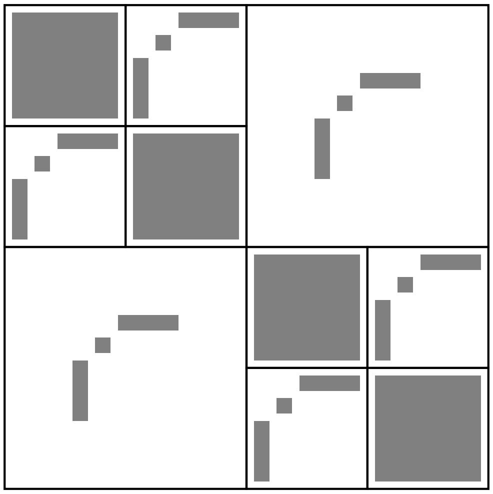
STRuctured **M**atrices **PACK**age User Guide

–

Distributed-Memory Dense Package

François-Henry Rouet*, Xiaoye S. Li*, Pieter Ghysels*

Version 1.1.1, September 2015



---

*Lawrence Berkeley National Laboratory, Computational Research Division, MS 50F-1650, One Cyclotron Road, Berkeley CA94720. `{fhrouet,xsli,pghysels}@lbl.gov`

Contents

1 Introduction

1.1 STRUMPACK

STRUMPACK – STRUctured Matrices PACKage – is a C++ library for computations with sparse and dense matrices. STRUMPACK has presently two main components: a distributed-memory dense matrix computations package and a shared-memory sparse direct solver. A sparse distributed-memory solver is in development. This manual is for the **distributed-memory, dense** component of STRUMPACK, SDP (STRUMPACK Dense Package).

STRUMPACK provides the usual building blocks of matrix computations: factorizations, matrix-vector products, etc. It relies on a kind of *structured matrices* (or *low-rank matrices*) called Hierarchically Semi Separable matrices (HSS). In many applications, such as boundary elements or finite elements, using HSS-based algorithms allows for fast solution of linear systems and/or fast computation of matrix-vector products. By "fast", we mean *faster* than traditional algorithms, e.g., $\mathcal{O}(n^3)$ factorizations and $\mathcal{O}(n^2)$ multiplications for dense matrices.

The main element of the HSS framework is the *compression phase*, i.e. computing an HSS representation of the input matrix. The HSS representation can be exact or approximate; the compression algorithm relies on a threshold that allows users to find the best trade-off between accuracy and performance for their application. Using a large threshold will allow fast operations at the cost of accuracy, while a very low threshold (close to machine precision) will ensure accuracy, at the cost of performance. For some applications, HSS algorithms will provide large speedups (with respect to traditional algorithms) even with low thresholds, while for some applications using a larger threshold is necessary. We provide more detail about this in the next sections.

The STRUMPACK project started at the Lawrence Berkeley National Laboratory in 2014 and is supported by the FASTMath SciDAC Institute funded by the Department of Energy.

1.2 Distributed-Memory Dense Package

This user guide describes the dense component of STRUMPACK, SDP (STRUMPACK Dense Package). SDP offers the following features:

- Compression of a dense matrix into HSS form, using either an explicit matrix or a matrix-free approach.
- Factorization of a dense matrix, either using classical LU (with ScaLAPACK) or in-house ULV factorization for HSS matrices.
- Solution of a linear system, either using classical triangular solution (with ScaLAPACK) or specialized solution algorithm for HSS matrices.
- Matrix-vector product (ditto).
- Partial factorization, computation of a compressed Schur complement, and partial forward/backward solutions.

Using these features, STRUMPACK can easily be used as:

- A standalone direct linear solver,
- A preconditioner with a choice of cost-accuracy trade-off (which can be used in iterative linear solvers or eigensolvers),
- A fast matrix-vector product for iterative linear solvers or eigensolvers.
- A dense kernel to be used within, e.g., a multifrontal solver.

Most of the features (e.g., factorization) can use exact algorithms (based on LAPACK/ScaLAPACK) or the in-house HSS algorithms in STRUMPACK, by simply changing a parameter (no change to function calls or data structures in most cases).

The main input of STRUMPACK is a dense matrix which can be either centralized or distributed following the popular 2D block-cyclic format used in ScaLAPACK and other codes. We provide more detail about this in the next sections.

SDP is described in detail in [2].

1.3 Version history

1.1.1 (09/03/2015)

- Parallel Octave/Matlab interface.
- Improved the Make system and the installation instructions.
- Bug fixes.

1.1.0 (08/27/2015)

- C interface.
- Fortran interface (using ISO C bindings).
- Octave/Matlab® interface (mex files).

1.0.0 (08/14/2015)

- Matrix-free compression.
- Schur complement features: partial factorization, RHS reduction, solution expansion.
- Element extraction from the compressed matrix.

0.9.0 (12/19/2014) First public release.

- Compression of explicit matrices.
- Factorization and triangular solution.
- Matrix-vector product.

2 Installation

2.1 Requirements

The following libraries are necessary:

- MPI.
- ScaLAPACK (and BLACS).
- LAPACK.
- BLAS.

