User Guide for the SPEX Software Package

VERSION 2.0.3, Jan 9, 2023

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

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

Contents

Chapter 1

SPEX Overview

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

SPEX Utilities Utility and auxiliary functions for all SPEX routines: interface to the GMP/MPFR library, memory management functions, the `SPEX_matrix` data structure, and various functions that are auxiliary to the factorization and solve functions. Please refer to Chapter 2 for further details.

SPEX Left 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; this is an asymptotically efficient complexity bound. Please refer to Chapter 3 for further details.

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

Required Packages: SPEX depends on the following packages:

- AMD [1, 2], available under a BSD 3-clause license and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`
- COLAMD [5, 4], available under a BSD 3-clause license and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`
- `SuiteSparse_config` [3], no license restrictions and distributed along with SPEX. May be independently obtained at `www.suitesparse.com`
- GNU GMP [7] and MPFR [6] libraries. Distributed under the LGPL3 and GPL2 and can be acquired and installed from `https://gmplib.org/` and `http://www.mpfr.org/`, respectively.

Within a Debian/Ubuntu based Linux system, a compatible version of GMP and MPFR can be installed with the following terminal commands:

```
sudo apt-get install libgmp-dev
sudo apt-get install libmpfr-dev libmpfr-doc
```

SPEX requires GMP 6.1.2 or later, and MPFR 4.0.2 or later. Be aware that the Debian package versions differ. The Debian `libgmp.so.10.3.2` corresponds to GMP 6.1.2, and `libmpfr.so.6.0.2` corresponds to MPFR 4.0.2. The same is true for the spack package manager. Do not rely on the suffix `X.Y.Z` in the `lib*.so.X.Y.Z` to determine the version. The CMake script will display the correct version numbers of GMP and MPFR, and report an error if they are not recent enough.

Chapter 2

SPEX Utilities

2.1 Overview

SPEX Util is a software package containing utility and auxiliary functions for the SPEX factorizations. Additionally, SPEX Util provides a wrapper class for the GNU Multiple Precision Arithmetic (GMP) [7] and GNU Multiple Precision Floating Point Reliable (MPFR) [6] libraries that prevent memory leaks and improve the overall stability of these external libraries. SPEX Util is written in ANSI C.

SPEX operates on matrices stored in any of the following 15 combinations of matrix formats and entry data-types: {Compressed Sparse Column (CSC), triplet, dense} × { `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t`, or `double`}. 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 15 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 2.7.

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 $a_{23} = 1$.

2.2 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 lourenco@usna.edu, 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.

2.3 Installation

Installation of SPEX requires cmake. An optional top-level `Makefile` is provided to simplify its use (just do `make ; make install`).

2.4 Managing the SPEX environment

2.4.1 SPEX_VERSION: the software package version

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

Macro	purpose
SPEX_VERSION	current version of the code (as a string)
SPEX_VERSION_MAJOR	major version of the code
SPEX_VERSION_MINOR	minor version of the code
SPEX_VERSION_SUB	sub version of the code

2.4.2 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_INCORRECT	The solution is incorrect.
-5	SPEX_UNSYMMETRIC	The input matrix is unsymmetric (for Cholesky)
-5	SPEX_PANIC	SPEX environment error

Either SPEX_initialize or SPEX_initialize_expert (but not both) must be called prior to using any other SPEX function. SPEX_finalize must be called as the last SPEX function.

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.

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

2.4.4 `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.

2.4.5 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.

2.5 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.

2.5.1 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 SPEX has not been initialized.

2.5.2 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 SPEX has not been initialized.

2.5.3 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. Returns ok as false if SPEX has not been initialized.

If the reallocation fails, p 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), unless SPEX has not been initialized.

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) ;
```

2.5.4 `SPEX_free`: free allocated memory

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

`SPEX_free` deallocates the memory previously allocated by a call to `SPEX_calloc`, `SPEX_malloc`, or `SPEX_realloc`. 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 ;                        \
}
```

No action is taken if SPEX has not been initialized.

2.6 The `SPEX_options` structure: parameter settings for SPEX

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. Modifying this struct provides control of column orderings, pivoting schemes, and other components of the factorization.

2.6.1 `SPEX_pivot`: enum for pivoting schemes

There are six available pivoting schemes provided in SPEX that can be selected with the `SPEX_options` structure. 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.
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.

2.6.2 `SPEX_col_order`: enum for column ordering schemes

The SPEX library provides three column ordering schemes: no pre-ordering, COLAMD, and AMD, selected via the `order` component in the `SPEX_options` structure described in Section 2.6.3.

0	`SPEX_NO_ORDERING`	No pre-ordering is performed on the matrix A, that is $Q = I$.
1	`SPEX_COLAMD`	The columns of A are permuted prior to factorization using the COLAMD [4] ordering. This is the default ordering.
2	`SPEX_AMD`	The nonzero pattern of $A + A^T$ is analyzed and the columns of A are permuted prior to factorization based on the AMD [2] ordering of $A + A^T$. This works well if A has a mostly symmetric pattern, but tends to be worse than COLAMD on matrices with unsymmetric pattern. [5].

2.6.3 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 (discussed in Section 2.6.1) which controls the type of pivoting used. Default value: SPEX_TOL_SMALLEST (3).

- option->order: An enum SPEX_col_order type (discussed in Section 2.6.2) which controls what column ordering is used. Default value: SPEX_COLAMD (1).

- 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.

- 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->check`: A `bool` which indicates whether the solution to the system should be checked. Intended for debugging only; the SPEX library is guaranteed to return the exact solution. Default value: `false`.

