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

Version 1.0.0, March 2020

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

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

Contents

1 Summary

SLIP LU is a software package designed to exactly solve unsymmetric sparse linear systems, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{Q}^{n \times n}$, $b \in \mathbb{Q}^{n \times m}$, and $\mathbf{x} \in \mathbb{Q}^{n \times m}$. This package performs a left-looking, roundoff-error-free (REF) LU factorization $PAQ = LDU$, where L and U are integer, D is diagonal, and P and Q are row and column permutations, respectively. It is important to note that the matrix D is never explicitly computed nor needed; thus the functional form of the factorization requires only the matrices L and U. The theory associated with this code is the Sparse Left-looking Integer-Preserving (SLIP) LU factorization [8]. Aside from solving sparse linear systems exactly, one of the key goals of this package is to provide a framework for other solvers to benchmark the reliability and stability of their linear solvers, as our final solution vector $\mathbf{x}$ is guaranteed to be exact. In addition, SLIP LU provides a wrapper class for the GNU Multiple Precision Arithmetic (GMP) [7] and GNU Multiple Precision Floating Point Reliable (MPFR) [6] libraries in order to prevent memory leaks and improve the overall stability of these external libraries. SLIP LU is written in ANSI C and is accompanied by a MATLAB interface.

The user's input matrix A and right hand side (RHS) vectors $\mathbf{b}$ are read from either `double`, `int`, `mpq_t`, `mpz_t`, or `mpfr_t` data types. A must be stored in either compressed sparse column form or sparse triplet form, while $\mathbf{b}$ must be stored as a dense matrix. A discussion of building each of these types of input is given in Section 6.

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

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

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

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

All left-hand side matrices (referred to as A henceforth) within this package are stored in compressed sparse column form (CSC). This data structure stores the matrix A as a sequence of three arrays:

- `A->p`: Column pointers; an array of size `n+1`. The row indices of column j are located in positions `A->p[j]` to `A->p[j+1]-1` of the array `A->i`. Data type: `int32_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: `int32_t`.

- `A->x`: Numeric entries. The entry `A->x[k]` is the numeric value of the kth nonzero in the matrix. Data type: `mpz_t`.

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

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

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

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

2 Availability

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

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

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

3 Installation

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

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

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

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

4 SLIP LU Data Structures

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

- `SLIP_options`: Contains numerous command parameters. Default values of these parameters are good for a general user; however, modifying this struct allows a user to control column orderings, pivoting schemes, and other components of the factorization.
- `SLIP_sparse`: A sparse matrix for SLIP LU. These matrices are stored in the CSC form with `mpz_t` entries.
- `SLIP_dense`: A dense matrix for SLIP LU. Primarily used for the RHS vector(s) $\mathbf{b}$.
- `SLIP_LU_analysis`: A symbolic analysis struct. Contains the column permutation and guesses for the number of nonzeros in L and U.

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

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

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

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

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

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

4.1 `SLIP_info`: status code returned by SLIP LU

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

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

4.2 `SLIP_pivot`: enum for pivoting schemes

There are six available pivoting schemes provided in SLIP LU. Users can set the pivoting method through the `SLIP_options` structure in Section 4.4. Note that the pivot is always nonzero, thus the smallest entry is the nonzero entry with the smallest magnitude. Also, the tolerance is specified by the `tol` component in `SLIP_options`. Please refer to Section 4.4 for details of this parameter. The pivoting schemes are described as follows:

0	`SLIP_SMALLEST`	The k-th pivot is selected as the smallest entry in the kth column.
1	`SLIP_DIAGONAL`	The k-th pivot is selected as the diagonal entry. If the diagonal entry is zero, this method instead selects the smallest pivot in the column.
2	`SLIP_FIRST_NONZERO`	The k-th pivot is selected as the first eligible nonzero in the column.
3	`SLIP_TOL_SMALLEST`	The k-th pivot is selected as the diagonal entry if the diagonal is within a specified tolerance of the smallest entry in the column. Otherwise, the smallest entry in the k-th column is selected. This is the default pivot selection strategy.
4	`SLIP_TOL_LARGEST`	The k-th pivot is selected as the diagonal entry if the diagonal is within a specified tolerance of the largest entry in the column. Otherwise, the largest entry in the k-th column is selected.
5	`SLIP_LARGEST`	The k-th pivot is selected as the largest entry in the k-th column.

