User Guide for the SPEX Software Package

VERSION 3.2.0, July 2, 2024

Jinhao Chen, Timothy A. Davis, Christopher Lourenco, Lorena Mejia-Domenzain, Erick Moreno-Centeno
Texas A&M University and US Naval Academy

Contact Information: Contact Chris Lourenco, chrisjlourenco@gmail.com, lourenco@usna.edu, or Tim Davis, timdavis@aldenmath.com, davis@tamu.edu, DrTimothyAldenDavis@gmail.com

CHAPTER 1

SPEX OVERVIEW

SPEX is a software package comprising several state-of-the-art SParse EXact linear algebra routines. It currently consists of the following:

SPEX Utilities Utility and auxiliary functions for all SPEX routines: interface to the GMP and MPFR libraries, memory management functions, the `SPEX_matrix`, `SPEX_factorization`, and `SPEX_symbolic_analysis` data structures, and various functions that are auxiliary to the factorization and solve functions. Please refer to Chapter 4 for further details.

SPEX LU Sparse exact left-looking LU factorization to solve the linear system $A\mathbf{x} = \mathbf{b}$. The solution time is proportional to the arithmetic work in the bit-complexity model; which is asymptotically efficient. Please refer to Chapter 5 for further details.

SPEX Cholesky and LDL Sparse exact left-looking and up-looking factorizations to solve the symmetric with nonzero leading principle minors linear system $A\mathbf{x} = \mathbf{b}$. The solution time is proportional to the arithmetic work in the bit-complexity model; this is an asymptotically efficient complexity bound. The methods can perform either a Cholesky or LDL factorization depending on the signs of the diagonal elements. Please refer to Chapter 6 for further details.

SPEX Backslash Routines to exactly solve the system $A\mathbf{x} = \mathbf{b}$ using either LU or LDL factorization. This is the simplest way to access the SPEX software package. Please refer to Chapter 7 for further details.

Location: https://github.com/clouren/SPEX and www.suitesparse.com

Required Packages: SPEX depends on the following packages:

- GNU GMP [6] and MPFR [5] libraries. Distributed under the LGPL3 and GPL2 and can be acquired and installed from https://gmplib.org/ and http://www.mpfr.org/, respectively.
- CMake, available under a BSD 3-clause license. May be independently obtained at https://cmake.org.

- AMD [1, 2], available under a BSD 3-clause license and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`
- COLAMD [4, 3], available under a BSD 3-clause license and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`
- `SuiteSparse_config`, available under a BSD 3-clause license and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`.

CHAPTER 2

SETTING UP SPEX

2.1 Licensing

Copyright: The copyright of this software is held by Christopher Lourenco, Jinhao Chen, Lorena Mejia-Domenzain, Erick Moreno-Centeno, and Timothy A. Davis.

Contact Info: Chris Lourenco, chrisjlourenco@gmail.com lourenco@usna.edu, or Tim Davis, timdavis@aldenmath.com, DrTimothyAldenDavis@gmail.com, or davis@tamu.edu

License: This software package is dual licensed under the GNU General Public License version 2 or the GNU Lesser General Public License version 3. Details of this license are in `SPEX/License/license.txt`. For alternative licenses, please contact the authors.

2.2 Installation

You must first install the GMP (v6.1.2 or later), MPFR (v4.0.2 or later) and CMake (v3.22 or later) packages. See `https://gmplib.org/`, `https://www.mpfr.org/`, and `https://cmake.org/` for details. The libraries may also be installed with package managers such as `spack` (`https://spack.io/`) or `homebrew` (`https://brew.sh/`).

2.2.1 Compiling and Installing SPEX using `cmake`

Installation of SPEX requires the `cmake` utility in Linux, MacOS, and Windows. In a terminal window (a Command Window in Windows) go to the top-level folder of the source distribution, where you will see the subfolders called `SPEX`, `AMD`, `COLAMD`, and `SuiteSparse_config` side-by-side, and then type the following commands. This will configure the packages, compile them, and install them:

```
cd build
cmake ..
cmake --build . --config Release
cmake --install .
```

The last command may require you to authenticate, for permission to install the `SPEX` package and its dependents (`AMD`, `COLAMD`, and `SuiteSparse_config`). On Linux or Mac, use this instead of last command above:

```
sudo cmake --install .
```

2.2.2 Compiling and Installing SPEX using `make`

If you have the `make` command, you can instead type these commands, in the same folder, which call upon `cmake` to do the same thing as above:

```
make
sudo make install
```

Then to run some demo programs, type `make demos`.

2.2.3 Installation location

If you cannot install SPEX in the default location, you can use `cmake` to change the installation location. Instead of the command `cmake ..` above, use `ccmake ..` which will pop up a list of options. Navigate down the list and select a different location for the `CMAKE_INSTALL_PREFIX` option.

If using `make`, try `make local` instead of `make`, which will configure SPEX and its dependent libraries in the `lib` folder located within the top-level folder. To compile and install SPEX system-wide for all users on your system, use `make global ; sudo make install`.

2.2.4 Using SPEX in your C/C++ program

After compiling and installing SPEX, you can use the code inside of a C/C++ program by including `#include "SPEX.h"`, and linking against `SPEX` and its dependencies. CMake `find_package` scripts or `pkgconfig` files are provided as well, to use in your own cmake scripts to find `SPEX` and its dependencies. They are located in the `lib/cmake` and `lib/pkgconfig` folders.

2.2.5 Using SPEX in MATLAB and Python

SPEX also includes MATLAB and Python interfaces. To install the MATLAB interface, you must first configure the `SPEX` library outside of MATLAB and build its dependencies. Follow the instructions above for configuring, compiling, and installing SPEX (installation is optional in this case).

Next, navigate to the `SPEX/MATLAB` folder from the MATLAB command window and type `spex_mex_install` which will install the MATLAB interfaces to all SPEX packages. These MATLAB functions can then be used outside of the `SPEX/MATLAB` folder by using the MATLAB `addpath` command, or `pathtool`. The Python interface does not need any additional installation, but does require the `Numpy`, `SciPy`, and `ctypes` libraries.

2.2.6 Configuration options

Cmake can be configured with options that control the creation of Python interface for SPEX, and the use of OpenMP. OpenMP is used in SPEX for thread-local storage. If OpenMP is not available, the compiler-supported thread-local storage mechanism is used instead (if available). Each of these options can be controlled just for SPEX, or for all packages in SuiteSparse:

- `SUITESPARSE_USE_PYTHON`:

  If `ON`, build all Python interfaces for SuiteSparse packages (currently only in SPEX). If `OFF`: do not build any Python interfaces. Default: `ON`.

- `SUITESPARSE_USE_OPENMP`:

  If `ON`, OpenMP is used in SuiteSparse if it is available. Default: `ON`.

- `SPEX_USE_PYTHON`:

  If `ON`, build Python interface for SPEX. If `OFF`: do not build the SPEX Python interface. Default: `SUITESPARSE_USE_PYTHON`.

- `SPEX_USE_OPENMP`:

  If `ON`, OpenMP is used in SPEX if it is available. Default: `SUITESPARSE_USE_OPENMP`.

CHAPTER 3

GENERAL SPEX DATA STRUCTURES AND MACROS

The following macros/data structures are defined in `SPEX.h` and are used in all SPEX functions.

3.1 SPEX_VERSION: the software package version

SPEX defines the following macros with `#define`. Refer to the `SPEX.h` file for details.

Macro	Purpose	Type
`SPEX_DATE`	Release date	string
`SPEX_VERSION_STRING`	Current version of the code	string
`SPEX_VERSION_MAJOR`	Major version of the code	integer
`SPEX_VERSION_MINOR`	Minor version of the code	integer
`SPEX_VERSION_SUB`	Sub version of the code	integer
`SPEX_VERSION`	Current version of the code	integer
`SPEX__VERSION`	Identical to `SPEX_VERSION`	integer
`SPEX_VERSION_NUMBER`	Macro to create `SPEX_VERSION`	macro

`SPEX_VERSION_NUMBER(major,minor,sub)` is a macro that creates a single integer from the three integers `major`, `minor`, `sub`. It is useful for checking relative version numbers. For example, a user application could include the following:

```
#include "SPEX.h"
#if SPEX_VERSION < SPEX_VERSION_NUMBER(3,2,0)
// SPEX version is prior to 3.2.0
#else
// SPEX is version 3.2.0 or later
#endif
```

`SPEX_VERSION` and `SPEX__VERSION` are identical. The latter is included because it follows the form of `Package__VERSION` for all packages in SuiteSparse.

See also the `SPEX_version` function described in Section 4.8.1.

3.2 SPEX_info: status codes returned by SPEX

Most SPEX functions return their status to the caller as their return value, an enumerated type called `SPEX_info`. All current possible values for `SPEX_info` are listed as follows:

0	`SPEX_OK`	The function was successfully executed.
-1	`SPEX_OUT_OF_MEMORY`	Out of memory
-2	`SPEX_SINGULAR`	The input matrix A is exactly singular.
-3	`SPEX_INCORRECT_INPUT`	One or more input arguments are incorrect.
-4	`SPEX_NOTSPD`	The input matrix is not SPD (thus can't use Cholesky)
-5	`SPEX_INCORRECT_ALGORITHM`	The algorithm is not compatible with the factorization
-6	`SPEX_PANIC`	SPEX environment error
-7	`SPEX_ZERODIAG`	Diagonal element is zero thus can't use LDL
-8	`SPEX_UNSYMMETRIC`	Input matrix is unsymmetric thus can't use LDL or Cholesky

3.3 SPEX_pivot: enum for pivoting schemes

There are six available pivoting schemes provided in SPEX that can be selected with the `SPEX_options` structure. Currently, these pivot options are only valid for LU factoization, as the symmetric routines pivot exclusively down the diagonal in order to maintain symmetry. If the matrix is non-singular (in an exact sense), then the pivot is always nonzero, and is chosen as the *smallest* nonzero entry, with the smallest magnitude. This may seem counter-intuitive, but selecting a small nonzero pivot leads to smaller growth in the number of digits in the entries of `L` and `U`. This choice does not lead to any kind of numerical inaccuracy, since SPEX is guaranteed to find an exact roundoff-error free factorization of a non-singular matrix (unless it runs out of memory), for any nonzero pivot choice.

The pivot tolerance for two of the pivoting schemes is specified by the `tol` component in `SPEX_options`. The pivoting schemes are as follows:

0	`SPEX_SMALLEST`	The k-th pivot is selected as the smallest entry in the k-th column. This is the default.
1	`SPEX_DIAGONAL`	The k-th pivot is selected as the diagonal entry. If the diagonal entry is zero, this method instead selects the smallest pivot in the column.
2	`SPEX_FIRST_NONZERO`	The k-th pivot is selected as the first eligible nonzero in the column.
3	`SPEX_TOL_SMALLEST`	The k-th pivot is selected as the diagonal entry if the diagonal is within a specified tolerance of the smallest entry in the column. Otherwise, the smallest entry in the k-th column is selected. This is the default pivot selection strategy.
4	`SPEX_TOL_LARGEST`	The k-th pivot is selected as the diagonal entry if the diagonal is within a specified tolerance of the largest entry in the column. Otherwise, the largest entry in the k-th column is selected.
5	`SPEX_LARGEST`	The k-th pivot is selected as the largest entry in the k-th column.

3.4 SPEX_preorder

The SPEX Library provides three ordering schemes: no ordering, COLAMD, and AMD. In LU factorization, the ordering is applied only to the columns, that is this ordering gives the matrix Q. In Cholesky and LDL factorizations, the ordering is applied to both the rows and columns, that is the ordering gives the matrices P and Q.

0	`SPEX_DEFAULT_ORDERING`	Use COLAMD for LU factorization. Use AMD for Cholesky or LDL factorization.
1	`SPEX_NO_ORDERING`	No pre-ordering is performed on the matrix A, that is $Q = I$.
2	`SPEX_COLAMD`	The permutation Q is found with COLAMD. [3]. This is recommended for LU factorization.
3	`SPEX_AMD`	The permutation Q is found with AMD [2]. This is recommended for Cholesky and LDL factorization.

For LU factorization, PAQ is factorized where P is determined during the numerical factorization using the method specified by `SPEX_pivot` (Section 3.3). For Cholesky or LDL factorization, PAQ is factorized where $P = Q^T$ is computed during the symbolic analysis. In both cases, Q is determined solely by the symbolic analysis.

3.5 SPEX_factorization_algorithm

This code tells SPEX which factorization algorithm should be used.

