User Guide for SLIP LU, A Sparse Left-Looking Integer Preserving LU Factorization

Version 1.0.0, March 2020

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

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

Contents

1 Summary

SLIP LU is a software package designed to exactly solve unsymmetric sparse linear systems, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{Q}^{n\times n}$, $b \in \mathbb{Q}^{n\times m}$, and $\mathbf{x} \in \mathbb{Q}^{n\times m}$. 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. It is important to note that the matrix D is never explicitly computed nor needed; thus the functional form of the factorization requires 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 $\mathbf{x}$ is guaranteed to be exact. In addition, SLIP LU provides a wrapper class for the GNU Multiple Precision Arithmetic (GMP) [7] and GNU Multiple Precision Floating Point Reliable (MPFR) [6] libraries in order to prevent memory leaks and improve the overall stability of these external libraries. SLIP LU is written in ANSI C and is accompanied by a MATLAB interface.

The user's input matrix A and right hand side (RHS) vectors $\mathbf{b}$ are read from either `double`, `int64_t`, `mpq_t`, `mpz_t`, or `mpfr_t` data types. A must be stored in either compressed sparse column form or sparse triplet form, while $\mathbf{b}$ must be stored as a dense matrix. A discussion of building each of these types of input is given in Section **??**.

The matrices L and U are computed using internal, integer-preserving routines with the big integer (`mpz_t`) data types from the GMP Library [7]. The matrices L and U are computed one column at a time, where each column is computed via the sparse REF triangular solve detailed in [8]. All divisions performed in the algorithm are guaranteed to be exact (i.e., integer); therefore, no greatest common divisor algorithms are needed to reduce the size of entries.

The permutation matrices P and Q are either user specified or determined dynamically during the factorization. For the matrix P, the default option is to use a partial pivoting scheme in which the diagonal entry in column k is selected if it is the same magnitude as the smallest entry of k-th column, otherwise the smallest entry is selected as the k-th pivot. In addition to this approach, the code allows diagonal pivoting, partial pivoting which selects the largest pivot, or various tolerance based diagonal pivoting schemes. For the matrix Q, the default ordering is the Column Approximate Minimum Degree (COLAMD) algorithm [4, 5]. Other approaches include using the Approximate Minimum Degree (AMD) ordering [1, 2], a user specified column ordering (i.e., the default column ordering applied to the input matrix). A discussion of how to select these permutations prior to factorization is given in Section 6.

Once the factorization $LDU = PAQ$ is computed, the vector $\mathbf{x}$ is computed via sparse REF forward and backward substitution. The forward substitution is a variant of the sparse REF triangular solve discussed above. The backward substitution is a typical column oriented sparse backward substitution. Both of these routines assume that the right hand side vector(s) $\mathbf{b}$ are dense. At the conclusion of the forward and backward substitution routines, the final solution vector(s) $\mathbf{x}$ are guaranteed to be exact and is stored using the GMP `mpq_t` data structure.

The final phase of SLIP LU comprises output routines. The final solution vector(s) is defaulted to be `mpq_t` data type. Alternatively, the solution vector(s) can be in `double` precision or to any user desired precision via the `mpfr_t` data type. One key advantage of utilizing SLIP LU with floating-point output is that the solution is guaranteed to be exact until this final conversion; meaning that roundoff errors are only introduced in the final conversion from rational numbers. Thus, the solution vector(s) output in `double` precision are accurate to machine roundoff (approximately 10^{-16}) and SLIP LU utilizes higher precision for the MPFR output; thus it is also accurate to user specified precision.

All left-hand side matrices (referred to as A henceforth) within this package are stored in compressed sparse column form (CSC). 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. Data type: `mpz_t`.

An example matrix A is stored as follows (notice that via C convention, the indexing is zero based).

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

```
A->p = [0, 3, 5, 8, 11]
A->i = [0, 1, 2, 2, 3, 1, 2, 3, 0,  1, 2]
A->x = [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`, with row indices 0, 1, and 2, and values $a_{03} = 1$, $a_{13} = 12$, and $a_{23} = 1$.

2 Availability

Copyright: This software is copyright by Christopher Lourenco, Jinhao Chen, Erick Moreno-Centeno, and Timothy Davis.
Contact Info: Contact Chris Lourenco, chrisjlourenco@gmail.com, or Tim Davis, timdavis@aldenmath.com, davis@tamu.edu, or DrTimothyAldenDavis@gmail.com
Licence: 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 can be seen in the directory SLIP_LU/License/license.txt. In short, SLIP LU is free to use for research purposes. For a commercial license, please contact the authors.
Location: `https://github.com/clouren/SLIP_LU` and `www.suitesparse.com`
Required Packages: SLIP LU requires the installation of AMD [1, 2], COLAMD [5, 4], `SuiteSparse_config` [3], the GNU GMP [7] and GNU MPFR [6] libraries. AMD and COLAMD are available under a BSD 3-clause license, and no license restrictions apply to `SuiteSparse_config`. Notice that AMD, COLAMD, and `SuiteSparse_config` are included in this distribution for users' convenience. The GNU GMP and GNU MPFR library can be acquired and installed from `https://gmplib.org/` and `http://www.mpfr.org/` respectively.

If a user is running Unix that is Debian/Ubuntu based, a compatible version of GMP and MPFR can be installed with the following terminal commands:

```
sudo apt-get install libgmp3-dev
sudo apt-get install libmpfr-dev libmpfr-doc libmpfr4 libmpfr4-dbg
```

3 Installation

Installation of SLIP LU requires the `make` utility in Linux/MacOS, or `Cygwin make` in Windows. With the proper compiler, typing `make` under the main directory will compile AMD, COLAMD and SLIP LU to the respective `SLIP_LU/Lib` folder. To further install the libraries onto your computer, simply type `make install`. Thereafter, to use the code inside of your program, precede your code with `#include "SLIP_LU.h"`.

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

