

# User Guide for NICSLU

Xiaoming Chen

chenxm05@mails.tsinghua.edu.cn

Nano Integrated Circuit and System (NICS) Laboratory

Tsinghua National Laboratory for Information Science and Technology

Department of Electronic Engineering, Tsinghua University

April 16, 2013

## Contents

<b>1</b>	<b>License</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>Matrix Format</b>	<b>4</b>
<b>4</b>	<b>Using NICSLU in a C/C++ Program</b>	<b>5</b>
4.1	Data Types Used in NICSLU . . . . .	5
4.2	The SNicsLU Structure . . . . .	6
4.2.1	Readable Members . . . . .	6
4.2.2	Writable Members . . . . .	7
4.3	Function Return Values . . . . .	8
4.4	NICSLU Routines . . . . .	9
4.4.1	NicsLU_Initialize . . . . .	9
4.4.2	NicsLU_Destroy . . . . .	9
4.4.3	NicsLU_CreateMatrix . . . . .	9
4.4.4	NicsLU_DestroyMatrix . . . . .	10
4.4.5	NicsLU_CreateThreads . . . . .	10
4.4.6	NicsLU_DestroyThreads . . . . .	10
4.4.7	NicsLU_CreateScheduler . . . . .	10
4.4.8	NicsLU_DestroyScheduler . . . . .	10
4.4.9	NicsLU_Analyze . . . . .	11
4.4.10	NicsLU_Factorize . . . . .	11
4.4.11	NicsLU_ReFactorize . . . . .	11
4.4.12	NicsLU_Factorize_MT . . . . .	11
4.4.13	NicsLU_ReFactorize_MT . . . . .	11

4.4.14	NicsLU_Solve . . . . .	11
4.4.15	NicsLU_SolveFast . . . . .	12
4.4.16	NicsLU_ResetMatrixValues . . . . .	12
4.4.17	NicsLU_Refine . . . . .	12
4.4.18	NicsLU_Residual . . . . .	12
4.4.19	NicsLU_Throughput . . . . .	13
4.4.20	NicsLU_Flops . . . . .	13
4.4.21	NicsLU_ThreadLoad . . . . .	13
4.4.22	NicsLU_Transpose . . . . .	13
4.4.23	NicsLU_DumpA . . . . .	14
4.4.24	NicsLU_DumpLU . . . . .	14
4.4.25	NicsLU_ConditionNumber . . . . .	15

## 5 History 15

## 1 License

NICSLU, Copyright©2011-2013 Tsinghua University. All Rights Reserved.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

## 2 Introduction

NICSLU is a high-performance and robust software package for solving large-scale sparse linear systems of equations ( $Ax = b$ ) on shared-memory machines. It is written by C, and can be easily used in C/C++ programs.

NICSLU solves  $Ax = b$  by Gaussian elimination method (LU factorization). It factorizes matrix  $A$  into product of a lower triangular matrix  $L$  and an upper triangular matrix  $U$  (i.e.  $A = LU$ , numerical factorization step), and then the solution of  $Ax = b$  is obtained by solving two triangular equations  $Ly = b$  and  $Ux = y$  (right-hand-solving step). Matrix  $A$  doesn't need to be symmetric or definite, but it must be square and full-rank, otherwise NICSLU cannot solve it.

Generally speaking, a simple description of sparse Gaussian elimination is as follows. Matrix  $A$  is factorized to:

$$LM_{n-1}R_{n-1} \cdots M_1R_1 = PD_rAD_cQ$$

where  $n$  is the dimension of  $A$ ;  $D_r$  and  $D_c$  are two diagonal matrices to scale  $A$  to enhance numerical stability;  $P$  and  $Q$  are row and column permutation matrices, which are used to maintain sparsity (i.e. reduce fill-ins);  $R_k$  is the column permutation matrix generated by partial pivoting that occurs at step  $k$  during numerical factorization;  $M_k$  is an upper triangular matrix whose  $k$ th row contains the multipliers. So  $Ax = b$  can be solved by:

$$\begin{aligned} x &= A^{-1}b \\ &= (D_r^{-1}P^{-1}LM_{n-1}R_{n-1} \cdots M_1R_1Q^{-1}D_c^{-1})^{-1}b \\ &= D_cQR_1^{-1}M_1^{-1} \cdots R_{n-1}^{-1}M_{n-1}^{-1}L^{-1}PD_rb \end{aligned}$$