All SPEX routines except basic memory management routines in Sections 2.4.5-2.5.1 and `SPEX_options` allocation routine in 2.6.4 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	2.6.4
`SPEX_FREE`	destroy `SPEX_options` structure	2.5.4

2.6.4 `SPEX_create_default_options`: create default `SPEX_options` structure

```
SPEX_options* SPEX_create_default_options
(
    void
) ;
```

`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 2.6.3. 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. For example, if a `NULL` pointer is passed to the symbolic analysis routines, COLAMD is used. As a result, defaults are desired, the `SPEX_options` struct need not be allocated. Returns `NULL` if SPEX has not been initialized.

2.7 The SPEX_matrix structure

All matrices for SPEX are stored as a SPEX_matrix structure (a pointer to a struct). The matrix can be held in three formats: CSC, triplet or dense matrix (as discussed in Section 2.7.1) with entries stored as 5 different types: mpz_t, mpq_t, mpfr_t, int64_t and double (as discussed in Section 2.7.2). This gives a total of 15 different combinations of matrix format and entry type. Note that not all functions accept all 15 matrix types. Indeed, most functions expect the input matrix A to be a CSC mpz_t matrix while vectors (such as x and b) are in dense format.

2.7.1 SPEX_kind: enum for matrix formats

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

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.

2.7.2 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.

2.7.3 SPEX_matrix structure

A matrix SPEX_matrix *A has the following components:

- 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->nz`: The number of nonzeros in the matrix A, if A is a triplet matrix (ignored for matrices in CSC or dense formats). Data Type: `int64_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->kind`: Indicating the kind of matrix A: CSC, triplet or dense. 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->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->scale`: A scaling parameter for matrix of `mpz_t` type. For all matrices whose entries are stored in data type other than `mpz_t`, `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`.

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`*`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` × `A->n`.

`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.

To simplify the access the entries in `A`, SPEX package provides the following macros (Note that the `TYPE` parameter in the macros is one of: `mpz`, `mpq`, `mpfr`, `int64` or `fp64`):

- `SPEX_1D(A,k,TYPE)`: used to access the kth entry in `SPEX_matrix *A` using 1D linear addressing for any matrix kind (CSC, triplet or dense), in any type with `TYPE` specified corresponding

- `SPEX_2D(A,i,j,TYPE)`: used to access the (i, j)th entry in a dense `SPEX_matrix *A`.

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

function name	description	section
`SPEX_matrix_allocate`	allocate a m-by-n `SPEX_matrix`	2.7.4
`SPEX_matrix_free`	destroy a `SPEX_matrix` and free its allocated memory	2.7.5
`SPEX_matrix_copy`	make a copy of a matrix, into another kind and/or type	2.7.6
`SPEX_matrix_nnz`	get the number of entries in a matrix	2.7.7

2.7.4 `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, or dense
    SPEX_type type,         // mpz, mpq, mpfr, int64, or double (fp64)
    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.
    bool init,              // If true, and the data types are mpz, mpq, or
                            // mpfr, the entries are initialized (using the
                            // appropriate SPEX_mp*_init function). If
                            // false, the mpz, mpq, and mpfr arrays are
                            // allocated but not initialized.
    const SPEX_options *option
) ;
```

`SPEX_matrix_allocate` allocates memory space for a m-by-n `SPEX_matrix` whose kind (CSC, triplet or dense) and data type (`mpz`, `mpq`, `mpfr`, `int64` or `fp64`) is specified. 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 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. Returns SPEX_PANIC if SPEX has not been initialized.

2.7.5 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 free. If default settings are desired, option can be input as NULL.

2.7.6 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,        // matrix to create (never shallow)
    // inputs, not modified:
    SPEX_kind kind,         // CSC, triplet, or dense
    SPEX_type type,         // mpz_t, mpq_t, mpfr_t, int64_t, or fp64
    SPEX_matrix *A,         // matrix to make a copy of (may be shallow)
    const SPEX_options *option
) ;
```

SPEX_matrix_copy creates a SPEX_matrix *C which is a modified copy of a SPEX_matrix *A. This function can convert between any pair of the 15 kinds of matrices, so the new matrix C can be of any type or kind different than A. On input

C must be non-NULL, and the value of *C is ignored; it is overwritten, output with the matrix C, which is a copy of A of kind kind and type type.

The input matrix is assumed to be valid. It can be checked first with SPEX_matrix_check (Section 2.7.8), if desired. Results are undefined for an invalid input matrix A. Returns SPEX_PANIC if SPEX has not been initialized.

2.7.7 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 2.7. For any matrix with invalid dimension(s), nnz is returned as -1. If default settings are desired, option can be input as NULL. Returns SPEX_PANIC if the SPEX working environment has not been initialized (e.g. via SPEX_initialize).

2.7.8 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 15 different matrix types (CSC, triplet, dense) $\times$ (mpz, mpq, mpfr, int64, fp64). The print level can be changed via option->print_level (refer to Section 2.6 for more details). If default settings are desired, option can be input as NULL. Returns SPEX_PANIC if SPEX has not been initialized.

2.7.9 SPEX_matrix_div: Divide dense matrix by scalar

```
SPEX_info SPEX_matrix_div // divides the x matrix by a scalar
(
    SPEX_matrix **x2_handle,    // x2 = x/scalar
    SPEX_matrix* x,             // input vector x
    const mpz_t scalar,         // the scalar
    const SPEX_options *option
);
```

This function divides the matrix x by the scalar scalar. On input, x2_handle is NULL and x must be SPEX_DENSE of SPEX_MPZ type. On success, x2_handle is returned as SPEX_DENSE with SPEX_MPQ entries each of which is equal to x / scalar.

