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

Version 1.0.2, July 14, 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 Overview

SLIP 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 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. 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.

For all primary computational routines in Section 8, 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. A discussion of how to use these matrix formats and data types in the `SLIP_matrix`, and to perform conversions between matrix types and formats in Section 10.1.3.

The matrices L and U are computed using 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 define the pivot ordering; Q is the 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 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], or no ordering ($Q = I$). A discussion of how to select these permutations prior to factorization is given in Section 8.

Once the factorization $LDU = PAQ$ is computed, the solution vector 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 require b stored 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.

Using the SLIP matrix copy function, any matrix in any of the 15 combinations of the set (CSC, triplet, dense) $\times$ (`mpz_t`, `mpq_t` , `mpfr_t`, `int64_t`, or `double`), can be copied and converted into any one of the 15 combinations.

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 x 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.

Most routines expect the input sparse matrix A to be stored in CSC format. 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, see Section 7.

An example matrix A with `mpz_t` type 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.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 Availability

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

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 convenience. The GNU GMP and GNU MPFR library can be acquired and installed from https://gmplib.org/ and http://www.mpfr.org/ respectively.

With 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 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 the `SLIP_backslash` function from within MATLAB. Note that `SLIP_install` does not add the correct directory to your path; therefore, if you want to use `SLIP_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 (type `doc startup` and `doc addpath` for more information).

4 Managing the SLIP LU environment

4.1 `SLIP_LU_VERSION`: the software package version

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

Macro	purpose
`SLIP_LU_VERSION`	current version of the code (as a string)
`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

4.2 `SLIP_info`: status code returned by SLIP LU

Most SLIP LU functions return their status to the caller as their return value, an enumerated type called `SLIP_info`. All current 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.
-5	SLIP_PANIC	SLIP LU environment error

Either `SLIP_initialize` or `SLIP_initialize_expert` (but not both) must be called prior to using any other SLIP LU function. `SLIP_finalize` must be called as the last SLIP LU function.

Subsequent SLIP LU sessions can be restarted after a call to `SLIP_finalize`, by calling either `SLIP_initialize` or `SLIP_initialize_expert` (but not both), followed by a final call to `SLIP_finalize` when finished.

4.3 `SLIP_initialize`: initialize the working environment

```
SLIP_info 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 and MPFR libraries. Either this function or `SLIP_initialize_expert` must be called prior to using any other function in the library. Returns `SLIP_PANIC` if SLIP LU has already been initialized, or `SLIP_OK` if successful.

4.4 `SLIP_initialize_expert`: initialize environment (expert version)

```
SLIP_info 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/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 SLIP_PANIC if SLIP LU has already been initialized, or SLIP_OK if successful.

4.5 SLIP_finalize: free the working environment

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

SLIP_finalize finalizes the working environment for SLIP LU library, and frees any internal workspace created by SLIP LU. It must be called as the last SLIP_* function called, except that a subsequent call to SLIP_initialize* may be used to start another SLIP LU session. Returns SLIP_PANIC if SLIP LU has not been initialized, or SLIP_OK if successful.

5 Memory Management

The routines in this section are used to allocate and free memory for the data structures used in SLIP LU. By default, SLIP LU 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 SLIP LU environment with SLIP_initialize_expert.

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. Returns NULL if SLIP LU has not been initialized.

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. Returns NULL if SLIP LU has not been initialized.

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 nitems_old × size_of_item. The object is reallocated to be of size nitems_new × 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. Returns `ok` as `false` if SLIP LU 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 SLIP LU 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 `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`. 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 `SLIP_FREE` is provided, which calls `SLIP_free` and then sets the freed pointer to `NULL`.

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

No action is taken if SLIP LU has not been initialized.

6 The SLIP_options object: parameter settings for SLIP LU

The SLIP_options object contains numerous parameters that may be modified to change the behavior of the SLIP LU 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.

6.1 SLIP_pivot: enum for pivoting schemes

There are six available pivoting schemes provided in SLIP LU that can be selected with the SLIP_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 SLIP LU 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 SLIP_options. The pivoting schemes are as follows:

0	SLIP_SMALLEST	The k-th pivot is selected as the smallest entry in the k-th 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.

6.2 SLIP_col_order: enum for column ordering schemes

The SLIP LU library provides three column ordering schemes: no pre-ordering, COLAMD, and AMD, selected via the `order` component in the `SLIP_options` structure described in Section 6.3.

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].

6.3 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 6.1) which controls the type of pivoting used. Default value: `SLIP_TOL_SMALLEST` (3).
- `option->order`: An enum `SLIP_col_order` type (discussed in Section 6.2) which controls what column ordering is used. Default value: `SLIP_COLAMD` (1).
- `option->tol`: A `double` tolerance for the 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 `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 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.

  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 SLIP LU library is guaranteed to return the exact solution. Default value: `false`.