NICSLU is based on the sparse left-looking algorithm proposed by Gilbert and Peierls [1], and KLU algorithm proposed by Davis [2]. We use a more efficient static pivoting algorithm (HSL\_MC64 algorithm) [3,4], which is combined with partial pivoting to achieve

higher numerical stability. We have developed a novel parallel algorithm, which obtains effective acceleration on shared-memory multi-core processors [5–7].

There are also some other similar software packages, such as SuperLU [8–10], PAR-DISO [11], etc. NICSLU is different from these software packages. NICSLU is well suited for extremely sparse matrices, such as the matrices in circuit simulation problems.

Currently NICSLU can be executed on Intel x86 or AMD64 (x86-64) hardware platforms, both Windows and Linux are supported.

**If you are using NICSLU in your research, please cite the following three papers:**

- [1] Xiaoming Chen, Wei Wu, Yu Wang, Hao Yu, Huazhong Yang, “An EScheduler-based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation”, Circuits and Systems II: Express Briefs, IEEE Transactions on, vol. 58, no. 10, pp. 702-706, oct. 2011.
- [2] Xiaoming Chen, Yu Wang, Huazhong Yang, “NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation”, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 32, no. 2, pp. 261-274, feb. 2013.
- [3] Xiaoming Chen, Yu Wang, Huazhong Yang, “An Adaptive LU Factorization Algorithm for Parallel Circuit Simulation”, Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, pp.359-364, Jan. 30, 2012-Feb. 2, 2012.

### 3 Matrix Format

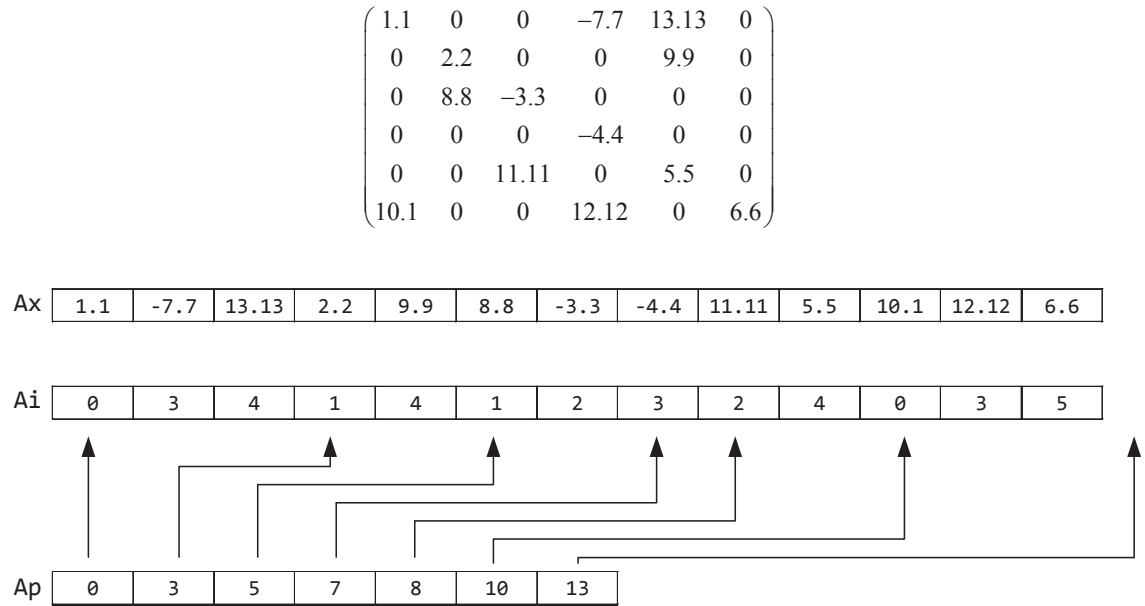


Figure 1: Example to illustrate the CSR format.