2.7.10 SPEX_matrix_mul: Multiply dense matrix by scalar

```
SPEX_matrix_mul   // multiplies x by a scalar
(
    SPEX_matrix *x,         // matrix to be multiplied
    const mpz_t scalar      // scalar to multiply by
);
```

This function divides the matrix x by the scalar scalar. On input, x must be SPEX_DENSE of SPEX_MPZ type. On success, x is overwritten by x * scalar.

2.8 SPEX_LU_analysis structure

The SPEX_LU_analysis data structure is used for storing the column permutation for LU factorization and the estimate of the number of nonzeros that may appear in L and U. This need not be modified or accessed in the user application; it simply needs to be passed in directly to the other functions that take it as in input parameter. A SPEX_LU_analysis structure has the following components:

- S->q: The column permutation stored as a dense int64_t vector of size $n+1$, where n is the number of columns of the analyzed matrix. Currently this vector is obtained via COLAMD, AMD, or is set to no ordering (i.e., $[0, 1, \ldots, n-1]$).

- S->lnz: An int64_t which is an estimate of the number of nonzeros in L. S->lnz must be in the range of $[n, n^2]$. If S->lnz is too small, the program may waste time performing extra memory reallocations. This is set during the symbolic analysis.

- `S->unz`: An `int64_t` which is an estimate of the number of nonzeros in U. `S->unz` must be in the range of $[n, n^2]$. If `S->unz` is too small, the program may waste time performing extra memory reallocations. This is set during the symbolic analysis.

The SPEX package provides the following functions to create and destroy a `SPEX_LU_analysis` structure:

function/macro name	description	section
`SPEX_LU_analyze`	create `SPEX_LU_analysis` structure	2.8.1
`SPEX_LU_analysis_free`	destroy `SPEX_LU_analysis` structure	2.8.2

2.8.1 `SPEX_LU_analyze`: perform symbolic analysis

```
SPEX_info SPEX_LU_analyze
(
    SPEX_LU_analysis **S, // symbolic analysis (column permutation
                          // and nnz L,U)
    const SPEX_matrix *A, // Input matrix
    const SPEX_options *option  // Control parameters
) ;
```

`SPEX_LU_analyze` performs the symbolic ordering for any LU factorzation in SPEX package. Currently, there are three options: no ordering, COLAMD, or AMD, which are passed in by `SPEX_options *option`. For more details, refer to Section 2.6.

The `SPEX_LU_analysis *S` is created by calling `SPEX_LU_analyze(&S, A, option)` with `SPEX_matrix *A` properly initialized as CSC matrix and `option` be `NULL` if default ordering (COLAMD) is desired. The value of `S` is ignored on input. On output, `S` is a pointer to the newly created symbolic analysis structure and `SPEX_OK` is returned upon successful completion, or `S = NULL` with error status returned if a failure occurred. Returns `SPEX_PANIC` if SPEX has not been initialized.

The analysis `S` is freed by `SPEX_LU_analysis_free`.

2.8.2 SPEX_LU_analysis_free: free SPEX_LU_analysis structure

```
SPEX_info SPEX_LU_analysis_free
(
    SPEX_LU_analysis **S, // Structure to be deleted
    const SPEX_options *option
) ;
```

SPEX_LU_analysis_free frees a SPEX_LU_analysis structure. Note that the input of the function is the pointer to the pointer of a SPEX_LU_analysis structure. This is because this function internally sets the pointer of a SPEX_LU_analysis to be NULL to prevent potential segmentation fault that could be caused by double free. If default settings are desired, option can be input as NULL. Returns SPEX_PANIC if SPEX has not been initialized.

2.9 SPEX wrapper functions for GMP and 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. For a detailed description of each function, refer to SPEX/SPEX/SPEX_Util/Source/SPEX_gmp.c.

If additional GMP and MPFR functions are needed in the end-user application, this wrapper mechanism can be extended to those functions. Below are instructions on how to do this.

Given 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), and they need not to be the same. 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, then use
    //SPEX_GMP_WRAPPER_START;
    // If this function is modifying mpz_t type (say TYPEa = mpz_t), then use
    //SPEX_GMPZ_WRAPPER_START(a) ;
    // If this function is modifying mpq_t type (say TYPEa = mpq_t), then use
    //SPEX_GMPQ_WRAPPER_START(a) ;
    // If this function is modifying mpfr_t type (say TYPEa = mpfr_t), then 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;
}
```

Note that, other than `SPEX_mpfr_fprintf`, `SPEX_gmp_fprintf`, `SPEX_gmp_printf` and `SPEX_gmp_fscanf`, 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. For example, for `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 Wrappter
    //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;
}
```