ScaLAPACK, BLACS, LAPACK and BLAS can often be found in bundles, for example in the Intel® MKL® library, the AMD® ACML® library, or the Cray® LibSci® library. They can also be built from scratch fairly easily. A good, high-performance, and freely available (and open source) implementation of BLAS and LAPACK is OpenBLAS (www.openblas.net); the version of ScaLAPACK and BLACS found at `www.netlib.org/scalapack` can be easily built on top of it. The ScaLAPACK Installer (same webpage) can build everything in one go.

The code was tested with the GNU, Intel® and PGI® compilers, the OpenMPI and MPICH MPI libraries, and the OpenBLAS, Intel® MKL® and Cray® LibSci® numerical libraries.

2.2 Running the examples

We recommend running the examples (and reading their source code, which is very concise) in order to become familiar with STRUMPACK.

To run the examples:

1. Go to the examples directory.
2. Create a `Makefile.inc` file following the examples shipped with STRUMPACK (`Makefile.gnu, Makefile.edison`). It defines compilers, flags, and library locations.
3. Compile the examples; you can do:
    - `make` to build everything (or make -j xx for parallel make). To do this you need C++, C, Fortran, and mex (Octave/Matlab) compilers.
    - `make cpp_examples` to build all the C++ examples. To do this you only need a C++ compiler.
    - `make c_example` to build the C example. To do this you need C and C++ compilers.
    - `make f90_example` to build the Fortran example. To do this you need C, C++, and Fortran compilers.
    - `make mex_example` to build the Octave/Matlab example. To do this you need C++ and mex compilers.
    - `make xxx` to build a specific example (e.g., "make solve". Cf. the list of example below).
4. Run the examples: they don't take any input parameters, and they can be run with arbitrary number of MPI ranks; e.g.,

    ```
    mpirun -np 4 ./solve
    ```

The list of examples (Make targets) is the following:

1. `solve` (`solve.cpp`) solves a linear system.

    Features: HSS compression (explicit matrix), ULV factorization, and triangular solution.
2. `matrixfree` (`matrixfree.cpp`) solves a matrix-free linear system.

    Features: matrix-free HSS compression, ULV factorization, and triangular solution.
3. `schur` (`schur.cpp`) solves a linear system with a hybrid approach.

    Features: HSS compression, partial factorization, Schur complement computation, Schur complement element extraction, RHS reduction, solution expansion, and Schur complement matrix-vector product.
4. `power` (`power.cpp`) is a simplistic power method.

    Features: HSS compression and HSS matrix-vector products.
5. `product` (`product.cpp`).

    Features: HSS matrix-vector products.
6. `selection` (`selection.cpp`).

    Features: Element extraction.
7. `communicator` (`communicator.cpp` illustrates the use of STRUMPACK with an MPI subcommunicator.
8. `c_example` (`c_example.c`) illustrates the use of the C interface; it is essentially a C version of solve.cpp.
9. `f90_example` (`f90_example.F90`) illustrates the use of the Fortran interface; it is essentially a Fortran version of solve.cpp.
10. `mex_example` (`mex_example.m`) illustrates the use of the Octave/Matlab interface; it is essentially an Octave/Matlab version of solve.cpp.

2.3 Using STRUMPACK Dense Package within your code.

If your code is in written in C++, you can use the native C++ interface of STRUMPACK Dense Package. The only thing you have to do is to include the main header file (found in the src/ directory):

```
#include "StrumpackDensePackage.hpp"
```

In this situation, there is no object to be linked with your code besides the abovementioned required libraries.

If you wish to use the C interface, you need to include the header file `src/StrumpackDensePackage.h` to your code, and you need to compile (and link with your code) file `StrumpackDensePackage_C.cpp`.

For the Fortran interface, you need to compile two files from the `src/` directory:

1. `StrumpackDensePackage.F90` (Fortran compiler).
2. `StrumpackDensePackage_C.cpp` (C++ compiler).

Then you can use the STRUMPACK Dense Package module inside you code:

```
use StrumpackDensePackage
```

You can to refer to the way target `f90_example` is built in `examples/Makefile`.

2.4 Compilation flags

A few STRUMPACK-specific compiler flags can be used when compiling:

- `-DHQR`: uses Householder RRQR instead of Modified Gram-Schmidt. This requires a Fortran compiler. Files src/*.f need to be compiled and linked against your code. Refer to examples/Makefile and src/Makefile.SDP.
- `-DRANDGEN`: uses a Mersenne Twister random number generator instead of the legacy C "rand" generator. With `-DRANDGENORMAL`, the distribution is normal, otherwise it is uniform. This requires a compiler compliant with C++11 standard.
- `-DWITH_PAPI`: experimental use of the PAPI library (icl.cs.utk.edu/papi/) that uses hardware counters to gather statistics. Statistics provided by PAPI are displayed when print_statistics() is called.
- `-DMEMTRACK`: replaces the standard memory allocators with in-house allocators that allow us to track memory peak usage (displayed when print_statistics() is called). This is *very* experimental and might be unstable on some systems.

