t  M,
                                        const char *         filename,
                                        const char *         matname,
                                        int *                info );
```

Arguments

`M`
Sparse or dense matrix to save to `filename`

`filename`
Name of Matlab file to load/save matrix from/to.

`matname`
Name of the matrix in the Matlab file.

5.2.5.4 Harwell-Boeing/Harwell-Rutherford

The Harwell-Boeing matrix format defines a file format to store sparse matrices and vectors. With the extended version, Harwell-Rutherford, a variety of other data can also be stored, e.g. coordinates. $\mathcal{H}$-Lib^{pro} supports reading and writing matrices in this format, although the special storage by elementary matrices is not yet supported.

The corresponding function to load matrices from files is:

Syntax

```
hlib_matrix_t hlib_hb_load_matrix ( const char *         filename,
                                    int *                info );
hlib_matrix_t hlib_hb_save_matrix ( const hlib_matrix_t  A,
                                    const char *         filename,
                                    int *                info );
```

Arguments

`A`
Matrix to save to `filename`

`filename`
Name of file to load matrix from.

5.2.5.5 PostScript

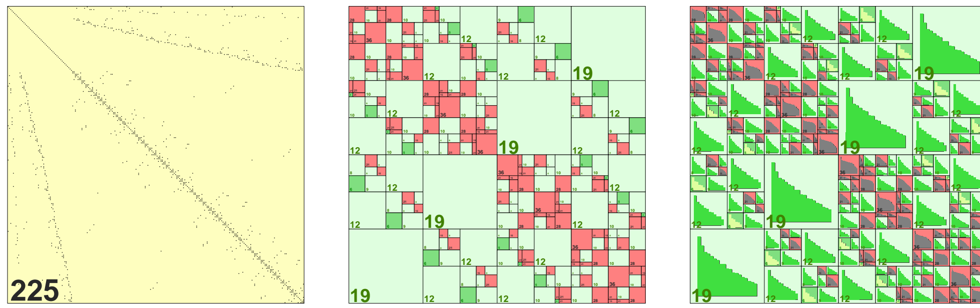
Matrices in $\mathcal{H}$-Lib^{pro} can also be printed in PostScript format. Here various options are available to define the kind of data in the resulting image:

HLIB_MATIO_SVD	print singular values of matrix in logarithmic scale
HLIB_MATIO_ENTRY	print each entry of matrix
HLIB_MATIO_PATTERN	print sparsity pattern (non-zero entries)

These options can also be combined by boolean "or", e.g. to print the SVD and the sparsity pattern the combination HLIB_MATIO_SVD || HLIB_MATIO_PATTERN is used. By default, only the structure of the matrix is printed. There, different kind of blocks are marked by different colours:

- dense matrix blocks
- low-rank matrix blocks
- non-admissible low-rank matrix blocks (see ???)
- sparse matrix blocks
- no matrix block exists, e.g. for symmetric matrices

In addition, special information about each block is printed. For dense and sparse blocks the dimension is shown in the lower left corner. For low-rank blocks, there the rank of the matrix is printed. If SVD is chosen to be printed for each block the actual rank of the matrix is printed at the centre of each block. For low-rank matrices this is only shown, if the SVD-rank differs from the rank of the matrix. An example for a sparse matrix and a dense matrix (with and without SVD) looks like:



The actual PostScript image is finally produced by the function

Syntax

```
void hlib_matrix_print_ps ( const hlib_matrix_t  A,
                            const char *          filename,
                            const int             options,
                            int *                 info );
```

Arguments

A
: Matrix to be printed.

filename
: Name of the PostScript file A shall be printed to.

options
: Combination of MATIO options or 0.

6 Algebra

6.1 Matrix Vector Multiplication

The multiplication of a matrix A with a vector x is implemented in $\mathcal{H}$-Lib^{pro} as the update of the destination vector y as:

$$y := \beta y + \alpha A x.$$

The special cases with $\beta = 0$ or $\alpha = 0$ are also handled efficiently. Furthermore, the multiplication with the adjoint matrix A^H is supported by $\mathcal{H}$-Lib^{pro}. The type of operation is chosen by a parameter of type

