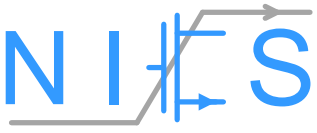
User Guide for NICSLU

—A Parallel Sparse Direct Solver for Circuit Simulation

(Version 201905)

Xiaoming Chen
chenxiaoming@ict.ac.cn
Nanoscale Integrated Circuit and System (NICS) Laboratory
Department of Electronic Engineering
Tsinghua University

May 8, 2019



Nanoscale Integrated Circuit
and System Laboratory

Contents

1 License

NICSLU, Copyright by NICS laboratory, Tsinghua University. NICSLU is protected by three Chinese patents (CN201110337789.6, CN201110076027.5, and CN201110088086.4).

An old and slow version of NICSLU was released under the GNU LGPL license. However, it is out of date and we will no longer maintain or update it.

Current and newer versions of NICSLU are distributed only for customization or research. For academic purpose, users can always use NICSLU for free. For commercial use, please contact the authors for details.

1.1 License File

In most cases, running NICSLU requires a valid license file. The license file is of size 512 bytes. The license may restrict the expiration date of using NICSLU, the operating system for running NICSLU, the maximum matrix dimension, the NICSLU version, etc. There are two alternative ways to set the license file for using NICSLU.

- Set the environment variable `NICSLU_LICENSE` to point to the license file. The content of the environment variable must contain both the path and file name, e.g., `NICSLU_LICENSE=/home/<user name>/nicslu.lic`. Only specifying the path without the file name does not work.
- If `NICSLU_LICENSE` is not set, rename the license file as `nicslu.lic` and put it along with the executable (with the original target executable instead of any symbolic link) or the NICSLU library (with the symbolic link in use if it exists).

The routine `NicsLU_Initialize` checks the license. If license check fails, it will return -100, -101, -102, -103, or -104 according to the reason of the failure. See Table 4 for details. The utility routine `PrintNicsLULicense` prints the license information.

2 Introduction

NICSLU is a high-performance and robust software package for solving large-scale sparse linear systems of equations ($\mathbf{Ax} = \mathbf{b}$) on shared-memory machines. It is written by pure C, and can be easily used in C/C++ programs. NICSLU is a black-box software and manages memories and threads by itself.

The Gaussian elimination method is widely used to solve the linear system $\mathbf{Ax} = \mathbf{b}$. Basically, matrix $\mathbf{A}$ can be factorized into the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ (i.e., $\mathbf{A} = \mathbf{LU}$), and then the solution of $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x}$, is computed by solving two triangular equations $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$. Matrix $\mathbf{A}$ does not need to be symmetric or definite, but it must be square and full-rank to ensure the solvability of the linear system.

Sparse Gaussian elimination can be described as follows. An $n \times n$ matrix $\mathbf{A}$ is factorized by

$$\mathbf{LU} = \mathbf{PD}_r\mathbf{AD}_c\mathbf{QM} \tag{1}$$

where $\mathbf{D}_r$ and $\mathbf{D}_c$ are two diagonal matrices to scale $\mathbf{A}$ to enhance numerical stability; $\mathbf{P}$ and $\mathbf{Q}$ are the row and column permutation matrices, which are used to maintain sparsity (i.e., minimize fill-ins); $\mathbf{M}$ is a column permutation matrix generated by partial pivoting.

After an LU factorization, $\mathbf{Ax} = \mathbf{b}$ can be solved by

$$\begin{aligned}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ &= \left(\mathbf{D}_r^{-1}\mathbf{P}^{-1}\mathbf{L}\mathbf{U}\mathbf{M}^{-1}\mathbf{Q}^{-1}\mathbf{D}_c^{-1}\right)^{-1}\mathbf{b} \\ &= \mathbf{D}_c\mathbf{Q}\mathbf{M}\left(\mathbf{U}^{-1}\left(\mathbf{L}^{-1}\left(\mathbf{P}\mathbf{D}_r\mathbf{b}\right)\right)\right).\end{aligned} \tag{2}$$

The fundamental algorithm of NICSLU is based on the sparse left-looking algorithm proposed by Gilbert and Peierls [1] and the KLU algorithm proposed by Davis [2]. We have further proposed many advanced techniques to enhance the performance of NICSLU. NICSLU improves existing algorithms mainly in the following four aspects.

- NICSLU can automatically select the most suitable algorithm from several integrated numerical kernels according to the sparsity of the matrix [3], so that NICSLU is an adaptive solver.
- An efficient static pivoting/scaling algorithm (HSL_MC64 algorithm) [4,5] is adopted combining with partial pivoting, dynamic scaling, and iterative refinement to achieve higher numerical stability.
- We have developed an innovative parallelization and scheduling framework for parallel LU factorization [6], which obtains effective acceleration on shared-memory multi-core processors [7, 8].
- A numerically stable pivoting-reduction technique in the left-looking algorithm was developed [9]. This algorithm reuses previous information as much as possible during successive factorizations in the Newton-Raphson method.