3 Algorithms

The different algorithms are described in detail in F.-H. Rouet, P. Ghysels, X. S. Li, A. Napov - *A distributed-memory package for dense Hierarchically Semi-Separable matrix computations*. Here we briefly recall what they do and how they are connected to each other. **The HSS Compression algorithm is the only one that requires parameter tuning by the user.** The others are mostly black-box.

3.1 HSS Compression

3.1.1 Idea

The compression algorithm computes a representation (the HSS form) of a dense matrix; the representation can be approximate or exact. After compression, the HSS representation can be used to perform matrix-vector products, factorizations, etc., and the input matrix is no longer needed.

The HSS representation is computed via *randomized sampling*. The idea is to generate a number of random vectors, generate samples of the row space and column space of the matrix using matrix-vector products, and compute the *HSS representation* (or *factors*; a collection of small dense matrices) of the input matrix. These HSS factors are computed using a *compression* algorithm that relies on a rank-revealing factorization. The quality of the compression can be tuned using a numerical *threshold*.

3.1.2 Variants

STRUMPACK SDP provides three variants of the compression phase, i.e., three ways of passing the input matrix to the compression algorithm:

1. Explicit matrix only: the implementation generates random vectors and computes the samples of the row space and column space using classical matrix-vector products (with LAPACK/ScaLAPACK). The random vectors and samples are not visible to the user.

2. Explicit matrix plus random vectors and samples: the user provides two sets of random vectors R^r and R^c and also provides the samples $S^r = AR^r$ and $S^c = A^*R^c$. The advantage is that users can use their own application-specific matrix-vector product, that may be faster than a traditional $\mathcal{O}(n^2)$ matrix-vector product. However, the user has to make sure that the random vector are “truly” random and that $S^r = AR^r$ and $S^c = A^*R^c$ hold, otherwise the HSS representation will most likely be inaccurate.

3. Matrix-free interface: the user provides random vectors, samples, and a function pointer to a **routine that provides access to arbitrary elements of the input matrix**. We explain this in detail in Section 4.3.3.

3.1.3 Parameters

The following parameters are important and should be tuned by the user for optimal performance:

- **The compression threshold** `tol_HSS`: if it is set very low (e.g., close to machine precision), the compression will be accurate but potentially slow, and the subsequent steps (solving a linear system, computing a matrix-vector product. . .) will also be accurate but potentially slow, i.e. potentially slower than using traditional algorithms (e.g., LU factorization). Conversely, if the threshold is aggressive (e.g., 10^{-2}), the compression and other operations will be fast but potentially inaccurate. This is very application dependant.

- **The sampling parameters**: when user does not provide random vectors and samples, the compression process starts with a set a of `min_rand_HSS` random vectors that are used to generate a sample of the input matrix. Different pieces of this sample are compressed using a rank-revealing factorization. If the rank found during a rank-revealing step is too close to the number of random vectors (the difference is smaller than `lim_rand_HSS`), the number of random vectors is increased by `inc_rand_HSS`. There is a limit `max_rand_HSS` on the number of random vectors that can be used. If `min_rand_HSS` is unnecessarily large, the compression algorithm will be slower than necessary.

  When the user provides samples and random vectors, the compression stops if the rank found during a rank-revealing step is too close to the number of random vectors. It is then the responsibility of the user to increase the sample size and restart the compression. The user can choose to restart the compression from scratch or to the resume the previous compression.

The choice of an algorithm for performing the rank-revealing factorization used at each step of the compression stage also has an influence on performance. The default algorithm is an in-house QR factorization with column pivoting that uses a Modified Gram-Schmidt scheme. By compiling with `-DHQR` it is replaced with an Householder reflections-based implementation that uses modified LAPACK/ScaLAPACK routines. In our experience, using `-DHQR` will improve performance on small number of processes but there should be almost no difference on large number of processes. The reason is that the sequential Householder LAPACK routine is a BLAS3 implementation and is faster than our in-house BLAS1 implementation of the Modified

Gram-Schmidt scheme. However, our parallel MGS implementation is as fast as ScaLAPACK's Householder routine. Therefore, there is a difference only in sequential parts of the compression.

3.2 Factorization

After compression, the matrix is factored using ULV factorization, which is essentially an LU factorization for HSS matrices. This is a black-box procedure, there is no parameter to tune.

3.3 Solution

After factorization, a linear system $Ax = b$ can be solved. This is a black-box procedure, there is no parameter to tune.

3.4 Iterative refinement

The accuracy of the solution can be improved using iterative refinement. The algorithm is essentially the following:

```
$err = \frac{||b-Ax||}{||b||}$;
$steps = 0$;
while $err >$ tol_IR and $steps <$ steps_IR do
    $steps{+}{+}$;
    $r = b - Ax$;
    $dx = A^{-1}r$;
    $x = x + dx$;
    $err = \frac{||b-Ax||}{||b||}$;
end
```

The tolerance (i.e., targeted accuracy) is set using `tol_IR` and the maximum of steps is set using `steps_IR`.