```
statments not yet tested: 0
```

If you want to use SLIP LU within MATLAB, from your installation of MATLAB, `cd` to the folder `SLIP_LU/SLIP_LU/MATLAB` then type `SLIP_install`. This should compile the necessary code so that you can use SLIP LU within MATLAB. Note that this file does not add the correct directory to your path; therefore, if you want SLIP LU as a default function, 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 (type doc startup and doc `addpath` for more information).

4 SLIP LU Data Structures

There are three important data structures used throughout the SLIP LU package: `SLIP_options`, `SLIP_matrix`, and `SLIP_LU_analysis`. We describe them briefly below and more in detail in this section.

- `SLIP_options`: Contains numerous command parameters. Default values of these parameters are good for a general user; however, modifying this struct allows a user to control column orderings, pivoting schemes, and other components of the factorization.

- `SLIP_matrix`: A sparse matrix for SLIP LU. These matrices are stored in the CSC form with `mpz_t` entries.

- `SLIP_LU_analysis`: A symbolic analysis struct. Contains the column permutation and guesses for the number of nonzeros in L and U.

Furthermore, five enumerated types (`enum`) are defined and used: `SLIP_info`, `SLIP_pivot`, `SLIP_col_order`, `SLIP_kind`, and `SLIP_type`. Again we briefly describe them below and in more detail later in this section.

- `SLIP_info`: Status codes for SLIP LU. Most function return a status indicating success or, in the case of failure, what went wrong.

- `SLIP_pivot`: Types of pivoting scheme available for the user.

- `SLIP_col_order`: Type of column preordering available for the user.
- `SLIP_kind`: Formats of matrix available for the user.
- `SLIP_type`: Data types of entries in matrix available for the user.

Lastly, SLIP LU defines the following strings with `#define`. Refer to `SLIP_LU.h` file for details.

Macro	purpose
`SLIP_LU_VERSION`	current version of the code
`SLIP_LU_VERSION_MAJOR`	major version of the code
`SLIP_LU_VERSION_MINOR`	minor version of the code
`SLIP_LU_VERSION_SUB`	sub version of the code
`SLIP_AUTHOR`	authors of the code

The remainder of this section describes each of these data structures and enumerated types.

4.1 `SLIP_info`: status code returned by SLIP LU

Most of SLIP LU functions return its status to the caller as its return value, an enumerated type called `SLIP_info`. All possible values for `SLIP_info` are listed as follows:

0	`SLIP_OK`	The function was successfully executed.
-1	`SLIP_OUT_OF_MEMORY`	out of memory
-2	`SLIP_SINGULAR`	The input matrix A is exactly singular.
-3	`SLIP_INCORRECT_INPUT`	One or more input arguments are incorrect.
-4	`SLIP_INCORRECT`	The solution is incorrect.

4.2 `SLIP_pivot`: enum for pivoting schemes

There are six available pivoting schemes provided in SLIP LU. Users can set the pivoting method through the `SLIP_options` structure in Section 4.6. Note that the pivot is always nonzero, thus the smallest entry is the nonzero entry with the smallest magnitude. Also, the tolerance is specified by the `tol` component in `SLIP_options`. Please refer to Section 4.6 for details of this parameter. The pivoting schemes are described as follows:

0	`SLIP_SMALLEST`	The k-th pivot is selected as the smallest entry in the kth column.
1	`SLIP_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	`SLIP_FIRST_NONZERO`	The k-th pivot is selected as the first eligible nonzero in the column.
3	`SLIP_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	`SLIP_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	`SLIP_LARGEST`	The k-th pivot is selected as the largest entry in the k-th column.

4.3 `SLIP_col_order`: enum for column ordering schemes

The SLIP LU library provides three column ordering schemes: no ordering, COLAMD, and AMD. Users can set the column ordering method through `order` component in the `SLIP_option` structure described in Section 4.6. In general, it is recommended that the user selects the COLAMD ordering, however, no preordering can be preferable if the user's matrix already has a good preordering.

0	`SLIP_NO_ORDERING`	No pre-ordering is performed on the matrix A, that is $Q = I$.
1	`SLIP_COLAMD`	The columns of A are permuted prior to factorization using the COLAMD [4] ordering. This is the default ordering.
2	`SLIP_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].

4.4 `SLIP_kind`: enum for matrix formats

The SLIP LU library provides three available matrix formats: sparse CSC (compressed sparse column), sparse triplet and dense. Details for matrices in SLIP LU are discussed in Section 4.7.

0	`SLIP_CSC`	Matrix is in compressed sparse column format.
1	`SLIP_TRIPLET`	Matrix is in sparse triplet format.
2	`SLIP_DENSE`	Matrix is in dense format.

4.5 `SLIP_type`: enum for data types of matrix entry

The SLIP LU library provides five data types for matrix entries: `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t` and `double`. Details for matrices in SLIP LU are discussed in Section 4.7.

0	`SLIP_MPZ`	Matrix entries are in `mpz_t` type.
1	`SLIP_MPQ`	Matrix entries are in `mpq_t` type.
2	`SLIP_MPFR`	Matrix entries are in `mpfr_t` type.
3	`SLIP_INT64`	Matrix entries are in `int64_t` type.
4	`SLIP_FP64`	Matrix entries are in `double` type.

4.6 `SLIP_options` structure

The `SLIP_options` struct stores key command parameters for various functions used in the SLIP LU package. The `SLIP_options* option` struct contains the following components:

- `option->pivot`: An enum `SLIP_pivot` type (discussed in Section 4.2) which controls the type of pivoting used. Default value: `SLIP_TOL_SMALLEST` (3).
- `option->order`: An enum `SLIP_col_order` type (discussed in Section 4.3) which controls what column ordering is used. Default value: `SLIP_COLAMD` (1).
- `option->tol`: A `double` which tells the tolerance used if the user selects a tolerance based pivoting scheme, i.e., `SLIP_TOL_SMALLEST` or `SLIP_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 SLIP, 3: all, with matrices and results. Default value: 0.