The detailed techniques and algorithms used in NICSLU are described in our book [10].

There are also some other popular software packages which do the same thing, such as SuperLU [11–13], PARDISO [14], etc. NICSLU is different from these software packages since NICSLU is specially designed for circuit simulation problems. NICSLU is well suited for extremely sparse matrices, and specially supports the case which requires many factorizations with the identical nonzero pattern but different values. NICSLU has been proved to be much faster than PARDISO and KLU on circuit matrices. **NICSLU has been tested in several state-of-the-art commercial circuit simulators and shown excellent performance, especially for large cases. If you want to use NICSLU is a SPICE-style circuit simulator, please read Section 6 carefully.**

NICSLU supports both **real number** and **complex number** matrices. Native real number and complex number kernels are both integrated in NICSLU.

The parallelism of NICSLU is implemented by low-level functions provided by Windows API (for Windows)/pthread library (for Linux). NICSLU does not require OpenMP.

If NICSLU is used in your research, please cite one or more of the following publications when you publish your work:

[1] Xiaoming Chen, Yu Wang, Huazhong Yang, "Parallel Sparse Direct Solver for Integrated Circuit Simulation", Springer Publishing, 1st edition, Feb. 2017. 136 pages.

[2] 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.

[3] 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.

[4] 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.

[5] Xiaoming Chen, Yu Wang, Huazhong Yang, "A Fast Parallel Sparse Solver for SPICE-based Circuit Simulators", Design, Automation, and Test in Europe (DATE) 2015, pp.205-210, 9-13 March, 2015.

[6] Xiaoming Chen, Lixue Xia, Yu Wang, Huazhong Yang, "Sparsity-Oriented Sparse Solver Design for Circuit Simulation", Design, Automation, and Test in Europe (DATE) 2016, pp.1580-1585, March 14-18, 2016.

3 Quick Start

Basically, to solve a linear system, at least five routines of NICSLU are required, which are listed below.

- `NicsLU_Initialize`
  This routines first checks the license. If license check is passed, it creates the handle of the solver.
- `NicsLU_Analyze`
  This routine pre-orders the matrix to reduce fill-ins and performs a symbolic factorization.
- `NicsLU_FactorizeMatrix`
  This routine performs the numerical factorization to get the LU factors. One can also call `NicsLU_Factorize` or `NicsLU_ReFactorize` to perform the numerical factorization. For the first factorization, it is equivalent to `NicsLU_Factorize`.
- `NicsLU_Solve`
  This routine performs forward/backward substitutions to obtain the solution of the linear system $\mathbf{Ax} = \mathbf{b}$, using the LU factors computed by the previous step. One can also call `NicsLU_SolveAndRefine` to perform substitutions with an automatically controlled iterative refinement.
- `NicsLU_Free`
  This routine frees all objects and destroys the handle of the solver.

Additional low-level routines are available for more functionalities and finer-grained operations of NICSLU. Please see Section 5 for more details. The matrix $\mathbf{A}$ is represented in a compressed row form. Please see Section 4 for more details. Below we show a simple example, which illustrates the basic usage of NICSLU. When running this code, please set the license file according to the methods mentioned in Section 1.1. We also provide other demos in the `demo` folder of the release package.

```
#include <stdio.h>
#include "nicslu.h"

int main()
{
```

```
    _double_t ax[13] = { 1.1, -7.7, 13.13, 2.2, 9.9, 8.8, -3.3, -4.4,
        11.11, 5.5, 10.1, 12.12, 6.6 };
    _uint_t ai[13] = { 0, 3, 4, 1, 4, 1, 2, 3, 2, 4, 0, 3, 5 };
    _uint_t ap[7] = { 0, 3, 5, 7, 8, 10, 13 };
    _double_t b[6] = { 35.95, 53.9, 7.7, -17.6, 60.83, 98.18 };
    _handle_t solver = NULL;
    _uint_t i;

    if (__FAIL(NicsLU_Initialize(&solver, NULL, NULL, NULL)))
    {
        printf("Failed to initialize\n");
        return -1;
    }
    NicsLU_Analyze(solver, 6, ax, ai, ap, MATRIX_ROW_REAL, NULL, NULL,
        NULL, NULL);
    NicsLU_FactorizeMatrix(solver, ax, 1);
    NicsLU_Solve(solver, b, NULL);
    NicsLU_Free(solver);

    for (i = 0; i < 6; ++i) printf("x[%d] = %g\n", i, b[i]);

    return 0;
}
```

The `ax`, `ai` and `ap` arrays represent the sparse row format of the matrix shown in Figure 1. The storage format will be explained in the next section. The solution of this example is $\mathbf{x} = [1, 2, 3, 4, 5, 6]^{\mathrm{T}}$. This example uses default configurations and does not return any statistics about the ordering, scaling, factorization, or solution. Please see Section 5 for more details of NICSLU's features.

