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

Version 1.0.0, November 2019

Christopher Lourenco, Jinhao Chen, Erick Moreno-Centeno, Timothy A. Davis

Contact Information: clouren@tamu.edu OR chrisjlourenco@gmail.com

Contents

1 Summary

SLIP_LU is a software package designed to exactly solve sparse linear systems, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^{n\times m}$, 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 integral, 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. SLIP_LU is written in C and comes with a MATLAB interface.

The matrices L and U are computed using internal, integer-preserving routines with the big integer (mpz_t) data types from the GNU Multiple Precision Arithmetic (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., integral); therefore, no greatest common divisor algorithms are needed to reduce the size of entries.

The 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 [5, 4]. 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 matrices prior to factorization is given in Section 5.

Once the factorization $LDU = PAQ$ is computed, the vector $\mathbf{x}$ is computed via sparse REF forward and backward substitution. The forward substitution assumes that $\mathbf{b}$ is dense. For the backward substitution, a typical column oriented sparse backward substitution is performed.

The final solution vector $\mathbf{x}$ is given as an exact rational vector using the GMP mpq_t data structure. Alternatively, the solution vector can be output as a double or variable precision floating point (mpfr_t) vector. A major advantage of this approach, instead of using other commercial packages, is that roundoff errors are only induced in the final conversion from rational number to floating point precision; therefore, this solution vector is accurate to machine roundoff.

All left-hand side matrices (i.e., A) within this package are stored in compressed column form (CCF). The matrix A is stored as a sequence of three arrays as follows:

A->p: Column pointers. Data type: int32_t

A->i: Row indices. Data type: int32_t

A->x: Numeric entries. 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]

2 Availability

Copyright: This software is copyright by Christopher Lourenco.

Contact Info: clouren@tamu.edu OR chrisjlourenco@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 Christopher Lourenco at clouren@tamu.edu or chrisjlourenco@gmail.com.

Location: TODO: Insert location here

Required Packages: `SLIP_LU` requires the installation of AMD [1, 2], COLAMD [5, 4], SuiteSparse [3], the GNU GMP [7] and GNU MPFR [6] libraries, Notice that AMD, COLAMD, and SuiteSparse are included in this distribution for users' convenience. The GNU GMP and GNU MPFR library can be installed from `https://gmplib.org/` and `http://www.mpfr.org/` respectively.

If a user is running Unix that is Debian/Ubuntu based, 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 or `Cygwin make` in Windows. With the proper compiler, type `make` under the main directory will compile AMD, COLAMD and SLIP_LU to respective /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"`.

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 Overview of User Data Structures

There are four important data structures used throughout the `SLIP_LU` package. The `SLIP_options` contains command parameters, `SLIP_sparse` and `SLIP_dense` are used to store the sparse and dense matrix, respectively, and `SLIP_LU_analysis` contains the column permutation. Furthermore, three enumerated types (`enum`) are defined and used, including `SLIP_pivot` for available pivot schemes, `SLIP_col_order` for available column ordering methods and `SLIP_info` for returned status of a function.

`SLIP_LU` also 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_PAPER`	name of associated paper
`SLIP_AUTHOR`	authors of the code
`SLIP_MPFR_ROUND`	type of MPFR rounding used, currently using `MPFR_RNDN`

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 singular.
-3	`SLIP_INCORRECT_INPUT`	One or more input arguments are incorrect.
-4	`SLIP_INCORRECT`	The solution is incorrect.

4.2 `SLIP_pivot`: code 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 4.4. Note that the pivot is always nonzero, thus the smallest entry is the nonzero entry with smallest digit (that is, smallest absolute value). Also, the tolerance is specified by the `tol` component in `SLIP_options`, for which the detail can be referred to Section 4.4. 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.
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`: code for column ordering schemes

The `SLIP_LU` library provides three column ordering schemes, which are list as follows. Users can set the column ordering method through `order` component in the `SLIP_option` structure described in Section 4.4. Please note that `SLIP_NO_ORDERING` can be used when a good column pre-ordering has been applied to the matrix A, or simply when the users want to disable the pre-ordering feature.

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.
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 was shown to have worse performance on real world unsymmetric instances versus COLAMD (refer to [5] and [8]).

4.4 SLIP_options Structure

The data structure SLIP_options stores key command parameters for the functions in this SLIP_LU package. A SLIP_options* option struct contains the following components:

option->check: A boolean indicates whether the solution to the linear system will be checked for accuracy. To check a solution, set option->check = true and SLIP_LU will go through extra function call to check if Ax = b is in exact arithmetic. Therefore, be warned that this check can be more time consuming than the factorization itself. Since the method has been proved to be mathematically correct [8], the returned solution can always ensure Ax = b in exact arithmetic, unless the source code of SLIP_LU has been manually modified. Default value: false.

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: 0.1.

option->print_level: An int32_t 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 int64_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 2. Default value: 128 (quad precision).

The SLIP_LU package uses 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	5.1.8
SLIP_FREE	destroy SLIP_options object	5.1.4

4.5 The SLIP_sparse Structure

All internal left-hand side matrices are stored in sparse compressed column form (CCF) via the SLIP_sparse structure. A sparse matrix SLIP_sparse *A has the following components:

A->m: Number of rows in the matrix. It is typically assumed that $m = n$. Data Type: int32_t

A->n: Number of columns in the matrix. It is typically assumed that $m = n$. Data Type: int32_t

A->nz: The number of nonzeros in the matrix A. Data Type: int32_t

A->nzmax: The allocated size of the vectors A->x and A->i. Note that A->nzmax $\geq$ A->nz. Data Type: int32_t

A->p: An array of size $n + 1$ which contains column pointers of A. Data Type: int32_t*

A->i: An array of size A->nzmax which contains the row indices of the nonzeros in A. The matrix is zero based therefore indices are in the range of $[0, n-1]$. Data Type: int32_t*

A->x: An array of size A->nzmax which contains the numeric values of the matrix. Data Type: mpz_t*

A->scale: A scaling parameter that ensures integrality if the input sparse matrix is stored as either double, variable precision floating point, or rational. Data Type: mpq_t

The SLIP_LU package has a set of functions to create, build and destroy a SLIP_LU sparse matrix, SLIP_sparse:

function name	description	section
SLIP_create_sparse	create empty sparse matrix	5.1.9
SLIP_build_sparse_ccf_double	build sparse matrix from double type ccf matrix	5.2.1
SLIP_build_sparse_ccf_int	build sparse matrix from int32_t type ccf matrix	5.2.2
SLIP_build_sparse_ccf_mpq	build sparse matrix from mpq_t type ccf matrix	5.2.3
SLIP_build_sparse_ccf_mpfr	build sparse matrix from mpfr_t type ccf matrix	5.2.4
SLIP_build_sparse_ccf_mpz	build sparse matrix from mpz_t type ccf matrix	5.2.5
SLIP_build_sparse_trip_double	build sparse matrix from double type triplet-format matrix	5.2.6
SLIP_build_sparse_trip_int	build sparse matrix from int32_t type triplet-format matrix	5.2.7
SLIP_build_sparse_trip_mpq	build sparse matrix from mpq_t type triplet-format matrix	5.2.8
SLIP_build_sparse_trip_mpfr	build sparse matrix from mpfr_t type triplet-format matrix	5.2.9
SLIP_build_sparse_trip_mpz	build sparse matrix from mpz_t type triplet-format matrix	5.2.10
SLIP_delete_sparse	destroy sparse matrix	5.1.10

4.6 The SLIP_dense Structure

All internal right-hand side matrices are stored as SLIP_dense structure. A dense matrix SLIP_dense *b has the following components:

b->m: Number of rows in the matrix. Data Type: int32_t

b->n: Number of columns in the matrix. Data Type: int32_t

b->x: An 2D array of size m-by-n which contains the numeric values of the matrix. Data Type: mpz_t**

b->scale: A scaling parameters that ensures integrality if the input dense matrix is stored as either double, variable precision floating point, or rational. Data Type: mpq_t