```
typedef enum { HLIB_MATOP_NORM,
               HLIB_MATOP_TRANS,
               HLIB_MATOP_ADJ } hlib_matop_t;
```

where HLIB_MATOP_NORM corresponds to Ax, HLIB_MATOP_TRANS to $A^T x$ and HLIB_MATOP_ADJ to $A^H x$.

The following two functions above computation. Both routines can handle real and complex valued matrices and vectors. The difference in the name only applies to the type of the constant factors.

Syntax

```
void hlib_matrix_mulvec  ( const hlib_real_t    alpha,
                           const hlib_matrix_t  A,     const hlib_vector_t x,
                           const hlib_real_t    beta,  hlib_vector_t       y,
                           const hlib_matop_t   matop, int *               info );

void hlib_matrix_cmulvec ( const hlib_complex_t alpha,
                           const hlib_matrix_t  A,     const hlib_vector_t x,
                           const hlib_complex_t beta,  hlib_vector_t       y,
                           const hlib_matop_t   matop, int *               info );
```

Arguments

A
: Sparse, dense or $\mathcal{H}$-matrix to multiply with.

x
: Argument vector of dimension hlib_matrix_cols(A) if Ax is performed and hlib_matrix_rows(A) in the case of $A^H x$.

y
: Result vector of the multiplication of dimension hlib_matrix_rows(A) if Ax is performed and hlib_matrix_cols(A) in the case of $A^H x$.

alpha,beta
: Scaling factors for the matrix-vector product and the destination vector.

matop
: Defines multiplication with A or adjoint matrix of A.

6.2 Matrix Addition

The sum of two matrices A and B in $\mathcal{H}$-Lib^{pro} is defined as

$$B := \alpha A + \beta B$$

with $\alpha, \beta \in \mathbb{R}$ or $\alpha, \beta \in \mathbb{C}$ depending on using real or complex arithmetic.

The two matrices for the matrix addition have to be *compatible*, i.e. of the same format. For example, it is not possible to add a sparse and a $\mathcal{H}$-matrix. Furthermore, $\mathcal{H}$-matrices have to be defined over the same block cluster tree.

When summing up $\mathcal{H}$-matrices, this is done up to a given accuracy. As usual, this accuracy is block-wise. Sparse and dense matrices are always added exactly.

Syntax

```
void hlib_matrix_add  ( const hlib_real_t alpha, const hlib_matrix_t A,
                        const hlib_real_t beta,  hlib_matrix_t       B,
                        const hlib_real_t eps,   int *               info );

void hlib_matrix_cadd ( const hlib_complex_t alpha, const hlib_matrix_t A,
                        const hlib_complex_t beta,  hlib_matrix_t       B,
                        const hlib_real_t    eps,   int *               info );
```

Arguments

`A, B`
: Matrices to be added. The result will be stored in `B`.

`alpha, beta`
: Additional scaling factors for both matrices.

`eps`
: Block-wise accuracy of the addition in the case of $\mathcal{H}$-matrices.

6.3 Matrix Multiplication

Matrix multiplication can only be performed with dense and $\mathcal{H}$-matrices. Furthermore, if the arguments are $\mathcal{H}$-matrices, they have to have compatible cluster trees, e.g. for the product

$$C := A \cdot B$$

the column cluster tree of A and the row cluster tree of B must be identical. This also applies to the row cluster trees of A and C as well as for the column cluster trees of B and C. Otherwise the function exits with a corresponding error code.

The multiplication itself is defined as the update to a matrix C with additional scaling arguments:

$$C := \alpha AB + \beta C.$$

The matrices A and B can also be transposed or conjugate transposed in the case of complex valued arithmetic.

Syntax