3.5 Accuracy checking

The quality of the HSS representation $\tilde{A}$ of the input matrix A can be checked by computing $||\tilde{A}-A||_F/||A||_F$ using the routine `check_compress`. $\tilde{A}$ is not stored as an explicit $n \times n$ matrix; the explicit form is computed using a matrix-matrix product $\tilde{A} \cdot I$ with I the identity matrix. Note that this is can be very expensive and should not be used for very large problems.

An alternative way to check accuracy can be to generate a few random vectors R and compute $||\tilde{A} \cdot R - A \cdot R||/||A \cdot R||$. This is illustrated in one of the examples shipped with the code.

3.6 Matrix-vector product

A matrix-vector (or matrix-matrix) product can be computed using the compressed form of the input matrix. This is a black-box procedure, there is no parameter to tune. Note that, when several right-hand sides are available simultaneously, we recommend using a single product with a block of vectors instead of multiple products with a single vector. Using a block favors cache reuse and usually improves performance dramatically.

3.7 Partial factorization, Schur complement computation, partial solution

The following partial factorization is useful in many situations:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} P_{11}^{*}L_{11} & 0 \\ A_{21}U_{11}^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & L_{11}^{-1}P_{11}A_{12} \\ 0 & S \end{bmatrix} \tag{1}$$

where S is the *Schur complement* $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$.

To solve a linear system $Ax = b$ using this factorization, one has to solve two block triangular systems:

$$\begin{cases} \begin{bmatrix} P_{11}^{*}L_{11} & 0 \\ A_{21}U_{11}^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} & \text{where } \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b \\ \begin{bmatrix} U_{11} & L_{11}^{-1}P_{11}A_{12} \\ 0 & S \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} & \text{where } \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x \end{cases}$$

The solution proceeds in three steps:

1. **Reduction phase:** (a.k.a *condensation*)

$$\begin{cases} y_1 = L_{11}^{-1}P_{11}b_1 \\ y_2 = b_2 - A_{21}U_{11}^{-1}y_1 \end{cases} \tag{2}$$

2. The Schur complement system (*reduced system*) is solved:

$$x_2 = S^{-1}y_2$$

3. **Expansion phase:**

$$x_1 = U_{11}^{-1}\left(y_1 - L_{11}^{-1}P_{11}A_{12}x_2\right) \tag{3}$$

In this framework, it is the responsibility of the user to solve the Schur complement system. One can use a direct method (maybe using another instance of STRUMPACK SDP), or an iterative method, using the **Schur complement matrix-vector product** provided by STRUMPACK SDP. STRUMPACK SDP can perform all the other operations using either traditional or HSS-based algorithms:

- Partial factorization.
- Schur complement computation.
- Reduction phase.
- Expansion.

One of the examples shipped with the code illustrates this approach; in the example, the reduced system is solved using Conjugate Gradient, relying on the (HSS) matrix-vector product provided by STRUMPACK.

4 User Interface

4.1 Matrix format

Whenever a matrix (or vector) is passed to SDP, it must be in 2D block-cyclic form with an accompanying BLACS descriptor. We refer the reader to [1] for more detail. Users already familiar with ScaLAPACK will have no problems using the routines in SDP; the signatures are similar to those in ScaLAPACK.

If the matrix in your application is centralized, you can use SDP and get (some) parallelism, but most routines will work sequentially. This is detailed for each routine in the following subsections. The best way to use the package is to distribute your matrix in 2D block-cyclic form by using the `PxGEMR2D` routine from ScaLAPACK. We provide a snippet of code to do this:

```
/* We initialize a context with only id 0 */
blacs_get_(&IZERO,&IZERO,&ctxtcent);
blacs_gridinit_(&ctxtcent,"R",&IONE,&IONE);

/* We initialize a context with all the processes */
blacs_get_(&IZERO,&IZERO,&ctxtglob);
blacs_gridinit_(&ctxtglob,"R",&IONE,&np);

/* We initialize a 2D grid of processes that will share A */
nprow=floor(sqrt((float)np));
npcol=np/nprow;
blacs_get_(&IZERO,&IZERO,&ctxt);
blacs_gridinit_(&ctxt,"R",&nprow,&npcol);
blacs_gridinfo_(&ctxt,&nprow,&npcol,&myrow,&mycol);

/* The input is generated in Acent on rank 0 */
if(myid==0) {
  [...]
  descinit_(descAcent,&n,&n,&nb,&nb,&IZERO,&IZERO,&ctxtcent,&n,&ierr);
} else {
  descset_(descAcent,&n,&n,&nb,&nb,&IZERO,&IZERO,&INONE,&IONE);
}

/* Distribute A into 2D block-cyclic form */
if(myid<nprow*npcol) {
  locr=numroc_(&n,&nb,&myrow,&IZERO,&nprow);
  locc=numroc_(&n,&nb,&mycol,&IZERO,&npcol);
  dummy=std::max(1,locr);
  A=new double[locr*locc];
  descinit_(descA,&n,&n,&nb,&nb,&IZERO,&IZERO,&ctxt,&dummy,&ierr);
} else {
  descset_(descA,&n,&n,&nb,&nb,&IZERO,&IZERO,&INONE,&IONE);
}
pgemr2d(n,n,Acent,IONE,IONE,descAcent,A,IONE,IONE,descA,ctxtglob);
delete[] Acent;
```

4.2 StrumpackDensePackage object and initialization

The `StrumpackDensePackage` class is the only class to be used. It is templated/parametrized by two types:

- The type of scalars (input matrix, vectors, etc.). It can be `dcomplex` (`std::complex<double>`, double precision complex), `scomplex` (`std::complex<float>`, single precision complex), `float` (single precision real) or `double` (double precision real).
- The type of reals (for norms, etc.). It can be `float` (single precision) or `double` (double precision).