4.3 `SLIP_col_order`: enum for column ordering schemes

The SLIP LU library provides three column ordering schemes: no ordering, COLAMD, and AMD. Users can set the column ordering method through `order` component in the `SLIP_option` structure described in Section 4.4. In general, it is recommended that the user selects the COLAMD ordering, however, no preordering can be preferable if the user's matrix already has a good preordering.

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

4.4 `SLIP_options` structure

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

- option->pivot: An enum SLIP_pivot type (discussed in Section 4.2) which controls the type of pivoting used. Default value: SLIP_TOL_SMALLEST (3).

- option->order: An enum SLIP_col_order type (discussed in Section 4.3) which controls what column ordering is used. Default value: SLIP_COLAMD (1).

- option->tol: A double which tells the tolerance used if the user selects a tolerance based pivoting scheme, i.e., SLIP_TOL_SMALLEST or SLIP_TOL_LARGEST. option->tol must be in the range of $(0, 1]$. Default value: 1 meaning that the diagonal entry will be selected if it has the same magnitude as the smallest entry in the k the column.

- option->print_level: An 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 uint64_t which specifies the precision used if the user desires multiple precision floating point numbers, (i.e., MPFR). This can be any integer larger than MPFR_PREC_MIN (value of 1 in MPFR 4.0.2 and 2 in some legacy versions) and smaller than MPFR_PREC_MAX (usually the largest possible int available in your system). Default value: 128 (quad precision).

- option->SLIP_MPFR_ROUND: A mpfr_rnd_t which determines the type of MPFR rounding to be used by SLIP LU. This is a parameter of the MPFR library. The options for this parameter are:

  - MPFR_RNDN: round to nearest (roundTiesToEven in IEEE 754-2008)
  - MPFR_RNDZ: round toward zero (roundTowardZero in IEEE 754-2008)
  - MPFR_RNDU: round toward plus infinity (roundTowardPositive in IEEE 754-2008)
  - MPFR_RNDD: round toward minus infinity (roundTowardNegative in IEEE 754-2008)
  - MPFR_RNDA: round away from zero
  - MPFR_RNDF: faithful rounding. This is not stable.

  By default, SLIP LU utilizes MPFR_RNDN. We refer the reader to the MPFR user guide available at https://www.mpfr.org/mpfr-current/mpfr.pdf for details on the MPFR rounding style and any other utilized MPFR convention.

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	5.8
SLIP_FREE	destroy SLIP_options object	5.4

4.5 The SLIP_sparse structure

All internal sparse matrices are stored in compressed sparse column (CSC) form 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. Internally, this parameter serves as an estimate on the amount of memory needed and is used to reduce the number of intermediate reallocations performed in the library. 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, as shown in the following table.

function name	description	section
SLIP_build_sparse_csc_double	build sparse matrix from double type CSC matrix	6.1
SLIP_build_sparse_csc_int	build sparse matrix from int32_t type CSC matrix	6.2
SLIP_build_sparse_csc_mpq	build sparse matrix from mpq_t type CSC matrix	6.3
SLIP_build_sparse_csc_mpfr	build sparse matrix from mpfr_t type CSC matrix	6.4
SLIP_build_sparse_csc_mpz	build sparse matrix from mpz_t type CSC matrix	6.5
SLIP_build_sparse_trip_double	build sparse matrix from double type triplet-format matrix	6.6
SLIP_build_sparse_trip_int	build sparse matrix from int32_t type triplet-format matrix	6.7
SLIP_build_sparse_trip_mpq	build sparse matrix from mpq_t type triplet-format matrix	6.8
SLIP_build_sparse_trip_mpfr	build sparse matrix from mpfr_t type triplet-format matrix	6.9
SLIP_build_sparse_trip_mpz	build sparse matrix from mpz_t type triplet-format matrix	6.10
SLIP_delete_sparse	destroy sparse matrix	5.9