0	`SPEX_ALGORITHM_DEFAULT`	See below.
1	`SPEX_LU_LEFT`	Left-looking LU factorization
2	`SPEX_CHOL_LEFT`	Left-looking Chokesy factorization
3	`SPEX_CHOL_UP`	Up-looking Cholesky factorization
4	`SPEX_LDL_LEFT`	Left-looking LDL factorization
5	`SPEX_LDL_UP`	Up-looking LDL factorization

This option is used slightly different within each family of algorithms, in `SPEX_*_analyze`, `SPEX_*_factorize`, and `SPEX_*_backslash`:

- `SPEX_lu_*`: the option can be `SPEX_ALGORITHM_DEFAULT` or `SPEX_LU_LEFT`. In both cases, a left-looking LU factorization method is used to analyze and factorize the matrix.

- `SPEX_cholesky_*`: the option can be `SPEX_ALGORITHM_DEFAULT`, `SPEX_CHOL_LEFT`, or `SPEX_CHOL_UP`. The default is to use an up-looking Cholesky factorization. Alternatively, `SPEX_CHOL_LEFT` selects a left-looking Cholesky method.

- `SPEX_ldl_*`: the option can be `SPEX_ALGORITHM_DEFAULT`, `SPEX_LDL_LEFT`, or `SPEX_LDL_UP`. The default is to use an up-looking LDL factorization. Alternatively, `SPEX_LDL_LEFT` selects a left-looking LDL method.

- `SPEX_backslash`: any option is permitted. The default algorithm is selected with `SPEX_ALGORITHM_DEFAULT`. In this case, an up-looking LDL factorization is first tried. If the matrix is unsymmetric, or if the factorization encounters zeros on the diagonal D, then the LDL factorization fails, and a left-looking LU factorization is then used. If the option is set to a particular algorithm, then it will be used instead. If the selected algorithm fails (say LDL is requested but the matrix is unsymmetric), SPEX does not fall back to trying a different algorithm (such as an LU factorization). This fallback is only available if the option is set to `SPEX_ALGORITHM_DEFAULT`.

3.6 `SPEX_options` structure

The `SPEX_options` struct stores key command parameters for various functions used in the SPEX package. The `SPEX_options option` struct contains the following components:

- `option->pivot`: An enum `SPEX_pivot` type which controls the type of pivoting used. Default value: `SPEX_SMALLEST` (0).
- `option->order`: An enum `SPEX_preorder` type which controls what column ordering is used. Default value: `SPEX_DEFAULT_ORDERING` which gives COLAMD for LU and AMD for Cholesky and LDL.
- `option->tol`: A `double` tolerance for the tolerance-based pivoting scheme, i.e., `SPEX_TOL_SMALLEST` or `SPEX_TOL_LARGEST`. `option->tol` must be in the range of $(0, 1]$. Default value: 1 meaning that the diagonal entry will be selected if it has the same magnitude as the smallest entry in the k the column. This option is only used for LU factorization.
- `option->print_level`: An `int` which controls the amount of output: 0: print nothing, 1: just errors, 2: terse, with basic stats from COLAMD/AMD and SPEX, 3: all, with matrices and results. Default value: 0.
- `option->prec`: An `int32_t` which specifies the precision used for multiple precision floating point numbers, (i.e., MPFR). This can be any integer larger than `MPFR_PREC_MIN` (value of 1 in MPFR 4.0.2 and 2 in some legacy versions) and smaller than `MPFR_PREC_MAX` (usually the largest possible integer available in your system). Default value: 128 (quad precision).
- `option->round`: A `mpfr_rnd_t` which determines the type of MPFR rounding to be used by SPEX. This is a parameter of the MPFR library. The options for this parameter are:
  - `MPFR_RNDN`: Round to nearest (roundTiesToEven in IEEE 754-2008)
  - `MPFR_RNDZ`: Round toward zero (roundTowardZero in IEEE 754-2008)
  - `MPFR_RNDU`: Round toward plus infinity (roundTowardPositive in IEEE 754-2008)
  - `MPFR_RNDD`: Round toward minus infinity (roundTowardNegative in IEEE 754-2008)

  - `MPFR_RNDA`: Round away from zero
  - `MPFR_RNDF`: Faithful rounding. This is not stable.

  Refer to the MPFR User Guide available at https://www.mpfr.org/mpfr-current/mpfr.pdf for details on the MPFR rounding style and any other utilized MPFR convention. Default value: `MPFR_RNDN`.

- `option->algo`: A `SPEX_factorization_algorithm` which indicates which type of factorization is being used.

All SPEX routines except basic memory management routines in Sections 4.2.3-4.3.1 and the `SPEX_options` allocation routine in 4.4.1 require `option` as an input argument. The construction of the `option` struct can be avoided by passing `NULL` for the default settings. Otherwise, the following functions create and destroy a `SPEX_options` structure:

Function/Macro Name	Description	Section
`SPEX_create_default_options`	create and return `SPEX_options` pointer with default parameters upon successful allocation	4.4.1
`SPEX_FREE`	destroy `SPEX_options` structure	4.3.4

3.7 SPEX_vector

`SPEX_vector` is a compressed sparse vector data structure which will be used for SPEX dynamic CSC matrices. This struct is not used in this version of SPEX version and its funcionality will be fully developed in a future release of SPEX; however the struct is provided here so that future versions of SPEX will have backward compatibility with this version of SPEX.

This is **NOT** intended to be used for building any n-by-1 vector (e.g., the right-hand-side vector $\mathbf{b}$ in $A\mathbf{x} = \mathbf{b}$), which should be considered as a n-by-1 `SPEX_matrix`. This struct contains the following components:

- `vector->nz`: The number of explicit entries in the vector. Data Type: `int64_t`.
- `vector->nzmax`: The size of the `i` and `x` arrays. Note that `nz` $\leq$ `nzmax`. Data Type: `int64_t`.
- `vector->i`: An array of size `nzmax` containing the row indices of all explicit entries in the vector. The last `(nzmax-nz)` entries are undefined. Data Type: `int64_t*`.
- `vector->x`: An array of size `nzmax` containing the numeric values of all explicit entries in the vector. The last `(nzmax-nz)` entries are undefined. Data Type: `mpz_t*`.
- `vector->scale`: Scaling parameter. The actual value of the k-th nonzero should be computed as `x[k]*scale`. Both `x[k]*scale` and `x[k]/mpq_denref(scale)` must be integer for all entries, where `mpq_denref(scale)` is a GMP macro that gives the denominator of `scale`. This is used to skip explicit update(s) for a column/row of

the factorization matrix, when all entries are to be multiplied with the same scaling factor(s). Data Type: `mpq_t`.

In the current release, the `SPEX_vector` is only used as a part of the `SPEX_matrix` struct, as a placeholder for future implementations, and is always a NULL pointer.

3.8 The `SPEX_matrix` structure

SPEX operates on matrices stored in any of the 16 different matrix formats: 15 of which are combinations of matrix formats and entry data-types: {Static Compressed Sparse Column (CSC), triplet, dense} × { `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t`, or `double`}, and the 16th of which is the dynamic CSC matrix with `mpz_t` entries. Using the SPEX matrix copy function, a matrix of any given form and data-type can be copied and converted into a matrix of any one of the 16 matrix-form and data-type combinations.

Most routines require the matrix to be in CSC form with `mpz_t` (i.e., arbitrary-sized integer) data type. This data structure stores the matrix A as a sequence of three arrays:

- `A->p`: Column pointers; an array of size `n+1`. The row indices of column j are located in positions `A->p[j]` to `A->p[j+1]-1` of the array `A->i`. Data type: `int64_t`.
- `A->i`: Row indices; an array of size equal to the number of entries in the matrix. The entry `A->i[k]` is the row index of the kth nonzero in the matrix. Data type: `int64_t`.
- `A->x`: Numeric entries. The entry `A->x[k]` is the numeric value of the kth nonzero in the matrix. The array `A->x` has a union type and must be accessed via a suffix according to the type of `A`. For details, please refer to Section 3.8.

An example matrix A with `mpz_t` type is stored as follows (note that indexing is zero based as per the C convention).

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 2 & 0 & 4 & 12 \\ 7 & 1 & 1 & 1 \\ 0 & 2 & 3 & 0 \end{bmatrix}$$

```
A->p     = [0, 3, 5, 8, 11]
A->i     = [0, 1, 2, 2, 3, 1, 2, 3, 0,  1, 2]
A->x.mpz = [1, 2, 7, 1, 2, 4, 1, 3, 1, 12, 1]
```

For example, the last column appears in positions 8 to 10 of `A->i` and `A->x.mpz`, with row indices 0, 1, and 2, and values $a_{03} = 1$, $a_{13} = 12$, and $a_{23} = 1$.

3.8.1 SPEX_kind: enum for matrix formats

The SPEX library provides four available matrix formats: sparse CSC (compressed sparse column), sparse triplet, dense and sparse dynamic CSC.

0	SPEX_CSC	Matrix is in compressed sparse column format.
1	SPEX_TRIPLET	Matrix is in sparse triplet format.
2	SPEX_DENSE	Matrix is in dense format.
3	SPEX_DYNAMIC_CSC	Matrix is in dynamic CSC format.

3.8.2 SPEX_type: enum for data types of matrix entries

The SPEX library provides five data types for matrix entries: `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t` and `double`.

0	SPEX_MPZ	Matrix entries are in `mpz_t` type: an integer of arbitrary size.
1	SPEX_MPQ	Matrix entries are in `mpq_t` type: a rational number with arbitrary-sized integer numerator and denominator.
2	SPEX_MPFR	Matrix entries are in `mpfr_t` type: a floating-point number of arbitrary precision.
3	SPEX_INT64	Matrix entries are in `int64_t` type.
4	SPEX_FP64	Matrix entries are in `double` type.

3.8.3 SPEX_matrix structure

A matrix `SPEX_matrix A` has the following components:

- `A->kind`: Indicating the kind/format of matrix A: CSC, triplet, dense or dynamic CSC. Data Type: `SPEX_kind`.

- `A->type`: Indicating the type of entries in matrix A: `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t` or `double`. Data Type: `SPEX_type`.

- `A->m`: Number of rows in the matrix. Data Type: `int64_t`.

- `A->n`: Number of columns in the matrix. Data Type: `int64_t`.

- `A->scale`: A scaling parameter for matrix of `mpz_t` type. For all matrices whose entries are stored in data type other than `mpz_t`, SPEX assumes and maintains `A->scale = 1`. This is used to ensure that entry can be represented as an integer in an `mpz_t` matrix if these entries are converted from non-integer type data (such as double, variable precision floating point, or rational). Data Type: `mpq_t`.

- `A->nzmax`: The allocated size of the vectors `A->i`, `A->j` and `A->x`. Note that `A->nzmax` $\geq$ `nnz(A)`, where `nnz(A)` is the return value of `SPEX_matrix_nnz(A,option)`. Data Type: `int64_t`.

- `A->nz`: The number of nonzeros in the matrix A, if A is a triplet matrix (ignored for matrices in CSC, dense or dynamic CSC formats). Data Type: `int64_t`.

- `A->p`: An array of size `A->n`+1 which contains column pointers of A, if A is a CSC matrix (`NULL` for matrices in triplet or dense formats). Data Type: `int64_t*`.
- `A->p_shallow`: A boolean indicating whether `A->p` is shallow. A *shallow* pointer is one that refers to a component of another matrix or data structure. If `A->p` is shallow, then it should not be modified as part of the `A` matrix, and it is not freed if `A` is freed. Data Type: `bool`.
- `A->i`: An array of size `A->nzmax` which contains the row indices of the nonzeros in A, if A is a CSC or triplet matrix (`NULL` for dense matrices). The matrix is zero-based, so row indices are in the range of [0, `A->m`−1]. Data Type: `int64_t*`.
- `A->i_shallow`: A boolean indicating whether `A->i` is shallow. Data Type: `bool`.
- `A->j`: An array of size `A->nzmax` which contains the column indices of the nonzeros in A, if A is a triplet matrix (`NULL` for matrices in CSC or dense formats). The matrix is zero-based, so column indices are in the range of [0, `A->n`−1]. Data Type: `int64_t*`.
- `A->j_shallow`: A boolean indicating whether `A->j` is shallow. Data Type: `bool`.
- `A->x`: An array of size `A->nzmax` which contains the numeric values of the matrix. This array is a union, and must be accessed via one of: `A->x.mpz`, `A->x.mpq`, `A->x.mpfr`, `A->x.int64`, or `A->x.fp64`, depending on the `A->type` parameter. Data Type: `union`.
- `A->x_shallow`: A boolean indicating whether `A->x` is shallow. Data Type: `bool`.
- `A->v`: If the matrix is a `SPEX_DYNAMIC_CSC` this is an array of size `A->n`, each of which is a dynamic column vector. Data Type: `SPEX_vector**`. Always NULL in the current release of SPEX.