MPFR Function	`SPEX_MPFR` **Function**	**Description**
`n = mpfr_asprintf(&buff, fmt, ...)`	`n = SPEX_mpfr_asprintf(&buff, fmt, ...)`	Print format to allocated string
`mpfr_free_str(buff)`	`SPEX_mpfr_free_str(buff)`	Free string allocated by MPFR
`mpfr_init2(x, size)`	`SPEX_mpfr_init2(x, size)`	Initialize x with size bits
`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_q(x, y, rnd)`	`SPEX_mpfr_set_q(x, y, rnd)`	$x = y$ (`mpq_t`)
`mpfr_set_z(x, y, rnd)`	`SPEX_mpfr_set_z(x, y, rnd)`	$x = y$ (`mpz_t`)
`mpfr_get_z(x, y, rnd)`	`SPEX_mpfr_get_z(x, y, rnd)`	(`mpz_t`) $x = y$
`x = mpfr_get_d(y, rnd)`	`SPEX_mpfr_get_d(x, y, rnd)`	(double) $x = y$
`mpfr_mul(x, y, z, rnd)`	`SPEX_mpfr_mul(x, y, z, rnd)`	$x = y * z$
`mpfr_mul_d(x, y, z, rnd)`	`SPEX_mpfr_mul_d(x, y, z, rnd)`	$x = y * z$
`mpfr_div_d(x, y, z, rnd)`	`SPEX_mpfr_div_d(x, y, z, rnd)`	$x = y/z$
`mpfr_ui_pow_ui(x, y, z, rnd)`	`SPEX_mpfr_ui_pow_ui(x, y, z, rnd)`	$x = y^z$
`mpfr_log2(x, y, rnd)`	`SPEX_mpfr_log2(x, y, rnd )`	$x = \log_2(y)$
`mpfr_free_cache()`	`SPEX_mpfr_free_cache()`	Free cache after log2

GMP Function	`SPEX_GMP` **Function**	**Description**
`n = gmp_fscanf(fp, fmt, ...)`	`n = SPEX_gmp_fscanf(fp, fmt, ...)`	Read from file fp
`mpz_init(x)`	`SPEX_mpz_init(x)`	Initialize x
`mpz_init2(x, size)`	`SPEX_mpz_init2(x, size)`	Initialize x to size bits
`mpz_set(x, y)`	`SPEX_mpz_set(x, y)`	$x = y$ (`mpz_t`)
`mpz_set_ui(x, y)`	`SPEX_mpz_set_ui(x, y)`	$x = y$ (signed int)
`mpz_set_si(x, y)`	`SPEX_mpz_set_si(x, y)`	$x = y$ (unsigned int)
`mpz_set_d(x, y)`	`SPEX_mpz_set_d(x, y)`	$x = y$ (double)
`x = mpz_get_d(y)`	`SPEX_mpz_get_d(x, y)`	$x = y$ (double out)
`mpz_set_q(x, y)`	`SPEX_mpz_set_q(x, y)`	$x = y$ (`mpz_t`)
`mpz_mul(x, y, z)`	`SPEX_mpz_mul(x, y, z)`	$x = y * z$
`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_submul(x, y, z)`	`SPEX_mpz_submul(x, y, z)`	$x = x - y * z$
`mpz_divexact(x, y, z)`	`SPEX_mpz_divexact(x, y, z)`	$x = y/z$
`gcd = mpz_gcd(x, y)`	`SPEX_mpz_gcd(gcd, x, y)`	$gcd = gcd(x,y)$
`lcm = mpz_lcm(x, y)`	`SPEX_mpz_lcm(lcm, x, y)`	$lcm = lcm(x,y)$
`mpz_abs(x, y)`	`SPEX_mpz_abs(x, y)`	$x = \|y\|$
`r = mpz_cmp(x, y)`	`SPEX_mpz_cmp(r, x, y)`	$r = 0$ if $x = y$, $r \neq 0$ if $x \neq y$
`r = mpz_cmpabs(x, y)`	`SPEX_mpz_cmpabs(r, x, y)`	$r = 0$ if $\|x\| = \|y\|$, $r \neq 0$ if $\|x\| \neq \|y\|$
`r = mpz_cmp_ui(x, y)`	`SPEX_mpz_cmp_ui(r, x, y)`	$r = 0$ if $x = y$, $r \neq 0$ if $x \neq y$
`sgn = mpz_sgn(x)`	`SPEX_mpz_sgn(sgn, x)`	$sgn = 0$ if $x = 0$
`size = mpz_sizeinbase(x, base)`	`SPEX_mpz_sizeinbase(size, x, base)`	size of x in base
`mpq_init(x)`	`SPEX_mpq_init(x)`	Initialize x
`mpq_set(x, y)`	`SPEX_mpq_set(x, y)`	$x = y$
`mpq_set_z(x, y)`	`SPEX_mpq_set_z(x, y)`	$x = y$ (`mpz`)
`mpq_set_d(x, y)`	`SPEX_mpq_set_d(x, y)`	$x = y$ (double)
`mpq_set_ui(x, y, z)`	`SPEX_mpq_set_ui(x, y, z)`	$x = y/z$ (unsigned int)
`mpq_set_num(x, y)`	`SPEX_mpq_set_num(x, y)`	$num(x) = y$
`mpq_set_den(x, y)`	`SPEX_mpq_set_den(x, y)`	$den(x) = y$
`mpq_get_den(x, y)`	`SPEX_mpq_get_den(x, y)`	$x = den(y)$
`x = mpq_get_d(y)`	`SPEX_mpq_get_d(x, y)`	(double) $x = y$
`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_mul(x, y, z)`	`SPEX_mpq_mul(x, y, z)`	$x = y * z$
`mpq_div(x, y, z)`	`SPEX_mpq_div(x, y, z)`	$x = y/z$
`r = mpq_cmp(x, y)`	`SPEX_mpq_cmp(r, x, y)`	$r = 0$ if $x = y$, $r \neq 0$ if $x \neq y$
`r = mpq_cmp_ui(x, n, d)`	`SPEX_mpq_cmp_ui(r, x, n, d)`	$r = 0$ if $x = n/d$, $r \neq 0$ if $x \neq n/d$
`r = mpq_equal(x, y)`	`SPEX_mpq_equal(r, x, y)`	$r = 0$ if $x = y$, $r \neq 0$ if $x \neq y$

2.10 Additional Useful SPEX Utility Functions

2.10.1 SPEX_cumsum: Cumulative sum of a vector

```
SPEX_info SPEX_cumsum
(
    int64_t *p,          // vector to store the sum of c
    int64_t *c,          // vector which is summed
    int64_t n            // size of c
);
```

SPEX_cumsum computes the cumulative sum of the array c and stores it in the array p. Specifically, $p_i = \sum_{j=1}^{i} c_j$. This is mainly used for matrix copy and some factorizations. On input, p and c must not be NULL and n must be at least 0. On completion, the contents of p are overwritten with the cumulative sum of c and SPEX_OK is returned.