NICSLU uses the compressed sparse row (CSR) format to store a sparse matrix, as illustrated in Fig. 1 illustrates. CSR uses five parameters to describe a sparse matrix, as listed in the below.

- **n**: (unsigned) integer, matrix dimension, i.e. the matrix is  $n \times n$ . NICSLU only supports square matrices.
- **nnz**: (unsigned) integer, the number of nonzeros in the matrix.
- **Ai**: (unsigned) integer array of length **nnz**, storing the column indices of all nonzeros.
- **Ax**: floating-point array of length **nnz**, storing the values of all nonzeros.
- **Ap**: (unsigned) integer array of length **n+1**, storing the location of the first nonzeros of each row in **Ai** and **Ax**. The first and last elements must be **Ap[0]=0** and **Ap[n]=nnz**. Values of the  $i$ th row of the matrix are stored in **Ax[Ap[i]]**, **Ax[Ap[i]+1]**,  $\dots$ , **Ax[Ap[i+1]-1]**, and the corresponding column indices of the nonzeros are stored in **Ai[Ap[i]]**, **Ai[Ap[i]+1]**,  $\dots$ , **Ai[Ap[i+1]-1]**, number of nonzeros of the  $i$ th row is **Ap[i+1]-Ap[i]**. The matrix is zero-based stored, which means the row and column indices are in the range from 0 to **n-1**.

The transposed format of CSR is compressed sparse column (CSC), which is stored in column-major.

## 4 Using NICSLU in a C/C++ Program

### 4.1 Data Types Used in NICSLU

NICSLU uses several self-defined data types, as listed in Table 1, in which the first column lists the data types used in NICSLU, and the second column lists the corresponding data types in standard C. The detailed definitions of the data types can be found in `nics_config.h`.

Table 1: Data types used in NICSLU.

data type	C type	meaning
<code>int__t</code>	<code>int</code>	32-bit integer
<code>uint__t</code>	<code>unsigned int</code>	32-bit unsigned integer
<code>real__t</code>	<code>double</code>	double-precision floating-point
<code>bool__t</code>	<code>unsigned char</code>	boolean value: TRUE or FALSE
<code>size_t/size__t</code>	<code>size_t</code>	32-bit or 64-bit* unsigned long integer
<code>byte__t</code>	<code>unsigned char</code>	byte, 8-bit

\* According to the hardware platform or the compiling configurations you used.

## 4.2 The SNicsLU Structure

The **SNicsLU** structure contains all the configurations, matrix data, LU factors, and statistical information for LU factorization in NICS LU. This object appears in most of NICS LU functions as the first parameter. Details of **SNicsLU** are given in **nicslu.h**. Only a few member parameters of **SNicsLU** can be read or written by users, which are listed below.

### 4.2.1 Readable Members

All the members in floating-point array **stat** are readable, and the meanings of each indexed member is as follows.

- **real\_t stat[0]**: analysis time, runtime (in seconds) of **NicsLU\_Analyze** function.
- **real\_t stat[1]**: factorization time, runtime (in seconds) of **NicsLU\_Factorize** or **NicsLU\_Factorize\_MT** function (according to your last calling).
- **real\_t stat[2]**: re-factorization time, runtime (in seconds) of **NicsLU\_ReFactorize** or **NicsLU\_ReFactorize\_MT** function (according to your last calling).
- **real\_t stat[3]**: right-hand-solving time, runtime (in seconds) of **NicsLU\_Solve** or **NicsLU\_SolveFast** function (according to your last calling).
- **real\_t stat[4]**: initialization time of the scheduler, runtime (in seconds) of **NicsLU\_CreateScheduler** function.
- **real\_t stat[5]**: total number of floating-point operations (FLOPs) to factorize the matrix, which is generated by **NicsLU\_Flops** function.
- **real\_t stat[6]**: condition number of the matrix, which is estimated by **NicsLU\_ConditionNumber** function. If MC64 scaling is used, the condition number is estimated after MC64 scaling to the matrix.
- **real\_t stat[7]**: estimated speedup if all the cores of the CPU are used, which is calculated by **NicsLU\_CreateScheduler** function.
- **real\_t stat[8]**: estimated upper bound of speedup attainable by NICS LU, regardless of the number of cores, which is calculated by **NicsLU\_CreateScheduler** function.
- **real\_t stat[9]**: number of cores on the computer. If super-threading is supported and enabled, **stat[9]** is twice of the number of physical cores.
- **real\_t stat[10]**: estimated number of FLOPs to factorize the matrix, which is calculated by **NicsLU\_CreateScheduler** function.