Specifically, for different kinds of `A` of size `A->m` × `A->n` with `nz` nonzero entries, its components are defined as:

- (0) `SPEX_CSC`: A sparse matrix in CSC (compressed sparse column) format. `A->p` is an `int64_t` array of size `A->n`+1, `A->i` is an `int64_t` array of size `A->nzmax` (with $nz \leq$ `A->nzmax`), and `A->x.TYPE` is an array of size `A->nzmax` of matrix entries (`TYPE` is one of `mpz`, `mpq`, `mpfr`, `int64`, or `fp64`). The row indices of column j appear in `A->i [A->p [j] ... A->p [j+1]-1]`, and the values appear in the same locations in `A->x.TYPE`. The `A->j` array is `NULL`. `A->nz` is ignored; the number of entries in `A` is given by `A->p [A->n]`. Row indices need not be sorted in each column, but duplicates cannot appear.
- (1) `SPEX_TRIPLET`: A sparse matrix in triplet format. `A->i` and `A->j` are both `int64_t` arrays of size `A->nzmax`, and `A->x.TYPE` is an array of values of the same size. The kth tuple has row index `A->i [k]`, column index `A->j [k]`, and value `A->x.TYPE [k]`, with $0 \leq k <$ `A->nz`. The `A->p` array is `NULL`. Triplets can be unsorted, but duplicates cannot appear.

- (2) `SPEX_DENSE`: A dense matrix. The integer arrays `A->p`, `A->i`, and `A->j` are all `NULL`. `A->x.TYPE` is a pointer to an array of size `A->m`-by-`A->n`, stored in column-oriented format. The value of $A(i,j)$ is `A->x.TYPE [p]` with `p` $= i + j *$`A->m`. `A->nz` is ignored; the number of entries in `A` is `A->m` $\times$ `A->n`.

- (3) `SPEX_DYNAMIC_CSC`: Currently unused

`A` may contain shallow components, `A->p`, `A->i`, `A->j`, and `A->x`. For example, if `A->p_shallow` is true, then a non-`NULL` `A->p` is a pointer to a read-only array, and the `A->p` array is not freed by `SPEX_matrix_free`. If `A->p` is `NULL` (for a triplet or dense matrix), then `A->p_shallow` has no effect.

The SPEX package has a set of functions to allocate, copy/convert, query and destroy a `SPEX_matrix` as shown in the following table.

Function Name	Description	Section
`SPEX_matrix_allocate`	allocate a m-by-n `SPEX_matrix`	4.5.1
`SPEX_matrix_free`	destroy a `SPEX_matrix` and free its allocated memory	4.5.2
`SPEX_matrix_copy`	make a copy of a matrix, into another kind and/or type	4.5.3
`SPEX_matrix_nnz`	get the number of entries in a matrix	4.5.4
`SPEX_matrix_check`	check the validity of a matrix and print it	4.5.5

3.9 The `SPEX_symbolic_analysis` structure

The symbolic analysis structure handles all preorderings and graphical struture information for each factorization within SPEX. Section 3.9.1 discusses an enum for the type of factorization. Section 3.9.2 discusses the components of this data structure.

3.9.1 `SPEX_factorization_kind`: enum for kind of factorization

The SPEX library currently provides three types of factorizations: LU, Cholesky, and LDL. The value `SPEX_QR_FACTORIZATION` is reserved for future development.

0	`SPEX_LU_FACTORIZATION`	LU factorization is being used
1	`SPEX_CHOLESKY_FACTORIZATION`	Cholesky factorization is being used
2	`SPEX_LDL_FACTORIZATION`	LDL factorization is used
3	`SPEX_QR_FACTORIZATION`	QR factorization is being used (reserved for future use)

3.9.2 `SPEX_symbolic_analysis` Data Structure

A symbolic analysis `SPEX_symbolic_analysis S` has the following components:

- `S->kind`: Indicating the kind of factorization either LU, Cholesky, or LDL. Data type: `SPEX_factorization_kind`

- `S->P_perm`: Row permutation for Cholesky/LDL and LU factorization. Data type: `int64_t*`

- `S->Pinv_perm`: Inverse row permutation for Cholesky/LDL and LU factorization. Data type: `int64_t*`

- `S->Q_perm`: Column permutation for LU factorization. This is always `NULL` and ignored for Cholesky/LDL factorization since its row and column permutations are the same. Data type: `int64_t*`

- `S->Qinv_perm`: Inverse column permutation for LU factorization. This is always `NULL` and ignored for Cholesky/LDL factorization since its inverse row and column permutations are the same. Data type: `int64_t*`

- `S->lnz`: Approximate number of nonzeros in L. In LU factorization, this is a crude estimate based on either AMD or COLAMD. In Cholesky/LDL factorization, if AMD is used, this is the exact number of nonzeros in L (excluding numeric cancellation). Data type: `int64_t`

- `S->unz`: Approximate number of nonzeros in U. In LU factorization, this is a crude estimate based on either AMD or COLAMD. In Cholesky/LDL factorization this is not used. Data type: `int64_t`

- `S->parent`: This is the elimination tree of the input matrix for Cholesky or LDL factorization. This is always `NULL` for LU factorization. Data type: `int64_t*`

- `S->cp`: Column pointers of L for Cholesky or LDL factorization. This is always `NULL` for LU factorization. Data type: `int64_t*`

This data type is constructed when analysis is called in the appropriate factorizations. See sections 5.3.1 and 6.3.1 for further details. To free this data structure, the function `SPEX_symbolic_analysis_free` is used and discussed further in section 4.6.

3.10 The `SPEX_factorization` data structure

The `SPEX_factorization` object holds an LU, Cholesky, or LDL numerical factorization. The components of the factorization structure are accessible to the user application. However, they should only be modified by calling SPEX methods. Changing them directly can lead to undefined behavior.

The components of a `SPEX_factorization F` are as follows:

- `F->kind`: Indicating the kind of factorization either LU, Cholesky, or LDL. Data type: `SPEX_factorization_kind`

- `F->updatable`: a flag that indicates whether the factorization is in an updatable format. Reserved for future development. Data type: `bool`

- `F->scale_for_A`: Scaling factor of the input matrix A. As discussed in section 3.8, all matrices in SPEX are integral, thus, if A must be scaled the scaling factor applied is stored here. Data type: `mpq_t`
- `F->L`: The lower triangular matrix for either LU, Cholesky, or LDL factorization. Data type: `SPEX_matrix`
- `F->U`: The upper triangular matrix for LU factorization. This is always `NULL` for Cholesky/LDL factorization. Data type: `SPEX_matrix`
- `F->Q`: The matrix for (future) QR factorization. Provided here so that future versions of SPEX have backward compatibility with this version of SPEX. Data type: `SPEX_matrix`
- `F->R`: The right triangular matrix for (future) QR factorization. Provided here so that future versions of SPEX have backward compatibility with this version of SPEX. Data type: `SPEX_matrix`
- `F->rhos`: An $n \times 1$ dense matrix containing the pivot values used for LU or Cholesky/LDL factorization. Data type: `SPEX_matrix`
- `F->P_perm`: Row permutation of the LU or Cholesky/LDL factors. Data type: `int64_t*`
- `F->Pinv_perm`: Inverse row permutation of the LU or Cholesky/LDL factors. Data type: `int64_t*`
- `F->Q_perm`: Column permutation of the LU factors. This is `NULL` and ignored for Cholesky/LDL factorization. Data type: `int64_t*`
- `F->Qinv_perm`: Inverse column permutation of the LU factors. This is `NULL` and ignored for Cholesky/LDL factorization. Data type: `int64_t*`

A `SPEX_factorization` is constructed by the appropriate factorization algorithms (see Sections 5.3.2 and 6.3.2 for further details). To free this data structure, the function `SPEX_factorization_free` is used and discussed further in section 4.7.1.

CHAPTER 4

SPEX UTILITIES

4.1 Overview

SPEX Util contains utility and auxiliary functions for the SPEX factorizations. Additionally, SPEX Util provides a wrapper class for the GNU Multiple Precision Arithmetic (GMP) [6] and GNU Multiple Precision Floating Point Reliable (MPFR) [5] libraries that prevent memory leaks and improve the overall stability of these external libraries by removing `abort()` conditions. SPEX Util is written in ANSI C.

4.2 Managing the SPEX environment

Either `SPEX_initialize` or `SPEX_initialize_expert` (but not both) must be called prior to using any other SPEX functions. Otherwise, all SPEX user-callable functions will return `SPEX_PANIC`. `SPEX_finalize` must be called as the last SPEX function. Note that if a user is working in a multi threaded environment then only one user thread should call the `SPEX_initialize` and `SPEX_finalize` functions.

Subsequent SPEX sessions can be restarted after a call to `SPEX_finalize`, by calling either `SPEX_initialize` or `SPEX_initialize_expert` (but not both), followed by a final call to `SPEX_finalize` when finished.

4.2.1 `SPEX_initialize`: initialize the working environment

```
SPEX_info SPEX_initialize
(
    void
) ;
```

`SPEX_initialize` initializes the working environment for SPEX functions. SPEX utilizes a specialized memory management scheme in order to prevent potential memory failures caused by GMP and MPFR libraries. Either this function or `SPEX_initialize_expert` must be called prior to using any other function in the library. By default, `SPEX_initialize` utilizes the `Suitesparse_malloc`, `Suitesparse_calloc`, `SuiteSparse_realloc`, and `Suitesparse_free` functions, which default to using the ANSI C `malloc`, `calloc`, `realloc`, and `free` functions. `SPEX_initialize` returns `SPEX_PANIC` if SPEX has already been initialized, or `SPEX_OK` if successful.

4.2.2 SPEX_initialize_expert: initialize environment (expert version)

```
SPEX_info SPEX_initialize_expert
(
    void* (*MyMalloc) (size_t),            // user-defined malloc
    void* (*MyCalloc) (size_t, size_t),    // user-defined calloc
    void* (*MyRealloc) (void *, size_t),   // user-defined realloc
    void  (*MyFree) (void *)               // user-defined free
) ;
```

`SPEX_initialize_expert` is the same as `SPEX_initialize` except that it allows for a redefinition of custom memory functions that are used for SPEX and GMP/ MPFR. The four inputs to this function are pointers to four functions with the same signatures as the ANSI C `malloc`, `calloc`, `realloc`, and `free` functions. That is:

```
#include <stdlib.h>
void *malloc (size_t size) ;
void *calloc (size_t nmemb, size_t size) ;
void *realloc (void *ptr, size_t size) ;
void free (void *ptr) ;
```

Returns `SPEX_PANIC` if SPEX has already been initialized, or `SPEX_OK` if successful.

4.2.3 SPEX_finalize: free the working environment

```
SPEX_info SPEX_finalize
(
    void
) ;
```

`SPEX_finalize` finalizes the working environment for SPEX library, and frees any internal workspace created by SPEX. It must be called as the last `SPEX_*` function called, except that a subsequent call to `SPEX_initialize*` may be used to start another SPEX session. Returns `SPEX_PANIC` if SPEX has not been initialized, or `SPEX_OK` if successful.

4.2.4 SPEX_thread_initialize: initialize working environment for a single thread

```
SPEX_info SPEX_thread_initialize
(
    void
);
```

`SPEX_thread_initialize` initializes the working environment of SPEX for a single user thread. If the user is working in a multithreaded environment, they must call this function at the beginning of each user thread. Returns `SPEX_OK` if successful or `SPEX_PANIC` if SPEX was already initialized.

This function is only required for a multithreaded user application that calls SPEX functions from threads other than the primary thread that called `SPEX_initialize`.

When the primary thread of the user application starts, it must call `SPEX_initialize`. When the user application enters a parallel region (say with OpenMP) or creates its own threads with a threading library, each user thread that calls any SPEX function must first call `SPEX_thread_initialize` when it starts, and `SPEX_thread_finalize` when it finishes.

An example usage can be found in the `SPEX/Demo` folder in the `spex_demo_threaded.c` main program.