- `option->prec`: An `uint64_t` which specifies the precision used if the user desires 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 `int` 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 SLIP LU. 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.

  By default, SLIP LU utilizes `MPFR_RNDN`. We refer the reader 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.

- `option->check`: A `bool` which inicates whether the solution to the system should be checked. Intended for debugging only; SLIP LU library is guaranteed to return the exact solution.

All SLIP LU routines described in Sections ?? and 6 require `option` as an input argument. However, users can avoid creating one by passing `NULL` if the default settings are desired. Otherwise, users can use the following function/macro to create and destroy a `SLIP_options` object.

function/macro name	description	section
`SLIP_create_default_options`	create and return `SLIP_options` pointer with default parameters upon successful allocation	5.8
`SLIP_FREE`	destroy `SLIP_options` object	5.4

4.7 The SLIP_matrix structure

All internal matrices are stored as `SLIP_matrix` structure, which can be CSC, triplet or dense matrix (as discussed in Section 4.4) with entries stored as `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t` and `double` (as discussed in Section 4.5). This gives a total of 15 different matrix types. Not all functions accept all 15 matrices types, however.

A matrix `SLIP_matrix *A` has the following components:

- `A->m`: Number of rows in the matrix. It is typically assumed that $m = n$. Data Type: `int64_t`

- `A->n`: Number of columns in the matrix. It is typically assumed that $m = n$. 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 `SLIP_matrix_nnz(A,option)`. Data Type: `int64_t`

- `A->kind`: Indicating the kind of matrix A: CSC, triplet or dense. Data Type: `SLIP_kind`

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

- `A->p`: An array of size $n+1$ 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. 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 therefore indices are in the range of $[0, n-1]$. 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 therefore indices are in the range of $[0, n-1]$. Data Type: `int64_t*`

- `A->j_shallow`: A boolean indicating whether `A->j` is shallow. Data Type: `bool`

- `A->x.TYPE`: An array of size `A->nzmax` which contains the numeric values of the matrix. `TYPE` should be mpz, mpq, mpfr, int64 or fp64 corresponding to the entry type indicated by `A->type`. Data Type: `union`

- `A->x_shallow`: A boolean indicating whether `A->x.TYPE` is shallow. Data Type: `bool`

- `A->scale`: A scaling parameter for matrix of `mpz_t` type. For all matrices whose type is not `mpz_t`, `A->scale = 1`. This is used to ensure integrality of each entry in mpz matrix if these entries are converted from non-integral 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) `SLIP_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; nz is `A->p [A->n]`.

- (1) `SLIP_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` $= nz$. The `A->p` array is NULL.

- (2) `SLIP_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; nz 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 `SLIP_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`, SLIP LU package provides the following macros (Note that the `TYPE` parameter in the macros is one of: mpz, mpq, mpfr, int64 or fp64):

- `SLIP_1D(A,k,TYPE)`: used to access the kth entry in `SLIP_matrix* A` using 1D linear addressing for any matrix kind (CSC, triplet or dense), in any type with `TYPE` specified corresponding

- `SLIP_2D(A,i,j,TYPE)`: used to access the (i, j)th entry in a dense `SLIP_matrix* A`

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

function name	description	section
`SLIP_matrix_allocate`	allocate a m-by-n `SLIP_matrix`	5.9
`SLIP_matrix_copy`	make a copy of a matrix, into another kind and/or type	**??**
`SLIP_matrix_nnz`	get the number of entries in a matrix	**??**
`SLIP_matrix_free`	destroy a `SLIP_matrix` and free its allocated memory	5.10

4.8 `SLIP_LU_analysis` structure

The `SLIP_LU_analysis` data structure is used for storing the column permutation for LU and the guess on nonzeros for L and U. Users do not need to modify this struct, just pass it into the functions. A `SLIP_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 a guess for 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 a guess for 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 SLIP LU package provides the following functions to create and destroy a `SLIP_LU_analysis` object:

function/macro name	description	section
`SLIP_LU_analyze`	create `SLIP_LU_analysis` object	6.1
`SLIP_LU_analysis_free`	destroy `SLIP_LU_analysis` object	5.11

5 Memory Management Routines

The routines in this section are used to allocate and free memory for the data structures used in SLIP LU. Note that, SLIP LU relies on the SuiteSparse memory management functions, `SuiteSparse_malloc`, `SuiteSparse_calloc`, `SuiteSparse_realloc`, and `SuiteSparse_free`.

5.1 `SLIP_calloc`: allocate initialized memory

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

`SLIP_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.

5.2 `SLIP_malloc`: allocate uninitialized memory

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

SLIP_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.

5.3 SLIP_realloc: resize allocated memory

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

SLIP_realloc is a wrapper for realloc. If p is non-NULL on input, it points to a previously allocated object of size old_size * size_of_item. The object is reallocated to be of size new_size * size_of_item. If p is NULL on input, then a new object of that size is allocated. On success, a pointer to the new object is returned. If the reallocation fails, p is not modified, and a flag is returned to indicate that the reallocation failed. If the size decreases or remains the same, then the method always succeeds (ok is returned as true).

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 SLIP_malloc succeeds but the SLIP_realloc fails, then the array remains unmodified, of size 10.