- `real__t stat[11]`: estimated  $NNZ(L+U)$ , which is calculated by `NicsLU_CreateScheduler` function.
- `real__t stat[12]`: estimated memory throughput (in bytes), which is calculated by `NicsLU_Throughput` function.
- `real__t stat[13]`: a suggestion. Non-zero suggests using `NicsLU_Factorize` function and zero suggests using `NicsLU_Factorize_MT` function. The suggestion is generated by `NicsLU_CreateScheduler` function.
- `real__t stat[14]`: number of off-diagonal pivots.
- `real__t stat[15]`: refinement time, runtime (in seconds) of `NicsLU_Refine`.
- `real__t stat[16]`: number of iterations of `NicsLU_Refine`.

Besides the above members in `stat` array, the following members are also readable.

- `size_t l_nnz, u_nnz`: the two members indicate the number of nonzeros in  $L$  and  $U$  after factorization, including the diagonals of  $L$  and  $U$  respectively.
- `size_t lu_nnz`: number of nonzeros in  $L + U - I$  after factorization, which is equal to `l_nnz + u_nnz - n`.

#### 4.2.2 Writable Members

All the writable members are in unsigned integer array `cfgi` and floating-point array `cfgf`.

- `uint__t cfgi[0]`: default value is 0. A flag to indicate the CSR or CSC mode. Zero indicates CSR and non-zero indicates CSC. If your matrix is stored in CSC format, NICSLU can also directly deal with it. In this case, NICSLU solves  $A^T x = b$ .
- `uint__t cfgi[1]`: default value is 1. A flag to indicate whether using MC64 to scale the matrix before factorization. MC64 scaling is strongly recommended.
- `uint__t cfgi[2]`: default value is 0. A flag to indicate the scaling method when factorizing the matrix. 1 indicates max-scaling, 2 indicates sum-scaling and other values indicate no scaling. Based on our experiments, the scaling methods may have effect in frequency-domain simulation, but they generally have no effect in time-domain transient simulation.
- `uint__t cfgi[3]`: default value is 10. It is a scheduling threshold for parallel LU factorization. It should be larger than or equal to the number of threads.
- `uint__t cfgi[4]`: default value is 2. It is used to pre-allocate memory for parallel LU factorization. If it is larger, NICSLU will use more memory, but during parallel LU factorization, less memory re-allocation will happen.

- `uint_t cfgi[7]`: default value is the number of created threads. This number indicates the actual number of threads used for parallel computation. You can set it before `NicsLU_Factorize_MT` or `NicsLU_ReFactorize_MT`. It cannot exceed the number of created threads.
- `real_t cfgf[0]`: default value is 0.001. It is the partial pivoting tolerance. If the diagonal entry has a magnitude greater than or equal to `cfgf[0]` times the largest magnitude of entries in the pivot row, then the diagonal entry is selected as the pivot, otherwise an off-diagonal pivot will be chosen.
- `real_t cfgf[1]`: default value is 3.0. It is also used to pre-allocate memory for parallel LU factorization. If it is larger, NICS LU will use more memory, but during parallel LU factorization, less memory re-allocation will happen.

If not necessary, these configurations (writable members) are not recommended to be changed.

### 4.3 Function Return Values

Each NICS LU function returns an integer (`int_t`) to indicate whether the function is executed successfully or not. The return values and their meanings are listed in the below. You should check the return value of each function to avoid failures of NICS LU.