The only constructor for `StrumpackDensePackage` is

```
StrumpackDensePackage<T,S>::StrumpackDensePackage(MPI_Comm user_comm)
```

The only parameter is an MPI communicator. It can be `MPI_COMM_WORLD` or a subcommunicator created by the user. We provide an example of usage with a subcommunicator; cf. Section 2.2.

4.3 HSS compression

4.3.1 Explicit matrix

```
void compress(T *A, int *descA);
```

Input: matrix A in 2D block-cyclic form with a BLACS descriptor `descA`.

Output: no user-available output. Internally, the `StrumpackDensePackage` object contains the HSS representation of A.

Remarks:

- If A is centralized, most of the work in sequential. The first step in the compression algorithm, and often the most time-consuming, is to compute a sample of A. This is done by the processes that own A, thus, if A is centralized, this is done sequentially. The rest of the compression procedure is done in parallel but the gains will probably be very limited.
- Refer to Section 3.1 to see how to set the parameters associated with the compression routine.
- If HSS is disabled (`StrumpackDensePackage::use_HSS` false), the routine does not do anything.
- If a matrix has been previously compressed, the routine will abort. Another `StrumpackDensePackage` must be used, or the current object must be destroy and re-created.

4.3.2 Explicit matrix, random vectors, and samples

```
void compress(T* A, int *descA, T **Rr, T **Rc, T** Sr, T** Sc, int *descRS);
```

Input: matrix A in 2D block-cyclic form with a BLACS descriptor `descA`; random vectors R^r and R^c, sample vectors S^r and S^c, with a BLACS descriptor `descRS`.

Output: no user-available output. Internally, the `StrumpackDensePackage` object contains the HSS representation of A.

Remarks:

- It is the responsibility of the user to ensure that `Rr` and `Rc` are random vectors, and that `Sr` and `Sc` define samples of the row and column space of A, i.e., $S^r = AR^r$ and $S^c = A^*R^c$.
- **Adaptive sampling is not available in this version. It is the responsibility of the user to check the maximum rank (`max_rank`) against the number of random vectors, increase the sampling size and restart the compression (after reinitializing the SDP object)**.

4.3.3 Matrix-free version

```
void compress(T **Rr, T**Rc, T**Sr, T** Sc, int *descRS, void (*foo)(void*,int*,int*,T*,int*) );
```

Input: Random vectors R^r and R^c, sample vectors S^r and S^c, with a BLACS descriptor `descRS`; function pointer to an element access routine (see below).

Output: no user-available output. Internally, the `StrumpackDensePackage` object contains the HSS representation of A.

The last argument is a pointer to a routine that **must be provided by the user**. It gives STRUMPACK access, on the fly, to selected elements of the input problem. Its parameters are the following:

- `submat` (**output**) is the scalar array containing the requested elements. It has to be filled by the user.

- `desc` (**input**) is the BLACS descriptor for `submat`. Cf. Section 4.1.
- `I` and `J` (**input**) are integer arrays containing the requested indices. The size of `I` (resp. `J`) is the number of rows (resp. `columns`) of `submat` and can be found in the descriptor `desc`. The values (indices) in `I` and `J` are **1-based**.
- `obj` (**input**) is a handle to a user-given object, that the user passes to the SDP instance before calling the compression routine (cf. Section 5). This allows users to access their data when they write the access routine without describing it to STRUMPACK.

It it also the responsibility of the user to indicate whether the element access routine involves interprocess communications or not. When the element access routine is "global" (involves interprocess communications), STRUMPACK has to modify its task scheduling in the compression to avoid deadlocks. On the other hand, when the element access routine is not global, STRUMPACK uses a more efficient scheduling. The parameter `StrumpackDensePackage::access_is_global` should be set to true when the element access routine is global.

The following function is a simplistic example of a routine that computes selected elements of a Cauchy matrix defined by $a_{ij} = \frac{1}{x_i - y_i}$. In this simple example, there is only one MPI process. The vectors x and y (user data) are accessed using the user object handle, assuming here that `obj` is an array containing two pointers. `T` is the scalar type.

```
void foo(void *obj, int *I, int *J, T* submat, int *desc) {
  int nI, nJ;
  int i, j;
  T *x, *y;
  T **ptr;

  nI=desc[2]; // Size of I
  nJ=desc[3]; // Size of J

  ptr=(T**)obj; // Get user object from the handle
  x=ptr[0];     // "
  y=ptr[1];     // "

  for(i=0;i<nI;i++)
    for(j=0;j<nJ;j++)
      submat[i+nI*j]=1/(x[I[i]-1]-y[J[j]-1]); // submat(i,j)=A(I[i],J[j])
}
```

Adaptive sampling is not available in this version. It is the responsibility of the user to check the maximum rank (`max_rank`) against the number of random vectors, increase the sampling size and restart the compression (after reinitializing the SDP object).