```
int *p ;
p = SLIP_malloc (10 * sizeof (int)) ;
if (p == NULL) { error here ... }
printf ("p points to an array of size 10 * sizeof (int)\n") ;
bool ok ;
p = SLIP_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") ;
SLIP_free (p) ;
```

5.4 SLIP_free: free allocated memory

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

SLIP_free deallocates the memory previously allocated by a call to SLIP_calloc, SLIP_malloc, or SLIP_realloc. Note that the default C free function can cause a segmentation fault if called multiple times on the same pointer or is called via other inappropriate behavior. To remedy this issue, this function frees the input pointer p only when it is not NULL. To further prevent the potential segmentation fault that could be caused by free, the following macro SLIP_FREE is provided, which sets the free'd pointer to NULL.

```
#define SLIP_FREE(p)                      \
{                                         \
    SLIP_free (p) ;                       \
    (p) = NULL ;                          \
}
```

5.5 SLIP_initialize: initialize the working environment

```
void SLIP_initialize
(
    void
) ;
```

SLIP_initialize initializes the working environment for SLIP LU functions. SLIP LU utilizes a specialized memory management scheme in order to prevent potential memory failures caused by GMP library. This function **must** be called prior to using the library.

5.6 SLIP_initialize_expert: initialize the working environment (expert version)

```
void SLIP_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
) ;
```

SLIP_initialize_expert is the same as SLIP_initialize. except that it allows for a redefinition of custom memory functions that are used for SLIP LU and GMP. 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) ;
```

5.7 SLIP_finalize: free the working environment

```
void SLIP_finalize
(
    void
) ;
```

SLIP_finalize finalizes the working evironment for SLIP LU library, and frees any internal workspace created by SLIP LU. It must be called as the last SLIP_* function called.

5.8 SLIP_create_default_options: create default SLIP_option object

```
SLIP_options* SLIP_create_default_options
(
    void
) ;
```

SLIP_create_default_options creates and returns a pointer to a SLIP_options struct with default parameters upon successful allocation, which are discussed in Section 4.6. To safely free the SLIP_options* option structure, simply use SLIP_FREE(option).

5.9 SLIP_matrix_allocate: allocate a m-by-n SLIP_matrix

```
SLIP_info SLIP_matrix_allocate
(
    SLIP_matrix **A_handle, // matrix to allocate
    SLIP_kind kind,         // CSC, triplet, or dense
    SLIP_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.
    bool init,              // If true, and the data types are mpz, mpq, or
                            // mpfr, the entries are initialized (using the
                            // appropriate SLIP_mp*_init function). If
                            // false, the mpz, mpq, and mpfr arrays are
                            // allocated but not initialized.
    const SLIP_options *option
) ;
```

`SLIP_matrix_allocate` allocate memory space for a m-by-n `SLIP_matrix` whose kind (CSC, triplet or dense) and data type (mpz, mpq, mpfr, int64 or double) is specified. If 'shallow' is true, all components (p,i,j,x) are NULL, and their shallow flags are all true. The user can then set A-¿p, A-¿i, A-¿j, and/or A-¿x accordingly, from their own arrays. If 'shallow' is false, components are allocated correspondingly (see Section 4.7 for more infomation). For data type as mpz, mpq or mpfr, the entries are initialized (using the appropriate `SLIP_mp*_init` function) if 'init' is true. Otherwise ('init' is false), the mpz, mpq or mpfr arrays are allocated but not initialized (which means that accessing their entry without further initialization would cause undefined behavior). The boolean 'init' is ignored for data type of double or int64.

5.10 `SLIP_matrix_free`: free a `SLIP_matrix`

```
SLIP_info SLIP_matrix_free
(
    SLIP_matrix **A_handle, // matrix to free
    const SLIP_options *option
) ;
```

`SLIP_matrix_free` frees the `SLIP_matrix *A`, which is then set to `NULL`. If default setting is desired, `option` can be input as NULL.

5.11 SLIP_LU_analysis_free: free SLIP_LU_analysis structure

```
void SLIP_LU_analysis_free
(
    SLIP_LU_analysis **S, // Structure to be deleted
    const SLIP_options *option
) ;
```

SLIP_LU_analysis_free deletes a SLIP_LU_analysis structure. Note that the input of the function is the pointer to the pointer of a SLIP_LU_analysis structure. This is because this function internally sets the pointer of a SLIP_LU_analysis to be NULL to prevent potential segmentation fault that could be caused by double free. If default setting is desired, option can be input as NULL.

6 Primary Computational Routines

These routines perform symbolic analysis prior to LU factorization, compute the LU factorization of the matrix A, and solve $Ax = b$ using the LU factorization of A.

6.1 SLIP_LU_analyze: perform symbolic analysis

```
SLIP_info SLIP_LU_analyze
(
    SLIP_LU_analysis **S, // symbolic analysis (column permutation
                          // and nnz L,U)
    const SLIP_matrix *A, // Input matrix
    const SLIP_options *option  // Control parameters
) ;
```

SLIP_LU_analyze performs the symbolic ordering for SLIP LU. Currently, there are three options: no ordering, COLAMD, or AMD, which are passed in by SLIP_option *option. For more details, users can refer to Section 4.6.

The SLIP_LU_analysis *S is created by calling SLIP_LU_analyze(&S, A, option) with SLIP_sparse *A properly initialized as CSC matrix and option could 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 object, or NULL if a failure occurred.

The analysis S is freed by SLIP_LU_analysis_free.

6.2 SLIP_LU_factorize: perform LU factorization

```
SLIP_info SLIP_LU_factorize
(
    // output:
    SLIP_matrix **L_handle,     // lower triangular matrix
    SLIP_matrix **U_handle,     // upper triangular matrix
    SLIP_matrix **rhos_handle,  // sequence of pivots
    int64_t **pinv_handle,      // inverse row permutation
    // input:
    const SLIP_matrix *A,        // matrix to be factored
    const SLIP_LU_analysis *S,   // stores guess on nnz
                                 // and column permutation
    const SLIP_options* option
) ;
```

SLIP_LU_factorize performs the SLIP LU factorization. This factorization is done via n (number of rows or columns of A) iterations of the sparse REF triangular solve function. The overall factorization is $PAQ = LDU$. This routine allows the user to separate factorization and solve. For example codes, please refer to either Demos/SLIPLU.c or Section 9.3.

On input, L, U, rhos, and pinv are undefined. A must be a CSC matrix with mpz_t type entries. Default setting will be used if option is input as NULL.

Upon successful completion, the function return SLIP_OK, and L and U are the lower and upper triangular matrices, rhos contains the sequence of pivots. The determinant of A can be obtained as rhos[n-1]. 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). Otherwise (in case of error occured), the function returns corresponding error code.

If an error occurs, L, U, rhos, and pinv are all returned as NULL, and an error code will be returned correspondingly.