```
void hlib_matrix_mul  ( const hlib_real_t  alpha,
                        const hlib_matop_t matop_A, const hlib_matrix_t A,
                        const hlib_matop_t matop_B, const hlib_matrix_t B,
                        const hlib_real_t  beta,    hlib_matrix_t       C,
                        const hlib_real_t  eps,     int *               info );

void hlib_matrix_cmul ( const hlib_complex_t alpha,
                        const hlib_matop_t   matop_A, const hlib_matrix_t A,
                        const hlib_matop_t   matop_B, const hlib_matrix_t B,
                        const hlib_complex_t beta,    hlib_matrix_t       C,
                        const hlib_real_t    eps,     int *               info );
```

Arguments

`A, B`
Dense or $\mathcal{H}$-matrices used as factors for the matrix multiplication.

`matop_A, matop_B`
Defines multiplication with `A`, `B` or the corresponding adjoint matrices $\texttt{A}^H$ and $\texttt{B}^H$.

`C`
Dense or $\mathcal{H}$-matrix containing the result of the matrix multiplication.

`alpha, beta`
Additional scaling arguments for the product and the destination matrix.

`eps`
Block-wise accuracy of the $\mathcal{H}$-arithmetic during the matrix multiplication.

6.4 Matrix Inversion

In $\mathcal{H}$-Lib^{pro}, the inverse of a matrix is computed using Gaussian elimination. This method is implemented for dense and $\mathcal{H}$-matrices, e.g. sparse matrices can not be inverted.

The following functions computes the corresponding inverse to the given matrix `A`, which will be overwritten by the result.

Syntax

```
void hlib_matrix_inv ( hlib_matrix_t       A,
                       const hlib_real_t   eps,
                       int *               info );
```

Arguments

`A`
Dense or $\mathcal{H}$-matrix to be inverted. `A` will be overwritten by the result.

`eps`
Block-wise accuracy of the $\mathcal{H}$-arithmetic during the inversion.

The quality of the computation can be checked by computing the spectral norm of $\|I - AB\|$, with B being the approximate inverse of A, by using function `hlib_matrix_inv_approx` (see Section 5.2.4).

If only the diagonal of the inverse of a given matrix A is of interest, a special function is available in $\mathcal{H}$-Lib^{pro}, which computes this in shorter time and returns the result in the form of a vector:

Syntax

```
hlib_vector_t hlib_matrix_inv_diag ( hlib_matrix_t      A,
                                     const hlib_real_t  eps,
                                     int *              info );
```

Arguments

A
: Dense or $\mathcal{H}$-matrix for which the diagonal of the inverse shall be returned. A will be overwritten during the computation.

eps
: Block-wise accuracy of the $\mathcal{H}$-arithmetic during computation.

Remark

The time for the computation of the diagonal of the inverse can be significantly decreased for $\mathcal{H}$-matrices built upon sparse matrices by using nested dissection (see Section 4.2.2) in the definition of the $\mathcal{H}$-matrix.

An alternative algorithm is the LU decomposition of a matrix, which allows the fast evaluation of the inverse operator. Since not the real inverse is computed, the decomposition can not be used for matrix arithmetic, e.g. matrix multiplication.

Function hlib_matrix_inv_lu computes the LU decomposition and overwrites A with a matrix representing the inverse of the factors, e.g. $(LU)^{-1}$. Therefore, evaluating A afterwards corresponds to A^{-1} and not A.

Syntax

```
void hlib_matrix_inv_lu ( hlib_matrix_t      A,
                          const hlib_real_t  eps,
                          int *              info );
```

Arguments

A
: Dense or $\mathcal{H}$-matrix to be decomposed using LU factorisation. A will be overwritten with the inverse of the factorisation result.

eps
: Block-wise accuracy of the $\mathcal{H}$-arithmetic during the LU decomposition.

Remark

Again, if the $\mathcal{H}$-matrix is defined by a sparse matrix, nested dissection (see Section 4.2.2) can be used to accelerate the computation of the LU factorisation.

6.5 Solving Linear Systems

Solving the linear system

$$Ax = b$$

with the matrix A and the right hand side b is accomplished by the $\mathcal{H}$-Lib^{pro} function

Syntax