The SLIP_LU package has a set of functions to create, build and destroy a SLIP_LU dense matrix, SLIP_dense:

function name	description	section
SLIP_create_dense	create empty dense matrix	5.1.11
SLIP_build_dense_double	build dense matrix from double type ccf matrix	5.2.11
SLIP_build_dense_int	build dense matrix from int32_t type ccf matrix	5.2.12
SLIP_build_dense_mpq	build dense matrix from mpq_t type ccf matrix	5.2.13
SLIP_build_dense_mpfr	build dense matrix from mpfr_t type ccf matrix	5.2.14
SLIP_build_dense_mpz	build dense matrix from mpz_t type ccf matrix	5.2.15
SLIP_delete_dense	destroy dense matrix	5.1.12

4.7 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* S has the following components:

S->q: The column permutation stored as a dense int32_t vector of size $n+1$, where n is the number of rows of the analyzed matrix. Currently this vector is obtained via COLAMD, AMD or no ordering.

S->lnz: An int32_t which is a guess for the number of nonzeros in L. S->lnz must be greater than or equal to n and less than or equal to n^2. If S->lnz is too large, the program may run out of memory. Alternatively, if this parameter is too small, the program may waste time performing extra memory reallocations. This is set during the symbolic analysis.

S->unz: An int32_t which is a guess for the number of nonzeros in U. S->unz must be greater than or equal to n and less than or equal to n^2. If S->unz is too large, the program may run out of memory. Alternatively, if this parameter 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_create_LU_analysis	create and return SLIP_LU_analysis pointer	5.1.13
SLIP_delete_LU_analysis	destroy SLIP_LU_analysis object	5.1.14

5 Overview of SLIP_LU User-Called Routines

This section provides a brief overview of the user callable routines in SLIP_LU. A list of this routines can be located in the SLIP_LU.h file. This section briefly describes each user callable function's purpose, syntax, input, and output.

5.1 Memory Management Routines

The routines in this section are used to allocate and free memory for the data structures used in SLIP_LU.

5.1.1 SLIP_calloc: allocate initialized memory

```
void *SLIP_calloc
(
    size_t n,           // Size of array
    size_t size         // Size to alloc
);
```

SLIP_calloc allocates block of memory for an array of n elements, each of them size bytes long, and initializes all its bits to zero. The function makes sure n and size at least be 1. If the function failed to allocate the requested block of memory, then a NULL pointer is returned.

5.1.2 SLIP_malloc: allocate uninitialized memory

```
void * SLIP_malloc
(
    size_t size         // Size to alloc
);
```

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. The function function makes sure size at least be 1. If the function failed to allocate the requested block of memory, then a NULL pointer is returned.

5.1.3 SLIP_realloc: resize allocated memory

```
void* SLIP_realloc
(
    void *p,             // Pointer to be realloced
    size_t old_size,     // Old size of this pointer
    size_t new_size      // New size of this pointer
);
```

SLIP_realloc attempts to resize the memory block pointed to by p that was previously allocated with a call to SLIP_malloc or SLIP_calloc. In the case when the function fails to allocate new block of memory as required and the newly required memory size is smaller than the old one, then the old block is kept unchanged and `SLIP_realloc` pretends to succeed. Otherwise, the function returns either `NULL` when it fails, or the new block of memory when it succeeds.

5.1.4 `SLIP_free`: free allocated memory

```
void SLIP_free
(
    void *p            // Pointer to be free'd
);
```

`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 created, which sets the free'd pointer to `NULL`.

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

5.1.5 `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. See the next section `SLIP_initialize_expert` for more details.

5.1.6 `SLIP_initialize_expert`: initialize the working environment (expert version)

```
void SLIP_initialize_expert
(
    void* (*MyMalloc) (size_t),                  // User defined malloc
    void* (*MyRealloc) (void *, size_t , size_t),// User defined realloc
    void (*MyFree) (void*, size_t)               // User defined free
);
```

`SLIP_initialize_expert` initializes the working environment for SLIP LU with custom memory functions that are used for GMP. If the user passes in their own `malloc`, `realloc`, or `free` function(s), we use those internally to process memory. If a NULL pointer is passed in for any function, then default functions are used.

The three functions are similar to ANSI C `malloc`, `realloc`, and `free` functions, but the calling syntax is not the same. Below are the definitions that **must** be followed, per the GMP specification:

```
void *MyMalloc (size_t size) ;  // same as the ANSI C malloc
void *MyRealloc (void *p, size_t oldsize, size_t newsize) ; // differs
void MyFree (void *p, size_t size) ; // differs
```

`MyMalloc` has the identical to the ANSI C `malloc`. MyRealloc adds a parameter, `oldsize`, which is the prior size of the block of memory to be reallocated. `MyFree` takes a second argument, which is the size of the block that is being free'd.

The default memory management functions used inside GMP are:

```
MyMalloc     slip_gmp_allocate
MyRealloc    slip_gmp_reallocate
MyFree       slip_gmp_free
```

The `slip_gmp_*` memory management functions are unique to SLIP LU Library. They provide an elegant workaround for how GMP manages its memory. By default, if GMP attempts to allocate memory, but it fails, then it simply terminates the user application. This behavior is not suitable for many applications (MATLAB in particular). Fortunately, GMP allows the user application (SLIP LU in this case) to pass in alternative memory manager functions, via `mp_set_memory_functions`. The `slip_gmp_*` functions do not return to GMP if the allocation fails, but instead use the `longjmp` feature of ANSI C to implement a try/catch mechanism. The memory failure can then be safely handled by SLIP LU, without memory leaks and without terminating the user application.

When SLIP LU is used via MATLAB, the following functions are used instead:

```
MyMalloc     mxMalloc
MyRealloc    slip_gmp_mex_realloc (a wrapper for mxRealloc)
MyFree       slip_gmp_mex_free (a wrapper for mxFree)
```

Note that these functions are not used by SLIP LU itself, but only inside GMP. The functions used by SLIP LU itself are `SLIP_malloc`, `SLIP_calloc`, `SLIP_realloc`, and `SLIP_free`, which are wrappers for the ANSI C `malloc`, `calloc`, `realloc`, and `free` (see Sections 5.1.1-5.1.4), or (if used inside MATLAB), for the MATLAB `mxMalloc`, `mxCalloc`, `mxRealloc`, and `mxFree` functions.

5.1.7 `SLIP_finalize`: free the working environment

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

SLIP_finalize frees the working environment for SLIP LU library. SLIP LU utilizes a specialized memory management scheme in order to prevent memory failures. Calling the function SLIP_finalize after you are finished using the library ensures all memory is freed.

5.1.8 SLIP_create_default_options: create default SLIP_option object

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

SLIP_create_default_options creates and returns SLIP_options pointer with default parameters upon successful allocation, which are discussed in Section 4.4. To safely free the SLIP_options* option structure, simply use SLIP_FREE(option).

5.1.9 SLIP_create_sparse: create empty sparse matrix

```
SLIP_sparse *SLIP_create_sparse
(
    void
);
```

SLIP_create_sparse returns a poiter to a created empty sparse matrix as SLIP_sparse pointer upon successful allocation. The returned matrix has size of 0-by-0 and scale parameter of 1. For methods to build the created sparse matrix, users can refer to the table in Section 4.5.

5.1.10 SLIP_delete_sparse: delete sparse matrix

```
void SLIP_delete_sparse
(
    SLIP_sparse **A // matrix to be deleted
);
```

SLIP_delete_sparse deletes the sparse matrix A. Note that the input of the function is the pointer to the pointer of a SLIP_sparse structure. This is because this function internally sets the pointer of a SLIP_sparse to be NULL to prevent potential segmentation fault that could be caused by double free.