4.2.5 `SPEX_thread_finalize`: finalize the working environment for a single thread

```
SPEX_info SPEX_thread_finalize
(
    void
);
```

`SPEX_thread_finalize` finalizes the working environment and frees any internal workspace created by SPEX for a single user thread. If the user is working in a multithreaded environment, they must call this function at the end of each user thread. Returns `SPEX_OK` if successful or `SPEX_PANIC` if SPEX was not initialized.

4.3 Memory Management

The routines in this section are used to allocate and free memory for the data structures used in SPEX. By default, SPEX relies on the SuiteSparse memory management functions, `SuiteSparse_malloc`, `SuiteSparse_calloc`, `SuiteSparse_realloc`, and `SuiteSparse_free`. By default, those functions rely on the ANSI C `malloc`, `calloc`, `realloc`, and `free`, but this may be changed by initializing the SPEX environment with `SPEX_initialize_expert`.

4.3.1 `SPEX_calloc`: allocate initialized memory

```
void *SPEX_calloc
(
    size_t nitems,      // number of items to allocate
    size_t size         // size of each item
) ;
```

`SPEX_calloc` allocates a block of memory for an array of `nitems` elements, each of them `size` bytes long, and initializes all its bits to zero. If any input is less than 1, it is treated as if equal to 1. If the function failed to allocate the requested block of memory, then a `NULL` pointer is returned. Returns `NULL` if the allocation fails.

4.3.2 SPEX_malloc: allocate uninitialized memory

```
void *SPEX_malloc
(
    size_t size         // size of memory space to allocate
) ;
```

SPEX_malloc allocates a block of size bytes of memory, returning a pointer to the beginning of the block. The content of the newly allocated block of memory is not initialized, remaining with indeterminate values. If size is less than 1, it is treated as if equal to 1. If the function fails to allocate the requested block of memory, then a NULL pointer is returned. Returns NULL if the allocation fails.

4.3.3 SPEX_realloc: resize allocated memory

```
void *SPEX_realloc      // pointer to reallocated block, or original block
                        // if the realloc failed
(
    int64_t nitems_new,     // new number of items
    int64_t nitems_old,     // previous/old number of items
    size_t size_of_item,    // size of each item
    void *p,                // pointer to reallocate
    bool *ok                // true if success, false on failure
) ;
```

SPEX_realloc is a wrapper for realloc. If p is non-NULL on input, it points to a previously allocated array of size nitems_old $\times$ size_of_item. The array is reallocated to be of size nitems_new $\times$ size_of_item. If p is NULL on input, then a new array of that size is allocated. On success, a pointer to the new array is returned.

If the reallocation fails, the space pointed to by p (if p is not NULL on input) is not modified , and ok is returned as false to indicate that the reallocation failed. If the size decreases or remains the same, then the method always succeeds (ok is returned as true). The return value of the function is the unchanged input value of p.

Typical usage: the following code fragment allocates an array of 10 int's, and then increases the size of the array to 20 int's. If the SPEX_malloc succeeds but the SPEX_realloc fails, then the array remains unmodified, of size 10.

```
int *p ;
p = SPEX_malloc (10 * sizeof (int)) ;
if (p == NULL) { error here ... }
printf ("p points to an array of size 10 * sizeof (int)\n") ;
bool ok ;
p = SPEX_realloc (20, 10, sizeof (int), p, &ok) ;
if (ok) printf ("p has size 20 * sizeof (int)\n") ;
else printf ("realloc failed; p still has size 10 * sizeof (int)\n") ;
SPEX_free (p) ;
```

4.3.4 SPEX_free: free allocated memory

```
void SPEX_free
(
    void *p         // Pointer to memory space to free
) ;
```

SPEX_free frees the memory previously allocated by a call to SPEX_calloc, SPEX_malloc, or SPEX_realloc. Results are undefined if p is not NULL on input, but was not obtained from one of these three functions. If p is NULL on input, then no action is taken (this is not an error condition). To guard against freeing the same memory space twice, the following macro SPEX_FREE is provided, which calls SPEX_free and then sets the freed pointer to NULL.

```
#define SPEX_FREE(p)                        \
{                                           \
    SPEX_free (p) ;                         \
    (p) = NULL ;                            \
}
```

4.4 SPEX_options helper function

The SPEX_options structure contains numerous parameters that may be modified to change the behavior of the SPEX functions. Default values of these parameters will lead to good performance in most cases. The following helper functions are provided.

4.4.1 SPEX_create_default_options: create default SPEX_options structure

```
SPEX_info SPEX_create_default_options
(
    SPEX_options *option_handle
) ;
```

SPEX_create_default_options creates and returns a pointer to a SPEX_options struct with default parameters upon successful allocation, which are discussed in Section 3.6. To safely free the SPEX_options option structure, simply use SPEX_FREE(option). All functions that require SPEX_options option as an input argument can have a NULL pointer passed instead. In this case, the default value of the corresponding command option is used.

4.5 SPEX_matrix helper functions

These functions provide several utilities for a SPEX_matrix.

4.5.1 SPEX_matrix_allocate: allocate an m-by-n SPEX_matrix

```
SPEX_info SPEX_matrix_allocate
(
    SPEX_matrix *A_handle,  // matrix to allocate
    SPEX_kind kind,         // CSC, triplet, dense or SPEX_DYNAMIC_CSC
    SPEX_type type,         // mpz, mpq, mpfr, int64, or double
    int64_t m,              // # of rows
    int64_t n,              // # of columns
    int64_t nzmax,          // max # of entries
    bool shallow,           // if true, matrix is shallow. A->p, A->i, A->j,
                            // A->x are all returned as NULL and must be set
                            // by the caller.  All A->*_shallow are returned
                            // as true. Ignored for SPEX_DYNAMIC_CSC
                            // kind matrix.
    bool init,              // If true, and the data types are mpz, mpq, or
                            // mpfr, the entries of A->x are initialized
                            // (using the proper SPEX_mp*_init function).
                            // If false, the mpz, mpq, and mpfr arrays are
                            // allocated but not initialized. Meaningless
                            // for data types FP64 or INT64. Ignored if kind
                            // is SPEX_DYNAMIC_CSC or shallow is true.
    const SPEX_options option
) ;
```

SPEX_matrix_allocate allocates memory space for a m-by-n SPEX_matrix whose kind (CSC, triplet, dense, or dynamic CSC) and data type (mpz, mpq, mpfr, int64 or fp64) is specified. On input, the SPEX matrix that A_handle points to is NULL. On output, A_handle points to a SPEX matrix of specified type, kind and size.

For a CSC, triplet or dense matrix, if shallow is true, all components (A->p, A->i, A->j, A->x) are returned as NULL, and their shallow flags are all true. The pointers A->p, A->i, A->j, and/or A->x can then be assigned from arrays in the calling application. If shallow is false, the appropriate individual arrays are allocated (via SPEX_calloc). The second boolean parameter init is used if the entries are mpz_t, mpq_t, or mpfr_t. Specifically, if init is true, the individual entries within A->x.TYPE are initialized using the appropriate SPEX_mp*_init function. Otherwise, if init is false, the A->x.TYPE array is allocated (via SPEX_calloc) and left that way. They are not otherwise initialized, and attempting to access the values of these uninitialized entries will lead to undefined behavior.

For a SPEX_DYNAMIC_CSC matrix, type, shallow and init are ignored (since it only allows mpz_t entries). Moreover, each column of the returned SPEX_DYNAMIC_CSC matrix will be allocated as SPEX_vector with no space for any entries. Additional reallocation for each column are performed as entries are added to the matrix.

4.5.2 SPEX_matrix_free: free a SPEX_matrix

```
SPEX_info SPEX_matrix_free
(
    SPEX_matrix *A_handle, // matrix to free
    const SPEX_options option
) ;
```

SPEX_matrix_free frees the SPEX_matrix A. Note that the input of the function is the pointer to the pointer of a SPEX_matrix structure. This is because this function internally sets the pointer of a SPEX_matrix to be NULL to prevent potential segmentation fault that could be caused by double call to SPEX_matrix_free.

4.5.3 SPEX_matrix_copy: make a copy of a SPEX_matrix with a potentially different matrix-format and data-type

```
SPEX_info SPEX_matrix_copy
(
    SPEX_matrix *C_handle,  // matrix to create (never shallow)
    // inputs, not modified:
    SPEX_kind C_kind,       // C->kind: CSC, triplet, dense, or dynamic
    SPEX_type C_type,       // C->type: mpz_t, mpq_t, mpfr_t, int64_t, or double
    const SPEX_matrix A,    // matrix to make a copy of (may be shallow)
    const SPEX_options option
) ;
```

SPEX_matrix_copy makes a deep copy of a SPEX_matrix A as a new SPEX_matrix C, which can be any of the 16 matrix formats discussed in Section 3.8. That is, the new matrix C can be exactly the same as A or any other type or kind different than A. On input, the SPEX matrix that C_handle points to must be NULL and will be ignored, and A is a valid matrix that can be potentially shallow. On output, C_handle points to the matrix C, which is a copy of A of kind kind and type type.

Results are undefined for an invalid input matrix A. Though all matrices generated from any SPEX user-callable functions are valid, they could become invalid when user directly modifies their component(s). To check the validity of the input matrix, call SPEX_matrix_check (Section 4.5.5).

4.5.4 SPEX_matrix_nnz: get the number of entries in a SPEX_matrix

```
SPEX_info SPEX_matrix_nnz     // return # of entries in A, or -1 on error
(
    int64_t *nnz,
    const SPEX_matrix A,          // matrix to query
    const SPEX_options *option
) ;
```

SPEX_matrix_nnz returns an integer, nnz, which is equal to the number of entries in a SPEX_matrix A. For details regarding how the number of entries is obtained for different kinds of matrices, refer to Section 3.8. For any matrix with invalid dimension(s), nnz is returned as -1.

4.5.5 SPEX_matrix_check: check and optionally print a SPEX_matrix

```
SPEX_info SPEX_matrix_check     // returns a SPEX status code
(
    const SPEX_matrix A,        // matrix to check
    const SPEX_options option   // defines the print level
) ;
```

SPEX_matrix_check checks the validity of a SPEX_matrix A in any of the 16 matrix formats discussed in Section 3.8. In addition, it optionally prints the matrix and any error found with proper print level specified by option->print_level. Specifically, SPEX_matrix_check prints nothing for print_level=0 (default); or just errors for print_level=1; or errors and terse output of the matrix for print_level=2; or errors and detailed output of the matrix for print_level=3. As mentioned, if default settings are desired, option can be input as NULL.

4.6 SPEX_symbolic_analysis helper function

4.6.1 SPEX_symbolic_analysis_free: free a symbolic analysis structure

```
SPEX_info SPEX_symbolic_analysis_free
(
    SPEX_symbolic_analysis *S_handle, // Structure to be deleted
    const SPEX_options option
);
```

SPEX_symbolic_analysis_free frees the memory of the SPEX_symbolic_analysis S that S_handle points to. On output, the symbolic analysis S is set to NULL.

4.7 SPEX_factorization helper function

4.7.1 SPEX_factorization_free: Free a SPEX factorization

```
SPEX_info SPEX_factorization_free
(
    SPEX_factorization *F_handle, // Structure to be deleted
    const SPEX_options option
);
```

SPEX_factorization_free frees the memory of the SPEX_factorization F that F_handle points to, and sets F to NULL.

4.8 Miscellaneous Utility Functions

4.8.1 SPEX_version: Return version of the code

```
SPEX_info SPEX_version
(
    int version [3],            // SPEX major, minor, and sub version
    char date [128]             // date of this version
)
```

`SPEX_version` returns the library version and date. The `version` array contains the three version numbers that are available at compile-time `#define`'d values: `SPEX_VERSION_MAJOR`, `SPEX_VERSION_MINOR`, and `SPEX_VERSION_SUB`, in that order. The `SPEX_version` function allows the user application to check which version of SPEX it has been linked with. The three `#define`'d values allow the user application to know which version of SPEX was used at compile-time, which might not be the same version that was linked later on. The `date` is the string `SPEX_DATE`, in the form `"Mar 31, 2023"` for example. The string is null-terminated.

4.8.2 SPEX_determine_symmetry: Determine if a matrix is symmetric

```
SPEX_info SPEX_determine_symmetry
(
    bool *is_symmetric,        // true if symmetric
    SPEX_matrix A,             // Input matrix to be checked for symmetry
    const SPEX_options option  // Command options
);
```

`SPEX_determine_symmetry` checks if `A` has a symmetric pattern and is also numerically symmetric. If A is numerically a symmetric matrix and has a symmetric pattern, `is_symmetric` is returned as `true`. Otherwise, `is_symmetric` is returned as `false`.