```
void hlib_solve  ( const hlib_matrix_t  A,
                   hlib_vector_t        x,
                   const hlib_vector_t  b,
                   hlib_solve_info_t    solve_info,
                   int *                info );
```

Arguments

`A, x, b`
Define linear equation system.

`solve_info`
If not `NULL`, used to return information about solution process.

By default, the exact type of solution technique used to solve the system is chosen by $\mathcal{H}$-Lib^pro depending on the characteristics of the matrix A, e.g. whether it is symmetric or positiv definite. In some cases, the user might want to change this default behaviour and explicitly define the solution algorithm. For this, $\mathcal{H}$-Lib^pro implements various iterative methods: Richardson, CG, BiCG-Stab, MINRES and GMRES iteration (see [Hac93], [vdV92], [PS75] and [SS86]). To modify the iteration algorithm, the following functions are available which make use of the Richardson, the the CG, BiCG-Stab and the MINRES iteration:

Syntax

```
void hlib_solver_richardson ( int * info );
void hlib_solver_cg         ( int * info );
void hlib_solver_bicgstab   ( int * info );
void hlib_solver_minres     ( int * info );
```

In the case of the GMRES-Iteration an additional parameter is expected which describes the dimension of the local Krylov subspace, e.g. when to restart.

Syntax

```
void hlib_solver_gmres ( const int restart, int * info );
```

Arguments

`restart`
Defines the number of iteration steps after which a restart is performed during the GMRES-iteration, e.g. the dimension of the constructed Krylov subspace.

To return to the default solver, e.g. automatic choice, one uses

Syntax

```
void hlib_solver_auto ( int * info );
```

The stopping criterion for the iterative solvers in $\mathcal{H}$-Lib^pro is defined with the function `hlib_solver_stopcrit`:

Syntax

```
void hlib_solver_stopcrit ( const int          maxit,
                            const hlib_real_t abs_red,
                            const hlib_real_t rel_red,
                            int *              info );
```

Arguments

`maxit`
Maximal number of iterations.

`abs_red`
Absolute reduction of the l_2-norm of the residual or negative, if this reduction shall not be checked.

`rel_red`
Relative reduction of the l_2-norm of the residual compared to the initial norm of the residual or negative, if this reduction shall not be checked.

The addition argument `solve_info` to `hlib_solve` can be used to get information about the solution process. The definition of the corresponding type is

```
typedef struct {
  unsigned int  converged;
  unsigned int  steps;
  hlib_real_t   res_norm;
  hlib_real_t   conv_rate;
} hlib_solve_info_t;
```

Here, the field `converged` is 1, if the iteration converged and 0 otherwise. The number of iteration steps is stored in `steps`. Similar, the norm of the final residual and the average convergence rate are put into `res_norm` and `conv_rate` respectively.

Usually, solving a linear systems involves *preconditioning* , e.g. solving the transformed system.

$$WAx = Wb,$$

This is also possible in $\mathcal{H}$-Lib^{pro} and implemented in the function

Syntax

```
void hlib_solve_precond  ( const hlib_matrix_t A, const hlib_matrix_t W,
                           hlib_vector_t       x, const hlib_vector_t b,
                           solve_info_t        solve_info,
                           int *               info );
```

Arguments

`W`
Preconditioner to the linear equation system.

Remark

When solving a preconditioned system, the residual is defined as

$$r = W(Ax - b),$$

e.g. is also preconditioned. This also applies to the corresponding norms supplied by the `solve_info` variable.

6.6 Changing Algebra behaviour

Two aspects of the $\mathcal{H}$-arithmetics can be influenced by the user: the absolute truncation error and the coarsening of matrix blocks.

The absolute truncation error defines the lower bound of the absolute value of the singular values taken into account during the truncation of low-rank blocks, i.e. all singular values and corresponding singular vectors which are smaller than the error bound are omitted. By default, this limit is 0, e.g. no limit is set. In some applications increasing this limit significantly reduces the runtime of $\mathcal{H}$-arithmetics, e.g. inversion or LU factorisation, since the rank of low-rank blocks is reduced without affecting the accuracy. Unfortunately, this is not valid for general problems, especially, if the entries in the matrix vary by a large magnitude.

The absolute truncation error is set for all subsequent $\mathcal{H}$-arithmetic functions by

Syntax