5.1.11 SLIP_create_dense: create empty dense matrix

```
SLIP_dense *SLIP_create_dense
(
    void
);
```

`SLIP_create_dense` returns a pointer to a created empty dense matrix as `SLIP_dense` pointer upon successful allocation. The returned matrix has size of 0-by-0 and scale parameter of 1. For methods to build the created dense matrix, users can refer to the table in Section 4.6.

5.1.12 `SLIP_delete_dense`: delete dense matrix

```
void SLIP_delete_dense
(
    SLIP_dense **A
);
```

`SLIP_delete_dense` deletes the dense matrix `A`. Note that the input of the function is the pointer to the pointer of a `SLIP_dense` structure. This is because this function internally sets the pointer of a `SLIP_dense` to be `NULL` to prevent potential segmentation fault that could be caused by double `free`.

5.1.13 `SLIP_create_LU_analysis`: create `SLIP_LU_analysis` structure

```
SLIP_LU_analysis *SLIP_create_LU_analysis
(
    int32_t n        // length of S->q
);
```

This function returns a pointer to a created `SLIP_LU_analysis` type with the length of `S->q` set as `n` (which needs to be 1 + number of rows of the analyzed matrix) upon successful allocation, otherwise, return NULL. Both `lnz` and `unz` are set to 0. For more information about the `SLIP_LU_analysis` structure, users can refer to Section 4.7

5.1.14 `SLIP_delete_LU_analysis`: delete `SLIP_LU_analysis` structure

```
void SLIP_delete_LU_analysis
(
    SLIP_LU_analysis **S // Structure to be deleted
);
```

`SLIP_delete_LU_analysis` 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`.

5.2 Matrix Building Routines

The routines in this section are used to build either the sparse matrix or the dense matrix.

5.2.1 `SLIP_build_sparse_ccf_double`: build sparse matrix using CCF with `double` entries

```
SLIP_info SLIP_build_sparse_ccf_double
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *p,           // The set of column pointers
    int32_t *I,           // set of row indices
    double *x,            // Set of values as doubles
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x and I vectors)
);
```

`SLIP_build_sparse_ccf_double` builds a sparse matrix using compressed column form inputs, where the entry values are `double` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.2 `SLIP_build_sparse_ccf_int`: build sparse matrix using CCF with `int32_t` entries

```
SLIP_info SLIP_build_sparse_ccf_int
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *p,           // The set of column pointers
    int32_t *I,           // set of row indices
    int32_t *x,           // Set of values as doubles
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x and I vectors)
);
```

`SLIP_build_sparse_ccf_int` builds a sparse matrix using compressed column form inputs, where the entry values are `int32_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.3 `SLIP_build_sparse_ccf_mpq`: build sparse matrix using CCF with `mpq_t` entries

```
SLIP_info SLIP_build_sparse_ccf_mpq
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *p,           // The set of column pointers
    int32_t *I,           // set of row indices
    mpq_t *x,             // Set of values as mpq_t rational numbers
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x and I vectors)
);
```

`SLIP_build_sparse_ccf_mpq` builds a sparse matrix using compressed column form inputs, where the entry values are `mpq_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.4 `SLIP_build_sparse_ccf_mpfr`: build sparse matrix using CCF with `mpfr_t` entries

```
SLIP_info SLIP_build_sparse_ccf_mpfr
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *p,           // The set of column pointers
    int32_t *I,           // set of row indices
    mpfr_t *x,            // Set of values as doubles
    int32_t n,            // dimension of the matrix
    int32_t nz,           // number of nonzeros in A (size of x and I vectors)
    SLIP_options *option  // command options containing the prec for mpfr
);
```

`SLIP_build_sparse_ccf_mpfr` builds a sparse matrix using compressed column form inputs, where the entry values are `mpfr_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.5 `SLIP_build_sparse_ccf_mpz`: build sparse matrix using CCF with `mpz_t` entries

```
SLIP_info SLIP_build_sparse_ccf_mpz
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *p,           // The set of column pointers
    int32_t *I,           // set of row indices
    mpz_t *x,             // Set of values in full precision int.
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x and I vectors)
);
```

`SLIP_build_sparse_ccf_mpz` builds a sparse matrix using compressed column form inputs, where the entry values are `mpz_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.6 `SLIP_build_sparse_trip_double`: build sparse matrix using triplet with `double` entries

```
SLIP_info SLIP_build_sparse_trip_double
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *I,         // set of row indices
    int32_t *J,         // set of column indices
    double *x,          // Set of values in double
    int32_t n,          // dimension of the matrix
```

```
    int32_t nz            // number of nonzeros in A (size of x, I,
                          // and J vectors)
);
```

`SLIP_build_sparse_trip_double` builds a sparse matrix using triplet form inputs, where the entry values are `double` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.7 `SLIP_build_sparse_trip_int`: build sparse matrix using triplet with `int32_t` entries

```
SLIP_info SLIP_build_sparse_trip_int
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *I,           // set of row indices
    int32_t *J,           // set of column indices
    int32_t *x,           // Set of values in int
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x, I,
                          // and J vectors)
);
```

`SLIP_build_sparse_trip_int` builds a sparse matrix using triplet form inputs, where the entry values are `int32_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.8 `SLIP_build_sparse_trip_mpq`: build sparse matrix using triplet with `mpq_t` entries

```
SLIP_info SLIP_build_sparse_trip_mpq
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *I,           // set of row indices
    int32_t *J,           // set of column indices
    mpq_t *x,             // Set of values as rational numbers
    int32_t n,            // dimension of the matrix
    int32_t nz            // number of nonzeros in A (size of x, I
                          // and J vectors)
);
```

`SLIP_build_sparse_trip_mpq` builds a sparse matrix using triplet form inputs, where the entry values are `mpq_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.9 `SLIP_build_sparse_trip_mpfr`: build sparse matrix using triplet with `mpfr_t` entries

```
SLIP_info SLIP_build_sparse_trip_mpfr
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *I,          // set of row indices
    int32_t *J,          // set of column indices
    mpfr_t *x,           // Set of values as mpfr_t
    int32_t n,           // dimension of the matrix
    int32_t nz,          // number of nonzeros in A (size of x, I,
                         // and J vectors)
    SLIP_options *option// command options containing the prec for mpfr
);
```

`SLIP_build_sparse_trip_mpfr` builds a sparse matrix using triplet form inputs, where the entry values are `mpfr_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.10 `SLIP_build_sparse_trip_mpz`: build sparse matrix using triplet with `mpz_t` entries

```
SLIP_info SLIP_build_sparse_trip_mpz
(
    SLIP_sparse *A_output,// It should be initialized but unused yet
    int32_t *I,          // set of row indices
    int32_t *J,          // set of column indices
    mpz_t *x,            // Set of values in full precision int
    int32_t n,           // dimension of the matrix
    int32_t nz           // number of nonzeros in A (size of x, I,
                         // and J vectors)
);
```

`SLIP_build_sparse_trip_mpz` builds a sparse matrix using triplet form inputs, where the entry values are `mpz_t` type. The input sparse matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_sparse` (in Section 5.1.9). WARNING: Using sparse matrix that has been built with `SLIP_build_sparse_*` as input would cause memory leak.

5.2.11 `SLIP_build_dense_double`: build dense matrix using 2D `double` array

```
SLIP_info SLIP_build_dense_double
(
    SLIP_dense *A_output, // Dense matrix, allocated but unused
    double **b,           // Set of values as doubles
    int32_t m,            // number of rows
    int32_t n             // number of columns
);
```

`SLIP_build_dense_double` builds a dense matrix using 2D `double` array. The input dense matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_dense` (in Section 5.1.11). WARNING: Using dense matrix that has been built with `SLIP_build_dense_*` as input would cause memory leak.