4.6 The SLIP_dense structure

All internal right-hand side matrices are stored as dense matrices, using the 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: A 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`, described in the following table:

function name	description	section
`SLIP_build_dense_double`	build `SLIP_dense` matrix from a 2D `double` array	6.11
`SLIP_build_dense_int`	build `SLIP_dense` matrix from a 2D `int32_t` array	6.12
`SLIP_build_dense_mpq`	build `SLIP_dense` matrix from a 2D `mpq_t` array	6.13
`SLIP_build_dense_mpfr`	build `SLIP_dense` matrix from a 2D `mpfr_t` array	6.14
`SLIP_build_dense_mpz`	build `SLIP_dense` matrix from a 2D `mpz_t` array	6.15
`SLIP_delete_dense`	destroy `SLIP_dense` matrix	5.10

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` structure 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 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 `int32_t` which is a guess for the number of nonzeros in L. `S->lnz` must be in the range of $[n, n^2]$. If `S->lnz` is too small, the program may waste time performing extra memory reallocations. This is set during the symbolic analysis.

- `S->unz`: An `int32_t` which is a guess for the number of nonzeros in U. `S->unz` must be in the range of $[n, n^2$. If `S->unz` is too small, the program may waste time performing extra memory reallocations. This is set during the symbolic analysis.

The SLIP LU package provides the following functions to create and destroy a `SLIP_LU_analysis` object:

function/macro name	description	section
`SLIP_LU_analyze`	create `SLIP_LU_analysis` object	7.1
`SLIP_delete_LU_analysis`	destroy `SLIP_LU_analysis` object	5.11

5 Memory Management Routines

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

5.1 SLIP_calloc: allocate initialized memory

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

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

5.2 SLIP_malloc: allocate uninitialized memory

```
void * SLIP_malloc
(
    size_t size         // Size 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. If the size is zero, it is treated as if equal to 1. If the function fails to allocate the requested block of memory, then a NULL pointer is returned.

5.3 SLIP_realloc: resize allocated memory

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

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.

5.4 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 provided, which sets the free'd pointer to NULL.

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

5.5 SLIP_initialize: initialize the working environment

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

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

5.6 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 identical parameters as the 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 of SLIP LU's GMP interface 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 5.1-5.4), or (if used inside MATLAB), for the MATLAB `mxMalloc`, `mxCalloc`, `mxRealloc`, and `mxFree` functions.

5.7 `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.

5.8 `SLIP_create_default_options`: create default `SLIP_option` object

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

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

5.9 `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`, which is then set to `NULL`.

5.10 SLIP_delete_dense: delete SLIP_dense matrix

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

SLIP_delete_dense deletes the SLIP_dense matrix A, which is then set to NULL.

5.11 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.

6 Matrix Building Routines

The routines in this section are used to build either a SLIP_sparse matrix or a SLIP_dense matrix.

6.1 SLIP_build_sparse_csc_double: build sparse matrix using CSC with double entries

```
SLIP_info SLIP_build_sparse_csc_double
(
    SLIP_sparse **A_handle,     // output matrix
    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_csc_double` builds a sparse matrix using compressed sparse column (CSC) form inputs, where the entry values are `double` type.

6.2 `SLIP_build_sparse_csc_int`: build sparse matrix using CSC with `int32_t` entries

```
SLIP_info SLIP_build_sparse_csc_int
(
    SLIP_sparse **A_handle,     // output matrix
    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_csc_int` builds a sparse matrix using compressed column form inputs, where the entry values are `int32_t` type.

6.3 `SLIP_build_sparse_csc_mpq`: build sparse matrix using CSC with `mpq_t` entries

```
SLIP_info SLIP_build_sparse_csc_mpq
(
    SLIP_sparse **A_handle,     // output matrix
    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_csc_mpq` builds a sparse matrix using compressed sparse column (CSC) form inputs, where the entry values are `mpq_t` type.