4.8.3 SPEX_transpose: Transpose a CSC mpz matrix

```
SPEX_info SPEX_transpose
(
    SPEX_matrix *C_handle,      // C = A'
    SPEX_matrix A,              // Matrix to be transposed
    const SPEX_options option
);
```

`SPEX_transpose` sets $C = A^T$. Currently, it is only supported if A is CSC and `mpz_t`. Returns `SPEX_OK` if successful; otherwise it returns the appropriate error code.

4.9 SPEX_gmp: SPEX wrapper functions for GMP/MPFR

SPEX provides a wrapper class for all GMP and MPFR functions used by SPEX. The wrapper class provides error-handling for out-of-memory conditions that are not handled by the GMP and MPFR libraries. These wrapper functions are used inside all SPEX functions, wherever any GMP or MPFR functions are used. These functions may also be called by the end-user application.

Each wrapped function has the same name as its corresponding GMP/MPFR function with the added prefix `SPEX_`. For example, the default GMP function `mpz_mul` is changed to `SPEX_mpz_mul`. Each SPEX GMP/MPFR function returns `SPEX_OK` if successful or the correct error code if not. The following table gives a brief list of each currently covered SPEX GMP/MPFR function. Each function is declared in `SPEX.h` and defined in `SPEX/SPEX_Util/Source/SPEX_gmp.c`.

MPFR Function	`SPEX_MPFR` **Function**	**Description**
`mpfr_asprintf(&buff, fmt, ...)`	`SPEX_mpfr_asprintf(&buff, fmt, ...)`	Print format to allocated string
`mpfr_clear(x)`	`SPEX_mpfr_clear(x)`	Safely free `mpfr_t` value
`mpfr_div_d(x, y, z, rnd)`	`SPEX_mpfr_div_d(x, y, z, rnd)`	$x = y/z$ (double)
`mpfr_free_cache()`	`SPEX_mpfr_free_cache()`	Free all caches and pools used by MPFR internally
`mpfr_free_str(buff)`	`SPEX_mpfr_free_str(buff)`	Free string allocated by MPFR
`x = mpfr_get_d(y, rnd)`	`SPEX_mpfr_get_d(x, y, rnd)`	(double) $x = y$
`mpfr_get_q(x, y)`	`SPEX_mpfr_get_q(x, y, rnd)`	(`mpq_t`) $x = y$
`x = mpfr_get_si(y, rnd)`	`SPEX_mpfr_get_si(x, y, rnd)`	(`int64_t`) $x = y$
`r = mpfr_get_z(x, y, rnd)`	`SPEX_mpfr_get_z(x, y, rnd)`	(`mpz_t`) $x = y$
`mpfr_init2(x, size)`	`SPEX_mpfr_init2(x, size)`	Initialize x with size bits
`mpfr_mul(x, y, z, rnd)`	`SPEX_mpfr_mul(x, y, z, rnd)`	$x = y * z$ (`mpfr_t`)
`mpfr_mul_d(x, y, z, rnd)`	`SPEX_mpfr_mul_d(x, y, z, rnd)`	$x = y * z$ (double)
`mpfr_set(x, y, rnd)`	`SPEX_mpfr_set(x, y, rnd)`	$x = y$
`mpfr_set_d(x, y, rnd)`	`SPEX_mpfr_set_d(x, y, rnd)`	$x = y$ (double)
`mpfr_set_null(x)`	`SPEX_mpfr_set_null(x)`	Initialize the (pointer) contents of a `mpfr_t` value
`mpfr_set_prec (x, size)`	`SPEX_mpfr_set_prec(x, size)`	Set the precision of an `mpfr_t` number
`mpfr_set_q(x, y, rnd)`	`SPEX_mpfr_set_q(x, y, rnd)`	$x = y$ (`mpq_t`)
`mpfr_set_si(x, y, rnd)`	`SPEX_mpfr_set_si(x, y, rnd)`	$x = y$ (`int64_t`)
`mpfr_set_z(x, y, rnd)`	`SPEX_mpfr_set_z(x, y, rnd)`	$x = y$ (`mpz_t`)
`sgn = mpfr_sgn(x)`	`SPEX_mpfr_sgn(sgn, x)`	$sgn = \mathrm{sgn}(x)$
`mpfr_ui_pow_ui(x, y, z, rnd)`	`SPEX_mpfr_ui_pow_ui(x, y, z, rnd)`	$x = y^z$ (`uint64_t`)

GMP Function	SPEX_GMP **Function**	**Description**
`gmp_fscanf(fp, fmt, ...)`	`SPEX_gmp_fscanf(fp, fmt, ...)`	Read from file fp
`mpq_abs(x, y)`	`SPEX_mpq_abs(x, y)`	$x = \|y\|$
`mpq_add(x, y, z)`	`SPEX_mpq_add(x, y, z)`	$x = y + z$
`mpq_canonicalize(x)`	`SPEX_mpq_canonicalize(x)`	Canonicalize x
`mpq_clear(x)`	`SPEX_mpq_clear(x)`	Safely free `mpq_t` value
`r = mpq_cmp(x, y)`	`SPEX_mpq_cmp(r, x, y)`	$r = \text{sgn}(x - y)$
`r = mpq_cmp_ui(x, n, d)`	`SPEX_mpq_cmp_ui(r, x, n, d)`	$r = \text{sgn}(x - n/d)$ (`uint64_t/uint64_t`)
`mpq_div(x, y, z)`	`SPEX_mpq_div(x, y, z)`	$x = y/z$
`r = mpq_equal(x, y)`	`SPEX_mpq_equal(r, x, y)`	$r \neq 0$ if $x = y$, $r = 0$ if $x \neq y$
`x = mpq_get_d(y)`	`SPEX_mpq_get_d(x, y)`	(double) $x = y$
`mpq_init(x)`	`SPEX_mpq_init(x)`	Initialize x
`mpq_mul(x, y, z)`	`SPEX_mpq_mul(x, y, z)`	$x = y * z$
`mpq_neg(x, y)`	`SPEX_mpq_neg(x, y)`	$x = -y$
`mpq_set(x, y)`	`SPEX_mpq_set(x, y)`	$x = y$
`mpq_set_d(x, y)`	`SPEX_mpq_set_d(x, y)`	$x = y$ (double)
`mpq_set_den(x, y)`	`SPEX_mpq_set_den(x, y)`	$den(x) = y$
`mpq_set_null(x)`	`SPEX_mpq_set_null(x)`	Initialize the (pointer) contents of a `mpq_t` value
`mpq_set_num(x, y)`	`SPEX_mpq_set_num(x, y)`	$num(x) = y$
`mpq_set_si(x, y, z)`	`SPEX_mpq_set_si(x, y, z)`	$x = y/z$ (`int64_t/uint64_t`)
`mpq_set_ui(x, y, z)`	`SPEX_mpq_set_ui(x, y, z)`	$x = y/z$ (`uint64_t/uint64_t`)
`mpq_set_z(x, y)`	`SPEX_mpq_set_z(x, y)`	$x = y$ (`mpz`)
`sgn = mpq_sgn(x)`	`SPEX_mpq_sgn(sgn, x)`	$sgn = \text{sgn}(x)$
`mpz_abs(x, y)`	`SPEX_mpz_abs(x, y)`	$x = \|y\|$
`mpz_add(x, y, z)`	`SPEX_mpz_add(x, y, z)`	$x = y + z$
`mpz_addmul(x, y, z)`	`SPEX_mpz_addmul(x, y, z)`	$x = x + y * z$
`mpz_cdiv_q(q, x, y)`	`SPEX_mpz_cdiv_q(q, x, y)`	$q = \text{ceil}(x/y)$
`mpz_cdiv_qr(q, r, x, y)`	`SPEX_mpz_cdiv_qr(q, r, x, y)`	$q = \text{ceil}(x/y), r = x - q * y$
`mpz_clear(x)`	`SPEX_mpz_clear(x)`	Safely free `mpz_t` value
`r = mpz_cmp(x, y)`	`SPEX_mpz_cmp(r, x, y)`	$r = \text{sgn}(x - y)$
`r = mpz_cmp_ui(x, y)`	`SPEX_mpz_cmp_ui(r, x, y)`	$r = \text{sgn}(x - y)$ (`uint64_t`)
`r = mpz_cmpabs(x, y)`	`SPEX_mpz_cmpabs(r, x, y)`	$r = \text{sgn}(\|x\| - \|y\|)$
`r = mpz_cmpabs_ui(x, y)`	`SPEX_mpz_cmpabs_ui(r, x, y)`	$r = \text{sgn}(\|x\| - \|y\|)$ (`uint64_t`)
`mpz_divexact(x, y, z)`	`SPEX_mpz_divexact(x, y, z)`	$x = y/z$
`mpz_fdiv_q(q, x, y)`	`SPEX_mpz_fdiv_q(q, x, y)`	$q = \text{floor}(x/y)$
`gcd = mpz_gcd(x, y)`	`SPEX_mpz_gcd(gcd, x, y)`	$gcd = \text{gcd}(x, y)$
`x = mpz_get_d(y)`	`SPEX_mpz_get_d(x, y)`	(double) $x = y$
`x = mpz_get_si(y)`	`SPEX_mpz_get_si(x, y)`	(`int64_t`) $x = y$
`mpz_init(x)`	`SPEX_mpz_init(x)`	Initialize x
`mpz_init2(x, size)`	`SPEX_mpz_init2(x, size)`	Initialize x to size bits
`lcm = mpz_lcm(x, y)`	`SPEX_mpz_lcm(lcm, x, y)`	$lcm = \text{lcm}(x, y)$
`mpz_mul(x, y, z)`	`SPEX_mpz_mul(x, y, z)`	$x = y * z$
`mpz_mul_si(x, y, z)`	`SPEX_mpz_mul_si(x, y, z)`	$x = y * z$(`int64_t`)
`mpz_neg(x, y)`	`SPEX_mpz_neg(x, y)`	$x = -y$
`mpz_set(x, y)`	`SPEX_mpz_set(x, y)`	$x = y$ (`mpz_t`)
`mpz_set_null(x)`	`SPEX_mpz_set_null(x)`	Initialize the (pointer) contents of a `mpz_t` value
`mpz_set_si(x, y)`	`SPEX_mpz_set_si(x, y)`	$x = y$ (`int64_t`)
`mpz_set_ui(x, y)`	`SPEX_mpz_set_ui(x, y)`	$x = y$ (`uint64_t`)
`sgn = mpz_sgn(x)`	`SPEX_mpz_sgn(sgn, x)`	$sgn = \text{sgn}(x)$
`size = mpz_sizeinbase(x, base)`	`SPEX_mpz_sizeinbase(size, x, base)`	size of x in base
`mpz_sub(x, y, z)`	`SPEX_mpz_sub(x, y, z)`	$x = y - z$
`mpz_submul(x, y, z)`	`SPEX_mpz_submul(x, y, z)`	$x = x - y * z$
`mpz_swap(x, y)`	`SPEX_mpz_swap(x, y)`	Swap the values of x and y

If additional GMP and MPFR functions are needed in the end-user application, this wrapper mechanism can be extended to those functions, which requires the source files of the SPEX library to be revised, (i.e., both `SPEX.h` and `SPEX_gmp.c`). Below are instructions on how to do this.