4 Matrix Format

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

- `n`: an (unsigned) integer, matrix dimension, i.e., the matrix is `n`×`n`. NICSLU only supports square matrices.
- `nnz`: an (unsigned) integer, the number of nonzeros in the matrix.
- `Ax`: a real number or complex number array of length `nnz`, storing the values of all nonzeros. `Ax` is stored in the row-major order. For a complex number matrix, each element of `Ax` is a complex number, i.e., two successive real numbers storing the real and imaginary parts. `Ax` means "`A`'s values".
- `Ai`: an (unsigned) integer array of length `nnz`, storing the column indexes of all nonzeros. `Ai` is also stored in the row-major order. `Ai` means "`A`'s indexes". Elements of `Ax` and `Ai` must be one-to-one matched.
- `Ap`: an (unsigned) integer array of length `n+1`, storing the position of the first nonzero of each row in `Ai` and `Ax`. The first and last elements must be `Ap[0]=0` and

$$\begin{pmatrix} 1.1 & 0 & 0 & -7.7 & 13.13 & 0 \\ 0 & 2.2 & 0 & 0 & 9.9 & 0 \\ 0 & 8.8 & -3.3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4.4 & 0 & 0 \\ 0 & 0 & 11.11 & 0 & 5.5 & 0 \\ 10.1 & 0 & 0 & 12.12 & 0 & 6.6 \end{pmatrix}$$

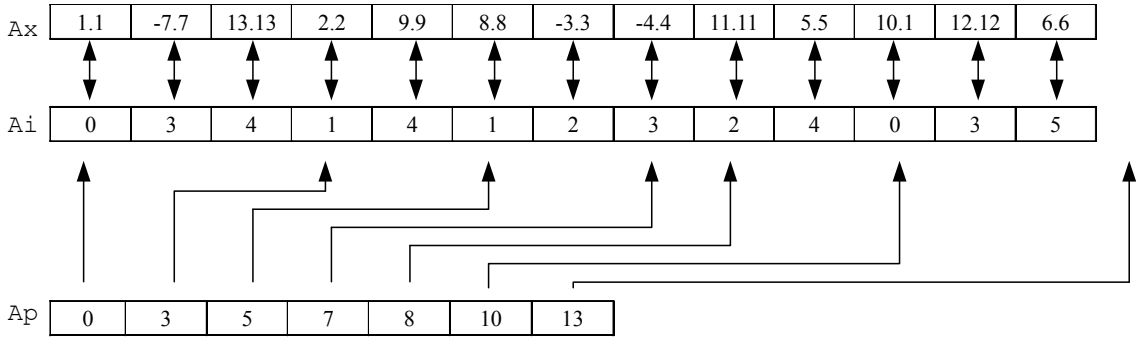


Figure 1: Example of the CSR format.

`Ap[n]=nnz` respectively. Values of the ith row of the matrix are stored in `Ax[Ap[i]]`, `Ax[Ap[i]+1]`, $\cdots$, `Ax[Ap[i+1]-1]`, and the corresponding column indexes of the nonzeros are stored in `Ai[Ap[i]]`, `Ai[Ap[i]+1]`, $\cdots$, `Ai[Ap[i+1]-1]`, number of nonzeros of the ith row is `Ap[i+1]-Ap[i]`. `Ap` means "`A`'s row pointers".

The transposed format, compressed sparse column (CSC), is also supported. CSR stores sparse matrices in the row-major order and CSC stores sparse matrices in the column-major order. NICSLU uses CSR by default. If your matrix is stored in CSC format, NICSLU can also directly handle it. In the CSC case, NICSLU solves $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ instead of $\mathbf{Ax} = \mathbf{b}$. The LU factors are identical in CSR and CSC cases.

Complex numbers should be stored in a packed complex form, namely, an array with the real and imaginary parts interleaved. Please note that NICSLU's interfaces only support real number arrays, so for a complex number matrix, `Ax` should be casted to a real number pointer.

The meanings of `Ai` and `Ap` are the same as those in KLU, but please note KLU uses CSC by default. If your solver is switched from PARDISO to NICSLU, please note that `Ai` (`Ap`) in NICSLU corresponds to `ja` (`ia`) in PARDISO. In addition, the column indexes of each row can be in any order for NICSLU, which is also different from PARDISO.

NOTE: NICSLU only supports C-type arrays, which means `Ai` and `Ap` are zero-based indexed.

NOTE: The CSR storage must not have duplicated entries.