6.4 `SLIP_build_sparse_csc_mpfr`: build sparse matrix using CSC with `mpfr_t` entries

```
SLIP_info SLIP_build_sparse_csc_mpfr
(
    SLIP_sparse **A_handle,     // output matrix
```

```
        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_csc_mpfr builds a sparse matrix using compressed sparse column (CSC) form inputs, where the entry values are mpfr_t type.

6.5 SLIP_build_sparse_csc_mpz: build sparse matrix using CSC with mpz_t entries

```
    SLIP_info SLIP_build_sparse_csc_mpz
    (
        SLIP_sparse **A_handle,     // output matrix
        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_csc_mpz builds a sparse matrix using compressed column form inputs, where the entry values are mpz_t type.

6.6 SLIP_build_sparse_trip_double: build sparse matrix using triplet with double entries

```
    SLIP_info SLIP_build_sparse_trip_double
    (
        SLIP_sparse **A_handle,     // output matrix
        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)
    ) ;
```

SLIP_build_sparse_trip_double builds a sparse matrix using triplet form inputs, where the entry values are double type.

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

```
SLIP_info SLIP_build_sparse_trip_int
(
    SLIP_sparse **A_handle,     // output matrix
    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.

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

```
SLIP_info SLIP_build_sparse_trip_mpq
(
    SLIP_sparse **A_handle,     // output matrix
    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)
) ;
```

`SLIP_build_sparse_trip_mpq` builds a sparse matrix using triplet form inputs, where the entry values are `mpq_t` type.

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

```
SLIP_info SLIP_build_sparse_trip_mpfr
(
    SLIP_sparse **A_handle,     // output matrix
    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)
        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.

6.10 SLIP_build_sparse_trip_mpz: build sparse matrix using triplet with mpz_t entries

```
    SLIP_info SLIP_build_sparse_trip_mpz
    (
        SLIP_sparse **A_handle,     // output matrix
        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)
    ) ;
```

SLIP_build_sparse_trip_mpz builds a sparse matrix using triplet form inputs, where the entry values are mpz_t type.

6.11 SLIP_build_dense_double: build SLIP_dense matrix from a 2D double array

```
    SLIP_info SLIP_build_dense_double
    (
        SLIP_dense **C_handle,      // Dense matrix to construct
        // inputs, not modified:
        double **B,                 // Set of values as doubles
        int32_t m,                  // number of rows
        int32_t n,                  // number of columns
        SLIP_options* option
    ) ;
```

SLIP_build_dense_double builds a SLIP_dense matrix C from a 2D double array B.

6.12 SLIP_build_dense_int: build SLIP_dense matrix from a 2D int32_t array

```
SLIP_info SLIP_build_dense_int
(
    SLIP_dense **C_handle,      // Dense matrix to construct
    // inputs, not modified:
    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 SLIP_dense matrix C from a 2D int32_t array B.

6.13 SLIP_build_dense_mpq: build SLIP_dense matrix from a 2D mpq_t array

```
SLIP_info SLIP_build_dense_mpq
(
    SLIP_dense **C_handle,      // Dense matrix to construct
    // inputs, not modified:
    mpq_t **B,                  // set of values as mpq_t
    int32_t m,                  // number of rows
    int32_t n                   // number of columns
) ;
```

SLIP_build_dense_mps builds a SLIP_dense matrix C from a 2D mpq_t array B.

6.14 SLIP_build_dense_mpfr: build SLIP_dense matrix from a 2D mpfr_t array

```
SLIP_info SLIP_build_dense_mpfr
(
    SLIP_dense **C_handle,      // Dense matrix to construct
    // inputs, not modified:
    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 with precision for mpfr
) ;
```

SLIP_build_dense_mpfr builds a dense matrix using 2D mpfr_t array.

6.15 SLIP_build_dense_mpz: build SLIP_dense matrix from a 2D mpz_t array

```
SLIP_info SLIP_build_dense_mpz
(
    SLIP_dense **C_handle,      // Dense matrix to construct
    // inputs, not modified:
    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_mps builds a SLIP_dense matrix C from a 2D mpz_t array B.