5.2.12 `SLIP_build_dense_int`: build dense matrix using 2D `int32_t` array

```
SLIP_info SLIP_build_dense_int
(
    SLIP_dense *A_output, // Dense matrix, allocated but unused
    int32_t **b,          // Set of values as ints
    int32_t m,            // number of rows
    int32_t n             // number of columns
);
```

`SLIP_build_dense_int` builds a dense matrix using 2D `int32_t` array. The input dense matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_dense` (in Section 5.1.11). WARNING: Using dense matrix that has been built with `SLIP_build_dense_*` as input would cause memory leak.

5.2.13 `SLIP_build_dense_mpq`: build dense matrix using 2D `mpq_t` array

```
SLIP_info SLIP_build_dense_mpq
(
    SLIP_dense *A_output, // dense matrix, allocated but unused
    mpq_t **b,            // set of values as mpq_t
    int32_t m,            // number of rows
    int32_t n             // number of columns
);
```

`SLIP_build_dense_mpq` builds a dense matrix using 2D `mpq_t` array. The input dense matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_dense` (in Section 5.1.11). WARNING: Using dense matrix that has been built with `SLIP_build_dense_*` as input would cause memory leak.

5.2.14 `SLIP_build_dense_mpfr`: build dense matrix using 2D `mpfr_t` array

```
SLIP_info SLIP_build_dense_mpfr
(
    SLIP_dense *A_output, // Dense matrix, allocated but unused
    mpfr_t **b,           // Set of values as mpfr_t
    int32_t m,            // number of rows
    int32_t n,            // number of columns
    SLIP_options *option  // command options containing the prec for mpfr
);
```

`SLIP_build_dense_mpfr` builds a dense matrix using 2D `mpfr_t` array. The input dense matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_dense` (in Section 5.1.11). WARNING: Using dense matrix that has been built with `SLIP_build_dense_*` as input would cause memory leak.

5.2.15 `SLIP_build_dense_mpz`: build dense matrix using 2D `mpz_t` array

```
SLIP_info SLIP_build_dense_mpz
(
    SLIP_dense *A_output, // Dense matrix, allocated but unused
    mpz_t **b,            // Set of values in full precision int.
    int32_t m,            // number of rows
    int32_t n             // number of columns
);
```

`SLIP_build_dense_mpz` builds a dense matrix using 2D `mpz_t` array. The input dense matrix `A_output` should be a created empty matrix. The best practice is to use the one returned from `SLIP_create_dense` (in Section 5.1.11). WARNING: Using dense matrix that has been built with `SLIP_build_dense_*` as input would cause memory leak.

5.3 Utility Routines

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

5.3.1 `SLIP_LU_analyze`: perform symbolic analysis

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

`SLIP_LU_analyze` performs the symbolic ordering for SLIP LU. Currently, there are three options: user-defined order, COLAMD, or AMD, which are passed in by `SLIP_option *option`. For more details, users can refer to Section 4.4.

5.3.2 `SLIP_LU_factorize`: perform LU factorization

```
SLIP_info SLIP_LU_factorize
(
    SLIP_sparse *L,          // lower triangular matrix
    SLIP_sparse *U,          // upper triangular matrix
    SLIP_sparse *A,          // matrix to be factored
    SLIP_LU_analysis *S,     // prior symbolic analysis
    mpz_t *rhos,             // sequence of pivots
    int32_t *pinv,           // inverse row permutation
    SLIP_options *option     // command options
```

```
);
```

`SLIP_LU_factorize` performs the SLIP LU factorization. This factorization is done via n (number of rows in 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 6.4.

On input, both `L` and `U` should be created using `SLIP_create_sparse` (see Section 5.1.9), and both `mpz_t *rhos` and `int32_t *pinv` should be allocated as arrays of size n using `SLIP_create_mpz_array` (see Section 5.4.15) and `SLIP_malloc` (see Section 5.1.2), respectively.

On output, `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, `p[i]` gives the row index in PA).

5.3.3 `SLIP_LU_solve`: solve the scaled linear system $LDUx = b$

```
SLIP_info SLIP_LU_solve      //solves the linear system LDU x = b
(
    mpq_t **x,               // rational solution to the system
    SLIP_dense *b,           // right hand side vector
    mpz_t *rhos,             // sequence of pivots
    SLIP_sparse *L,          // lower triangular matrix
    SLIP_sparse *U,          // upper triangular matrix
    int32_t *pinv            // row permutation
);
```

`SLIP_LU_solve` obtains the solution to the scaled linear system $LDUx = b$ upon a successful factorization.

On input, `mpq_t **x` should be allocated as a 2D array of same size as `b` using `SLIP_create_mpq_mat` (see Section 5.4.7). The function is called upon successful return from `SLIP_LU_factorize`.

Upon completion, `x` contains the solution to the *scaled* linear system. Like some of the other routines discussed in this section, this function is primarily internal; thus for usage information please refer to either Demos/SLIPLU.c or Section 6.4.

5.3.4 `SLIP_permute_x`: permute solution back to original form

```
SLIP_info SLIP_permute_x
(
    mpq_t **x,              // Solution vector
    int32_t n,              // Size of solution vector
    int32_t numRHS,         // number of RHS vectors
    SLIP_LU_analysis *S     // symbolic analysis with the column ordering Q
);
```

`SLIP_permute_x` permutes the solution vector(s) `x` so that they are with respect to the chosen column permutation (that is, this function computes $Q\mathbf{x}$). The function is called upon successful return from `SLIP_LU_solve`.

Like some of the other routines discussed in this section, thus function is primarily internal; thus for usage information please refer to either Demos/SLIPLU.c or Section 6.4.

5.3.5 `SLIP_check_solution`: check if $A_{scaled}x = b_{scaled}$

```
SLIP_info SLIP_check_solution
(
    SLIP_sparse *A,             // input matrix
    mpq_t **x,                  // solution vector
    SLIP_dense *b               // right hand side
);
```

`SLIP_check_solution` checks the solution of the linear system. This function returns either `SLIP_CORRECT` or `SLIP_INCORRECT`.

This function is provided simply for integrity or as troubleshoot code. It is mostly not needed since the algorithm is designed to be exact. To use it correctly, `SLIP_check_solution` must be called before `SLIP_scale_x`. WARNING: `SLIP_check_solution` could return `SLIP_INCORRECT` if it is called after `SLIP_solve_double` (in Section 5.3.9), `SLIP_solve_mpq` (in Section 5.3.10) or `SLIP_solve_mpfr` (in Section 5.3.11).

5.3.6 `SLIP_scale_x`: scale solution with scaling factors of A and b

```
SLIP_info SLIP_scale_x
(
    mpq_t **x,                  // Solution matrix
    SLIP_sparse *A,             // matrix A
    SLIP_dense *b               // right hand side
);
```

`SLIP_scale_x` scales solution vector with scaling factors of A and b. SLIP LU will scale the user's input matrix to ensure everything is integral; thus, once the rational solution vector `x` is obtained, it must be properly scaled so that it is accurate. Again, this is mainly an internal function, thus for usage, please refer to either Demos/SLIPLU.c or Section 6.4.

5.3.7 `SLIP_get_double_soln`: obtain solution in `double` type

```
SLIP_info SLIP_get_double_soln
(
    double **x_doub,          // double soln of size n*numRHS to Ax = b
    mpq_t  **x_mpq,           // mpq solution to Ax = b. x is of size n*numRHS
    int32_t n,                // Dimension of A, number of rows of x
    int32_t numRHS            // Number of right hand side vectors
) ;
```

`SLIP_get_double_soln` converts the `mpq_t**` solution vector obtained from `SLIP_LU_solve` and `SLIP_permute_x` to `double**`. This process introduces round-off error.