6.3 SLIP_LU_solve: solve the linear system $Ax = b$

```
SLIP_info SLIP_LU_solve         // solves the linear system LD^(-1)U x = b
(
    // Output
    SLIP_matrix **X_handle,     // rational solution to the system
    // input:
    const SLIP_matrix *b,       // right hand side vector
    const SLIP_matrix *A,       // Input matrix
    const SLIP_matrix *L,       // lower triangular matrix
```

```
        const SLIP_matrix *U,       // upper triangular matrix
        const SLIP_matrix *rhos,    // sequence of pivots
        const SLIP_LU_analysis *S,  // symbolic analysis struct
        const int64_t *pinv,        // inverse row permutation
        const SLIP_options* option
    ) ;
```

SLIP_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 SLIP_LU_factorize, which computes L, U, rhos, and pinv.

On input, SLIP_matrix *x are undefined. A, L and U should be CSC mpz matrices while b and rhos should be dense mpz matrices. All matrices should have matched dimensions. (Since L, U and rhos are computed from SLIP_LU_factorize, they will have matched dimension with A. Yet, b should be guaranteed to have same number of rows as that of A.) Default setting will be used if option is input as NULL.

Upon successful completion, the function returns SLIP_OK, and x contains the solution of mpq_t 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; SLIP LU library is guaranteed to return the exact solution. Otherwise (in case of error occured), the function returns corresponding error code.

Like some of some other routines discussed in this section, this function is primarily for advanced users who might want intermediate calculation results; thus for usage information please refer to either Demos/SLIPLU.c or Section 9.3.

6.4 SLIP_backslash: solve $Ax = b$ and return x in user desired type

```
    SLIP_info SLIP_backslash
    (
        // Output
        SLIP_matrix **X_handle,       // Final solution vector
        // Input
        SLIP_type type,               // Type of output desired:
                                      // Must be SLIP_MPQ, SLIP_MPFR,
                                      // or SLIP_FP64
        const SLIP_matrix *A,         // Input matrix
        const SLIP_matrix *b,         // Right hand side vector(s)
        const SLIP_options* option
    ) ;
```

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

On input, `SLIP_matrix *x` are undefined. `type` must be one of: `SLIP_MPQ`, `SLIP_MPFR` or `SLIP_FP64` to specify the data type of the solution entries. `A` should be a square CSC mpz matrix while `b` should be a dense mpz matrix. In addition, `A->m` should be equal to `b->m`. Default setting will be used if `option` is input as NULL.

Upon successful completion, the function returns `SLIP_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; SLIP LU library is guaranteed to return the exact solution. Otherwise (in case of error occured), the function returns corresponding error code.

For a complete example, users can refer to `Demos/example2.c`. Here is an brief example of how to use this code:

```
/* Create and populate A, b, and option */
/* A has size of n-by-n, b has size of n-by-numRHS */
SLIP_matrix *x;
/* we want the solution in double format with default setting*/
SLIP_backslash(&x, SLIP_FP64, A, b, NULL) ;
```

7 Additional Routines for `SLIP_matrix` (TODO rename this)

This section contains additional routines to copy, query and check a `SLIP_matrix` structure.

7.1 `SLIP_matrix_copy`: make a copy of a `SLIP_matrix`

```
SLIP_info SLIP_matrix_copy
(
    SLIP_matrix **C,        // matrix to create (never shallow)
    // inputs, not modified:
    SLIP_kind kind,         // CSC, triplet, or dense
    SLIP_type type,         // mpz_t, mpq_t, mpfr_t, int64_t, or double
```

```
        SLIP_matrix *A,         // matrix to make a copy of (may be shallow)
        const SLIP_options *option
    ) ;
```

SLIP_matrix_copy create a SLIP_matrix *C which is a modified copy of a SLIP_matrix *A. The new matrix C can have a different kind and type than A. If default setting is desired, option can be input as NULL.

The input matrix is assumed to be valid. It can be checked first with SLIP_matrix_check, if desired. If the input matrix A is not valid, results are undefined.

7.2 SLIP_matrix_nnz: get the number of entries in a SLIP_matrix

```
    int64_t SLIP_matrix_nnz     // return # of entries in A, or -1 on error
    (
        const SLIP_matrix *A,         // matrix to query
        const SLIP_options *option
    ) ;
```

SLIP_matrix_nnz returns the number of entries in a SLIP_matrix *A. For details regarding how the number of entries is obtained for different kinds of matrices, users can refer to Section 4.7. For any invalid matrix, this funtion returns -1. If default setting is desired, option can be input as NULL.

7.3 SLIP_matrix_check: check and print a SLIP_matrix

```
    SLIP_info SLIP_matrix_check     // returns a SLIP_LU status code
    (
        const SLIP_matrix *A,       // matrix to check
        const SLIP_options* option  // defines the print level
    ) ;
```

SLIP_matrix_check check the validity of a SLIP_matrix *A in any of the 15 different matrix types (CSC, triplet, dense) $\times$ (mpz, mpq, mpfr, int64, double). Users can adjust the print level by changing option->print_level (refer to Section 4.6 for more details). If default setting is desired, option can be input as NULL.

8 SLIP LU Wrapper Functions for GMP and MPFR

SLIP LU provides a wrapper class for all GMP and MPFR functions used by SLIP LU. 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 SLIP LU 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 `SLIP_`. For example, the default GMP function `mpz_mul` is changed to `SLIP_mpz_mul`. Each SLIP GMP/MPFR function returns `SLIP_OK` if successful or the correct error code if not. The following table gives a brief list of each currently covered SLIP GMP/MPFR function. For a detailed description of each function, please refer to `SLIP_LU/Source/SLIP_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, we give 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. In order to apply our wrapper to a new function, one can create it as follows:

```
SLIP_info SLIP_gmpfunc
(
    TYPEa a,
    TYPEb b,
    ...
)
{
    // Start the GMP Wrappter
    // uncomment one of the followings that meets the needs
    // If this function is not modifying any GMP type variable, then use
    //SLIP_GMP_WRAPPER_START;
    // If this function is modifying mpz_t type (say TYPEa = mpz_t), then use
    //SLIP_GMPZ_WRAPPER_START(a) ;
    // If this function is modifying mpq_t type (say TYPEa = mpq_t), then use
    //SLIP_GMPQ_WRAPPER_START(a) ;
    // If this function is modifying mpz_t type (say TYPEa = mpz_t), then use
    //SLIP_GMPFR_WRAPPER_START(a) ;

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

    //Finish the wrapper and return ok if successful.
    SLIP_GMP_WRAPPER_FINISH;
    return SLIP_OK;
}
```