Consider a GMP function `void gmpfunc(TYPEa a, TYPEb b, ...)`, where `TYPEa` and `TYPEb` can be GMP type data (`mpz_t`, `mpq_t` and `mpfr_t`, for example) or non-GMP type data (`int`, `double`, for example). A wrapper for a new GMP or MPFR function can be created by following this outline:

```
SPEX_info SPEX_gmpfunc
(
    TYPEa a,
    TYPEb b,
    ...
)
{
    // Start the GMP Wrappter
    // uncomment one of the following:
    // If this function is not modifying any GMP/MPFR type variable, use
    //SPEX_GMP_WRAPPER_START;
    // If this function is modifying mpz_t type (say TYPEa = mpz_t), use
    //SPEX_GMPZ_WRAPPER_START(a) ;
    // If this function is modifying two variables of mpz_t type (say
    // TYPEa = mpz_t, TYPEb = mpz_t), use
    //SPEX_GMPZ_WRAPPER_START2(a, b) ;
    // If this function is modifying mpq_t type (say TYPEa = mpq_t), use
    //SPEX_GMPQ_WRAPPER_START(a) ;
    // If this function is modifying mpfr_t type (say TYPEa = mpfr_t), use
    //SPEX_GMPFR_WRAPPER_START(a) ;

    // Call the GMP function
    gmpfunc(a,b,...) ;

    //Finish the wrapper and return ok if successful.
    SPEX_GMP_WRAPPER_FINISH;
    return SPEX_OK;
}
```

All of the wrapped GMP/MPFR functions always return `SPEX_info` to the caller. Therefore, for some GMP/MPFR functions that have their own return value, that value becomes a parameter instead. For example, for the GMP function
`int mpq_cmp(const mpq_t a, const mpq_t b)`, the return value becomes a parameter of the wrapped function. In general, a GMP/MPFR function in the form of
`TYPEr gmpfunc(TYPEa a, TYPEb b, ...)`, the wrapped function can be constructed as follows:

```
SPEX_info SPEX_gmpfunc
(
    TYPEr *r,        // return value of the GMP/MPFR function
    TYPEa a,
    TYPEb b,
    ...
)
{
    // Start the GMP Wrapper (see details above)
    //SPEX_GMP_WRAPPER_START;

    // Call the GMP function
    *r = gmpfunc(a,b,...) ;

    //Finish the wrapper and return ok if successful.
    SPEX_GMP_WRAPPER_FINISH;
    return SPEX_OK;
}
```

4.10 SPEX Helper Macros

In addition to the functionality described in this section; SPEX offers several helper macros to increase ease for the end user application. The first two macros form a simple try/catch mechanism which can be used to wrap functions for error handling. The next two give an easy interface to access individual entries in a matrix.

4.10.1 `SPEX_TRY` and `SPEX_CATCH`

In a robust application, the return values from SPEX should be checked and properly handled in the case an error occurs. SPEX is written in C and thus it cannot rely on the try/catch mechanism of C++. Thus, `SPEX_TRY` and `SPEX_CHECK` aim to provide a similar mechanism within the SPEX environment. We provide `SPEX_TRY` and leave `SPEX_CATCH` to the user to define.

```
#define SPEX_TRY(method)            \
{                                   \
    SPEX_info info = (method) ;     \
    if (info != SPEX_OK)            \
    {                               \
        SPEX_CATCH (info) ;         \
    }                               \
}
```

An example definition of a `SPEX_CATCH` is below. This example assumes that the user needs to free a matrix and return an error code.

```
#define SPEX_CATCH(info)                        \
{                                               \
    SPEX_matrix_free (&A, NULL) ;               \
    fprintf (stderr, "SPEX failed: info %d,"    \
    "line %d, file %s\n",                       \
    info, __LINE__, __FILE__) ;                 \
    return (info) ;                             \
}
```

With this mechanism, the user can safely wrap any SPEX function which returns `SPEX_info` with `SPEX_TRY`.

4.10.2 `SPEX_1D`: Access matrix entries with 1D linear indexing.

```
#define SPEX_1D(A,k,type) ((A)->x.type [k])
```

This access the kth entry of a matrix stored in any kind (CSC, triplet, dense) of any type (`mpq_t, mpz_t, int64_t, double, int`). For example, to return the kth entry of a CSC matrix with `mpz_t` data types, one would use `SPEX_1D(A, k, mpz)`.

4.10.3 `SPEX_2D`: Access dense matrix entries with 2D indexing.

```
#define SPEX_2D(A,i,j,type) SPEX_1D (A, (i)+(j)*((A)->m), type)
```

This accesses the (i, j) entry of a dense matrix of any type (`mpq_t, mpz_t, int64_t`, `double, int`). For example to return the (i, j) entry of a dense matrix with `mpq_t` data types, one would use `SPEX_2D(A, i, j, mpq)`. This macro cannot be used with sparse (CSC or dynamic CSC) or triplet matrices.

CHAPTER 5

SPEX LU

5.1 Overview

SPEX LU is a software package designed to exactly solve unsymmetric sparse linear systems, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{Q}^{n \times n}$, $\mathbf{b} \in \mathbb{Q}^{n \times r}$, and $\mathbf{x} \in \mathbb{Q}^{n \times r}$. This package performs a left-looking, roundoff-error-free (REF) LU factorization $PAQ = LDU$, where L and U are integer, D is diagonal, and P and Q are row and column permutations, respectively. Note that, in order to solve a linear system, the matrix D is never explicitly computed nor needed; thus this package uses only the matrices L and U. The theory associated with this code is the Sparse Left-looking Integer-Preserving (SLIP) LU factorization [7]. Aside from solving sparse linear systems exactly, one of the key goals of this package is to provide a framework for other solvers to benchmark the reliability and stability of their linear solvers, as our final solution vector x is guaranteed to be exact. SPEX LU is written in ANSI C and is accompanied by a MATLAB interface.

Version 1.1.2 of SPEX Left LU was published in ACM TOMS as: Lourenco, C., Chen, J., Moreno-Centeno, E., & Davis, T. A. (2022). Algorithm 1021: SPEX Left LU, Exactly Solving Sparse Linear Systems via a Sparse Left-looking Integer-preserving LU Factorization. ACM Transactions on Mathematical Software (TOMS), 48(2), 1-23.

5.2 Licensing

Copyright: The copyright of this software is held by Christopher Lourenco, Jinhao Chen, Erick Moreno-Centeno, and Timothy A. Davis.

Contact Info: Contact Chris Lourenco, chrisjlourenco@gmail.com, or Tim Davis, timdavis@aldenmath.com, davis@tamu.edu, or DrTimothyAldenDavis@gmail.com

License: This software package is dual licensed under the GNU General Public License version 2 or the GNU Lesser General Public License version 3. Details of this license are in `SPEX/License/license.txt`. For alternative licenses, please contact the authors.

5.3 Factorization and Solve Routines

To factorize and solve a linear system $A\mathbf{x} = \mathbf{b}$ via the SPEX Left LU factorization, a user must call analyze, factorize, and solve. The functions are explained below:

5.3.1 SPEX_lu_analyze: symbolic analysis for LU factorization

```
SPEX_info SPEX_lu_analyze
(
    SPEX_symbolic_analysis *S_handle, // symbolic analysis including
                                   // column permutation and nnz of L and U
    const SPEX_matrix A,           // Input matrix
    const SPEX_options option      // Control parameters, if NULL, use default
) ;
```

`SPEX_lu_analyze` performs symbolic analysis for the SPEX LU factorization. On input, the `SPEX_symbolic_analysis S` that `S_handle` points to is undefined; `A` must be a square matrix of `SPEX_CSC` kind; and `option` contains any command parameters (default settings are used if the `option` input is `NULL`). The `option->algo` must be either `SPEX_ALGORITHM_DEFAULT` or `SPEX_LU_LEFT`. On output, `S` contains the column preordering of `A` and estimates on the number of nonzeros in L and U. The type of column permutation can be selected by changing the value of `option->order`. COLAMD is the default fill-reducing ordering, but AMD can be effective if the nonzero pattern of `A` is mostly symmetric.

5.3.2 SPEX_lu_factorize: Compute the LU factorization of A

```
SPEX_info SPEX_lu_factorize
(
    // output:
    SPEX_factorization *F_handle,   // LU factorization
    // input:
    const SPEX_matrix A,            // matrix to be factored
    const SPEX_symbolic_analysis S, // symbolic analysis
    const SPEX_options option       // command options
) ;
```

`SPEX_lu_factorize` performs the left-looking LU factorization. On input, the `SPEX_factorization F` that `F_handle` points to is undefined; `A` must be a square matrix of `SPEX_CSC SPEX_MPZ` format; `S` is obtained from `SPEX_lu_analyze` that contains the column ordering of `A`; and `option` contains any command parameters (default settings are used if `option` is input as `NULL`). Note that `option->algo` must be either `SPEX_ALGORITHM_DEFAULT` or `SPEX_LU_LEFT`. The symbolic analysis `S` is typically constructed with the same input matrix `A`, or another matrix with the same sparsity pattern. However, the only strict requirement is that the symbolic factorization `S` is created by analyzing a matrix with the dimensions as the input matrix to the numerical factorization. If the sparsity pattern is different, the numerical factorization can still be successful but the fill-in could be excessive.

On output, A, S, and option are unmodified and F contains the SPEX LU factorization of A. If any error occurs, F is returned as NULL, and an appropriate error code is returned.

5.3.3 SPEX_lu_solve: solve the linear system

```
SPEX_info SPEX_lu_solve // solves the linear system LD^(-1)U x = b
(
    // Output
    SPEX_matrix *x_handle,     // rational solution to the system
    // input/output:
    SPEX_factorization F,      // The LU factorization.
    // input:
    const SPEX_matrix b,       // right hand side vector
    const SPEX_options option  // Command options
) ;
```

SPEX_lu_solve obtains the solution of mpq_t type to the linear system $A\mathbf{x} = \mathbf{b}$ upon a successful factorization. This function may be called after a successful return from SPEX_lu_factorize.

On input, SPEX_matrix x that x_handle points to is undefined; F must be a valid LU factorization; and b must be a dense mpz_t matrix with same number of rows as F->L; Default settings are used if option is input as NULL. Upon successful completion, the function returns SPEX_OK, and x contains the solution of mpq_t type with dense format to the linear system $Ax = b$. In case of failure, x is returned as NULL and the appropriate error code is returned.

5.3.4 SPEX_lu_backslash: solve a linear system

```
SPEX_info SPEX_lu_backslash
(
    // Output
    SPEX_matrix *X_handle,        // Final solution vector
    // Input
    SPEX_type type,               // Type of output desired. Must be
                                  // SPEX_FP64, SPEX_MPFR, or SPEX_MPQ
    const SPEX_matrix A,          // Input matrix of SPEX_CSC SPEX_MPZ
    const SPEX_matrix b,          // Right hand side vector(s). Must be
                                  // SPEX_DENSE SPEX_MPZ
    const SPEX_options option     // Command options (Default if NULL)
) ;
```

SPEX_lu_backslash solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a dense matrix of mpq_t, mpfr_t or double entries. This function performs symbolic analysis, factorization, and solving all in one function. It can be thought of as an exact version of an LU factorization based MATLAB sparse backslash.

On input, SPEX_matrix x that X_handle points to is undefined. type must be one of: SPEX_MPQ, SPEX_MPFR or SPEX_FP64 to specify the data type of the solution entries. A should be a square CSC mpz_t matrix while b should be a dense mpz_t matrix. In addition, A->m should be equal to b->m. Note that option->algo must be either SPEX_ALGORITHM_DEFAULT or SPEX_LU_LEFT. Default settings are used if option is input as NULL.

Upon successful completion, the function returns `SPEX_OK`, and `x` contains the solution of data type specified by `type` to the linear system $A\mathbf{x} = \mathbf{b}$. In case of failure, `x` is returned as `NULL` and the appropriate error code is returned.

CHAPTER 6

SPEX CHOLESKY AND LDL

6.1 Overview

SPEX Cholesky is a software package designed to exactly solve symmetric linear systems, $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{Q}^{n\times n}$, $\mathbf{b} \in \mathbb{Q}^{n\times r}$, and $\mathbf{x} \in \mathbb{Q}^{n\times r}$. The package contains an exact Cholesky factorization, appropriate for SPD matrices, and an exact LDL factorization, appropriate for symmetric matrices with nonzero leading principle minors. This package performs either a left-looking or up-looking sparse roundoff-error-free Cholesky or LDL factorization $PAP^T = LDL^T$ where L is integer, and P is the symmetric permutation.

Note that, in order to solve a linear system, the matrix D is never explicitly computed nor needed; thus this package uses only the matrix L. The theory associated with this code can be found at [8]. SPEX Cholesky is written in ANSI C and is accompanied by MATLAB and Python interfaces.

6.2 Licensing

Copyright: The copyright of this software is held by Christopher Lourenco, Lorena Mejia Domenzain, Jinhao Chen, Erick Moreno-Centeno, and Timothy A. Davis.

Contact Info: Contact Chris Lourenco, chrisjlourenco@gmail.com, or Tim Davis, timdavis@aldenmath.com, davis@tamu.edu, or DrTimothyAldenDavis@gmail.com

License: This software package is dual licensed under the GNU General Public License version 2 or the GNU Lesser General Public License version 3. Details of this license are in `SPEX/License/license.txt`. For alternative licenses, please contact the authors.