All SLIP LU routines except basic memory management routines in Sections 4.5-5.1 and `SLIP_options` allocation routine in 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 `SLIP_options` object:

function/macro name	description	section
`SLIP_create_default_options`	create and return `SLIP_options` pointer with default parameters upon successful allocation	6.4
`SLIP_FREE`	destroy `SLIP_options` object	5.4

6.4 `SLIP_create_default_options`: create default `SLIP_options` 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 6.3. To safely free the `SLIP_options* option` structure, simply use `SLIP_FREE(option)`. All functions that require `SLIP_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 `SLIP_options` struct need not be allocated. Returns `NULL` if SLIP LU has not been initialized.

7 The `SLIP_matrix` object

All matrices for SLIP LU are stored as a `SLIP_matrix` object (a pointer to a `struct`). The matrix can be held in three formats: CSC, triplet or dense matrix (as discussed in Section 7.1) with entries stored as 5 different types: `mpz_t`, `mpq_t`, `mpfr_t`, `int64_t` and `double` (as discussed in Section 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.

7.1 `SLIP_kind`: enum for matrix formats

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

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.

7.2 `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`.

0	`SLIP_MPZ`	Matrix entries are in `mpz_t` type: an integer of arbitrary size.
1	`SLIP_MPQ`	Matrix entries are in `mpq_t` type: a rational number with arbitrary-sized integer numerator and denominator.
2	`SLIP_MPFR`	Matrix entries are in `mpfr_t` type: a floating-point number of arbitrary precision.
3	`SLIP_INT64`	Matrix entries are in `int64_t` type.
4	`SLIP_FP64`	Matrix entries are in `double` type.

7.3 `SLIP_matrix` structure

A matrix `SLIP_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 `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 `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. 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` $\times$ `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; 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) `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`. The `A->p` array is `NULL`. Triplets can be unsorted, but duplicates cannot appear.

- (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; the number of entries in `A` is `A->m` $\times$ `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`	7.4
`SLIP_matrix_free`	destroy a `SLIP_matrix` and free its allocated memory	7.5
`SLIP_matrix_copy`	make a copy of a matrix, into another kind and/or type	7.6
`SLIP_matrix_nnz`	get the number of entries in a matrix	7.7

7.4 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 (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 SLIP_mp*_init function). If
                            // false, the mpz, mpq, and mpfr arrays are
                            // allocated but not initialized.
    const SLIP_options *option
) ;
```

`SLIP_matrix_allocate` allocates memory space for a m-by-n `SLIP_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 `SLIP_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 `SLIP_mp*_init`) function. Otherwise, if `init` is false, the `A->x.TYPE` array is allocated (via `SLIP_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 `SLIP_PANIC` if SLIP LU has not been initialized.

7.5 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`. Note that the input of the function is the pointer to the pointer of a `SLIP_matrix` structure. This is because this function internally sets the pointer of a `SLIP_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`.

7.6 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 fp64
    SLIP_matrix *A,         // matrix to make a copy of (may be shallow)
    const SLIP_options *option
) ;
```

`SLIP_matrix_copy` creates a `SLIP_matrix *C` which is a modified copy of a `SLIP_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 `SLIP_matrix_check` (Section 7.8), if desired. Results are undefined for an invalid input matrix `A`. Returns `SLIP_PANIC` if SLIP LU has not been initialized.

7.7 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, refer to Section 7. For any matrix with invalid dimension(s), this function returns -1. If default settings are desired, `option` can be input as `NULL`. Returns -1 if SLIP LU has not been initialized.

7.8 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` checks the validity of a `SLIP_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 6 for more details). If default settings are desired, `option` can be input as `NULL`. Returns `SLIP_PANIC` if SLIP LU has not been initialized.

8 Primary Computational Routines

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

8.1 SLIP_LU_analysis structure

The `SLIP_LU_analysis` data structure is used for storing the column permutation for LU 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 `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 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 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	8.2
`SLIP_LU_analysis_free`	destroy `SLIP_LU_analysis` object	8.3

8.2 `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_options *option`. For more details, refer to Section 6.

The `SLIP_LU_analysis *S` is created by calling `SLIP_LU_analyze(&S, A, option)` with `SLIP_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 object and `SLIP_OK` is returned upon successful completion, or `S = NULL` with error status returned if a failure occurred. Returns `SLIP_PANIC` if SLIP LU has not been initialized.

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

8.3 `SLIP_LU_analysis_free`: free `SLIP_LU_analysis` structure

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

`SLIP_LU_analysis_free` frees 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 settings are desired, `option` can be input as `NULL`. Returns `SLIP_PANIC` if SLIP LU has not been initialized.

8.4 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,   // column permutation and estimates
                                 // of nnz in L and U
    const SLIP_options* option
) ;
```

`SLIP_LU_factorize` performs the SLIP 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 `SLIP_LU/Demos/SLIPLU.c` or Section 10.3.

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 `SLIP_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 ;
SLIP_mpq_init (determinant) ;
SLIP_mpq_set_z (determinant, rhos->x.mpz[rhos->n-1]) ;
SLIP_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 `SLIP_PANIC` if SLIP LU 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.

8.5 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` 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 `SLIP_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 SLIP LU 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 `SLIP_LU/Demos/SLIPLU.c` or Section 10.3. Returns `SLIP_PANIC` if SLIP LU has not been initialized.

8.6 SLIP_backslash: solve $Ax = b$

```
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 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, `SLIP_matrix *x` is 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_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 `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 occurred), the function returns corresponding error code.