5 Using NICSLU in C/C++ Programs

NICSLU provides a standard C interface so it can be easily used in C/C++ programs. This section explains how to use NICSLU in C or C++ programs in detail. The demo code also provides a fast and simple demonstration of the usage of NICSLU.

5.1 Data Types

Table 1: Data types used in NICSLU.

Data type	C date type	Description
_handle_t	void *	Handle or context, internal object
_int_t	int or long long	32-bit or 64-bit[a] integer
_uint_t	unsigned int or Unsigned long long	32-bit or 64-bit[a] unsigned integer
_double_t	double	Double-precision floating-point
_bool_t	unsigned char	Boolean value: __TRUE or __FALSE
_byte_t	unsigned char	Byte, 8-bit
_complex_t	double [2][b]	Complex number
_size_t	size_t	Return type of sizeof
_uint64_t	Unsigned long long	64-bit unsigned integer

[a] This depends on whether the macro __NICS_INT64 is defined. See Section 7.2 for more details. On Windows, an alternate type of long long is __int64.

[b] Please do not define a complex number type like this:
struct complex {double real; double image;};
The C standard does not guarantee the continuity of member variables within a structure due to possible alignment requirements or optimizations.

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 equivalent data types in standard C. The detailed definitions of these data types can be found in nics_common.h.

5.2 Configurations

Users can easily control the features of NICSLU via the configuration array _double_t cfg[32]. The pointer of the array can be retrieved from NicsLU_Initialize. Its specification is listed in Table 2. Users can ignore the configuration array, then NICSLU uses the default configurations. Some configuration parameters will be explained in detail in the following contents. NICSLU will change a configuration parameter if it is out of range.

NOTE: Users should not change configuration parameters which are not listed in Table 2, because some undocumented configuration parameters are also used by NICSLU. Optionally changing undocumented configuration parameters may cause problems.

NOTE: The calling overhead of the built-in timer may be quite large (e.g., hundreds of CPU cycles) on some platforms, so we suggest turning off the timer (default option) as long as it is not necessary.

5.2.1 Guidance on Adjusting Configurations

The optimal configurations for best performance depend on the hardware and characteristics of the matrix. Although there are many parameters that can be adjusted, only a few of them can potentially have great impacts on the performance. Among all the parameters listed in Table 2, cfg[1], cfg[12], cfg[5], cfg[6] and cfg[7] may have some impacts on the performance.

cfg[1] is the tolerance for partial pivoting which controls the magnitude of pivots. Increasing cfg[1] may improve the accuracy of the solution, but can increase fill-ins dramatically. Increasing cfg[1] cannot improve the condition number of the matrix so it

Table 2: Specification of the _double_t cfg[32] array.

Index	Default	Range	Description
0	0	any	Timer control. 0: no timer; >0: high-precision timer; <0: low-precision timer
1	10^{-3}	[10^{-8},1]	Partial pivoting threshold
2	-1	any	Synchronization method. ≤0: blocked wait; >0: spin wait
3	10	0/1/±2/±3/±4 /±5/±6/±7/±8 /±9/±10	Ordering method. 0: no ordering; 1: user ordering; ±2: AMD; ±3: mAMD[a]; ±4: COLAMD; ±5: METIS; ±6: AMF; ±7: mAMF1[b]; ±8: mAMF2; ±9: mAMF3; ±10: the best one in AMD, mAMD, AMF, mAMF1, mAMF2 and mAMF3. Negative values mean that NicsLU_Analyze will output the ordering. See Section 5.6.3 for more details
4	10.0	any	Threshold for dense row detection in AMD, AMF, and COLAMD
5	0	any	Pre-ordering and scaling. 0: static pivoting; >0: static pivoting and scaling; <0: symbolic pattern-based zero -free permutation. See Section 5.2.1 for more details
6	4	[1, $\sqrt{n}$]	Parameter for pre-allocating memory
7	1.5	[1.25,3.0]	Dynamic memory growth factor
8	80	≥8	Minimum number of columns of supernodes
9	4	4∼100	Number of initial rows of supernodes
10	8	≥2	Pipeline scheduling parameter
11	0.95	[0.5,1.0]	Parameter for controlling load balance
12	0	any	Dynamic scaling method. 0: no scaling; >0: max-scaling; <0: sum-scaling
13	1	0/1	Enable automatic control for serial/parallel factorization?
14	3	1∼20	Maximum number of iterations for iterative refinement
15	10^{-10}	[$2.22\times10^{-16}, 10^{-4}$]	Residual requirement for iterative refinement
16	10^{12}	[10^{8},10^{30}]	Pseudo condition number threshold for doing refinement
17	5.0	[0,1000]	Threshold to determine whether to call re-factorization or factorization in NicsLU_FactorizeMatrix
18	0	any	Pivot perturbation threshold. 0 or negative value means no pivot perturbation
19	2	≥ 1.25	Threshold for garbage collection

[a] Modified AMD.
[b] Modified AMF.

does not help reduce the singularity. If you want to increase cfg[1], it is recommended to set cfg[5]>0, which helps reduce fill-ins.