4.4 Compression accuracy checking

`S check_compression(T *A, int *descA);`

Input: matrix A in 2D block-cyclic form with a BLACS descriptor `descA`.

Output: a real scalar containing $\frac{||\tilde{A}-A||_F}{||A||_F}$ with $\tilde{A}$ the compressed (HSS) representation of A.

Remarks:

- **This is very expensive and memory-consuming**.
- If A was not previously compressed, the routine exits and reports an error message.

4.5 Factorization

```
void factor(T *A, int *descA);
```

Input: if HSS is used (`use_HSS` true), no input is needed; `A` can be a `NULL` pointer. Otherwise, matrix A in 2D block-cyclic form with a BLACS descriptor `descA`

Output: no user-available output. Internally, the `StrumpackDensePackage` objects contains the factors of A.

Remarks:

- If HSS is used (`use_HSS` true), this is an ULV factorization. If a matrix has been previously factored with ULV factorization, the routine will abort. Another `StrumpackDensePackage` object must be used to hold another ULV factorization.
- If HSS is disabled (`use_HSS` true), this is a standard LU factorization performed with ScaLAPACK (`PxGETRF`). If the matrix is centralized, this is will performed sequentially. If a matrix has been previously factored with LU factorization, the routine will abort. Another `StrumpackDensePackage` object must be used to LU hold another factorization.
- As explained above, a `StrumpackDensePackage` cannot hold several ULV factors or several LU factors, however it can hold one ULV factorization and one LU factorization simultaneously.

4.6 Partial factorization

```
void partially_factor(T* A, int *descA, int nfact);
```

Input: if HSS is used (`use_HSS` true), no input is needed; `A` can be a `NULL` pointer. Otherwise, matrix A in 2D block-cyclic form with a BLACS descriptor `descA`. `nfact` is the number of variables to eliminated, i.e., the size of the (1,1) in the partial factorization 1 (Section 3.7). If HSS kernels are used (`use_HSS` is true), then `nfact` must be equal to `split_HSS`.

Output: no user-available output.Internally, the `StrumpackDensePackage` objects contains the factors of A.

4.7 Schur complement computation

```
void compute_schur();
```

Input: no input. `partially_factor` must have been previously called.

Output: no user-available output. Internally, the `StrumpackDensePackage` objects contains a representation of the Schur complement of A (a traditional Schur complement if `use_HSS` is false, or a *compressed* Schur complement is `use_HSS` is true.

4.8 Solution

```
void solve(T *X, int *descX, T *B, int *descB);
```

Input: X is the solution and B the right-hand side of the system to be solved. X and B can have multiple columns. They must have the same size and be distributed in 2D block-cyclic form over the same set of processes, using the same block size.

Output: X is the solution of $A \cdot X = B$ where A is a matrix previously factored with `factor`.

Remarks:

- If `use_HSS` is true, a special HSS triangular solution algorithm is used. The matrix A must have been factored using ULV factorization, i.e., by calling `factor` with `use_HSS` true.
- If `use_HSS` is false, a traditional triangular solution algorithm is used. The matrix A must have been factored using LU factorization, i.e., by calling `factor` with `use_HSS` false.

4.9 Right-hand side reduction

```
void reduce_RHS(T *X, int *descX, T* B, int *descB);
```

Input: X is the reduced RHS and B the right-hand side of the system to be solved. X and B can have multiple columns. They must have the same size and be distributed in 2D block-cyclic form over the same set of processes, using the same block size.

Output: X is the reduced RHS corresponding to vector y in Equation 2 in Section 3.7.

Remarks:

- If `use_HSS` is true, a special HSS triangular solution algorithm is used. The matrix A must have been **partially** factored using ULV factorization, i.e., by calling `partially_factor` with `use_HSS` true.
- If `use_HSS` is false, a traditional triangular solution algorithm is used. The matrix A must have been **partially** factored using LU factorization, i.e., by calling `partially_factor` with `use_HSS` false.

4.10 Solution expansion

```
void expand_solution(T *X, int *descX, T *B, int *descB);
```

Input: the bottom part of X (last $n-$`nfact` must contain the solution of the Schur complement system, x_2, as in Section 3.7. The top part of B (first `nfact` rows) must contain y_1. X and B can have multiple columns. They must have the same size and be distributed in 2D block-cyclic form over the same set of processes, using the same block size.

Output: the top part of X (first `nfact` rows) is filled so that X is the solution of the whole problem.