```
void hlib_set_abs_eps ( const hlib_real_t eps );
```

Coarsening of matrix blocks is the technique of replacing subblocks in a matrix by a new low-rank or dense matrix of corresponding size to reduce the memory consumption without affecting the per block approximation properties. This was already applied during matrix construction (see Section 5.2.2) and leads to a significant reduction in memory costs. A similar reduction can also be observed during $\mathcal{H}$-matrix arithmetics. Unfortunately, the coarsening process does not come for free, the agglomeration of matrix blocks involves the trunction of low-rank matrices which, depending on the rank, can be time consuming. Therefore, coarsening is deactivated during $\mathcal{H}$-algebra by default. To activate this feature, one has to call the following function:

Syntax

```
void hlib_set_coarsening ( const int build, const int arith );
```

Arguments

`build`
: Activate coarsening during $\mathcal{H}$-matrix construction if `build` is non-zero and deactivate otherwise.

`arith`
: Activate coarsening during $\mathcal{H}$-matrix algebra if `arith` is non-zero and deactivate otherwise.

7 Miscellaneous Functions

The following functions are mostly included in $\mathcal{H}$-Lib^{pro} for convenience and are usually available by other libraries or even the operating system itself.

7.1 Measuring Time

Two types of time can be measured by $\mathcal{H}$-Lib^{pro}: the CPU time and the wall clock time. The first corresponds to the time spent by the program actually computing things. This type of time has the advantage, that the load on the machine does not have any influence on the value of the time. In contrast to this, the wall clock time is the actual time as measured by a real clock. This type of time is dependent on the load on the computer system and therefore might be different between two runs of the program.

The actual functions to obtain both types are:

Syntax

```
double hlib_walltime ();
double hlib_cputime  ();
```

The absolute value of these functions is normally not usable. Only the difference between two measurements returns the passed time.

7.2 Progress Meter

Depending on the verbosity level chosen by the user, $\mathcal{H}$-Lib^{pro} will print the progress of the computation to the screen. The corresponding information can also be requested by the user to implement a different display of the progress. For this, a callback function has to be provided of type

```
typedef void (* hlib_progressfn_t) ( const double * values,
                                     int *          cancel,
                                     void *         arg );
```

The parameter `value` stores the minimal, the maximal and the current value of the progress, which can be accessed by the constants

```
enum { HLIB_PROGRESS_MIN, HLIB_PROGRESS_MAX, HLIB_PROGRESS_VAL };
```

The second argument allows the user to interrupt the current execution, e.g. matrix construction or LU factorisation, by setting the value of `cancel` to anything different from 0.

Attention

By interrupting a $\mathcal{H}$-Lib^{pro} computation, the state of the result is not defined, e.g. matrices do not contain meaningfull data.

Finally, the fourth argument `arg` is an optional argument by the user supplied to the function

Syntax

```
void hlib_set_progress_cb ( hlib_progressfn_t  fn,
                            void *             arg );
```

Arguments

`fn`
Callback function to be called upon change in the progress of any computation or `NULL` to revert to the default behaviour.

`arg`
Optional argument passed through to the callback function.

which will set `fn` as the new progress function.

7.3 Quadrature Rules

Since $\mathcal{H}$-Lib^{pro} is capable of discretising integral equations, different quadrature rules are implemented as part of the computations. These rules are also exported so that external routines can benefit from them.

7.3.1 Gaussian Quadrature

Quadrature rules for Gaussian quadrature of order n over the interval $[0, 1]$ are constructed by the function

Syntax