If cfg[5] is 0, NICSLU tries to permute the matrix to obtain a zero-free diagonal based on both the symbolic pattern and numerical values passed into NicsLU_Analyze. If it is positive, NICSLU tries to permute the matrix for a zero-free diagonal and scale the matrix based on both the symbolic pattern and numerical values. Otherwise (negative value), NICSLU only tries to permute the matrix for a zero-free diagonal based on the symbolic pattern. In most cases, cfg[5]=0 can generate good results in both performance and accuracy.

For the scaling-related options cfg[5] and cfg[12], we do not have a method to automatically decide whether scaling can improve the performance. Based on our experience,

Table 3: Specification of the `const _double_t stat[32]` array.

Index	Description
0	Runtime of the last symbolic analysis
1	Runtime of the last factorization or re-factorization
2	Runtime of the last solving
3	Estimated number of floating-point operations
4	Estimated number of nonzeros in $\mathbf{L}+\mathbf{U}-\mathbf{I}$
5	Structural symmetry of the matrix, valid only when using AMD or AMF ordering
6	Runtime of iterative refinement
7	Number of iterations performed by iterative refinement
8	Number of nonzeros in $\mathbf{L}+\mathbf{U}-\mathbf{I}$
9	Number of nonzeros in $\mathbf{L}$
10	Number of nonzeros in $\mathbf{U}$
11	Number of off-diagonal pivots
12	Number of supernodes
13	Number of perturbed pivots
14	Number of factorizations executed
15	Number of re-factorizations executed
29	License expiration date (format: YYYYMMDD)
30	Compilation time (format: YYYYMMDD.HHMMSS)
31	Version

Table 4: Specification of return values of NICSLU routines.

Value	Description
0	Everything is OK
-100	No license found
-101	Invalid license (e.g., the license is damaged)
-102	License expired
-103	License restricted (matrix dimension, OS, or version is restricted)
-104	License check error
-1	Invalid handle
-2	Routine arguments have errors, e.g., a `NULL` pointer is specified
-3	Out of memory, no enough virtual memory (`malloc` or `realloc` fails)
-4	Structurally singular or symbolic zeros on the diagonal
-5	Numerically singular
-6	Invalid input (e.g., an index is out of range)
-7	The CSR/CSC storage has duplicated entries
-8	Threads have not been created
-9	Threads cannot be created
-10	Matrix is not analyzed
-11	Matrix is not factorized
-12	Abnormal numerical values, namely, `inf` or `nan`
-13	32-bit integer is not large enough to store the matrix, please use 64-bit integer
-14	Cannot open the specified file
-255	An unknown failure has occurred. Please contact the author
+1	Setting thread schedule failed
+2	Static pivoting is invalid
+3	The same number of threads are already created

cfg[5]>0 is generally good for ill-conditioned matrices (with large condition number) but has little effect for well-conditioned matrices. cfg[12] only has effect for a few matrices but has no effect for most matrices.

cfg[6] is used for pre-allocating memories for factorizations. If it is bigger, NICSLU will pre-allocate more memories so memory re-allocations during factorizations can be reduced, but there may be more useless memories. cfg[7] controls the growth ratio when a memory re-allocation happens. A larger cfg[7] will re-allocate more memories so further re-allocations may be reduced.

5.3 Statistics

Users can obtain the statistics information of NICSLU from the const _double_t stat[32] array. The pointer of the array can be retrieved from NicsLU_Initialize. It collects runtime of some routines, and statistics of the factorization and the factors. The specification of the stat array is listed in Table 3. The runtime reported is wall time in seconds with a high-resolution of up to microsecond (μs).

NOTE: Users can only read these statistics parameters but must not write them.

5.4 Error Code & Message

All NICSLU routines return an integer (int) value to indicate whether the routine is executed successfully or not. The return values and their descriptions are listed in Table 4. Negative values indicate fatal failures so the solver must stop. Positive values indicate a warning is generated but the solver can continue without affecting the correctness of the solution. A few macros defined in nics_common.h can be utilized to conveniently check the return values: __SUCCESS(code), __FAIL(code), and __WARNING(code).

The error message can be retrieved from a string whose pointer can be retrieved from NicsLU_Initialize. The string stores the last error message, which includes a terminating null character ('\0') but does not include a newline character ('\n'). Users can use a printf or puts function to print the message. But please note that the error message can only be retrieved when the solver handle is valid. This means that, any failure caused by NicsLU_Initialize or NULL handle does not produce an error message. Among all the routines of NICSLU, only PrintNicsLULicense may directly produce messages on the screen. The other routines generate error messages through the error message string which must be explicitly printed by the users.