- `NICS_OK`: value 0. The function is executed successfully.
- `NICS LU_GENERAL_FAIL`: value -1. A simple failure has occurred.
- `NICS LU_ARGUMENT_ERROR`: value -2. There are some errors with the function arguments; for example, you specify `NULL` to a pointer that is not allowed to be `NULL`.
- `NICS LU_MEMORY_OVERFLOW`: value -3. No enough memory.
- `NICS LU_FILE_CANNOT_OPEN`: value -4. The specified file cannot be opened. It is not used in NICS LU.
- `NICS LU_MATRIX_STRUCTURAL_SINGULAR`: value -5. The matrix is structural singular, i.e. the matrix is not structural full-rank.
- `NICS LU_MATRIX_NUMERIC_SINGULAR`: value -6. The matrix is numerical singular, i.e. there is one row/column that does not contain any nonzero elements.
- `NICS LU_MATRIX_INVALID`: value -7. The matrix is invalid because there are some errors in the CSR/CSC storage. For example, an index is out of range.
- `NICS LU_MATRIX_ENTRY_DUPLICATED`: value -8. The matrix has duplicated entries in the CSR/CSC storage.



- `NICSLU_THREADS_NOT_INITIALIZED`: value -9. The threads are not created yet.
- `NICSLU_MATRIX_NOT_INITIALIZED`: value -10. The matrix is not created yet.
- `NICSLU_SCHEDULER_NOT_INITIALIZED`: value -11. The scheduler is not created yet.
- `NICSLU_SINGLE_THREAD`: value -12. When creating only 1 thread, this error occurs, since the main thread does not require to be explicitly created.
- `NICSLU_THREADS_INIT_FAIL`: value -13. The specified threads cannot be created.
- `NICSLU_MATRIX_NOT_ANALYZED`: value -14. The matrix is not analyzed yet.
- `NICSLU_MATRIX_NOT_FACTORIZED`: value -15. The matrix is not factorized yet.
- `NICSLU_NUMERIC_OVERFLOW`: value -16. Numerical overflow has occurred during factorization.

## 4.4 NICSLU Routines

### 4.4.1 `NicsLU_Initialize`

```
int__t NicsLU_Initialize(SNicsLU *nicslu);
```

This function initializes the `SNicsLU` structure and sets the default configurations. It must be called first, before any other NICSLU function called.

### 4.4.2 `NicsLU_Destroy`

```
int__t NicsLU_Destroy(SNicsLU *nicslu);
```

This function destroys the `SNicsLU` structure and frees all the memory allocated by NICSLU. It must be called at last, otherwise memory leak will occur.

### 4.4.3 `NicsLU_CreateMatrix`

```
int__t NicsLU_CreateMatrix(SNicsLU *nicslu, uint__t n, uint__t nnz, real__t
*ax, uint__t *ai, uint__t *ap);
```

This function initializes the matrix which will be used by NICSLU. The matrix is described by the CSR/CSC format (i.e. `n`, `nnz`, `ax`, `ai`, `ap`), which is described in Section 3. If your matrix is stored in CSC format, you can also directly use it, and after calling this function, `nicslu->cfgi[0]` should be set to a non-zero value.

If you need to change the configurations of NICSLU, they should be set after calling this function.

#### 4.4.4 NicsLU\_DestroyMatrix

```
int__t NicsLU_DestroyMatrix(SNicsLU *nicslu);
```

This function destroys the matrix and frees memory used by the matrix. It is contained in `NicsLU_Destroy`, so it can be skipped when you finish your computation.

In this function, all the configurations are set to default values.

#### 4.4.5 NicsLU\_CreateThreads

```
int__t NicsLU_CreateThreads(SNicsLU *nicslu, uint__t thread, bool__t check);
```

This function creates the threads for parallel computation. The second argument (`thread`) specifies the number of threads, including the main thread. The last argument (`check`) specifies whether to check the number of threads or not. If it is `TRUE`, then this function will check your specified thread number, and if the thread number is larger than the number of cores of your computer, the thread number will be set to core number.

We strongly recommend `check = TRUE`.

If you only want to run single-threaded LU factorization (i.e. sequential factorization), this function is not required, you should directly call the sequential version of factorization and re-factorization functions.

#### 4.4.6 NicsLU\_DestroyThreads

```
int__t NicsLU_DestroyThreads(SNicsLU *nicslu);
```

This function destroys the threads and frees memory used by the threads. It is contained in `NicsLU_Destroy`, so it can be skipped when you finish your computation.