```
void
hlib_gauss_quadrature_1d ( const unsigned int order,
                           double * points, double * weights,
                           int    * info );
```

Arguments

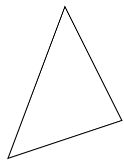
`n`
Order of the quadrature.

`points`
Array of size `order` where the quadrature points will be stored.

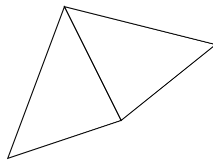
`weights`
Array of size `order` where the quadrature weights will be stored.

7.3.2 Quadrature Rules for Triangles

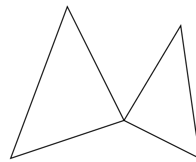
$\mathcal{H}$-Lib^{pro} also provides quadrature rules for the integration over a pair of triangles, e.g. when computing integral equations on a surface grid. These rules were developed by Stefan Sauter (see [SS04]). There different rules apply to different cases of triangle interaction:



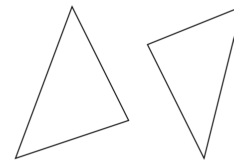
same triangle



common edge

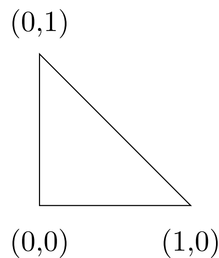


common vertex



separated triangles

The quadrature points are build for each triangle individually, where the triangle itself is the standard 2d simplex



Therefore, you have to transform the computed coordinates to your triangles.

Computing the quadrature rules for equal triangles in done with

Syntax

```
void
hlib_sauter_quadrature_eq ( const unsigned int order,
                            double             * tri1_pts[2],
                            double             * tri2_pts[2],
                            double             * weights,
                            int                * info );
```

Arguments

`order`
Order of the quadrature.

`tri1_pts, tri2_pts`
Array where the 2d quadrature coordinates for both triangles are stored. The array have to be of size $6 \cdot \texttt{order}^4$.

`weights`
Array of size $6 \cdot \texttt{order}^4$ holding the quadrature weights.

Similar defined are the functions for triangles with a common edge, a common vertex or separated triangles:

Syntax

```
void
hlib_sauter_quadrature_edge ( const unsigned int order,
                              double             * tri1_pts[2],
                              double             * tri2_pts[2],
                              double             * weights,
                              int                * info );
```

Arguments

`tri1_pts, tri2_pts, weights`
Arrays of size $5 \cdot \texttt{order}^4$.

Syntax

```
void
hlib_sauter_quadrature_vtx ( const unsigned int order,
                             double             * tri1_pts[2],
                             double             * tri2_pts[2],
                             double             * weights,
                             int                * info );
```

Arguments

`tri1_pts, tri2_pts, weights`
Arrays of size $2 \cdot \texttt{order}^4$.

Syntax