On input, `double **x_doub` should be allocated using `SLIP_create_double_mat` in Section 5.4.1.

5.3.8 `SLIP_get_mpfr_soln`: obtain solution in `mpfr_t` type

```
SLIP_info SLIP_get_mpfr_soln
(
    mpfr_t **x_mpfr,       // mpfr solution of size n*numRHS to Ax = b
    mpq_t  **x_mpq,        // mpq solution of size n*numRHS to Ax = b.
    int32_t n,             // Dimension of A, number of rows of x
    int32_t numRHS         // Number of right hand side vectors
);
```

`SLIP_get_mpfr_soln` converts the `mpq_t**` solution vector obtained from `SLIP_LU_solve` and `SLIP_permute_x` to `mpfr_t**`. This process introduces round-off error.

On input, `mpfr_t **x_mpfr` should be allocated using `SLIP_create_mpfr_mat` in Section 5.4.5.

5.3.9 `SLIP_solve_double`: solve $Ax = b$ and return x in `double` type

```
SLIP_info SLIP_solve_double
(
    double **x_doub,          // Solution vector stored as an double
    SLIP_sparse *A,           // Compressed column form full precision matrix A
    SLIP_LU_analysis *S,      // Column ordering
    SLIP_dense *b,            // Right hand side vectrors
    SLIP_options *option      // Control parameters
);
```

`SLIP_solve_double` solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a matrix accurate to double precision. This is a "all-in-one" function that performs factorization, solving, permutation and scaling. However, symbolic analysis `SLIP_LU_analysis` should be performed before calling this function.

On output, this `x_doub` contains the solution to the linear system in double precision and the function returns `SLIP_OK`.

For the full example, users can refer to Demos/example3.c. Here is an brief example of how to use this code:

```
/* Create and populate A, b, and option */
/* A has size of nrows-by-nrows, b has size of nrows-by-numRHS */

SLIP_LU_analysis* S = SLIP_create_LU_analysis(nrows+1);

SLIP_LU_analyze(S, A, option);

double** x = SLIP_create_double_mat(nrows, numRHS);

SLIP_solve_double(x, A, S, b, option);
```

5.3.10 `SLIP_solve_mpq`: solve $Ax = b$ and return x in `mpq_t` type

```
SLIP_info SLIP_solve_mpq
(
    mpq_t **x_mpq,           // Solution vector stored as an mpq_t array
    SLIP_sparse *A,          // Compressed column form full precision matrix A
    SLIP_LU_analysis *S,     // Column ordering
    SLIP_dense *b,           // Right hand side vectrors
    SLIP_options *option     // Control parameters
);
```

`SLIP_solve_mpq` solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a matrix of `mpq_t` numbers. This is a "all-in-one" function that performs factorization, solving, permutation and scaling. However, symbolic analysis `SLIP_LU_analysis` should be performed before calling this function.

On output, this `x_mpq` contains the exact solution to the linear system as `mpq_t` numbers and the function returns `SLIP_OK`

For the full 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 nrows-by-nrows, b has size of nrows-by-numRHS */

SLIP_LU_analysis* S = SLIP_create_LU_analysis(nrows+1);

SLIP_LU_analyze(S, A, option);

mpq_t** x = SLIP_initialize_mpq_mat(nrows, numRHS);

SLIP_solve_mpq(x, A, S, b, option);
```

5.3.11 `SLIP_solve_mpfr`: solve $Ax = b$ and return x in `mpfr_t` type

```
SLIP_info SLIP_solve_mpfr
(
    mpfr_t **x_mpfr,         // Solution vector stored as an mpfr_t array
    SLIP_sparse *A,          // Compressed column form full precision matrix A
    SLIP_LU_analysis *S,     // Column ordering
    SLIP_dense *b,           // Right hand side vectrors
    SLIP_options *option     // Control parameters
);
```

`SLIP_solve_mpq` solves the linear system $A\mathbf{x} = \mathbf{b}$ and returns the solution as a matrix of `mpfr_t` numbers. This is a "all-in-one" function that performs factorization, solving, permutation and scaling. However, symbolic analysis `SLIP_LU_analysis` should be performed before calling this function.

On output, this `x_mpfr` contains the exact solution to the linear system as `mpfr_t` numbers and the function returns `SLIP_OK`

Here is an brief example of how to use this code:

```
/* Create and populate A, b, and option */
/* A has size of nrows-by-nrows, b has size of nrows-by-numRHS */

SLIP_LU_analysis* S = SLIP_create_LU_analysis(nrows+1);

SLIP_LU_analyze(S, A, option);

option->prec = 128; // Quad

mpfr_t** x = SLIP_create_mpfr_mat(nrows, numRHS, option);

SLIP_solve_mpfr(x, A, S, b, option);
```

5.4 Miscellaneous Routines

This section contains miscellaneous routines that may be of interest to the user.

5.4.1 `SLIP_create_double_mat`: create a m-by-n `double` matrix

```
double** SLIP_create_double_mat
(
    int32_t m,      // number of rows
    int32_t n       // number of columns
);
```

`SLIP_create_double_mat` allocates a `double` matrix of size $m \times n$ and sets each entry equal to zero, where $A[i][j]$ is the (i, j)th entry. $A[i]$ is a pointer to row i, of size n.

5.4.2 `SLIP_delete_double_mat`: delete a m-by-n `double` matrix

```
void SLIP_delete_double_mat
(
    double*** A,    // dense matrix
    int32_t m,      // number of rows of A
    int32_t n       // number of columns of A
);
```

`SLIP_delete_double_mat` frees the memory associated with a `double` matrix of size $m \times n$, and set `**A=NULL`.

5.4.3 `SLIP_create_int_mat`: create a m-by-n `int32_t` matrix

```
int32_t** SLIP_create_int_mat
(
    int32_t m,      // number of rows
    int32_t n       // number of columns
);
```

`SLIP_create_int_mat` allocates a `int32_t` matrix of size $m \times n$ and sets each entry equal to zero, where $A[i][j]$ is the (i, j)th entry. $A[i]$ is a pointer to row i, of size n.

5.4.4 `SLIP_delete_int_mat`: delete a m-by-n `int32_t` matrix

```
void SLIP_delete_int_mat
(
    int32_t*** A,   // dense matrix
    int32_t m,      // number of rows
    int32_t n       // number of columns
);
```

`SLIP_delete_int_mat` frees the memory associated with a `int32_t` matrix of size $m \times n$, and set `**A=NULL`.

5.4.5 `SLIP_create_mpfr_mat`: create a m-by-n `mpfr_t` matrix

```
mpfr_t** SLIP_create_mpfr_mat
(
    int32_t m,      // number of rows
    int32_t n,      // number of columns
    SLIP_options *option   // command options containing the prec for mpfr
);
```

`SLIP_create_mpfr_mat` allocates a `mpfr_t` matrix of size $m \times n$ and sets each entry equal to zero, where $A[i][j]$ is the (i, j)th entry. $A[i]$ is a pointer to row i, of size n. The floating point precision associated with each entry is given by option->prec.

5.4.6 `SLIP_delete_mpfr_mat`: delete a m-by-n `mpfr_t` matrix

```
void SLIP_delete_mpfr_mat
(
    mpfr_t ***A,    // Dense mpfr matrix
    int32_t m,      // number of rows of A
    int32_t n       // number of columns of A
);
```

`SLIP_delete_mpfr_mat` frees the memory associated with a `mpfr_t` matrix of size $m \times n$, and set `**A=NULL`. When using the GMP library for `SLIP LU`, it is highly recommended that the user uses the defined `SLIP_delete*` functions in order to avoid memory leaks or potential segmentation faults.

5.4.7 `SLIP_create_mpq_mat`: create a m-by-n `mpq_t` matrix