#### 4.4.7 NicsLU\_CreateScheduler

```
int__t NicsLU_CreateScheduler(SNicsLU *nicslu);
```

This function creates the task scheduler for parallel LU factorization. If you want to run parallel factorization or parallel re-factorization, it should be called after `NicsLU.Analyze`.

**If this function returns 1, it indicates that the matrix is not suitable for parallel factorization (i.e. the parallel performance may be even worse than the sequential performance, for the specified matrix). It returns 0 if the matrix is suitable for parallel factorization. So we suggest you choose the proper factorization function according to the return value of this function. The suggestion is only for factorization but not re-factorization. `NicsLU.ReFactorize_MT` can always achieve speedups than `NicsLU.ReFactorize`.**

The suggestion can be also obtained by `nicslu->stat[13]`.

#### 4.4.8 NicsLU\_DestroyScheduler

```
int__t NicsLU_DestroyScheduler(SNicsLU *nicslu);
```

This function destroys the scheduler and frees memory used by the scheduler. It is contained in `NicsLU_Destroy`, so it can be skipped when you finish your computation.

#### 4.4.9 NicsLU\_Analyze

```
int__t NicsLU_Analyze(SNicsLU *nicslu);
```

This function analyzes the matrix, including row/column ordering and MC64 scaling. It must be called after `NicsLU_CreateMatrix` and before any factorization or re-factorization.

#### 4.4.10 NicsLU\_Factorize

```
int__t NicsLU_Factorize(SNicsLU *nicslu);
```

This function performs the numerical LU factorization (i.e.  $A = LU$ ) with partial pivoting. It must be called after `NicsLU_Analyze`.

#### 4.4.11 NicsLU\_ReFactorize

```
int__t NicsLU_ReFactorize(SNicsLU *nicslu, real__t *ax);
```

If you want to factorize another matrix with different entry values but with the same nonzero structure, this function can be used. This function is without partial pivoting, so it uses the same pivoting order as the last `NicsLU_Factorize` or `NicsLU_Factorize_MT` called. It must be called after `NicsLU_Factorize` or `NicsLU_Factorize_MT` is called at least once. This function executes faster than `NicsLU_Factorize`; however, it may cause numerical stability problem. Array `ax` specifies the new matrix values in CSR/CSC format.

#### 4.4.12 NicsLU\_Factorize\_MT

```
int__t NicsLU_Factorize_MT(SNicsLU *nicslu);
```

It is the parallel version of `NicsLU_Factorize`. `NicsLU_CreateScheduler` and `NicsLU_CreateThreads` should be called before this function.

#### 4.4.13 NicsLU\_ReFactorize\_MT

```
int__t NicsLU_ReFactorize_MT(SNicsLU *nicslu, real__t *ax);
```

It is the parallel version of `NicsLU_ReFactorize`. `NicsLU_CreateScheduler` and `NicsLU_CreateThreads` should be called before this function.

#### 4.4.14 NicsLU\_Solve

```
int__t NicsLU_Solve(SNicsLU *nicslu, real__t *rhs);
```

This function performs right-hand-solving (i.e.  $Ly = b$  and  $Ux = y$ ) to obtain the solution of  $Ax = b$ . It can be called after any factorization or re-factorization functions.

Array `rhs` is used for both input and output. On input, it should store the right-hand-vector ( $b$ ); on output, it is overwritten by the solution vector ( $x$ ).

#### 4.4.15 NicsLU\_SolveFast

```
int__t NicsLU_SolveFast(SNicsLU *nicslu, real__t *rhs);
```

It is a faster version of `NicsLU_Solve`. When there are many zeros in the right-hand-vector ( $b$ ), this function can be faster than `NicsLU_Solve`.

#### 4.4.16 NicsLU\_ResetMatrixValues

```
int__t NicsLU_ResetMatrixValues(SNicsLU *nicslu, real__t *ax);
```

Since `NicsLU_ReFactorize` and `NicsLU_ReFactorize_MT` are performed without partial pivoting, they may cause numerical stability problem. If you want to factorize a new matrix with the same nonzero pattern, and with partial pivoting to avoid the potential numerical stability problem, then this function should be used to reset the matrix data. And then `NicsLU_Factorize` or `NicsLU_Factorize_MT` can be used to factorize the new matrix with partial pivoting. Array `ax` specifies the new matrix values in CSR/CSC format.