NOTE: Users should check the return value of any NICSLU routine to avoid any failures.

5.5 Solver Handle

A *handle* means an object or a context. The handle of NICSLU is an internal data structure which maintains all necessary data for NICSLU. The handle is created by NicsLU_Initialize, and passed into all other NICSLU routines (except the utility routines) as the first argument. If handle is NULL when calling NICSLU routines (except NicsLU_Initialize), the routine returns -1 immediately.

NOTE: One handle cannot be processed by multiple threads simultaneously.

5.6 Low-Level Routines

This subsection and the subsequent two subsections will introduce the routines of NICSLU. Current NICSLU provides 23 user-callable routines, including 16 low-level routines, 2 high-level routines, and 5 utility routines. The C header file `nicslu.h` defines the interface of the NICSLU routines. NICSLU uses only one set of routines for both real number and complex number matrices, so any complex number pointer must be casted to a real number pointer when calling NICSLU routines.

5.6.1 NicsLU_Initialize

```
int NicsLU_Initialize
(
    _handle_t *solver,
    _double_t **cfg,
    const _double_t **stat,
    const char **last_err
);
```

This routine creates the solver handle and internal objects, and sets the default configurations. However, before initializing these, this routine first checks the license. If the license check fails, this routine returns an error code indicating the reason of the failure without initializing anything. The first argument `solver` will return the handle. The next two arguments `cfg` and `stat` return the pointers of the configuration array and the statistics array. The last argument `last_err` returns the pointer of the error message string. If you do not want to change configurations, get the statistics information, or retrieve the error message, they can be `NULL`. The three pointers retrieve the head addresses of three internal arrays. This also means that users do not need to allocate spaces for them.

NOTE: This is the only routine from which the pointers of the configuration array, the statistics array, and the error message string can be obtained. If you pass `NULL` pointer to `cfg`, `stat`, or `last_err`, you will have no chance to obtain these pointers later.

Usage example:

```
_handle_t solver = NULL;
_double_t *cfg = NULL;
const _double_t *stat = NULL;
const char *last_err = NULL;
...
int err = NicsLU_Initialize(&solver, &cfg, &stat, &last_err);
if (__FAIL(err)) ... /*deal with failure and exit*/

cfg[0] = ...; /*change configurations before matrix analysis*/
...
err = NicsLU_Analyze(...);
if (__FAIL(err))
{
    puts(last_err); /*print error message and exit*/
    ...
}
```

```
...
NicsLU_Free(solver);
solver = NULL;
```

5.6.2 NicsLU_Free

```
int NicsLU_Free
(
    _handle_t solver
);
```

This routine frees all the memory allocated by NICSLU and destroys the handle. If the handle is `NULL`, it does nothing.

NOTE: Each `NicsLU_Initialize` call must match an `NicsLU_Free` call, otherwise memory leak will occur. Call this routine only once for one handle, otherwise segmentation fault will occur.

5.6.3 NicsLU_Analyze

```
int NicsLU_Analyze
(
    _handle_t solver,
    _uint_t n,
    const _double_t *ax,
    const _uint_t *ai,
    const _uint_t *ap,
    _matrix_type_t mtype,
    _uint_t *row_perm,
    _uint_t *col_perm,
    _double_t *row_scale,
    _double_t *col_scale
);
```

This routine creates and analyzes the matrix, including row/column ordering, calculation of the static scaling factors, and doing symbolic factorization. The arguments `ax`, `ai` and `ap` specify the CSR arrays of the matrix, which are described in Section 4. This routine must be called before any factorization or re-factorization. `row_perm` and `col_perm` are two arrays of length `n` which are used to specify a user-defined ordering or retrieve the ordering. `row_perm[i]=j` (`col_perm[i]=j`) means that row (column) i in the permuted matrix is row (column) j is the original matrix. `row_scale` and `col_scale` (can be `NULL`) are two arrays of length `n` which are used to retrieve the row and column scaling vectors (for the permuted matrix). Users can calculate the permuted and scaled matrix by the two permutation arrays and two scaling arrays. If this routine is called more than once, all memories associated with the previous matrix as well as the existing threads will be freed/destroyed.

NICSLU provides eight different ordering methods: AMD [15], a modified version of AMD (mAMD), approximate minimum fill-in (AMF) [16], three modified versions of AMF (mAMF1, mAMF2 and mAMF3), COLAMD [17], and METIS [18]. `cfg[3]` is used to select the ordering method. By default, NICSLU selects the best one from AMD, mAMD, AMF, mAMF1, mAMF2 and mAMF3. Generally speaking, AMD and AMF can provide

reasonably good orderings in most cases, while AMF is better than AMD in most cases, and AMF is as fast as AMD. COLAMD, which is not recommended, is suitable for only very dense matrices. METIS is suitable for large dense matrices, but METIS can be very slow in some extreme cases.