2.10.2 SPEX_check_solution: Validate solution vector (for debugging)

```
SPEX_info SPEX_check_solution
(
    const SPEX_matrix *A,         // Input matrix
    const SPEX_matrix *x,         // Solution vectors
    const SPEX_matrix *b,         // Right hand side vectors
    const SPEX_options* option    // Command options
);
```

Given a solution vector x, check the solution of the linear system Ax = b. This is done by computing a rational-arthmetic A*x == b. Since all routines in SPEX are gauranteed to be exact, this function is for debugging purposes. On input, A is SPEX_CSC of SPEX_MPZ type, x is SPEX_DENSE of SPEX_MPQ type and b is SPEX_DENSE of SPEX_MPZ type. On success, SPEX_OK is returned.

Chapter 3

SPEX Left LU

3.1 Overview

SPEX Left LU is a software package designed to exactly solve unsymmetric sparse linear systems, $Ax = b$, 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 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 [8]. 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 Left LU is written in ANSI C and is accompanied by a MATLAB interface.

For all primary computational routines in Section 3.4.1, the input argument A must be stored in a compressed sparse column (CSC) matrix with entries in `mpz_t` type (referred to as CSC `mpz_t` matrix henceforth), while b must be stored as a dense `mpz_t` matrix (i.e., a dense matrix with entries in `mpz_t` type). However, the original data type of entries in the input matrix A and right hand side (RHS) vectors b can be any one of: `double`, `int64_t`, `mpq_t`, `mpz_t`, or `mpfr_t`, and their format(s) are allowed to be CSC, sparse triplet, or dense. Section 3.5.1 discusses the `SPEX_matrix` structure, the allowed matrix formats and types, and how to perform conversions across different formats/types.

The matrices L and U are computed using integer-preserving routines with the arbitrary-sized integer (`mpz_t`) data type from the GMP Library [7]. The matrices

L and U are computed in a left-looking fashion: one column at a time via the sparse REF triangular solve detailed in [8]. All divisions performed in the algorithm are *guaranteed* to be exact (i.e., with zero reminder); therefore, no greatest common divisor algorithms are needed.

The permutation matrices P and Q define the pivot ordering; Q is a given fill-reducing column ordering, and P is determined dynamically during the factorization. For the matrix P, the default option is to use a partial pivoting scheme in which the (non-zero) entry with the smallest magnitude in column k is selected as pivot; where ties are broken in favor of the diagonal entry (and arbitrarily otherwise). In addition to this scheme, the code allows diagonal pivoting, partial pivoting selecting the entry with largest magnitude, or various tolerance-based diagonal pivoting schemes. For the matrix Q, the default ordering is obtained via the Column Approximate Minimum Degree (COLAMD) algorithm [4, 5]. Other approaches include using the Approximate Minimum Degree (AMD) ordering [1, 2], or no ordering ($Q = I$). Section 2.8.1 discusses how to select the desired column ordering prior to factorization.

Once the factorization $LDU = PAQ$ is computed, the solution vector x is computed via the sparse REF forward and backward substitution algorithms [8]. Forward substitution is a variant of the sparse REF triangular solve discussed above. Backward substitution is a typical column-oriented sparse backward substitution. Both of these routines require b to be given as a dense `mpz_t` matrix. At the conclusion of the forward and backward substitution routines, the final solution vector(s) x are guaranteed to be exact. The solution x is returned as a dense `mpq_t` matrix.

A key advantage of utilizing SPEX Left LU with floating-point output is that the solution is guaranteed to be exact until the final (and only) rational-to-floating point conversion. Note that this final conversion is done in higher precision. Thus, the solution x output in `double` precision are accurate to machine roundoff (approximately 10^{-16}). In addition, the returned solution can also be accurate to any user-specified precision by using MPFR output.

3.2 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.

3.3 Installation

Installation of SPEX requires cmake. An optional top-level `Makefile` is provided to simplify its use (just do `make ; make install`).

To run the statement coverage tests (Linux required), go to the `Tcov` folder and type `make`. The last line of output should read:

```
statements not yet tested: 0
```

If you want to use SPEX Left LU within MATLAB, `cd` in MATLAB to the folder `SPEX/SPEX/SPEX_Left_LU/MATLAB` then type `SPEX_Left_LU_install`. This compiles the necessary code so that you can use the `SPEX_Left_LU_backslash` function within MATLAB. Note that `SPEX_Left_LU_install` does not add the correct directory to your path; therefore, if you want to use `SPEX_Left_LU_backslash` in future sessions, type `pathtool` and save your path for future MATLAB sessions. If you cannot save your path because of file permissions, edit your `startup.m` by adding `addpath` commands (for more information on how this is done, please type `doc startup` and `doc addpath`).