#### 4.4.17 NicsLU\_Refine

```
int__t NicsLU_Refine(SNicsLU *nicslu, real__t *x, real__t *b, real__t eps,
uint__t maxiter);
```

When necessary, this function can be used to refine the solution. However, it is not always successful. The refinement is implemented as follows:

```
compute residual  $r = Ax - b$ ;
while  $\|r\| > eps$ 
    solve  $Ad = r$ ;
    update solution  $x = x - d$ ;
    update residual  $r = Ax - b$ ;
end while
```

The residual is based on the 1-norm. Array `x` should be the solution vector on input; on output, it will be updated by the refinement. Array `b` is the right-hand-vector (input). `eps` is the precision, when the residual is smaller than `eps`, the refinement ends. `maxiter` is used to control the refinement iterations. If `maxiter` is nonzero, the refinement will end when the number of iterations reaches `maxiter`; otherwise the number of iterations has no limit, but it will also end when the residual reaches a minimum value.

#### 4.4.18 NicsLU\_Residual

```
int__t NicsLU_Residual(uint__t n, real__t *ax, uint__t *ai, uint__t *ap,
real__t *x, real__t *b, real__t *error, int__t norm, uint__t mode);
```

This function calculates the residual error of  $\|Ax - b\|$ .

`ax`, `ai` and `ap` are the CSR/CSC storage of matrix A. Array `x` is the solution vector and `b` is the right-hand-vector, both are inputs. `norm` indicates the norm of the residual: 1 indicates the 1-norm, 2 indicates the 2-norm and other values indicate the infinite-norm. `mode` indicates the CSR/CSC mode: zero indicates CSR and non-zero indicates CSC. On output, `*error` returns the residual error. `error` cannot be a NULL pointer.

#### 4.4.19 NicsLU\_Throughput

```
int__t NicsLU_Throughput(SNicsLU *nicslu, real__t *thr);
```

This function estimates the memory throughput (in bytes) that is required to factorize the matrix. Parameter `*thr` returns the throughput if `thr` is not NULL. It is an estimation of the throughput, the actual memory throughput may not be equal to the estimated value. The throughput can be also obtained by `nicslu->stat[12]`.

#### 4.4.20 NicsLU\_Flops

```
int__t NicsLU_Flops(SNicsLU *nicslu, real__t *flops);
```

This function calculates the number of FLOPs that are required to factorize the matrix. Argument `*flops` returns the number of FLOPs if `flops` is not NULL. The number of FLOPs can be also obtained by `nicslu->stat[5]`.

#### 4.4.21 NicsLU\_ThreadLoad

```
int__t NicsLU_ThreadLoad(SNicsLU *nicslu, uint__t thread, real__t
**thread_flops);
```

This function calculates the number of FLOPs of each thread such that one can evaluate load-balance of the parallel algorithm. Parameter `thread` specifies the number of threads and the pointer `*thread_flops` should be NULL. On output, this function will allocate memory for `*thread_flops`, which is a floating-point array, with the length of the thread number. The number of FLOPs of thread No. `i` is stored in `(*thread_flops)[i]`. The thread number specified here may not equal to the actual thread number used in parallel factorization.

Example:

```
real__t *thread_flops;
thread_flops = NULL;
/*factorizing the matrix here ...*/
NicsLU_ThreadLoad(&nicslu, 8, &thread_load);
/*to obtain flops of thread i, visit thread_load[i]*/
```

#### 4.4.22 NicsLU\_Transpose

```
int__t NicsLU_Transpose(uint__t n, uint__t nnz, real__t *ax, uint__t *ai,
uint__t *ap);
```

This function transposes a matrix stored in CSR/CSC format. On input, you should specify `n`, `nnz`, `ax`, `ai`, `ap` to be the original matrix; on output, `ax`, `ai`, `ap` will be overwritten by the transposed matrix.