```
void
hlib_sauter_quadrature_sep ( const unsigned int order,
                             double             * tri1_pts[2],
                             double             * tri2_pts[2],
                             double             * weights,
                             int                * info );
```

Arguments

`tri1_pts, tri2_pts, weights`
Arrays of size $\texttt{order}^4$.

Bibliography

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

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

[BGH03] S. Börm, L. Grasedyck, and W. Hackbusch. Hierarchical Matrices. Technical report, Lecture note 21, MPI Leipzig, 2003.

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

[GL96] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 3rd edition, 1996.

[Hac93] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* B.G. Teubner, Stuttgart, 1993.

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

[Kri] Ronald Kriemann. *HLIBpro User manual.* Max-Planck-Institute for Mathematics in the Sciences, Leipzig.

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

[PS75] C.C. Paige and M.A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, September 1975.

[SS86] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Comput.*, 7(3):856–869, 1986.

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

[vdV92] H. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bicg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

Index

A

ACA 4, 38
ACA+ 39
ACAFull 39
adaptive cross approximation 4, 38
addition 50
algebra 49

B

BiCG-Stab 53
binary space partitioning 22

C

CG 53
cluster tree 21
  construction 22
  management 21
coarsening 42, 55
coordinate 19
  I/O 21
  management 20
  periodicity 20

D

data types 16
dot product 32

E

error
  function 16
error handling 13

F

FFT 33
finalisation 13
Frobeniusnorm 44

G

Gaussian quadrature 58
GMRES 53

H

HLIB_NTHREADS 17

I

initialisation 13
inversion 11, 51

L

LU factorisation 7, 10, 51

M

matrix 34
  addition 50
  coarsening 42
  inversion 11, 51
  LU decomposition 51
  multiplication 50
  norm 44
MINRES 53
multiplication 50

N

nested dissection 10, 22, 52
norm
  Frobenius 44
  matrix 44
  spectral 44
  vector 32

P

parallel 17
preconditioning 54
progress meter 57

Q

quadrature 58
  Gaussian 58
  triangles 58

R

reference counting 17
Richardson 53

S

singular value decomposition 6, 39
spectral norm 44
SVD 39

T
threads 17
time
  CPU 57
  wall clock 57
timing 57
triangle quadrature 58

V
vector 29
  dot product 32
  norm 32

Function and Datatype Index

A

hlib_adm_t ... 26

B

hlib_bct_build ... 4, 10, 11, 26
hlib_bct_bytesize ... 27
hlib_bct_column_ct ... 27
hlib_bct_free ... 8, 11, 27
hlib_bct_print_ps ... 4, 10, 11, 27
hlib_bct_row_ct ... 27
hlib_blockcluster_t ... 26
hlib_bsp_t ... 22

C

hlib_ccoeff_fn_t ... 38
hlib_cluster_t ... 21
hlib_coeff_fn_t ... 38
hlib_complex_t ... 16
hlib_coord_free ... 20
hlib_coord_import ... 4, 19
hlib_cputime ... 57
hlib_ct_build_alg ... 11, 24
hlib_ct_build_alg_nd ... 10, 24
hlib_ct_build_alg_part ... 24
hlib_ct_build_bsp ... 4, 22
hlib_ct_build_bsp_nd ... 22
hlib_ct_build_bsp_part ... 24
hlib_ct_bytesize ... 21
hlib_ct_free ... 8, 11, 21
hlib_ct_print_ps ... 4, 10, 11, 25

D

hlib_done ... 8, 11, 13

E

hlib_error_desc ... 4, 16
hlib_errorfn_t ... 16

G

hlib_gauss_quadrature_1d ... 58

H

hlib_hb_load_matrix ... 47
hlib_hformat_load_matrix ... 46