Note that, other than `SLIP_mpfr_fprintf`, `SLIP_gmp_fprintf`, `SLIP_gmp_printf` and `SLIP_gmp_fscanf`, all of the wrapped GMP/MPFR functions always return `SLIP_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, ...)`, users can create the wrapped function as follows:

```
SLIP_info SLIP_gmpfunc
(
    TYPEr *r,        // return value of the GMP/MPFR function
    TYPEa a,
    TYPEb b,
    ...
)
{
    // Start the GMP Wrappter
    // uncomment one of the followings that meets the needs
    //SLIP_GMP_WRAPPER_START;
    //SLIP_GMPZ_WRAPPER_START(a) ;
    //SLIP_GMPQ_WRAPPER_START(a) ;
    //SLIP_GMPFR_WRAPPER_START(a) ;

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

    //Finish the wrapper and return ok if successful.
    SLIP_GMP_WRAPPER_FINISH;
    return SLIP_OK;
}
```

MPFR Function	SLIP_MPFR Function	Description
n = mpfr_fprintf(fp, format, ...)	n = SLIP_mpfr_fprintf(fp, format, ...)	Print format to file fp
mpfr_init2(x, size)	SLIP_mpfr_init2(x, size)	Initialize x with size bits
mpfr_set(x, y, rnd)	SLIP_mpfr_set(x, y, rnd)	$x = y$
mpfr_set_d(x, y, rnd)	SLIP_mpfr_set_d(x, y, rnd)	$x = y$ (double)
mpfr_set_q(x, y, rnd)	SLIP_mpfr_set_q(x, y, rnd)	$x = y$ (mpq)
mpfr_set_z(x, y, rnd)	SLIP_mpfr_set_z(x, y, rnd)	$x = y$ (mpz)
mpfr_get_z(x, y, rnd)	SLIP_mpfr_get_z(x, y, rnd)	(mpz) $x = y$
x = mpfr_get_d(y, rnd)	SLIP_mpfr_get_d(x, y, rnd)	(double) $x = y$
mpfr_mul(x, y, z, rnd)	SLIP_mpfr_mul(x, y, z, rnd)	$x = y * z$
mpfr_mul_d(x, y, z, rnd)	SLIP_mpfr_mul_d(x, y, z, rnd)	$x = y * z$
mpfr_div_d(x, y, z, rnd)	SLIP_mpfr_div_d(x, y, z, rnd)	$x = y/z$
mpfr_ui_pow_ui(x, y, z, rnd)	SLIP_mpfr_ui_pow_ui(x, y, z, rnd)	$x = y^z$
mpfr_log2(x, y, rnd)	SLIP_mpfr_log2(x, y, rnd)	$x = \log_2(y)$
mpfr_free_cache()	SLIP_mpfr_free_cache()	Free cache after log2

GMP Function	SLIP_GMP Function	Description
n = gmp_fprintf(fp, format, ...)	n = SLIP_gmp_fprintf(fp, format, ...)	Print format to file fp
n = gmp_printf(format, ..)	n = SLIP_gmp_printf(format, ...)	Print to screen
n = gmp_fscanf(fp, format, ...)	n = SLIP_gmp_fscanf(fp, format, ...)	Read from file fp
mpz_init(x)	SLIP_mpz_init(x)	Initialize x
mpz_init2(x, size)	SLIP_mpz_init2(x, size)	Initialize x to size bits
mpz_set(x, y)	SLIP_mpz_set(x, y)	$x = y$ (mpz)
mpz_set_ui(x, y)	SLIP_mpz_set_ui(x, y)	$x = y$ (signed int)
mpz_set_si(x, y)	SLIP_mpz_set_si(x, y)	$x = y$ (unsigned int)
mpz_set_d(x, y)	SLIP_mpz_set_d(x, y)	$x = y$ (double)
x = mpz_get_d(y)	SLIP_mpz_get_d(x, y)	$x = y$ (double out)
mpz_set_q(x, y)	SLIP_mpz_set_q(x, y)	$x = y$ (mpq)
mpz_mul(x, y, z)	SLIP_mpz_mul(x, y, z)	$x = y * z$
mpz_add(x, y, z)	SLIP_mpz_add(x, y, z)	$x = y + z$
mpz_addmul(x, y, z)	SLIP_mpz_addmul(x, y, z)	$x = x + y * z$
mpz_submul(x, y, z)	SLIP_mpz_submul(x, y, z)	$x = x - y * z$
mpz_divexact(x, y, z)	SLIP_mpz_divexact(x, y, z)	$x = y/z$
gcd = mpz_gcd(x, y)	SLIP_mpz_gcd(gcd, x, y)	$gcd = gcd(x, y)$
lcm = mpz_lcm(x, y)	SLIP_mpz_lcm(lcm, x, y)	$lcm = lcm(x, y)$
mpz_abs(x, y)	SLIP_mpz_abs(x, y)	$x = \|y\|$
r = mpz_cmp(x, y)	SLIP_mpz_cmp(r, x, y)	$r = 0$ if $x = y$ $r \neq 0$ if $x \neq y$
r = mpz_cmpabs(x, y)	SLIP_mpz_cmpabs(r, x, y)	$r = 0$ if $\|x\| = \|y\|$ $r \neq 0$ if $\|x\| \neq \|y\|$
r = mpz_cmp_ui(x, y)	SLIP_mpz_cmp_ui(r, x, y)	$r = 0$ if $x = y$ $r \neq 0$ if $x \neq y$
sgn = mpz_sgn(x)	SLIP_mpz_sgn(sgn, x)	$sgn = 0$ if $x = 0$
size = mpz_sizeinbase(x, base)	SLIP_mpz_sizeinbase(size, x, base)	size of x in base
mpq_init(x)	SLIP_mpq_init(x)	Initialize x
mpq_set(x, y)	SLIP_mpq_set(x, y)	$x = y$
mpq_set_z(x, y)	SLIP_mpq_set_z(x, y)	$x = y$ (mpz)
mpq_set_d(x, y)	SLIP_mpq_set_d(x, y)	$x = y$ (double)
mpq_set_ui(x, y, z)	SLIP_mpq_set_ui(x, y, z)	$x = y/z$ (unsigned int)
mpq_set_num(x, y)	SLIP_mpq_set_num(x, y)	$num(x) = y$
mpq_set_den(x, y)	SLIP_mpq_set_den(x, y)	$den(x) = y$
mpq_get_den(x, y)	SLIP_mpq_get_den(x, y)	$x = den(y)$
x = mpq_get_d(y)	SLIP_mpq_get_d(x, y)	(double) $x = y$
mpq_abs(x, y)	SLIP_mpq_abs(x, y)	$x = \|y\|$
mpq_add(x, y, z)	SLIP_mpq_add(x, y, z)	$x = y + z$
mpq_mul(x, y, z)	SLIP_mpq_mul(x, y, z)	$x = y * z$
mpq_div(x, y, z)	SLIP_mpq_div(x, y, z)	$x = y/z$
r = mpq_cmp(x, y)	SLIP_mpq_cmp(r, x, y)	$r = 0$ if $x = y$ $r \neq 0$ if $x \neq y$
r = mpq_cmp_ui(x, n, d)	SLIP_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)	SLIP_mpq_equal(r, x, y)	$r = 0$ if $x = y$ $r \neq 0$ if $x \neq y$