```
mpq_t** SLIP_create_mpq_mat
(
    int32_t m,       // number of rows
    int32_t n        // number of columns
);
```

`SLIP_create_mpq_mat` allocates a `mpq_t` matrix of size $m \times n$ and sets each entry equal to zero, where $A[i][j]$ is the (i, j)th entry. $A[i]$ is a pointer to row i, of size n.

5.4.8 `SLIP_delete_mpq_mat`: delete a m-by-n `mpq_t` matrix

```
void SLIP_delete_mpq_mat
(
    mpq_t***A,       // dense mpq matrix
    int32_t m,       // number of rows of A
    int32_t n        // number of columns of A
);
```

`SLIP_delete_mpq_mat` frees the memory associated with a `mpq_t` matrix of size $m \times n$, and set `**A=NULL`. When using the GMP library for `SLIP LU`, it is highly recommended that the user uses the defined `SLIP_delete*` functions in order to avoid memory leaks or potential segmentation faults.

5.4.9 `SLIP_create_mpz_mat`: create a m-by-n `mpz_t` matrix

```
mpz_t** SLIP_create_mpz_mat
(
    int32_t m,       // number of rows
    int32_t n        // number of columns
);
```

`SLIP_create_mpz_mat` allocates a `mpz_t` matrix of size $m \times n$ and sets each entry equal to zero, where $A[i][j]$ is the (i, j)th entry. $A[i]$ is a pointer to row i, of size n.

5.4.10 `SLIP_delete_mpz_mat`: delete a m-by-n `mpz_t` matrix

```
void SLIP_delete_mpz_mat
(
    mpz_t ***A,       // The dense mpz matrix
    int32_t m,        // number of rows of A
    int32_t n         // number of columns of A
);
```

`SLIP_delete_mpz_mat` frees the memory associated with a `mpz_t` matrix of size $m \times n$, and set `**A=NULL`. When using the GMP library for `SLIP LU`, it is highly recommended that the user uses the defined `SLIP_delete*` functions in order to avoid memory leaks or potential segmentation faults.

5.4.11 `SLIP_create_mpfr_array`: create a `mpfr_t` of length n

```
mpfr_t* SLIP_create_mpfr_array
(
    int32_t n,       // size of the array
    SLIP_options *option   // command options containing the prec for mpfr
);
```

`SLIP_create_mpfr_array` allocates a `mpfr_t` matrix of length n and sets each entry equal to zero, where $A[i]$ is an entry of type `mpfr_t`. The floating point precision associated with each entry is given by option->prec.

5.4.12 `SLIP_delete_mpfr_array`: delete a `mpfr_t` of length n

```
void SLIP_delete_mpfr_array
(
    mpfr_t** x,      // mpfr array to be deleted
    int32_t n        // size of x
);
```

`SLIP_delete_mpfr_array` frees the memory associated with a `mpfr_t` array of size n, and set `*x=NULL`. When using the GMP library for `SLIP LU`, it is highly recommended that the user uses the defined `SLIP_delete*` functions in order to avoid memory leaks or potential segmentation faults.

5.4.13 `SLIP_create_mpq_array`: create a `mpq_t` of length n

```
mpq_t* SLIP_create_mpq_array
(
    int32_t n        // size of the array
);
```

`SLIP_create_mpq_array` allocates a `mpq_t` matrix of length n and sets each entry equal to zero, where $A[i]$ is an entry of type `mpq_t`.

5.4.14 `SLIP_delete_mpq_array`: delete a `mpq_t` of length n

```
void SLIP_delete_mpq_array
(
    mpq_t** x,       // mpq array to be deleted
    int32_t n        // size of x
);
```

SLIP_delete_mpq_array frees the memory associated with a mpq_t array of size n, and set *x=NULL. When using the GMP library for SLIP LU, it is highly recommended that the user uses the defined SLIP_delete* functions in order to avoid memory leaks or potential segmentation faults.

5.4.15 SLIP_create_mpz_array: create a mpz_t of length n

```
mpz_t* SLIP_create_mpz_array
(
    int32_t n       // Size of x
);
```

SLIP_create_mpz_array allocates a mpz_t matrix of length n and sets each entry equal to zero, where $A[i]$ is an entry of type mpz_t.

5.4.16 SLIP_delete_mpz_array: delete a mpz_t of length n

```
void SLIP_delete_mpz_array
(
    mpz_t ** x,      // mpz array to be deleted
    int32_t n        // Size of x
);
```

SLIP_delete_mpz_array frees the memory associated with a mpz_t array of size n, and set *x=NULL. When using the GMP library for SLIP LU, it is highly recommended that the user uses the defined SLIP_delete* functions in order to avoid memory leaks or potential segmentation faults.

5.4.17 SLIP_spok: check and print a SLIP_sparse matrix

```
SLIP_info SLIP_spok  // returns a SLIP_LU status code
(
    SLIP_sparse *A,      // matrix to check
    int32_t print_level // 0: print nothing, 1: just errors, 2: terse, 3: all
) ;
```

SLIP_spok check the validity of a SLIP_sparse matrix in compressed-sparse column form. Derived from SuiteSparse/MATLAB_TOOLS/spok.

5.5 Miscellaneous Routines2

5.5.1 SLIP_read_matrix

```
int SLIP_read_matrix
(
    SLIP_mat* A,
```

```
    char* filename
);
```

This function reads in an integral matrix stored in matrix market format from the file `filename`. On success, the value `SLIP_OK` is returned. The following code snippet shows how to use this function.

```
SLIP_mat* A = SLIP_initialize_mat();
char* myfile = "myfile.txt";

SLIP_read_matrix(A, myfile);
```

5.5.2 `SLIP_read_rhs`

Thus function's syntax is:

```
int SLIP_read_rhs
(
    mpz_t** b,
    int n,
    char* filename
);
```

This function assumes `b` is $n \times 1$ and reads in the integral right hand side vector from the file `filename`.

6 Using SLIP_LU in C

Using SLIP_LU in C has three steps: 1) initialize and populate data structures, 2) perform symbolic analysis, 3) factorize the matrix A and solve the linear system for each $\mathbf{b}$ vector, and 4) free all used memory and finalize. Steps 1 and 2 are discussed in Subsections 6.1 and 6.2. Factorizing A and solving the linear $A\mathbf{x} = \mathbf{b}$ can be done in one of two ways. If the user is only interested in obtaining the solution vector $\mathbf{x}$, SLIP_LU provides a simple interface for this purpose which is discussed in Section 6.3. Alternatively, if the user wants the actual L and U factors, please refer to Section 6.4. Finally, step 4 is discussed in Section 6.5. 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 $\mathbf{b} \in \mathbb{Z}^{n\times r}$).

6.1 SLIP_LU Initialization and Population of Data Structures

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

6.1.1 Initializing the Environment

SLIP_LU is built upon the GNU GMP library [7] and provides wrappers to all GMP functions in order to regulate memory management and ensure that the user does not have to directly interface with the GMP library. Moreover, SLIP_LU strives to not cause out of memory errors. To ensure all of these precautions, SLIP_LU requires initialization. This can be done by beginning your code with the statement:

```
SLIP_initialize();
```

6.1.2 Initializing Data Structures

SLIP_LU assumes four specific input options for all functions. These are:

- SLIP_sparse* A: A contains the user's input matrix. If the input matrix was already an integral matrix, A is the user's input and A->scale=1. Otherwise, the input matrix is not integral and A contains the user's scaled input matrix.
- 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 4.4.
- SLIP_dense* b: b contains the user's right hand side vector(s). If the input right hand side vectors were already integral, b contains them directly and b->scale=1. Otherwise, b is the scaled input right hand side vector(s).

SLIP_LU provides four separate functions to allocate memory for and initialize each of these data structures. For more details, users can refer to Sections 4.4-4.7. Thus, after calling SLIP_initialize();, the user should call the following:

```
/* Assume SLIP LU environment has been initialized */
```

```
SLIP_sparse * A = SLIP_create_sparse();             // Initialize memory for A
SLIP_LU_analysis* S = SLIP_create_LU_analysis(n+1); // Initialize memory for S and column
                                                    // ordering vector of size n+1
SLIP_options * option = SLIP_create_default_options(); // Initialize memory for option
SLIP_dense * b = SLIP_create_dense();               // Initialize memory for b
```

6.1.3 Populating Data Structures

Of the four aforementioned data structures, `S` is populated during symbolic analysis (Section 6.2), `option` is initialized to default values and can be modified if the user desires (please refer to Section 4.4 for the contents of `option`) and `A` and `b` are populated by the user. `SLIP_LU` allows the input numerical data for `A` and `b` to come in one of 5 options: `int32_t`, `double`, `mpfr_t`, `mpq_t`, and `mpz_t`. Moreover, `A` can be stored in either triplet form or compressed column form. Compressed column 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 $A(i[k], j[k]) = x[k]$.

If the input matrix is stored in compressed column form, the functions `SLIP_build_sparse_ccf_*` can be used. Details of these functions are described in Sections 5.2.1-5.2.5.

The user should use the function that matches the data type of their available `x`. The following code snippet will show how to use these functions. Note that this snippet serves as partially working code (i.e., select the one you'd want to use and delete the surrounding if statements).

```
/* Assume everything has been declared and initialized */

/* Get the matrix A. Assume that everything is stored in
   compressed column form. This means that int* I is the
   set of row indices, int* p are the column pointers, x
   is the array of values, n is the size of the matrix and
   nz is the number of nonzeros in the matrix. We will show
   how to obtain for each possible data type of x (again,
   to have working code, select the one that fits your code
   and delete the rest)  */

if(X IS mpz_t)
{
    SLIP_build_sparse_ccf_mpz(A, p, I, x, n, nz);
}
else if (X IS double)
{
    SLIP_build_sparse_ccf_double(A, p, I, x, n, nz);
}
else if (X IS int32_t)
{
    SLIP_build_sparse_ccf_int(A, p, I, x, n, nz);
}
else if (X IS mpq_t)
{
    SLIP_build_sparse_ccf_mpq(A, p, I, x, n, nz);
}
else if (X IS mpfr_t)
{
```

```
    SLIP_build_sparse_ccf_mpfr(A, p, I, x, n, nz, option);
}
```

Conversely, if the input matrix is stored in triplet form, the functions `SLIP_build_sparse_trip_*` are used. Details of these functions are described in Sections 5.2.6-5.2.10.

The user should use the function that matches the data type of their available `x`. The following code snippet will show how to use these functions. Note that this snippet serves as partially working code (i.e., select the one you'd want to use and delete the surrounding if statements).

```
/* Assume everything has been declared and initialized */

/* Get the matrix A. Assume that everything is stored in
   compressed column form. This means that int* I is the
   set of row indices, int* J is the set of column indices,
   x is the array of values, n is the size of the matrix and
   nz is the number of nonzeros in the matrix. We will show
   how to obtain for each possible data type of x (again,
   to have working code, select the one that fits your code
   and delete the rest)  */

if(X IS mpz_t)
{
    SLIP_build_sparse_trip_mpz(A, I, J, x, n, nz);
}
else if (X IS double)
{
    SLIP_build_sparse_trip_double(A, I, J, x, n, nz);
}
else if (X IS int32_t)
{
    SLIP_build_sparse_trip_int(A, I, J, x, n, nz);
}
else if (X IS mpq_t)
{
    SLIP_build_sparse_trip_mpq(A, I, J, x, n, nz);
}
else if (X IS mpfr_t)
{
    SLIP_build_sparse_trip_mpfr(A, I, J, x, n, nz, option);
}
```

Lastly, the right hand side vectors `b` are populated via the `SLIP_build_dense_*` functions. Details of these functions are described in Sections 5.2.11-5.2.14.

The user should use the function that matches the data type of their available `b`. The following code snippet will show how to use this function. Note that this snippet serves as partially working code (i.e., select the one you'd want to use and delete the surrounding if statements).

```
if (b2 IS mpz_t)
```

```
{
    SLIP_build_dense_mpz(b, b2, n, numRHS);
}
else if (b2 IS double)
{
    SLIP_build_dense_double(b, b2, n, numRHS);
}
else if (b2 IS int32_t)
{
    SLIP_build_dense_int(b, b2, n, numRHS);
}
else if (b2 IS mpq_t)
{
    SLIP_build_dense_mpq(b, b2, n, numRHS);
}
else if (b2 IS mpfr_t)
{
    SLIP_build_dense_mpfr(b, b2, n, numRHS, option);
}
```

6.2 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:

```
/* Assume A has been populated, and option and S have been initialized. */

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

6.3 Easy SLIP_LU for Solving Linear Systems (no L and U)

After initializing the necessary data structures and performing symbolic analysis, SLIP_LU obtains the solution to $A\mathbf{x} = \mathbf{b}$. Using the "easy" interface of SLIP_LU requires only that the user decides what data type that he/she wants **x** to be stored as. SLIP_LU allows **x** to be returned as either double, mpq_t, or mpfr_t with an associated precision. This is done by using one of the following functions: SLIP_solve_double (Section 5.3.9), SLIP_solve_mpq (Section 5.3.10) or SLIP_solve_mpfr (Section 5.3.11).

Below, we show sample syntax to use each of these functions. As above, this code snippet contains all of the potential options, thus a user can merely copy the one they desire and paste into their code.

```
/* Assume that A, S, option, and b have been properly declared and populated. */

if (USER WANTS MPQ)
{
    // The solution is a dense matrix of size n*numRHS
    mpq_t** soln = SLIP_create_mpq_mat(n, numRHS);
    int ok = SLIP_solve_mpq(soln, A, S, b, option);
}
```

```
else if (USER WANTS DOUBLE)
{
    // The solution is a dense matrix of size n*numRHS
    double** soln = SLIP_create_double_mat(n, numRHS);
    int ok = SLIP_solve_double(soln, A, S, b, option);
}
else if (USER WANTS MPFR)
{
    // The solution is a dense matrix of size n*numRHS
    mpfr_t** soln = SLIP_create_mpfr_mat(n, numRHS, option);
    int ok = SLIP_solve_mpfr(soln, A, S, b, option);
}
```

On success, each of these functions return `SLIP_OK` (see Section 4.1).

6.4 Complex `SLIP_LU` for Solving Linear Systems (with L and U)

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 methods described in the previous subsection. Particularly, the following steps must be performed: 1) allocate memory for L, U, the solution vector(s) (stored as `mpq_t`) $\mathbf{x}$, and others, 2) compute the factorization $PAQ = LDU$, 3) solve the linear system $P^{-1}LDUQ^{-1}\mathbf{x} = \mathbf{b}$, 4) permute the solution vector(s), 5) scale the solution vector if the scaling factors of A and b are not zero, and 6) 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.