7 Primary Computational Routines

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

7.1 SLIP_LU_analyze: perform symbolic analysis

```
SLIP_info SLIP_LU_analyze
(
    SLIP_LU_analysis **S, // symbolic analysis
    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.

The SLIP_LU_analysis object is created by this function, and the value of S is ignored on input. On output, S is a pointer to the newly created symbolic analysis object, or NULL if a failure occurred.

The analysis S is freed by SLIP_delete_LU_analysis.

7.2 SLIP_LU_factorize: perform LU factorization

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

`SLIP_LU_factorize` performs the SLIP LU factorization. This factorization is done via n (number of rows or columns of A) iterations of the sparse REF triangular solve function. The overall factorization is $PAQ = LDU$. This routine allows the user to separate factorization and solve. For example codes, please refer to either `Demos/SLIPLU.c` or Section 10.4.

On input, `L`, `U`, `rhos`, and `pinv` are undefined.

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, `pinv[i]` gives the row index in PA).

If an error occurs, `L`, `U`, `rhos`, and `pinv` are all returned as `NULL`.

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

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

`SLIP_LU_solve` obtains the solution to the scaled linear system $LDUx = b$ 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, `mpq_t **x` should be allocated as a 2D array of same size as `b` using `SLIP_create_mpq_mat` (see Section 8.7).

Upon completion, `x` contains the solution to the *scaled* linear system. Like some of some other routines discussed in this section, this function is primarily for advanced users who might want intermediate calculation results; thus for usage information please refer to either `Demos/SLIPLU.c` or Section 10.4.

7.4 `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`.

7.5 `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 7.9), `SLIP_solve_mpq` (in Section 7.10) or `SLIP_solve_mpfr` (in Section 7.11).

7.6 `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 integer; thus, once the rational solution vector `x` is obtained, it must be properly scaled so that it is accurate. Again, this is mainly for advanced users with needs for intermediate calculation results, thus for usage, please refer to either `Demos/SLIPLU.c` or Section 10.4.

7.7 `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 8.1.

7.8 `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 8.5.

7.9 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,           // CSC 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 function performs factorization, solving, permutation and scaling. It must be preceded by a call to the SLIP_LU_analysis function, which constructs the symbolic analysis object S.

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

For a complete 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 n-by-n, b has size of n-by-numRHS */
SLIP_LU_analysis *S ;
SLIP_LU_analyze(&S, A, option) ;
double** x = SLIP_create_double_mat(n, numRHS) ;
SLIP_solve_double(x, A, S, b, option) ;
```

7.10 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,           // CSC 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 function performs factorization, solving, permutation and scaling. It must be preceded by a call to the SLIP_LU_analysis function, which constructs the symbolic analysis object S.

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

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

```
/* Create and populate A, b, and option */
/* A has size of n-by-n, b has size of n-by-numRHS */
SLIP_LU_analysis *S ;
SLIP_LU_analyze(&S, A, option) ;
mpq_t** x = SLIP_initialize_mpq_mat(n, numRHS) ;
SLIP_solve_mpq(x, A, S, b, option) ;
```

7.11 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,         // CSC 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 function performs factorization, solving, permutation and scaling. It must be preceded by a call to the SLIP_LU_analysis function, which constructs the symbolic analysis object S.

On output, 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 n-by-n, b has size of n-by-numRHS */
```

```
SLIP_LU_analysis *S ;
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) ;
```

8 Miscellaneous Routines (TODO rename this)

This section contains miscellaneous routines that may be of interest to the user. **TODO: this is a confusing name for this section. And "may be of interest" is misleading, since functions like `SLIP_create_double_mat` is used in many places.**

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

```
double** SLIP_create_double_mat
(
    int32_t m,     // number of rows (must be > 0)
    int32_t n      // number of columns (must be > 0)
) ;
```

`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. `NULL` is returned if $m \leq 0$ or $n \leq 0$ or out of memory.

8.2 `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 sets `**A=NULL`.

```
TODO: any time a code needs a triple star pointer, something is wrong.
This data structure should be redesigned. Also for integer matrix.
        double*** A,   // dense matrix
```

8.3 SLIP_create_int_mat: create a m-by-n int32_t matrix

```
int32_t** SLIP_create_int_mat
(
    int32_t m,     // number of rows (must be > 0)
    int32_t n      // number of columns (must be > 0)
) ;
```

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. NULL is returned if $m \leq 0$ or $n \leq 0$ or out of memory.

8.4 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 sets **A=NULL.

8.5 SLIP_create_mpfr_mat: create a m-by-n mpfr_t matrix

```
mpfr_t** SLIP_create_mpfr_mat
(
    int32_t m,     // number of rows (must be > 0)
    int32_t n,     // number of columns (must be > 0)
    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. NULL is returned if $m \leq 0$ or $n \leq 0$ or out of memory.