9 Using SLIP LU in C

Using SLIP LU in C has four 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.

Steps 1 is discussed in Subsections 9.1. Perform symbolic analysis and factorizing A and solving the linear $Ax = b$ can be done in one of two ways. If the user is only interested in obtaining the solution vector x, SLIP LU provides a simple interface for this purpose which is discussed in Section 9.2. Alternatively, if the user wants the actual L and U factors, please refer to Section 9.3. Finally, step 3 is discussed in Section 9.4. 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 $\mathbf{b} \in \mathbb{Z}^{n\times r}$).

9.1 SLIP LU Initialization and Population of Data Structures

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

9.1.1 Initializing the Environment

SLIP LU 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 SLIP LU to properly handle memory management failures, which GMP/MPFR does not handle. It may also allow the user to not need any direct access to the GMP/MPFR libraries. To enable this mechanism, SLIP LU requires initialization. The following must be done before using any other SLIP LU function:

```
SLIP_initialize ( ) ;
// or SLIP_initialize_expert (...); if custom memory functions are desired
```

9.1.2 Initializing Data Structures

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

- `SLIP_matrix* A` and `SLIP_matrix *b`: `A` contains the user's input matrix, while `b` contains the user's right hand side vector(s). If the input matrix was already an integer matrix, `A` is the user's input and `A->scale`=1. Otherwise, the input matrix is not integer and `A` contains the user's scaled input matrix. `b` is handled in the same way.

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

- `SLIP_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 `SLIP_options` structure, please refer to Section 4.6. If default setting is desired, NULL can be given instead.

9.1.3 Populating Data Structures

Of the three data structures discussed in Section 9.1.2, `S` is constructed during symbolic analysis (Section **??**), `option` can be provided as NULL unless user desires setting different from default. (please refer to Section 4.6 for the contents of `option`.)

SLIP LU allows the input numerical data for `A` and `b` to come in one of 5 typess: `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 1. Conversely, triplet form stores the contents of the matrix A in three arrays `i`, `j`, and `x` where the kth nonzero entry is stored as $A(i[k], j[k]) = x[k]$. SLIP LU assumes that dense form stores entries 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, user can refer to `Demo/example2.c` on how to read in data and construct `A` and `b`. If the data for matrices are readily stored in vectors corresponding to CSC form, sparse triplet form or dense form, user can allocate a shallow `SLIP_matrix` and assign vectors accordingly, then use `SLIP_matrix_copy` to get a `SLIP_matrix` in the desired kind and type. For more details, user can refer to `Demo/example.c`. In a case when `A` is available in the format other than CSC mpz, and/or `b` is available in the format other than dense mpz, the following code snippet shows how to get `A` and `b` in a requred 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) */

SLIP_matrix *A, *b;
// A is a copy of the A1. A is a CSC matrix with mpz_t entries
SLIP_matrix_copy(&A, SLIP_CSC,   SLIP_MPZ, A1, option);
// b is a copy of the b1. b is dense with mpz_t entries.
SLIP_matrix_copy(&b, SLIP_DENSE, SLIP_MPZ, b1, option);
```

9.2 Simple SLIP LU Routines for Solving Linear Systems

After initializing the necessary data structures, SLIP LU obtains the solution to $Ax = b$ using the "simple" interface of SLIP LU, which requires only that user decides what data type that he/she wants for the entries of `SLIP_matrix *x`. SLIP LU allows entries of `x` to be `double`, `mpq_t`, or `mpfr_t` with an associated precision. This is done by using `SLIP_backslash` (Section 6.4).

The following code snippet shows how to get solution as a dense `mpq_t` matrix. User can modify accordingly to meet one's need.