Remarks:

- If `use_HSS` is true, a special HSS triangular solution algorithm is used. The matrix A must have been **partially** factored using ULV factorization, i.e., by calling `partially_factor` with `use_HSS` true.
- If `use_HSS` is false, a traditional triangular solution algorithm is used. The matrix A must have been **partially** factored using LU factorization, i.e., by calling `partially_factor` with `use_HSS` false.

4.11 Solution accuracy checking

```
S check_solution(T *A, int *descA, T *X, int *descX, T *B, int *descB);
```

Input: A, X and B matrices distributed in 2D block cyclic form over the same set of processes, with the same block size. A, X, and B have the same row count, X and B have the same column count.

Output: relative residual $||A \cdot X - B||_F / ||B||_F$

Remark: the computation is performed using traditional PBLAS/ScaLAPACK operations.

4.12 Solution iterative refinement

```
S refine(T *A, int *descA, T *X, int *descX, T *B, int *descB);
```

Input: A, X and B matrices distributed in 2D block cyclic form over the same set of processes, with the same block size. A, X, and B have the same row count, X and B have the same column count.

Output: X is the result of the refinement and is (hopefully) a better solution to $A \cdot X = B$.

Remark: the algorithm is described in Section 3.4.

4.13 Matrix-vector product

```
void product(char Trans, T alpha, T *A, int *descA, T *B, int *descB,
             T beta, T *C, int *descC);
```

Input: A, B and C matrices distributed in 2D block cyclic form over the same set of processes, with the same block size. A, B, and C have the same row count, B and C have the same column count.

Output: B contains $\alpha A \cdot B + \beta B$ (same as the usual GEMM).

Remarks:

- If `use_HSS` is true, an HSS matrix-vector product algorithm is used, but the matrix must have been previously compressed (and input A is actually not used, but descA must be correct).
- If `use_HSS` is false, a standard product is computed with PBLAS. If A is centralized, the operation is sequential.

4.14 Schur complement matrix-vector product

```
void schur_product(char Trans, T alpha, T *X, int *descX, T beta, T *B, int *descB);
```

Input: X and B matrices of the same size, distributed in 2D block cyclic form over the same set of processes, with the same block size. The number of rows of X and B must be $n-$`nfact`, after calling `partially_factor(...,nfact)` and `compute_schur()`.

Output: B contains $\alpha S \cdot X + \beta B$, where S is the Schur complement as defined in Section 3.7.

4.15 Element extraction

```
void extract(T *A, int *descA, T *B, int *descB, int *I, int nI, int *J, int nJ);
```

Input: Matrix A (resp. B in 2D block-cyclic form, described by `descA` (resp. `descB`). If `use_HSS` is true, `A` and `descA` can be `NULL` (but `compress` must have been previously called). Integer vectors `I` (size `nI`) and `J` (size `nJ`) **replicated** on all the processes in the communicator. The values in `I` and `J` **must be between 1 and** n.

Output: B contains $A(I, J)$.

4.16 Schur complement element extraction

```
void extract_schur(T *B, int *descB, int *I, int nI, int *J, int nJ);
```

Input: Matrix B in 2D block-cyclic form, described by `descB`. Integer vectors `I` (size `nI`) and `J` (size `nJ`) **replicated** on all the processes in the communicator. The values in `I` and `J` **must be between 1 and** $n-$**`nfact`**. Functions `partially_factor(...,nfact)` and `compute_schur()` must have been called previously.

Output: B contains $S(I, J)$, where S is the Schur complement as defined in Section 3.7.

4.17 Statistics

```
void print_statistics();
```

Input: none

Output: prints various statistics on screen (flops, memory…).

4.18 C interface

We provide a C interface to SDP. Its usage is very similar to the native C++ interface, the main difference being that the C++ interface is templated/parametrized. The structures `StrumpackDensePackage_C_?` (with `?=double` for double precision real arithmetic, `?=float` for single precision real arithmetic, `?=dcomplex` for complex double precision arithmetic, or `?=scomplex` for complex single precision arithmetic) mimic the C++ object `StrumpackDensePackage`; they have the same fields (e.g., `use_HSS`, `tol_IR`…). The function names are prefixed with `SDP_C_?` (with `?=[double|float|dcomplex|scomplex]`) and they have the same signature as the methods of the `StrumpackDensePackage` C++ class, except they have an additional argument corresponding to the structure on which the function has to operate.

Remarks:

- The constructor of the C++ interface is replaced with an initialization routine that **must be called** before doing anything with the SDP structure: `void SDP_C_?_init`.
- In the C++ interface, overloading is used and the compression routine has three different versions with the same name (`compress`). In the C interface, these three versions have three different names:
  - `SDP_C_?_compress_A(&sdp_C, A, descA)` (explicit matrix).
  - `SDP_C_?_compress_ARS(&sdp_C, A, descA, Rr, Rc, Sr, Sc, descRS)` (explicit matrix and user given random vectors and samples).
  - `SDP_C_?_compress_mtxfree(&sdp_C, Rr, Rc, Sr, Sc, descRS, elements)` (matrix-free compression).
- The `dcomplex` and `scomplex` types are simply structures with two fields (`r` real part, `i` imaginary part). In the C++ interface, the types `std::complex<?>` are used.
- Boolean variables in the C++ interface are integers in the C interface (with the usual convention that nonzero==true).