3.4 Factorization and Solve Routines

3.4.1 SPEX_Left_LU_factorize: perform LU factorization

```
SPEX_info SPEX_Left_LU_factorize
(
    // output:
    SPEX_matrix **L_handle,     // lower triangular matrix
    SPEX_matrix **U_handle,     // upper triangular matrix
    SPEX_matrix **rhos_handle,  // sequence of pivots
    int64_t **pinv_handle,      // inverse row permutation
    // input:
    const SPEX_matrix *A,        // matrix to be factored
    const SPEX_LU_analysis *S,   // column permutation and estimates
                                 // of nnz in L and U
    const SPEX_options* option
) ;
```

SPEX_Left_LU_factorize performs the SPEX Left-looking LU factorization. This factorization is done via n (number of rows or columns of the square matrix A) iterations of the sparse REF triangular solve function. The overall factorization is $PAQ = LDU$. This routine allows the factorization and solve to be split into separate phases. For example codes, refer to either SPEX/SPEX/SPEX_Left_LU/Demos/spexlu_denmo.c or Section 3.5.3.

On input, L, U, rhos, and pinv are undefined and ignored. A must be a CSC mpz matrix. Default settings are used if option is input as NULL.

Upon successful completion, the function returns SPEX_OK, and L and U are lower and upper triangular matrices, respectively, which are CSC matrices of type mpz. rhos contains the sequence of pivots as an n-by-1 dense vector of type mpz.

After factorizing the matrix, the determinant of A can be obtained from rhos[n-1] and A->scale as follows:

```
mpq_t determinant ;
SPEX_mpq_init (determinant) ;
SPEX_mpq_set_z (determinant, rhos->x.mpz[rhos->n-1]) ;
SPEX_mpq_div (determinant, determinant, A->scale) ;
```

The output array pinv contains the inverse row permutation (that is, the row index in the permuted matrix PA. For the ith row in A, pinv[i] gives the row index in PA).

Returns SPEX_PANIC if SPEX has not been initialized. Otherwise, if another error occurs, L, U, rhos, and pinv are all returned as NULL, and an error code will be returned correspondingly.

3.4.2 SPEX_Left_LU_solve: solve the linear system $Ax = b$

```
SPEX_info SPEX_Left_LU_solve    // solves the linear system LD^(-1)U x = b
(
    // Output
    SPEX_matrix **X_handle,     // rational solution to the system
    // input:
    const SPEX_matrix *b,       // right hand side vector
    const SPEX_matrix *A,       // Input matrix
    const SPEX_matrix *L,       // lower triangular matrix
    const SPEX_matrix *U,       // upper triangular matrix
    const SPEX_matrix *rhos,    // sequence of pivots
    const SPEX_LU_analysis *S,  // symbolic analysis struct
    const int64_t *pinv,        // inverse row permutation
    const SPEX_options* option
) ;
```

SPEX_Left_LU_solve obtains the solution of mpq_t type to the linear system $Ax = b$ upon a successful factorization. This function may be called after a successful return from SPEX_Left_LU_factorize, which computes L, U, rhos, and pinv.

On input, SPEX_matrix *x is undefined. A, L and U must be CSC mpz_t matrices while b and rhos must be dense mpz_t matrices. All matrices must have matched dimensions: the matrices L and U must be square lower and upper triangular matrices the same size as A, and rhos must be a dense n-by-1 vector. The input matrix b must have same number of rows as A. 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 $Ax = b$. If desired, option->check can be set to true to enable a post-check of the solution of this function. However, this is intended for debugging only; the SPEX library is guaranteed to return the exact solution. Otherwise (in case of error occurred), the function returns corresponding error code.

This function is primarily for applications that require intermediate results. For additional information, refer to either SPEX/SPEX/SPEX_Left_LU/Demos/spexlu_demo.c or Section 3.5.3. Returns SPEX_PANIC if SPEX has not been initialized.

3.4.3 SPEX_Left_LU_backslash: solve $Ax = b$

```
SPEX_info SPEX_Left_LU_backslash
(
    // Output
    SPEX_matrix **X_handle,       // Final solution vector
    // Input
    SPEX_type type,               // Type of output desired:
                                  // Must be SPEX_MPQ, SPEX_MPFR,
                                  // or SPEX_FP64
    const SPEX_matrix *A,         // Input matrix
    const SPEX_matrix *b,         // Right hand side vector(s)
    const SPEX_options* option
) ;
```

SPEX_Left_LU_backslash solves the linear system $Ax = b$ and returns the solution as a dense matrix of mpq_t, mpfr_t or double numbers. This function performs symbolic analysis, factorization, and solving all in one line. It can be thought of as an exact version of MATLAB sparse backslash.

On input, SPEX_matrix *x 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.

Upon successful completion, the function returns SPEX_OK, and x contains the solution of data type specified by type to the linear system $Ax = b$. If desired, option->check can be set to true to enable solution checking process in this function. However, this is intended for debugging only; SPEX library is guaranteed to return the exact solution. Otherwise (in case of error occurred), the function returns corresponding error code.

Returns SPEX_PANIC if SPEX has not been initialized.

For a complete example, refer to SPEX/SPEX/SPEX_Left_LU/Demos/example.c, SPEX/SPEX/SPEX_Left_LU/Demos/example2.c, or Section 3.5.2.

3.5 Using SPEX Left LU in C

Using SPEX Left LU in C has three steps:

1. initialize and populate data structures,
2. perform symbolic analysis, factorize the matrix A and solve the linear system for each b vector, and

3. free all used memory and finalize.

Step 1 is discussed in Section 3.5.1. For Step 2, performing symbolic analysis and factorizing A and solving the linear $Ax = b$ can be done in one of two ways. If only the solution vector x is required, SPEX Left LU provides a simple interface for this purpose which is discussed in Section 3.5.2. Alternatively, if the L and U factors are required, refer to Section 3.5.3. Finally, step 3 is discussed in Section 3.5.4. For the remainder of this section, `n` will indicate the dimension of A (that is, $A \in \mathbb{Z}^{n\times n}$) and `numRHS` will indicate the number of right hand side vectors being solved (that is, if `numRHS`$= r$, then $b \in \mathbb{Z}^{n\times r}$).