```
SLIP_matrix *x;
SLIP_type my_type = SLIP_MPQ; // SLIP_MPQ, SLIP_MPFR, SLIP_FP64
SLIP_backslash(&x, my_type, A, b, option) ;
```

On successful return, this function returns `SLIP_OK` (see Section 4.1).

9.3 Expert SLIP LU Routines

If a user wishes to perform the SLIP LU factorization of the matrix A while capturing information about the factorization itself and solving the linear system, extra steps must be performed that are all done internally in the method described in the previous subsection. Particularly, the following steps must be performed: 1) declare `L`, `U`, the solution matrix (stored as dense `mpq_t`) `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 user's desired form. Below, we discuss each of these steps followed by an example of putting it all together.

9.3.1 Declare Workspace

Using SLIP LU 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
// option is declared and set to meet to user's need
SLIP_matrix *U = NULL;
SLIP_matrix *x = NULL;
SLIP_matrix *rhos = NULL;
int64_t* pinv = NULL;
SLIP_LU_analysis* S = NULL;
```

9.3.2 SLIP LU Symbolic Analysis

The symbolic analysis phase of SLIP LU computes the column permutation and guesses for the number of nonzeros in L and U. This function is called as:

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

9.3.3 Computing the Factorization

The matrices `L` and `U`, the pivot sequence `rhos`, and the row permutation `pinv` are computed via the `SLIP_LU_factorize` function (Section 6.2). Upon successful completion, this function returns `SLIP_OK`.

9.3.4 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 `SLIP_LU_solve` function (Section 6.3).

Upon successful completion, this function returns `SLIP_OK`.

In this step, user can set `option->check` to `true` to enable the solution check process as discussed in Section 6.3. The process can verify that the solution vector x satisfies $Ax = b$ in perfect precision intended for debugging. Note that this is entirely optional and not necessary. The solution returned is guaranteed to be exact. It appears here just as a verification that SLIP LU 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. Also, note that this process can be quite time consuming; thus it is not recommended to be used in general.

9.3.5 Converting the Solution Vector to the User's Desired Form

Upon completion of the above routines, the solution to the linear system is in a dense `mpq_t` matrix. SLIP LU allows this to be converted into any form of matrix in the set of (CSC, sparse triplet, dense) × (`mpfr_t`, `mpq_t`, `double`) using `SLIP_matrix_copy`. The following code snippet shows how to get solution as a dense double matrix.

```
SLIP_kind my_kind = SLIP_DENSE;  // SLIP_CSC, SLIP_TRIPLET or SLIP_DENSE
SLIP_type my_type = SLIP_FP64;   // SLIP_MPQ, SLIP_MPFR, or SLIP_FP64

SLIP_matrix* my_x = NULL;        // New output
Create copy which is stored as my_kind and my_type:
SLIP_matrix_copy( &my_x, my_kind, my_type, x, option);
```

9.4 SLIP LU Freeing Memory

As described in Section 5, SLIP LU provides a number of functions/macros to handle this for the user. Below, we briefly summarize which memory freeing routine should be used for specific data types:

- `SLIP_matrix*`: A `SLIP_matrix* A` data structure can be freed with a call to `SLIP_matrix_free(&A, NULL) ;`
- `SLIP_LU_analysis*`: A `SLIP_LU_analysis* S` data structure can be freed with a call to `SLIP_LU_analysis_free(&S, NULL) ;`
- All others including `SLIP_options*`: These data structures can be freed with a call to the macro `SLIP_FREE()`, for example, `SLIP_FREE(option)` for `SLIP_options* option`.

After all usage of the SLIP LU routines is finished, one must call `SLIP_finalize()` (Section 5.7) to finalize usage of the library.

9.5 Examples of Using SLIP LU in a C Program

The `SLIP_LU/Demo` folder contains three sample C codes which utilize SLIP LU. These files demonstrate the usage of SLIP LU 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 `SLIP_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 `SLIP_backslash` function, and, the solution is given as a matrix of rational numbers.

- `SLIPLU.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 9.3. This file also allows command line arguments (discussed in README.txt) and can be used to replicate the results from [8].

10 Using SLIP LU in MATLAB

After following the installation steps discussed in Section 3, using the SLIP LU factorization within MATLAB can be done via the `SLIP_LU.m` and the `SLIP_get_options` functions. First, this section will describe the `SLIP_get_options` struct in Section 10.1 then we describe how to use the factorization in Section 10.2. Again, recall that by default the SLIP LU MATLAB routines are not natively installed into your MATLAB installation; thus if you want to use them in a different directory please add the `SLIP_LU/MATLAB` folder to your path.

10.1 `SLIP_get_options.m`

Much like the C routines described throughout, the SLIP LU MATLAB interface has various parameters that the user can modify to control the factorization. In MATLAB, these are stored in a struct (hereafter referred to as the "options" struct). Notice that this struct is optional for the user to use and can be avoided if one wishes to use only default options. The options struct can be accessed by typing the following into the MATLAB command window:

```
option = SLIP_get_options;
```

The elements of the options struct are as follows:

- `option.pivot`: This parameter controls the pivoting scheme used. The factorization selects a pivot element in each column as follows:
  - 0: smallest pivot,
  - 1: diagonal pivot if possible, otherwise smallest pivot,
  - 2: first nonzero pivot in each column,

  - 3: (default) diagonal pivot with a tolerance (`option.tol`) for the smallest pivot,
  - 4: diagonal pivot with a tolerance (`option.tol`) for the largest pivot,
  - 5: largest pivot.

  It is recommended that the user always selects either 3 or 1 for this parameter UNLESS they are trying to extract the Doolittle factors, then 5 may be appropriate (due to the size of numbers in Doolittle).

- `option.order`: This parameter controls the column ordering used. 0: none, 1: COLAMD, 2: AMD. It is usually recommended that the user keep this at COLAMD unless they already have a good column permutation.

- `option.tol`: This parameter determines the tolerance used if one of the threshold pivoting schemes is chosen. The default value is 0.1 and this parameter can take any value in the range (0,1).

10.2 `SLIP_LU.m`

The `SLIP_LU.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.

The SLIP LU function expects as input a sparse matrix A and dense set of right hand side vectors $\mathbf{b}$. Optionally, the user can also pass in the options struct. Currently, there are 2 ways to use this function outlined below:

- `x = SLIP_LU(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`.

- `x = SLIP_LU(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using user specified settings from the options struct.

References

[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.