6.4.1 Allocating New Memory

Using `SLIP_LU` in this form requires that memory be allocated for L, U, and the solution vector(s). The solution vectors are **required** to be stored as a `mpq_t**` array. Additionally, `SLIP_LU` utilizes a `mpz_t` array which stores the pivot elements (referred to as `rhos`) and the inverse row permutation (refereed to as `pinv`). The following code snippet shows how to allocate these entries.

```
/* Purpose: Allocate memory for L, U, and x */

SLIP_sparse* L = SLIP_create_sparse();
SLIP_sparse* U = SLIP_create_sparse();

// x is of size n * numRHS
mpq_t** x = SLIP_create_mpq_mat(n, numRHS);

// rhos is the sequence of pivots
mpz_t* rhos = SLIP_create_mpz_array(n);

// pinv is the inverse row permutation
int32_t* pinv = (int32_t*) SLIP_malloc(n*sizeof(int32_t));
```

6.4.2 Computing the Factorization

The matrices L and U are computed via the `SLIP_LU_factorize` function (Section 5.3.2).

Upon successful completion, this function returns `SLIP_OK`.

6.4.3 Solving the Linear System

After factorization, the next step is to solve the linear system and store the solution as a set of rational number `mpq_t` in the previously allocated `x` data structure. This solution is done via the `SLIP_LU_solve` function (Section 5.3.3).

Upon successful completion, this function returns `SLIP_OK`.

Note: The solution vector given here is NOT the solution to $A\mathbf{x} = \mathbf{b}$ because it has not been properly permuted and scaled. Recall that when solving a system via the SLIP LU factorization, two systems are solved: $LD\mathbf{y} = P\mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$. The solution here is the solution to $Y\mathbf{x} = \mathbf{y}$ and must still be permuted by the column permutation Q which is discussed in the next subsection.

6.4.4 Permuting the Solution Vectors

Permuting the solution vector(s) is done via the function `SLIP_permute_x` (Section 5.3.4).

Upon successful completion, this function returns `SLIP_OK`. At the conclusion of this routine, `x` contains the solution to the scaled system $A_{int}\mathbf{x} = \mathbf{b}_{int}$.

6.4.5 Scaling the Solution Vectors

Scaling the solution vector(s) is done via the function `SLIP_scale_x` (Section 5.3.6).

Upon successful completion, this function returns `SLIP_OK`. At the conclusion of this routine, `x` contains the solution to the system $A\mathbf{x} = \mathbf{b}$.

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

Upon completion of the above routines, the solution to the linear system is given by the `mpq_t** x`. `SLIP_LU` allows this to be converted into either a double precision matrix or a `mpfr_t` precision matrix via the functions `SLIP_get_double_soln` (Section 5.3.7) or `SLIP_get_mpfr_soln` (Section 5.3.8). Below, we show how to call these functions.

```
if (USER WANTS DOUBLE)
{
double** x2 = SLIP_create_double_mat(n, numRHS);
SLIP_get_double_soln(x2, x, n, numRHS);
}
else if (USER WANTS MPFR)
{
```

```
mpfr_t** x2 = SLIP_create_mpfr_mat(n, numRHS, option);
SLIP_get_mpfr_soln(x2, x, n, numRHS);
}
```

6.5 SLIP_LU Freeing all Used Memory

Upon finishing using SLIP_LU all memory must be freed. As described in Sections 5.1 and 5.4, SLIP_LU provides a number of functions to handle this for the user. Below, we briefly summarize which memory freeing routine should be used for specific data types:

- SLIP_sparse*: A SLIP_sparse* A data structure can be freed with a call to SLIP_delete_sparse(&A);
- SLIP_LU_analysis*: A SLIP_LU_analysis* S data structure can be freed with a call to SLIP_delete_LU_analysis(&S);
- SLIP_dense*: The SLIP_dense* b of dimension n * numRHS can be cleared with a call to SLIP_delete_dense(&b).
- 2D array created via SLIP_create_*_mat: The 2D array **x of dimension n * numRHS can be cleared with a call to SLIP_delete_*_mat(&x, n, numRHS).
- 1D array of GMP data type created via SLIP_create_*_array: The 1D array *x of size n can be cleared with a call to SLIP_delete_*_array(&x, n).
- All others including SLIP_options*: These data structures can be freed with a call to the macro SLIP_FREE(), e.g., SLIP_FREE(option) for SLIP_options* option.

Note: after usage of the SLIP_LU routines are finished, one must call SLIP_finalize() (Section 5.1.7) to finalize usage of the library.

6.6 Examples of Using SLIP_LU in a C Program

The "Demo" folder contains six 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_solve_double function is used; thus the output is given as a double matrix.

example2.c This example reads in a matrix stored in integral matrix market 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_solve_mpq function. Thus, the solution is given as a set of rational numbers.

example3.c This example creates an input matrix and right hand side vector stored as mpfr_t numbers. Then, it shows how to create the input matrix A and right hand side vector **b** and solves the linear system using the SLIP_solve_double function, outputting the solution in double precision.

example4.c This example is nearly identical to example3 except that the input has multiple right hand side vectors and all input numbers are stored as double precision numbers.

example5.c This example creates a random set of right hand side vectors, reads in a matrix from a file, and solves the associated linear system outputting the solution as a double matrix.

SLIPLU.c This example reads in a matrix and right hand side vector from a file and solves the linear system $A\mathbf{x} = \mathbf{b}$ using the techniques discussed in Section 6.4. This file also allows command line arguments (discussed in README.txt) and can be used to replicate the results from [8].

7 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 7.1 then we describe how to use the factorization in Section 7.2. 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.

7.1 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) which contains 9 elements. 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.column`: This parameter controls the column ordering used. 0 (default): COLAMD, 1: AMD, 2: no column ordering. It is usually recommended that the user keep this at COLAMD unless they already have a good column permutation.

`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, o.w., smallest pivot, 2: first nonzero pivot in each column, 3 (default): diagonal pivot with a tolerance for the smallest pivot, 4: diagonal pivot with a tolerance 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.int`: Set this parameter equal to 1 if the input matrix is already integral. Otherwise, if the input matrix has any decimal entries, scaling must be performed to obtain an integral input matrix. ****IMPORTANT****If the input matrix is not integral and this parameter is set equal to 1, the values will be truncated.

`option.intb`: Set this parameter equal to 1 if the input right hand side vector(s) are already integral. Like the input matrix, if **b** contains any fractional entries, scaling must be performed to ensure integrality.

`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).

7.2 SLIP_LU.m

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

The SLIP_LU function expects as input a sparse matrix A and dense set of right hand side vectors **b**. Optionally, the user can also pass in the options struct. Currently, there are 6 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.

`[x d] = SLIP_LU(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ and the absolute value of the determinant of A using default settings. In some cases, this method of obtaining the determinant of A is faster than the default MATLAB `d = det(A)` approach.

`[x d] = SLIP_LU(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ and the absolute value of the determinant of A using user defined settings via the options struct.

`[L U P Q x] = SLIP_LU(A,b)` returns lower and upper triangular matrices, L and U, and permutation matrices P and Q such that $LU = PAQ$. In addition this function returns the solution to the system $A\mathbf{x} = \mathbf{b}$. This function uses default parameters the returned L and U matrices are Doolittle factors. If desired, these Doolittle factors can be used to solve $A\mathbf{x} = \mathbf{b}$ as: `x = Q * U \ (L \ (P*b)))`. Note that the $\mathbf{x}$ obtained from using the Doolittle factors is NOT exact to roundoff unlike the $\mathbf{x}$ obtained from the `SLIP_LU` function.

`[L U P Q x] = SLIP_LU(A,b,option)` returns lower and upper triangular matrices, L and U, and permutation matrices P and Q such that $LU = PAQ$. In addition this function returns the solution to the system $A\mathbf{x} = \mathbf{b}$. This function uses user specified parameters the returned L and U matrices are Doolittle factors. If the user wishes their L and U factors to be traditional (i.e. the magnitude of all entries in L are ≤ 1 in absolute value), then set `option.pivot = 5`.

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, W. Hager, and I. Duff, *Suitesparse*, htt p://faculty. cse. tamu. edu/davis/suitesparse. html, (2014).

[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.