6.3 Factorization and Solve Routines

To factorize and solve a linear system $A\mathbf{x} = \mathbf{b}$ via the SPEX Cholesky or SPEX LDL factorization, a user must call analyze, factorize, and solve. There are four user callable functions for each SPEX Cholesky and SPEX LDL factorization. The functions are explained below:

6.3.1 SPEX_cholesky_analyze: symbolic analysis for Cholesky factorization

```
SPEX_info SPEX_cholesky_analyze
(
    // Output
    SPEX_symbolic_analysis* S_handle, // Symbolic analysis data structure
    // Input
    const SPEX_matrix A,         // Input matrix of SPEX_CSC
    const SPEX_options option    // Command options (Default if NULL)
);
```

`SPEX_cholesky_analyze` performs symbolic analysis for the SPEX Cholesky factorization. On input, the `SPEX_symbolic_analysis S` that `S_handle` points to is undefined; `A` must be an SPD matrix of `SPEX_CSC` kind; and `option` contains any command parameters (default settings are used if `option` is input as `NULL`). `option->algo` must be either `SPEX_ALGORITHM_DEFAULT`, `SPEX_CHOL_LEFT`, or `SPEX_CHOL_UP`. On output, `S` contains the row and column ordering of `A`, the exact number of nonzeros in L, the elimination tree of `A`, and the column pointers of L. The type of ordering can be chosen with `option->order`. AMD is the default ordering, but sometimes COLAMD can give good results.

6.3.2 SPEX_cholesky_factorize: Compute the Cholesky factorization of A

```
SPEX_info SPEX_cholesky_factorize
(
    // Output
    SPEX_factorization *F_handle,  // Cholesky factorization struct
    //Input
    const SPEX_matrix A,       // CSC MPZ Matrix to be factored
    const SPEX_symbolic_analysis S,// Symbolic analysis struct from
                               // SPEX_cholesky_analyze.
    const SPEX_options option  // command options, option->algo can be
                               // either SPEX_CHOL_UP (default) or SPEX_CHOL_LEFT.
);
```

`SPEX_cholesky_factorize` performs the SPEX Cholesky factorization via either the up-looking (default) or left-looking manner (specified by `option->algo`). On input, the `SPEX_factorization F` that `F_handle` points to is undefined; `A` must be a symmetric positive definite matrix of `SPEX_CSC SPEX_MPZ` format; `S` is obtained from `SPEX_cholesky_analyze` that contains the column/row ordering of `A`; and `option` contains any command parameters (default settings are used if `option` is input as `NULL`). On output, `A`, `S`, and `option` are unmodified and `F` contains the SPEX Cholesky factorization of `A`.

The symbolic analysis `S` must be obtained by a call to `SPEX_cholesky_analyze` with an input matrix `A` that has the identical sparsity pattern as the matrix given as input to this `SPEX_cholesky_factorize` method. Results are undefined if this condition does not hold.

If error occurs, `F` is returned as `NULL`, and an appropriate error code is returned.

6.3.3 SPEX_cholesky_solve: solve the linear system

```
SPEX_info SPEX_cholesky_solve   // solves the linear system LD^(-1)L^T x = b
(
    // Output
    SPEX_matrix *x_handle,  // rational solution to the system.
    // input/output:
    SPEX_factorization F,   // The non-updatable Cholesky factorization.
    // input:
    const SPEX_matrix b,    // Right hand side vector
    const SPEX_options option // command options
);
```

SPEX_cholesky_solve obtains the solution of mpq_t type to the linear system $A\mathbf{x} = \mathbf{b}$ upon a successful factorization. This function may be called after a successful return from SPEX_cholesky_factorize.

On input, SPEX_matrix x that x_handle points to is undefined; F must be a valid Cholesky factorization and b must be dense mpz_t matrix with the same number of rows as F->L. Default settings are used if option is input as NULL. Upon successful completion, the function returns SPEX_OK, and x contains the solution of mpq_t type with dense format to the linear system $A\mathbf{x} = \mathbf{b}$. In case of failure, x is returned as NULL and the appropriate error code is returned.

6.3.4 SPEX_cholesky_backslash: solve a linear system

```
SPEX_info SPEX_cholesky_backslash
(
    // Output
    SPEX_matrix *x_handle,        // Final solution vector(s)
    // Input
    SPEX_type type,               // Type of output desired. Must be
                                  // SPEX_FP64, SPEX_MPFR, or SPEX_MPQ
    const SPEX_matrix A,          // Input matrix of SPEX_CSC SPEX_MPZ
    const SPEX_matrix b,          // Right hand side vector(s). Must be
                                  // SPEX_DENSE SPEX_MPZ
    const SPEX_options option     // Command options (Default if NULL)
);
```

SPEX_cholesky_backslash solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a dense matrix of mpq_t, mpfr_t or double entries. This function performs symbolic analysis, factorization, and solving all in one function. It can be thought of as an exact version of MATLAB sparse backslash for symmetric positive-definite matrices. If A is not symmetric positive-definite, this function will return an appropriate error code.

On input, SPEX_matrix x that x_handle points to is undefined. type must be one of: SPEX_MPQ, SPEX_MPFR or SPEX_FP64 to specify the data type of the solution entries. A should be a square CSC mpz_t matrix while b should be a dense mpz_t matrix. In addition, A->m should be equal to b->m. Note that option->algo must be either SPEX_ALGORITHM_DEFAULT, SPEX_CHOL_LEFT, or SPEX_CHOL_UP. Default settings are used if option is input as NULL.

Upon successful completion, the function returns SPEX_OK, and x contains the solution of data type specified by type to the linear system $A\mathbf{x} = \mathbf{b}$. In case of failure, x is returned as NULL and the appropriate error code is returned.

6.3.5 SPEX_ldl_analyze: symbolic analysis for LDL factorization

```
SPEX_info SPEX_ldl_analyze
(
    // Output
    SPEX_symbolic_analysis *S_handle, // Symbolic analysis data structure
    // Input
    const SPEX_matrix A,          // Input matrix of SPEX_CSC
    const SPEX_options option     // Command options (Default if NULL)
);
```

SPEX_ldl_analyze performs symbolic analysis for the SPEX LDL factorization. On input, the SPEX_symbolic_analysis S that S_handle points to is undefined; A must be a symmetric matrix of SPEX_CSC kind with a zero-free diagonal; and option contains the command parameters (default settings are used if option is input as NULL). option->algo must be either SPEX_ALGORITHM_DEFAULT, SPEX_LDL_LEFT, or SPEX_LDL_UP. On output, S contains the row and column ordering of A, the exact number of nonzeros in L, the elimination tree of A, and the column pointers of L. The type of ordering can be chosen with option->order. AMD is the default ordering, but sometimes COLAMD can give good results.

6.3.6 SPEX_ldl_factorize: Compute the LDL factorization of A

```
SPEX_info SPEX_ldl_factorize
(
    // Output
    SPEX_factorization *F_handle,  // ldl factorization struct
    //Input
    const SPEX_matrix A,       // CSC MPZ Matrix to be factored
    const SPEX_symbolic_analysis S,// Symbolic analysis struct from
                               // SPEX_ldl_analyze.
    const SPEX_options option  // command options,
);
```

SPEX_ldl_factorize performs the SPEX LDL factorization via either the up-looking factorization or left-looking factorization (based on the choice of option->algo). On input, the SPEX_factorization F that F_handle points to is undefined; A must be a symmetric matrix of SPEX_CSC SPEX_MPZ format with a zero-free diagonal; S is obtained from SPEX_ldl_analyze that contains the column/row ordering of A; and option contains any command parameters (default settings are used if option is input as NULL). On output, A, S, and option are unmodified and F contains the SPEX LDL factorization of A.

The symbolic analysis S must be obtained by a call to SPEX_ldl_analyze with an input matrix A that has the identical sparsity pattern as the matrix given as input to this SPEX_ldl_factorize method. Results are undefined if this condition does not hold.

If error occurs, F is returned as NULL, and an appropriate error code is returned.

6.3.7 SPEX_ldl_solve: solve the linear system

```
SPEX_info SPEX_ldl_solve   // solves the linear system LD^(-1)L^T x = b
(
    // Output
    SPEX_matrix *x_handle,  // rational solution to the system.
    // input/output:
    SPEX_factorization F,   // The ldl factorization of A
    // input:
    const SPEX_matrix b,    // Right hand side vector
    const SPEX_options option // command options
);
```

SPEX_ldl_solve obtains the solution of mpq_t type to the linear system $A\mathbf{x} = \mathbf{b}$ upon a successful factorization. This function may be called after a successful return from SPEX_ldl_factorize.

On input, SPEX_matrix x that x_handle points to is undefined; F must be a valid LDL factorization and b must be dense mpz_t with same number of rows as F->L. Default settings are used if option is input as NULL. Upon successful completion, the function returns SPEX_OK, and x contains the solution of mpq_t type with dense format to the linear system $A\mathbf{x} = \mathbf{b}$. In case of failure, x is returned as NULL and the appropriate error code is returned.

6.3.8 SPEX_ldl_backslash: solve a linear system

```
SPEX_info SPEX_ldl_backslash
(
    // Output
    SPEX_matrix *x_handle,       // Final solution vector(s)
    // Input
    SPEX_type type,              // Type of output desired. Must be
                                 // SPEX_FP64, SPEX_MPFR, or SPEX_MPQ
    const SPEX_matrix A,         // Input matrix of SPEX_CSC SPEX_MPZ
    const SPEX_matrix b,         // Right hand side vector(s). Must be
                                 // SPEX_DENSE SPEX_MPZ
    const SPEX_options option    // Command options (Default if NULL)
);
```

SPEX_ldl_backslash solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a dense matrix of mpq_t, mpfr_t or double entries. This function performs symbolic analysis, factorization, and solving all in one line. If A is not symmetric with nonzero leading principle minors, this function will return an appropriate error code and LU factorization should be used.

On input, SPEX_matrix x that x_handle points to is undefined. type must be one of: SPEX_MPQ, SPEX_MPFR or SPEX_FP64 to specify the data type of the solution entries. A should be a square CSC mpz_t matrix while b should be a dense mpz_t matrix. In addition, A->m should be equal to b->m. Note that option->algo must be either SPEX_ALGORITHM_DEFAULT, SPEX_LDL_LEFT, or SPEX_LDL_UP. Default settings are used if option is input as NULL.

Upon successful completion, the function returns `SPEX_OK`, and `x` contains the solution of data type specified by `type` to the linear system $A\mathbf{x} = \mathbf{b}$. In case of failure, `x` is returned as `NULL` and the appropriate error code is returned.

CHAPTER 7

SPEX BACKSLASH

7.1 Overview

SPEX Backslash is a software package designed to exactly solve sparse linear systems, $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{Q}^{n \times n}$, $b \in \mathbb{Q}^{n \times r}$, and $x \in \mathbb{Q}^{n \times r}$. This package acts as either a wrapper to the SPEX LU, Cholesky, and LDL backslash methods, or it can determine the appropriate factorization to apply based on the structure of the input matrix.

SPEX Backslash is written in ANSI C and is accompanied by MATLAB and Python interfaces.

7.2 Licensing

Copyright: The copyright of this software is held by Christopher Lourenco, Lorena Mejia Domenzain, Jinhao Chen, Erick Moreno-Centeno, and Timothy A. Davis.

Contact Info: Contact Chris Lourenco, chrisjlourenco@gmail.com, or Tim Davis, timdavis@aldenmath.com, davis@tamu.edu, or DrTimothyAldenDavis@gmail.com

License: This software package is dual licensed under the GNU General Public License version 2 or the GNU Lesser General Public License version 3. Details of this license are in `SPEX/License/license.txt`. For alternative licenses, please contact the authors.

7.3 SPEX_backslash: Exactly solve sparse linear systems

```
SPEX_info SPEX_backslash
(
    // Output
    SPEX_matrix *X_handle,       // On output: Final solution vector
                                 // On input: undefined
    // Input
    SPEX_type type,              // Type of output desired. Must be
                                 // SPEX_FP64, SPEX_MPFR, or SPEX_MPQ
    const SPEX_matrix A,         // Input matrix of SPEX_CSC SPEX_MPZ
    const SPEX_matrix b,         // Right hand side vector(s). Must be
                                 // SPEX_DENSE SPEX_MPZ
    const SPEX_options option    // Command options (Default if NULL)
);
```

`SPEX_backslash` exactly solves the linear system $A\mathbf{x} = \mathbf{b}$ using the appropriate factorization. On input, `SPEX_matrix x` that `X_handle` points to is undefined. `type` must be one of: `SPEX_MPQ`, `SPEX_MPFR` or `SPEX_FP64` to specify the data type of the solution entries. `A` should be a square CSC `mpz_t` matrix while `b` should be a dense `mpz_t` matrix. In addition, `A->m` should be equal to `b->m`. Default settings are used if `option` is input as `NULL`.