3.5.1 SPEX Left LU initialization and population of data structures

This section discusses how to initialize and populate the global data structures required for SPEX Left LU.

Initializing the environment

SPEX is built upon the GNU Multiple Precision Arithmetic (GMP) [7] and GNU Multiple Precision Floating Point Reliable (MPFR) [6] libraries and provides wrappers to all GMP/MPFR functions it uses. This allows SPEX to properly handle memory management failures, which GMP/MPFR does not handle. To enable this mechanism, SPEX requires initialization. The following must be done before using any other SPEX function:

```
SPEX_initialize ( ) ;
// or SPEX_initialize_expert (...); if custom memory functions are desired
```

Initializing data structures

SPEX assumes three specific input options for all functions. These are:

- `SPEX_matrix* A` and `SPEX_matrix *b`: `A` contains the input coefficient matrix, while `b` contains the right hand side vector(s) of the linear system $Ax = b$.

- `SPEX_LU_analysis* S`: `S` contains the column permutation used for A as well as estimates of the number of nonzeros in L and U.

- `SPEX_options* option`: `option` contains various control options for the factorization including column ordering used, pivot selection scheme, and others. For a full list of the contents of the `SPEX_options` structure, refer to Section 2.6. If default settings are desired, `option` can be set to `NULL`.

Populating data structures

Of the three data structures discussed in Section 3.5.1, `S` is constructed during symbolic analysis (Section 2.8.1), and `option` is an optional parameter for selecting non-default parameters. Refer to Section 2.6 for the contents of `option`.

SPEX allows the input numerical data for `A` and `b` to come in one of 5 types: `int64_t`, `double`, `mpfr_t`, `mpq_t`, and `mpz_t`. Moreover, both `A` and `b` can be stored in CSC form, sparse triplet form or dense form. CSC form is discussed in Section 2.1. The triplet form stores the contents of the matrix A in three arrays `i`, `j`, and `x` where the kth nonzero entry is stored as $A(i[k], j[k]) = x[k]$. SPEX stores its dense matrices in in column-oriented format, that is, the (i, j)th entry in `A` is `A->x.TYPE[p]` with $p = i + j$*`A->m`.

If the data for matrices are in file format to be read, refer to `SPEX/SPEX/SPEX_Left_LU/Demo /example2.c` on how to read in data and construct `A` and `b`. If the data for matrices are already stored in vectors corresponding to CSC form, sparse triplet form or dense form, allocate a shallow `SPEX_matrix` and assign vectors accordingly, then use `SPEX_matrix_copy` to get a `SPEX_matrix` in the desired kind and type. For more details, refer to `SPEX/SPEX/SPEX_Left_LU/Demo/example.c`. In a case when `A` is available in format other than CSC `mpz`, and/or `b` is available in format other than dense `mpz`, the following code snippet shows how to get `A` and `b` in a required format.

```
/* Get the matrix A. Assume that A1 is stored in CSC form
   with mpfr_t entries, while b1 is stored in triplet form
   with mpq_t entries. (for A1 and b1 in any other form,
   the exact same code will work) */

SPEX_matrix *A, *b;
// A is a copy of the A1. A is a CSC matrix with mpz_t entries
SPEX_matrix_copy(&A, SPEX_CSC,   SPEX_MPZ, A1, option);
// b is a copy of the b1. b is a dense matrix with mpz_t entries.
SPEX_matrix_copy(&b, SPEX_DENSE, SPEX_MPZ, b1, option);
```

3.5.2 Simple SPEX Left LU routines for solving linear systems

After initializing the necessary data structures, SPEX obtains the solution to $Ax = b$ using the simple interface of SPEX Left LU, `SPEX_Left_LU_backslash`. The `SPEX_Left_LU_backslash` function can return `x` as `double`, `mpq_t`, or `mpfr_t` with an associated precision. See Section 3.4.3 for more details. The following code snippet shows how to get solution as a dense `mpq_t` matrix.

```
SPEX_matrix *x;
SPEX_type my_type = SPEX_MPQ; // SPEX_MPQ, SPEX_MPFR, SPEX_FP64
SPEX_Left_LU_backslash(&x, my_type, A, b, option) ;
```

On successful return, this function returns `SPEX_OK` (see Section 2.4.2).

3.5.3 Expert SPEX Left LU routines

If the L and U factors from the SPEX Left LU factorization of the matrix A are required, the steps performed by `SPEX_Left_LU_backslash` can be done with a sequence of calls to SPEX functions:

1. declare `L`, `U`, the solution matrix `x`, and others,
2. perform symbolic analysis,
3. compute the factorization $PAQ = LDU$,
4. solve the linear system $Ax = b$, and
5. convert the final solution into the final desired form.

These steps are discussed below, along with examples.