Returns `SLIP_PANIC` if SLIP LU has not been initialized.

For a complete example, refer to `SLIP_LU/Demos/example.c`, `SLIP_LU/Demos/example2.c`, or Section 10.2.

9 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, 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 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:

```
SLIP_info SLIP_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
    //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 mpfr_t type (say TYPEa = mpfr_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, ...)`, the wrapped function can be constructed as follows:

```
SLIP_info SLIP_gmpfunc
(
    TYPEr *r,        // return value of the GMP/MPFR function
    TYPEa a,
    TYPEb b,
    ...
)
{
    // Start the GMP Wrappter
    //SLIP_GMP_WRAPPER_START;

    // 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_asprintf(&buff, fmt, ...)	n = SLIP_mpfr_asprintf(&buff, fmt, ...)	Print format to allocated string
mpfr_free_str(buff)	SLIP_mpfr_free_str(buff)	Free string allocated by MPFR
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_t)
mpfr_set_z(x, y, rnd)	SLIP_mpfr_set_z(x, y, rnd)	$x = y$ (mpz_t)
mpfr_get_z(x, y, rnd)	SLIP_mpfr_get_z(x, y, rnd)	(mpz_t) $x = y$
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_fscanf(fp, fmt, ...)	n = SLIP_gmp_fscanf(fp, fmt, ...)	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_t)
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)	$x = y$ (mpz_t)
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 $x \neq y$

10 Using SLIP LU in C

Using SLIP 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 10.1. 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, SLIP LU provides a simple interface for this purpose which is discussed in Section 10.2. Alternatively, if the L and U factors are required, refer to Section 10.3. Finally, step 3 is discussed in Section 10.4. 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 $b \in \mathbb{Z}^{n \times r}$).

10.1 SLIP LU initialization and population of data structures

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

10.1.1 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. 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
```

10.1.2 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 input coefficient matrix, while `b` contains the right hand side vector(s) of the linear system $Ax = b$.
- `SLIP_LU_analysis* S`: `S` contains the column permutation used for A as well as estimates of 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, refer to Section 6. If default settings are desired, `option` can be set to `NULL`.

10.1.3 Populating data structures

Of the three data structures discussed in Section 10.1.2, `S` is constructed during symbolic analysis (Section 8.2), and `option` is an optional parameter for selecting non-default parameters. Refer to Section 6 for the contents of `option`.

SLIP LU 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 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 $A(i[k], j[k]) = x[k]$. SLIP LU 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 `SLIP_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 `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, refer to `SLIP_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) */

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 a dense matrix with mpz_t entries.
SLIP_matrix_copy(&b, SLIP_DENSE, SLIP_MPZ, b1, option);
```

10.2 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, `SLIP_backslash`. The `SLIP_backslash` function can return `x` as `double`, `mpq_t`, or `mpfr_t` with an associated precision. See Section 8.6 for more details. The following code snippet shows how to get solution as a dense `mpq_t` matrix.

```
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 4.2).

10.3 Expert SLIP LU routines

If the L and U factors from the SLIP LU factorization of the matrix A are required, the steps performed by `SLIP_backslash` can be done with a sequence of calls to SLIP LU 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.

10.3.1 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
SLIP_matrix *L = NULL;
SLIP_matrix *U = NULL;
SLIP_matrix *x = NULL;
SLIP_matrix *rhos = NULL;
int64_t* pinv = NULL;
SLIP_LU_analysis* S = NULL;

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

10.3.2 SLIP LU symbolic analysis

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

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

10.3.3 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 8.4). Upon successful completion, this function returns `SLIP_OK`.

10.3.4 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 8.5). Upon successful completion, this function returns `SLIP_OK`.

In this step, `option->check` can be set to `true` to enable the solution check process as discussed in Section 8.5. 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 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 (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.

10.3.5 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. SLIP LU 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 `SLIP_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.

```
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);
```

10.4 Freeing memory

As described in Section 5, SLIP LU provides a number of functions/macros to free SLIP LU objects:

- `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, `SLIP_finalize()` must be called (Section 4.5) to finalize usage of the library.

10.5 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 10.3. This file also allows command line arguments (discussed in `README.md`) and can be used to replicate the results from [8].

11 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_backslash.m` function. First, this section describes the `option` struct in Section 11.1. The use of the factorization is discussed in Section 11.2. The `SLIP_LU/MATLAB` folder must be in your MATLAB path.

11.1 Optional parameter settings

The SLIP 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).

11.2 SLIP_backslash.m

The `SLIP_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 SLIP 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 = SLIP_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 = SLIP_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 11.1 for details.

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.