For example, the following C++ code:

```
#include "StrumpackDensePackage.hpp"
StrumpackDensePackage<double,double> sdp(MPI_COMM_WORLD);
sdp.use_HSS=true;
sdp.compress(A,descA);
```

…can be called using the C interface with:

```
#include "StrumpackDensePackage.h"
StrumpackDensePackage_C_double sdp;
SDP_C_init(&sdp,MPI_COMM_WORLD);
sdp.use_HSS=1;
SDP_C_double_compress_A(&sdp,A,descA);
```

4.19 Fortran interface

We also provide a Fortran interface to SDP; it relies on ISO C bindings to call the functions of the C interface. Its usage is straightforward. The snippet of C++ code mentioned in the previous section becomes the following in the Fortran interface:

```
use StrumpackDensePackage
type(StrumpackDensePackage_F90_double) :: sdp
call SDP_F90_dcomplex_init(sdp,MPI_COMM_WORLD)
sdp%use_HSS=1
call SDP_F90_dcomplex_compress_A(sdp,C_LOC(A),C_LOC(descA))
```

The only remarkable point is that the Fortran arrays (e.g., the matrix) must be converted to C pointers using the `C_LOC` function before being passed to the SDP routines.

4.20 Octave/Matlab® interface

We provide an interface for Octave/Matlab®. Only real double precision and complex double precision arithmetics are available. Matrix-free HSS compression is are not available.

Usage is similar to all the other interfaces; the example code becomes the following:

```
sdp=SDP_mex_double_init();
sdp.use_HSS=1;
SDP_mex_double_compress_A(sdp,A);
```

Remarks:

- The interface was mostly tested with Octave. Support for Matlab is experimental.
- With Matlab, one might need to run `mpiInit` before running SDP.
- The interface works in parallel; e.g., one can run `mpirun -np 4 octave mex_example.m`.
- The interface assumes that all the objects (matrices, vectors, scalars…) are duplicated. E.g., when one writes `x=ones(5,1)` in an `.m` script and runs Octave with `mpirun`, the `x` object is replicated on all the processes (every process owns a 5 × 1 array), and the interface relies on this assumption. The interface internally builds 2D block-cyclic versions of the input objects so that they can be passed to SDP. Output arrays are also duplicated.

5 Parameters

5.1 General parameters

`use_HSS:` true to use HSS algorithms, false to use standard PBLAS/ScaLAPACK algorithms. Default: true.

`verbose:` if false, the routines do not print anything on screen; otherwise, they print some information. Default: true.

5.2 HSS compression

Most important parameters (described in Section 3.1):

`tol_HSS:` tolerance used for the HSS compression routine.

`min_rand_HSS:` starting number of random vectors for sampling.

`max_rand_HSS:` maximum number of random vectors for sampling.

`split_HSS:` for use with a partial factorization. It must be set prior to the compression, and must be equal to the `nfact` parameter of the `partially_factor` routine.

`obj:` for matrix-free compression only; corresponds to the `obj` parameter in the element access routine provided by the user. Cf. Section 4.3.3 and the `matrixfree.cpp` example shipped with the code.

`access_is_global:` for matrix-free compression only. Must be set to true when the user-given element access routine involves interprocess communication. Can be set to false otherwise. True by default.

`sched_strat:` for compression with an explicit matrix. Values: 1 or 2. Default: 1; 2 is an alternative scheduling strategy that can pay off in some situations for problems that are very compressible.

Secondary (default parameters should be fine in most case):

`lim_rand_HSS`: restarting criterion. If d is the current number of random vectors used for sampling and the rank computed at a step of the compression stage is larger than d-`lim_rand_HSS`, more random vectors are added and the compression restarts.

`inc_rand_HSS` number of random vectors to be added when the compression restarts.

`levels_HSS`: number of levels in the HSS tree.

`block_HSS`: size of the index set associated with each leaf of the HSS tree. It is another way of defining the number of levels in the tree (`levels_HSS`$=\log_2(n/$`block_HSS`$+1)$; `levels_HSS` has higher priority.

5.3 Iterative refinement

`tol_IR`: stopping criterion for iterative refinement (cf. Section 3.4).

`steps_IR`: number of steps of iterative refinement.

`fast_IR`: if true, approximate HSS matrix-vector products are used instead of exact matrix-vector products during the iterative refinement. This is usually not numerically stable and is disabled by default.

6 Acknowledgements

We wish to thank our collaborator Artem Napov (Université Libre de Bruxelles) for his insight on the algorithmic aspects.

We wish to thank people who sent us test problems and helped testing the code:

- Guillaume Sylvand (Airbus).
- Ana Manic (Colorado State University).
- Jeremiah Jones (Arizona State University).
- Umberto Villa (Lawrence Livermore National Laboratory).

7 Copyright notice

8 License agreement

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK users' guide*, vol. 4, SIAM, 1997.

[2] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, Submitted to ACM Transactions on Mathematical Software, (2014).