The behavior of `SPEX_Backslash` depends on the `option->algo` parameter. If `option->algo` is left as default: the symmetry of A is checked. If A is symmetric, an up-looking LDL factorization is attempted. If the factorization succeeds, $\mathbf{x}$ is returned. Otherwise, a left-looking LU factorization is attempted. If `option->algo` is not set to `SPEX_ALGORITHM_DEFAULT` the request algorithm requested is used with no substitutions. For example, if `option->algo` is set to `SPEX_CHOL_LEFT`, then `SPEX_Backslash` essentially serves as a wrapper to the SPEX left-looking Cholesky factorization method, `SPEX_cholesky_backslash`. An appropriate error code is returned if the selected algorithm is not appropriate for the given matrix (e.g., if `option->algo` is `SPEX_LDL_LEFT` but the matrix is not symmetric).

Upon successful completion, the function returns `SPEX_OK`, and `x` contains the solution of data type specified by `type` to the linear system $A\mathbf{x} = \mathbf{b}$. If an error occurs, `x` is returned as `NULL` in `X_handle`, and the appropriate error code is returned.

CHAPTER 8

USING SPEX IN MATLAB

The MATLAB interface of SPEX can be installed by navigating to the `MATLAB` folder and typing `spex_mex_install`. Doing so installs SPEX and allows the use of four MATLAB functions `spex_lu_backslash.m`, `spex_cholesky_backslash.m`, `spex_ldl_backslash.m` and `spex_backslash.m`. Section 8.1 describes the `option` input parameter for each method. The use of each SPEX method is discussed in Section 8.2. The `SPEX/SPEX/MATLAB` folder must be in your MATLAB path.

8.1 Optional parameter settings

The SPEX MATLAB interface includes an `option` struct as an optional input parameter that modifies the behavior of each method. If this parameter is not provided, default parameter settings are used. The elements of the `option` struct are listed below. Any fields not present in the struct are treated as their default values.

- `option.pivot`: This parameter is a string that controls the pivoting scheme used. When selecting a pivot entry in a given column, the factorization method uses one of the following pivoting strategies. This parameter only applies to LU factorization:
  - `'smallest'`: (default) smallest pivot,
  - `'diagonal'`: diagonal pivot if possible, otherwise smallest pivot,
  - `'first'`: first nonzero pivot in each column,
  - `'tol smallest'`: diagonal pivot with a tolerance (`option.tol`) for the smallest pivot,
  - `'tol largest'`: diagonal pivot with a tolerance (`option.tol`) for the largest pivot,
  - `'largest'`: largest pivot.
- `option.order`: This parameter is a string controls the fill-reducing column preordering used.
  - `'none'`: no column ordering; factorize `A` as-is.
  - `'colamd'`: COLAMD ordering (default for LU)

  - 'amd': AMD ordering (default for Cholesky and LDL)

- `option.tol`: This parameter determines the tolerance used if one of the threshold pivoting schemes is chosen. The default value is 1 and this parameter can take any value in the range $(0, 1]$. This is only valid for LU factorization.

- `option.solution`: a string determining how `x` is to be returned:

  - 'double': `x` is converted to a 64-bit floating-point approximate solution. This is the default.
  - 'vpa': `x` is returned as a `vpa` array with `option.digits` digits (default is given by the MATLAB `digits` function). The result may be inexact, if an entry in `x` cannot be represented in the specified number of digits. To convert this `x` to double, use `x=double(x)`.
  - 'char': `x` is returned as a cell array of strings, where `x {i} = 'numerator/denominator'` and both `numerator` and `denominator` are arbitrary-length strings of decimal digits. The result is always exact, although `x` cannot be directly used in MATLAB for numerical calculations. It can be inspected or analyzed using MATLAB string manipulation. To convert `x` to `vpa`, use `x=vpa(x)`. To convert `x` to double, use `x=double(vpa(x))`.

- `option.digits`: the number of decimal digits to use for `x`, if `option.solution` is 'vpa'. Must be in range 2 to 2^{29}.

- `option.print`: display the inputs and outputs (0: nothing (default), 1: just errors, 2: terse, 3: all).

8.2 SPEX m files for use

8.2.1 spex_lu_backslash.m

The `spex_lu_backslash.m` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^{n\times m}$ and $\mathbf{b} \in \mathbb{R}^{n\times m}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision.

This function expects as input a matlab matrix A and dense set of right hand side vectors $\mathbf{b}$. A does not necessarily have to be sparse even though SPEX will internally treat it as so. Optionally, `option` struct can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x = spex_lu_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution vectors are more accurate than the solution obtained via `x = A \ b`. The solution `x` is returned as a MATLAB double matrix.

- `x = spex_lu_backslash(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` struct.

If the result `x` is held as a MATLAB double matrix, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a MATLAB `vpa` array, or as a cell array of strings; See Section 8.1 for details.

8.2.2 spex_cholesky_backslash.m

The `spex_cholesky_backslash.m` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathtt{R}^{n\times n}$, $\mathbf{x} \in \mathtt{R}^{n\times m}$ and $\mathbf{b} \in \mathtt{R}^{n\times m}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision. Note that A must be SPD otherwise this function returns an error.

This function expects as input a matrix A and dense set of right hand side vectors b. A does not necessarily have to be sparse even though SPEX will internally treat it as so. Optionally, the `option` struct can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x = spex_cholesky_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution vectors are more accurate than the solution obtained via `x = A \ b`. The solution `x` is returned as a MATLAB double matrix.

- `x = spex_cholesky_backslash(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` struct.

If the result `x` is held as a MATLAB double matrix, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a MATLAB `vpa` array, or as a cell array of strings; See Section 8.1 for details.

8.2.3 spex_ldl_backslash.m

The `spex_ldl_backslash.m` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathtt{R}^{n\times n}$, $\mathbf{x} \in \mathtt{R}^{n\times m}$ and $\mathbf{b} \in \mathtt{R}^{n\times m}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision. Note that all leading principal minors of A must be nonzero. Otherwise, this function returns an error.

This function expects as input a matrix A and dense set of right hand side vectors b. A does not necessarily have to be sparse even though SPEX will internally treat it as so. Optionally, the `option` struct can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x = spex_ldl_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution vectors are more accurate than the solution obtained via `x = A \ b`. The solution `x` is returned as a MATLAB double matrix.

- `x = spex_ldl_backslash(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` struct.

If the result `x` is held as a MATLAB double matrix, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a MATLAB `vpa` array, or as a cell array of strings; See Section 8.1 for details.

8.2.4 spex_backslash.m

The `spex_backslash.m` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathtt{R}^{n\times n}$, $\mathbf{x} \in \mathtt{R}^{n\times m}$ and $\mathbf{b} \in \mathtt{R}^{n\times m}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision.

This function expects as input a matrix A and dense set of right hand side vectors b. A does not necessarily have to be sparse even though SPEX will internally treat it as so. Optionally, the `option` struct can be passed in. If A is numerically symmetric, it attempts an LDL factorization. If the LDL fails or if the matrix is not numerically symmetric it performs an LU factorization. Currently, there are 2 ways to use this function outlined below:

- `x = spex_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution vectors are more accurate than the solution obtained via `x = A \ b`. The solution `x` is returned as a MATLAB double matrix.

- `x = spex_backslash(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` struct.

If the result `x` is held as a MATLAB double matrix, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a MATLAB `vpa` array, or as a cell array of strings; See Section 8.1 for details.

8.2.5 spex_mex_demo.m

This function provides a demo of the SPEX library. It shows the usage for an exact solution as well as error checking and tuning the parameters. The typical output of this function may be seen in the provided `MATLAB/html` folder.

CHAPTER 9

USING SPEX IN PYTHON

The Python interface of SPEX can be installed by navigating to the Python folder and typing make. Doing so allows the use of the Python SPEX library. First, this section describes the `Option` object in Section 9.1. The use of SPEX to solve $A\mathbf{x} = \mathbf{b}$ is discussed in Section 9.2.

9.1 Optional parameter settings

The SPEX Python interface includes an object as an optional input parameter that modifies behaviour. If this is not provided, default parameter settings are used.

- output: This parameter is a string that determines how the solution is to be returned
  - `'double'`: `x` is converted to a 64-bit floating-point approximate solution. This is the default.
  - `'string'`: `x` is returned as an array of strings.
- ordering: This parameter is a string that controls the fill-reducing column preordering used. By default it is initialized as None, if this option is chosen, the solve functions use the appropriate default ordering (AMD for Cholesky and COLAMD for Left LU).
  - `'none'`: no column ordering; factorize A as-is.
  - `'colamd'`: COLAMD ordering
  - `'amd'`: AMD ordering

9.2 Functions in Python SPEX

9.2.1 `lu_backslash`

The `lu_backslash` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^{n \times 1}$ and $\mathbf{b} \in \mathbb{R}^{n \times 1}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision.

The LU function expects as input a `scipy` sparse matrix A and a right hand side vector $\mathbf{b}$. Optionally, `option` object can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x=SPEX.lu_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution `x` is returned as a `numpy` double array.
- `x=SPEX.lu_backslash(A,b,options)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` object.

If the result `x` is held as a `numpu` double array, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a list of strings; See Section 9.1 for details.

9.2.2 cholesky_backslash

The `cholesky_backslash` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathtt{R}^{n\times n}$, $\mathbf{x} \in \mathtt{R}^{n\times 1}$ and $\mathbf{b} \in \mathtt{R}^{n\times 1}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision. Note that A must be symmetric positive definite.

The Cholesky function expects as input a `scipy` sparse matrix A and a right hand side vector $\mathbf{b}$. Optionally, `option` object can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x=SPEX.cholesky_backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution `x` is returned as a `numpy` double array.
- `x=SPEX.cholesky_backslash(A,b,options)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` object.

If the result `x` is held as a `numpu` double array, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a list of strings; See Section 9.1 for details.

9.2.3 backslash

The `backslash` function solves the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathtt{R}^{n\times n}$, $\mathbf{x} \in \mathtt{R}^{n\times 1}$ and $\mathbf{b} \in \mathtt{R}^{n\times 1}$. The final solution vector(s) obtained via this function are exact prior to their conversion to double precision. Note that A must be symmetric positive definite.

The Backslash function expects as input a `scipy` sparse matrix A and a right hand side vector $\mathbf{b}$. Optionally, `option` object can be passed in. If A is numerically symmetric, it attempts a Cholesky factorization. If the Cholesky fails or if the matrix is not numerically symmetric it performs an LU factorization. Currently, there are 2 ways to use this function outlined below:

- `x=SPEX.backslash(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution `x` is returned as a `numpy` double array.
- `x=SPEX.backslash(A,b,options)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using non-default settings from the `option` object.

If the result `x` is held as a `numpu` double array, in conventional floating-point representation (`double`), it is guaranteed to be exact only if the exact solution can be held in `double` without modification.

The solution `x` may also be returned as a list of strings; See Section 9.1 for details.

9.3 Demo

There is a file that provides a demo of the SPEX library in Python `demo.py`. It shows the usage for an exact solution as well as tuning the parameters.

BIBLIOGRAPHY

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] ——, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 381–388.

[3] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, *Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 377–380.

[4] ——, *A column approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 353–376.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, *MPFR: a multiple-precision binary floating-point library with correct rounding*, ACM Transactions on Mathematical Software (TOMS), 33 (2007), p. 13.

[6] T. Granlund et al., *GNU MP 6.0 Multiple Precision Arithmetic Library*, Samurai Media Limited, 2015.

[7] C. Lourenco, A. R. Escobedo, E. Moreno-Centeno, and T. A. Davis, *Exact solution of sparse linear systems via left-looking roundoff-error-free LU factorization in time proportional to arithmetic work*, SIAM Journal on Matrix Analysis and Applications, 40 (2019), pp. 609–638.

[8] C. J. Lourenco and E. Moreno-Centeno, *Exactly solving sparse rational linear systems via roundoff-error-free cholesky factorizations*, SIAM Journal on Matrix Analysis and Applications, 43 (2022), pp. 439–463.