#### 4.4.23 NicsLU\_DumpA

```
int__t NicsLU_DumpA(SNicsLU *nicslu, real__t **ax, uint__t **ai, uint__t **ap);
```

This function stores matrix  $A$  into CSR format after factorization. The exported matrix is different from the original matrix since row/column ordering and MC64 scaling may be performed after analysis and factorization. Pointers `*ax`, `*ai`, `*ap` must be NULL, otherwise a memory exception or memory leak will occur. This function will allocate memory for these pointers.

Example:

```
real__t *ax;
uint__t *ai, *ap;
ax = NULL;
ai = NULL;
ap = NULL;
/*factorizing the matrix here ...*/
NicsLU_DumpA(nicslu, &ax, &ai, &ap);
```

#### 4.4.24 NicsLU\_DumpLU

```
int__t NicsLU_DumpLU(SNicsLU *nicslu, real__t **lx, uint__t **li, size_t
**lp, real__t **ux, uint__t **ui, size_t **up);
```

This function stores the factorized LU factors into CSR format. Pointers `*lx`, `*li`, `*lp`, `*ux`, `*ui`, `*up` must be NULL, otherwise a memory exception or memory leak will occur. This function will allocate memory for these pointers. The exported CSR arrays contain the diagonals of  $L$  and  $U$ . The number of nonzeros of  $L$  and  $U$  can be obtained from `nicslu->l.nnz` and `nicslu->u.nnz`.

Example:

```
real__t *lx, *ux;
uint__t *li, *ui;
size_t *lp, *up;
lx = ux = NULL;
li = ui = NULL;
lp = up = NULL;
/*factorizing the matrix here ...*/
NicsLU_DumpLU(nicslu, &lx, &li, &lp, &ux, &ui, &up);
```

#### 4.4.25 NicsLU\_ConditionNumber

```
int__t NicsLU_ConditionNumber(SNicsLU *nicslu, real__t *cond);
```

This function estimates the condition number of the matrix, using the 1-norm. If MC64 scaling is used, the condition number is reported based on the scaled matrix, otherwise it's calculated based on the original matrix. Argument `*cond` returns the condition number if `cond` is not NULL. The condition number can be also obtained by `nicslu->stat[6]`.

## 5 History

### 2013, Apr 16. version 2.0

- \* NICS LU is distributed under the GNU LGPL license.

### 2012, Dec 16. version 1.2

- \* the framework of NICS LU has a few changes, current framework is the final one and will not be changed in future versions.
- \* add function: `NicsLU_MemoryUsage`.
- \* add function: `NicsLU_DumpA`.
- \* add function: `NicsLU_SolveFast`.
- \* `NicsLU_ResetMatrixData` is changed to `NicsLU_ResetMatrixValues`.
- \* `NicsLU_ResidualError` is changed to `NicsLU_Residual`, the arguments are also changed.
- \* memory usage optimization.
- \* demo programs are changed.
- \* bug fixes.

### 2011, Oct 19. version 1.1

- \* add function: `NicsLU_Throughput`.
- \* add function: `NicsLU_ThreadLoad`.
- \* `NicsLU_CreateScheduler` doesn't need to be called after `NicsLU_CreateThreads` anymore.
- \* correct an error in `NicsLU_ResidualError`.
- \* some small improvements.
- \* some small bug fixes.

2011, Jul 20. version 1.0

\* the first version is released.

## References

- [1] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9(5):862–874, 1988.
- [2] T. A. Davis and E. P. Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, September 2010.
- [3] I. S. Duff and J. Koster. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, July 1999.
- [4] I. S. Duff and J. Koster. On Algorithms For Permuting Large Entries to the Diagonal of a Sparse Matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, July 2000.
- [5] X. Chen, Y. Wang, and H. Yang. NICS LU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(2):261–274, feb. 2013.
- [6] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang. An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 58(10):702–706, oct. 2011.
- [7] X. Chen, Y. Wang, and H. Yang. An adaptive LU factorization algorithm for parallel circuit simulation. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 359–364, 30 2012-feb. 2 2012.
- [8] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, May 1999.
- [9] J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.
- [10] X. S. Li and J. W. Demmel. SuperLU\_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.
- [11] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, April 2004.