Declare workspace

Using SPEX in this form requires the intermediate variables be declared, such as `L`, `U`, etc. The following code snippet shows the detailed list.

```
// A and b are in required type and ready to use
SPEX_matrix *L = NULL;
SPEX_matrix *U = NULL;
```

```
SPEX_matrix *x = NULL;
SPEX_matrix *rhos = NULL;
int64_t* pinv = NULL;
SPEX_LU_analysis* S = NULL;

// option needs no declaration if default setting is desired
// only declare option for further modification on default setting
SPEX_options *option = SPEX_create_default_options();
```

SPEX Left LU symbolic analysis

The symbolic analysis phase of an LU factorization entails computing the symbolic column ordering and estimating the number of nonzeros in L and U. This is performed by calling the following function:

```
SPEX_LU_analyze (&S, A, option) ;
```

Computing the factorization

The matrices `L` and `U`, the pivot sequence `rhos`, and the row permutation `pinv` are computed via the `SPEX_Left_LU_factorize` function (Section 3.4.1). Upon successful completion, this function returns `SPEX_OK`.

Solving the linear system

After factorization, the next step is to solve the linear system and store the solution as a dense matrix `x` with entries of rational number `mpq_t`. This solution is done via the `SPEX_Left_LU_solve` function (Section 3.4.2). Upon successful completion, this function returns `SPEX_OK`.

In this step, `option->check` can be set to `true` to enable the solution check process as discussed in Section 3.4.2. The process can verify that the solution vector x satisfies $Ax = b$ in perfect precision intended for debugging. This step is not needed, since the solution returned is guaranteed to be exact. It appears here simply as debugging tool, and as a verification that SPEX is computing its expected result. This test can fail only if it runs out of memory, or if there is a bug in the code (in which case, please notify the authors). Also, note that this process can be quite time consuming; thus it is not recommended to be used in general.

Converting the solution vector to the final desired form

Upon completion of the above routines, the solution to the linear system is in a dense `mpq_t` matrix. SPEX allows this to be converted into any form of matrix in the set of (CSC, sparse triplet, dense) $\times$ (`mpfr_t`, `mpq_t`, `double`) using `SPEX_matrix_copy`. The following code snippet shows how to get solution as a dense `double` matrix; since this involves a floating-point representation, the solution `my_x` will no longer be exact, even though `x` is the exact solution.

```
SPEX_kind my_kind = SPEX_DENSE;  // SPEX_CSC, SPEX_TRIPLET or SPEX_DENSE
SPEX_type my_type = SPEX_FP64;   // SPEX_MPQ, SPEX_MPFR, or SPEX_FP64
SPEX_matrix* my_x = NULL;        // New output
// Create copy which is stored as my_kind and my_type:
SPEX_matrix_copy( &my_x, my_kind, my_type, x, option);
```

3.5.4 Freeing memory

As described in Section 2.5, SPEX provides a number of functions/macros to free SPEX structures:

- `SPEX_matrix*`: A `SPEX_matrix* A` data structure can be freed with a call to `SPEX_matrix_free(&A, NULL) ;`
- `SPEX_LU_analysis*`: A `SPEX_LU_analysis* S` data structure can be freed with a call to `SPEX_LU_analysis_free(&S, NULL) ;`
- All others including `SPEX_options*`: These data structures can be freed with a call to the macro `SPEX_FREE()`, for example, `SPEX_FREE(option)` for `SPEX_options* option`.

After all usage of the SPEX routines is finished, `SPEX_finalize()` must be called (Section 2.4.5) to finalize usage of the library.

3.5.5 Examples

The `SPEX/SPEX/SPEX_Left_LU/Demo` folder contains three sample C codes which utilize SPEX. These files demonstrate the usage of SPEX as follows:

- `example.c`: This example generates a random dense 50×50 matrix and a random dense 50×1 right hand side vector b and solves the linear system. In this function, the `SPEX_Left_LU_backslash` function is used; and the output is given as a double matrix.

- `example2.c`: This example reads in a matrix stored in triplet format from the `ExampleMats` folder. Additionally, it reads in a right hand side vector from this folder and solves the associated linear system via the `SPEX_Left_LU_backslash` function, and, the solution is given as a matrix of rational numbers.

- `spexlu_demo.c`: This example reads in a matrix and right hand side vector from a file and solves the linear system $Ax = b$ using the techniques discussed in Section 3.5.3. This file also allows command line arguments (discussed in `README.md`) and can be used to replicate the results from [8].

3.6 Using SPEX Left LU in MATLAB

After following the installation steps discussed in Section 3.3, using the SPEX Left LU factorization within MATLAB can be done via the `SPEX_Left_LU_backslash.m` function. First, this section describes the `option` struct in Section 3.6.1. The use of the factorization is discussed in Section 3.6.2. The `SPEX/SPEX/SPEX_Left_LU/MATLAB` folder must be in your MATLAB path.

3.6.1 Optional parameter settings

The SPEX Left LU MATLAB interface includes an `option` struct as in optional input parameter that modifies behavior. 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:
  - `'smallest'`: smallest pivot,
  - `'diagonal'`: diagonal pivot if possible, otherwise smallest pivot,
  - `'first'`: first nonzero pivot in each column,
  - `'tol smallest'`: (default) 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)
  - `'amd'`: AMD ordering

  The `'colamd'` is recommended for most cases. The `'AMD'` ordering is suitable if the nonzero pattern of `A` is mostly symmetric. In this case, `option.pivot = 'diagonal'` is a useful option.

- `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 $(0, 1]$.

- `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).

3.6.2 SPEX_Left_LU_backslash.m

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

The SPEX Left LU function expects as input a sparse matrix A and dense set of right hand side vectors b. Optionally, `option` struct can be passed in. Currently, there are 2 ways to use this function outlined below:

- `x = SPEX_Left_LU_backslash(A,b)` returns the solution to $Ax = b$ 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_Left_LU_backslash(A,b,option)` returns the solution to $Ax = b$ 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 3.6.1 for details.

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. Davis, *SuiteSparse*, 2020. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[4] 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.

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

[6] 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.

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

[8] 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.