8.6 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
) ;
```

```
TODO: any time a code needs a triple star pointer, something is wrong.
This data structure is terribly confusion:
        mpfr_t ***A,   // Dense mpfr matrix
```

`SLIP_delete_mpfr_mat` frees the memory associated with a `mpfr_t` matrix of size $m \times n$, and sets `**A=NULL`.

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

```
mpq_t** SLIP_create_mpq_mat
(
    int32_t m,     // number of rows (must be > 0)
    int32_t n      // number of columns (must be > 0)
) ;
```

`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. `NULL` is returned if $m \leq 0$ or $n \leq 0$ or out of memory.

8.8 `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 sets `**A=NULL`.

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

```
mpz_t** SLIP_create_mpz_mat
(
    int32_t m,     // number of rows (must be > 0)
    int32_t n      // number of columns (must be > 0)
) ;
```

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. NULL is returned if $m \leq 0$ or $n \leq 0$ or out of memory.

8.10 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 sets **A=NULL.

8.11 SLIP_create_mpfr_array: create a mpfr_t of length n

```
mpfr_t* SLIP_create_mpfr_array
(
    int32_t n,     // size of the array (must be > 0)
    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. NULL is returned if $n \leq 0$ or out of memory.

8.12 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 sets *x=NULL.

8.13 SLIP_create_mpq_array: create a mpq_t of length n

```
    mpq_t* SLIP_create_mpq_array
    (
        int32_t n      // size of the array (must be > 0)
    ) ;
```

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. NULL is returned if $n \leq 0$ or out of memory.

8.14 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 sets *x=NULL.

8.15 SLIP_create_mpz_array: create a mpz_t of length n

```
    mpz_t* SLIP_create_mpz_array
    (
        int32_t n      // Size of x (must be > 0)
    ) ;
```

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. NULL is returned if $n \leq 0$ or out of memory.

8.16 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 sets *x=NULL.

8.17 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.

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, please refer to SLIP_LU/Source/SLIP_gmp.c.

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

Given a GMP function void gmpfunc(TYPEa a, TYPEb b, ...), where TYPEa and TYPEb can be GMP type data (mpz_t, mpq_t and mpfr_t, for example) or non-

GMP type data (`int`, `double`, for example), and they need not to be the same. In order to apply our wrapper to a new function, one can create it as follows:

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

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

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

Note that, other than `SLIP_mpfr_fprintf`, `SLIP_gmp_fprintf`, `SLIP_gmp_printf` and `SLIP_gmp_fscanf`, all of the wrapped GMP/MPFR functions always return `SLIP_info` to the caller. Therefore, for some GMP/MPFR functions that have their own return value. For example, fo `int mpq_cmp(const mpq_t a, const mpq_t b)`, the return value becomes a parameter of the wrapped function. In general, a GMP/MPFR function in the form of `TYPEr gmpfunc(TYPEa a, TYPEb b, ...)`, users can create the wrapped function as follows:

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

```
{
    // Start the GMP Wrappter
    // uncomment one of the followings that meets the needs
    //SLIP_GMP_WRAPPER_START;
    //SLIP_GMPZ_WRAPPER_START(a) ;
    //SLIP_GMPQ_WRAPPER_START(a) ;
    //SLIP_GMPFR_WRAPPER_START(a) ;

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

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

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

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

10 Using SLIP LU in C

Using SLIP LU in C has four 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 10.1 and 10.2. 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 10.3. Alternatively, if the user wants the actual L and U factors, please refer to Section 10.4. Finally, step 4 is discussed in Section 10.5. 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 $\mathbf{b} \in \mathbb{Z}^{n \times r}$).

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 GMP library [7] and provides wrappers to all GMP functions it uses. This allows SLIP LU to properly handle memory management failures, which GMP does not handle. It may also allow the user to not need any direct access to the GMP library. To enable this mechanism, SLIP LU requires initialization. The following must be done before using any other SLIP LU function:

```
SLIP_initialize ( ) ;
```

10.1.2 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 integer matrix, `A` is the user's input and `A->scale`=1. Otherwise, the input matrix is not integer and `A` contains the user's scaled input matrix.

- `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 integer, `b` contains them directly and `b->scale`=1. Otherwise, `b` is the scaled input right hand side vector(s).

10.1.3 Populating Data Structures

Of the four data structures discussed in Section 10.1.2, `S` is constructed during symbolic analysis (Section 10.2), `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. TODO: no, we have functions that create A and b.

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_csc_*` can be used. Details of these functions are described in Sections 6.1-6.5.

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).

```
/* 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_csc_mpz(&A, p, I, x, n, nz) ;
}
else if (X IS double)
{
    SLIP_build_sparse_csc_double(&A, p, I, x, n, nz) ;
}
else if (X IS int32_t)
{
    SLIP_build_sparse_csc_int(&A, p, I, x, n, nz) ;
}
else if (X IS mpq_t)
{
    SLIP_build_sparse_csc_mpq(&A, p, I, x, n, nz) ;
}
else if (X IS mpfr_t)
{
    SLIP_build_sparse_csc_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 6.6-6.10.

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).

```
/* 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 6.11-6.14.

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) ;
}
```

10.2 SLIP LU Symbolic Analysis

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

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

10.3 Simple SLIP LU Routines for Solving Linear Systems

After initializing the necessary data structures and performing symbolic analysis, SLIP LU obtains the solution to $A\mathbf{x} = \mathbf{b}$. Using the "simple" interface of SLIP LU requires only that the user decides what data type that he/she wants $\mathbf{x}$ to be stored as. SLIP LU allows $\mathbf{x}$ 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 7.9), `SLIP_solve_mpq` (Section 7.10) or `SLIP_solve_mpfr` (Section 7.11).

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.

```
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).

10.4 Expert SLIP LU Routines

If a user wishes to perform the SLIP LU factorization of the matrix A while capturing information about the factorization itself and solving the linear system, extra steps must be performed that are all done internally in the 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.

10.4.1 Allocating Memory

Using SLIP LU in this form requires that memory be allocated for the solution vector(s). The solution vectors are stored as an `mpq_t**` array. The following code snippet shows how to allocate the solution vector(s).

```
// Allocate memory for x, of size n-by-numRHS
mpq_t** x = SLIP_create_mpq_mat(n, numRHS) ;
```

10.4.2 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 7.2). Upon successful completion, this function returns `SLIP_OK`.

10.4.3 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 7.3).

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.

10.4.4 Permuting the Solution Vectors

Permuting the solution vector(s) is done via the function `SLIP_permute_x` (Section 7.4).

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

10.4.5 Scaling the Solution Vectors

Scaling the solution vector(s) is done via the function `SLIP_scale_x` (Section 7.6).

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

10.4.6 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 7.7) or `SLIP_get_mpfr_soln` (Section 7.8). 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) ;
    }
```

10.5 SLIP LU Freeing all Used Memory

Upon finishing using SLIP LU all memory must be freed. As described in Sections 5 and 8, 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`-by-`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()`, for example, `SLIP_FREE(option)` for `SLIP_options* option`.

Note: after usage of the SLIP LU routines are finished, one must call `SLIP_finalize()` (Section 5.7) to finalize usage of the library.

10.6 Examples of Using SLIP LU in a C Program

The `SLIP_LU/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 $\mathbf{b}$ 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 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_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 $\mathbf{b}$ 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 10.4. This file also allows command line arguments (discussed in README.txt) 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_LU.m` and the `SLIP_get_options` functions. First, this section will describe the `SLIP_get_options` struct in Section 11.1 then we describe how to use the factorization in Section 11.2. 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.

11.1 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, otherwise 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).

11.2 `SLIP_LU.m`

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

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

- `x = SLIP_LU(A,b)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using default settings. The solution vectors are more accurate than the solution obtained via `x = A \ b`.
- `x = SLIP_LU(A,b,option)` returns the solution to $A\mathbf{x} = \mathbf{b}$ using user specified settings from the options struct.

References

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

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

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

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

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

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

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

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