hlib_hformat_load_vector ... 33
hlib_hformat_save_matrix ... 46
hlib_hformat_save_vector ... 33

I

hlib_init ... 3, 9, 13

L

hlib_load_coord ... 21
hlib_load_matrix ... 9
hlib_load_vector ... 9
HLIB_LRAPX_ACAPLUS ... 5
hlib_lrapx_t ... 39

M

HLIB_MATIO_PATTERN ... 9
HLIB_MATIO_SVD ... 6
hlib_matlab_load_matrix ... 47
hlib_matlab_load_vector ... 34
hlib_matlab_save_matrix ... 47
hlib_matlab_save_vector ... 34
hlib_matop_t ... 49
hlib_matrix_add ... 50
hlib_matrix_build_ccoeff ... 39
hlib_matrix_build_coeff ... 5, 39
hlib_matrix_build_sparse ... 10, 11, 37
hlib_matrix_bytesize ... 5, 10, 44
hlib_matrix_cadd ... 50
hlib_matrix_centry_get ... 44
hlib_matrix_cmul ... 50
hlib_matrix_cmulvec ... 49
hlib_matrix_col_vector ... 9, 42
hlib_matrix_cols ... 42
hlib_matrix_copy ... 43
hlib_matrix_copy_blockdiag ... 43
hlib_matrix_copy_blockdiag_eps ... 43
hlib_matrix_copy_eps ... 8, 43
hlib_matrix_copyto ... 43
hlib_matrix_copyto_eps ... 43

hlib_matrix_entry_get . . . 44
hlib_matrix_free . . . 6, 8, 11, 44
hlib_matrix_import_ccrs . . . 35
hlib_matrix_import_cdense . . . 36
hlib_matrix_import_crs . . . 35
hlib_matrix_import_dense . . . 36
hlib_matrix_inv . . . 11, 51
hlib_matrix_inv_diag . . . 51
hlib_matrix_inv_lu . . . 8, 10, 52
hlib_matrix_mul . . . 50
hlib_matrix_mulvec . . . 49
hlib_matrix_norm_frobenius . . . 45
hlib_matrix_norm_frobenius_diff . . . 45
hlib_matrix_norm_inv_approx . . . 8, 45
hlib_matrix_norm_spectral . . . 45
hlib_matrix_norm_spectral_diff . . . 6, 45
hlib_matrix_norm_spectral_inv . . . 45
hlib_matrix_print_ps . . . 5, 8–11, 48
hlib_matrix_row_vector . . . 42
hlib_matrix_rows . . . 42
hlib_matrix_t . . . 34

P

hlib_progressfn_t . . . 57

S

hlib_samg_load_coord . . . 21
hlib_samg_load_matrix . . . 46
hlib_samg_load_vector . . . 33
hlib_samg_save_matrix . . . 46
hlib_samg_save_vector . . . 33
hlib_sauter_quadrature_edge . . . 59
hlib_sauter_quadrature_eq . . . 59
hlib_sauter_quadrature_sep . . . 60
hlib_sauter_quadrature_vtx . . . 60
hlib_set_abs_eps . . . 55
hlib_set_admissibility . . . 26
hlib_set_bsp_type . . . 22
hlib_set_coarsening . . . 42, 55
hlib_set_error_fn . . . 16
hlib_set_nthreads . . . 17
hlib_set_progress_cb . . . 58
hlib_set_verbosity . . . 3, 13
hlib_solve . . . 7, 9, 52
hlib_solve_info_t . . . 54
hlib_solve_precond . . . 8, 10, 11, 54
hlib_solver_auto . . . 53
hlib_solver_bicgstab . . . 53
hlib_solver_cg . . . 53
hlib_solver_gmres . . . 53
hlib_solver_minres . . . 53
hlib_solver_richardson . . . 53
hlib_solver_stopcrit . . . 53

V

hlib_vector_alloc_cscalar . . . 29
hlib_vector_alloc_scalar . . . 29
hlib_vector_assign . . . 32
hlib_vector_axpy . . . 32
hlib_vector_bytesize . . . 31
hlib_vector_caxpy . . . 32
hlib_vector_cfill . . . 30
hlib_vector_copy . . . 31
hlib_vector_cscale . . . 32
hlib_vector_dot . . . 32
hlib_vector_entry_cget . . . 30
hlib_vector_entry_cset . . . 30
hlib_vector_entry_get . . . 30
hlib_vector_entry_set . . . 30
hlib_vector_fft . . . 33
hlib_vector_fill . . . 30
hlib_vector_fill_rand . . . 31
hlib_vector_free . . . 8, 11, 31
hlib_vector_ifft . . . 33
hlib_vector_import_array . . . 7, 29
hlib_vector_import_carray . . . 29
hlib_vector_norm2 . . . 32
hlib_vector_norm_inf . . . 32
hlib_vector_scale . . . 32
hlib_vector_size . . . 31
hlib_vector_t . . . 29
hlib_vector_axpy . . . 7
hlib_vector_copy . . . 7
hlib_vector_fill . . . 7
hlib_vector_norm2 . . . 7

W

hlib_walltime . . . 57