Based on the value of `cfg[3]`, the ordering has the following 16 cases.

- `cfg[3] = 0`: the natural order is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 1`: a user-defined ordering is used. `row_perm` and `col_perm` specify the ordering.
- `cfg[3] = 2`: AMD is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 3`: the modified AMD is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 4`: COLAMD is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 5`: METIS is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 6`: AMF is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 7`: the modified AMF1 is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 8`: the modified AMF2 is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 9`: the modified AMF3 is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = 10` (**default**): the best one in AMD, mAMD, AMF, mAMF1, mAMF2 and mAMF3 is used. `row_perm` and `col_perm` can be `NULL`.
- `cfg[3] = -2`: AMD is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -3`: the modified AMD is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -4`: COLAMD is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -5`: METIS is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -6`: AMF is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -7`: the modified AMF1 is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -8`: the modified AMF2 is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -9`: the modified AMF3 is used. `row_perm` and `col_perm` return the ordering.
- `cfg[3] = -10`: the best one in AMD, mAMD, AMF, mAMF1, mAMF2 and mAMF3 is used. `row_perm` and `col_perm` return the ordering.

NOTE: If a user-defined ordering or the natural order is used, the permutation or the natural order must guarantee a symbolically zero-free diagonal; otherwise this routine returns -4 (symbolic singular).

The argument `mtype` is an enumeration variable to specify the type of the matrix. It can be one of the following four values.

- `MATRIX_ROW_REAL` (value 0): real matrix stored in CSR format.
- `MATRIX_COLUMN_REAL` (value 1): real matrix stored in CSC format. In this case, NICSLU solves $\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$.
- `MATRIX_ROW_COMPLEX` (value 2): complex matrix stored in CSR format.
- `MATRIX_COLUMN_COMPLEX` (value 3): complex matrix stored in CSC format. In this case, NICSLU solves $\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$.

If an application involves successively solving multiple linear systems with identical nonzero pattern of **A** but different numerical values, this routine needs to be performed only once for the first matrix. For subsequent systems, only factorizations (or re-factorizations) and substitutions are required. This feature is important in circuit simulation, see Section 6 for details.

Providing valid and good numerical values of `ax` when calling this routine is strongly recommended. NICSLU can use these values to calculate static scaling factors and permute large elements to the diagonal. If you need to factorize the matrix in multiple iterations with the identical symbolic pattern and different numerical values, the values provided to this routine should be representative. The numerical values of `ax` can also be invalid when calling this routine. For example, all values are zeros. In this case, the static scaling feature will be automatically disabled, and NICSLU will use a pure symbolic pattern-based permutation method to obtain a zero-free diagonal. Besides, `cfg[5]` can also be used to control the zero-free permutation method. See Section 5.2.1 for more details.

A note on METIS library: AMD, AMF, and COLAMD are integrated in NICSLU but METIS is not. If you choose to use METIS, you will need to provide a METIS shared library (dynamic-link library). **The file must be named `metis.dll` on Windows or `libmetis.so` on Linux and put along with the executable or the NICSLU library**. NICSLU links METIS at runtime and calls the `METIS_NodeND` routine if a dynamic link is successful. If METIS is enabled and NICSLU does not find the METIS library when calling `NicsLU_Analyze`, it returns -14 (cannot open file). If you do not use METIS, the METIS shared library is not required.

NOTE: If the matrix is stored in a column-wise format, the actual meanings of `row_perm` and `col_perm`, and `row_scale` and `col_scale` are interchanged. In other words, in this case, `row_perm` and `row_scale` are actually the column permutation and column scaling vector for the actual matrix, and `col_perm` and `col_scale` are actually the row permutation and row scaling vector for the actual matrix.

5.6.4 NicsLU_CreateThreads

```
int NicsLU_CreateThreads
(
    _handle_t solver,
    int threads
);
```

This routine creates threads for parallel factorizations or re-factorizations. The argument `threads` specifies the number of threads that will be created. If `threads` is 0, this routines will create threads as the same number of physical cores on the computer. If this routine is called more than once with the same number of threads, it returns +3 immediately; otherwise it first destroys the existing threads and then creates the new threads. The created threads will not exit until `NicsLU_DestroyThreads` or `NicsLU_Free` is called. `NicsLU_Analyze` must be called before calling this routine.

`cfg[2]` is used to control the waiting method for thread synchronization. `cfg[2]`$\leq$`0` (default is 0) indicates using the blocked waiting method and `cfg[2]`$>$`0` indicates using the busy (spin) waiting method. Busy waiting may increase the performance but CPU is occupied even when no tasks are running. `cfg[2]` is effective only before calling `NicsLU_CreateThreads`. In other words, changing `cfg[2]` after `NicsLU_CreateThreads`

has no effect.

NOTE: The specified number of threads cannot exceed the dimension of the matrix or the number of logical cores on the computer.

NOTE: If your CPU supports hyper-threading, it is not recommended to use all the logical threads. Instead, it is recommended to only use all the physical cores. Using hyper-threading may even degrade the performance as NICSLU is a compute-intensive program.

5.6.5 NicsLU_DestroyThreads

```
int NicsLU_DestroyThreads
(
    _handle_t solver
);
```

This routine destroys the threads and frees memory used by the threads. Multiple calls for the same handle have no effect.

5.6.6 NicsLU_SetThreadSchedule

```
int NicsLU_SetThreadSchedule
(
    _handle_t solver,
    _thread_sched_t op,
    int param
);
```

This routines changes threads' scheduling policy by binding threads to cores or changing threads' priority. The second argument `op` is an enumeration variable to specify the operation. The last argument `param` specifies the parameter for the corresponding operation. `op` can be one of the following two values.

- `THREAD_BINDING` (value 0): binding threads to cores (`param >= 0`) or unbinding threads from cores (`param < 0`). Binding threads to cores may increase the performance for very sparse matrices. The argument `param` specifies the minimum core id that will be bound. Threads are bound to successive cores from `param`, i.e., threads are bound to cores numbered `param`, `param+1`, `param+2`, ... If this routine is called after `NicsLU_CreateThreads` is called, then all the created threads are bound to the specified cores; otherwise only the calling thread is bound to the specified core.
- `THREAD_PRIORITY` (value 1): changing threads' priority. The priority is used by the operating system for thread scheduling. A high priority may increase the performance when the system load is very high. The argument `param` specifies the priority, which can be an integer value in the interval of $[0, 99]$. 0 is the normal (default) priority and 99 is the highest priority. If this routine is called after `NicsLU_CreateThreads` is called, then the priority of all the created threads are changed; otherwise only the priority of the calling thread is changed.

NOTE: Elevating threads' priority typically requires the administrator/root permission.

NOTE: This routine is experimental. It is not recommended using this routine in practical applications, as binding threads to cores or changing threads' priority may dis-

turb the operating system's regular thread scheduling strategy, and thus, it may lead to performance degradation or other issues in some cases. A common case is that the operating system may response slow or even not response when the priority is set too high.

5.6.7 NicsLU_Factorize

```
int NicsLU_Factorize
(
    _handle_t solver,
    const _double_t *ax,
    int threads
);
```

This routine performs the numerical LU factorization (i.e., $\mathbf{A} = \mathbf{LU}$) with partial pivoting. It can be called after `NicsLU_Analyze` is called. The argument `ax` specifies the matrix data of the CSR storage, which must correspond to the index order passed into `NicsLU_Analyze`. The argument `threads` specifies the number of threads which will be used for LU factorizations. If `threads` is 0, this routine will use all the created threads. To perform a parallel factorization, `NicsLU_CreateThreads` must be called before.

LU factorization with partial pivoting involves strong data dependence and dynamic update of the dependence. As a result, parallel factorization cannot always have benefits than serial factorization. NICSLU has an adaptive feature that it can automatically judge whether a serial factorization or a parallel factorization should be used. To enable this feature, set `cfg[13]=1` (default). This feature is strongly recommended. This feature also affects `NicsLU_ReFactorize` and `NicsLU_FactorizeMatrix`. If the adaptive feature is enabled and NICSLU selects to use a serial factorization for the specified matrix, NICSLU always uses a serial factorization regardless how many threads are specified in this routine.

NICSLU has a feature of pivot perturbation, although it is not recommended. When factorizing an ill-conditioned matrix, some pivots may be too small so numerical instability may appear. If this feature is enabled, NICSLU will set small pivots to a bigger value if the pivot is smaller than a threshold. This feature also affects `NicsLU_ReFactorize` and `NicsLU_FactorizeMatrix`. However, pivot perturbation cannot be applied to complex number matrices.

This routine integrates a pivoting-reduction technique [9]. Subsequent calls to this routine may spend much less time than the first.

5.6.8 NicsLU_ReFactorize

```
int NicsLU_ReFactorize
(
    _handle_t solver,
    const _double_t *ax,
    int threads
);
```

If you want to factorize another matrix with different entry values but with the identical nonzero pattern, this routine can be used. This routine can be called after `NicsLU_Factorize` is called at least once. It does not perform partial pivoting, so it uses the pivoting order obtained in the last `NicsLU_Factorize` call. It runs faster than

NicsLU_Factorize, especially for extremely sparse matrices; however, it may cause numerical instability. See Section 6 for more details. To perform a parallel re-factorization, NicsLU_CreateThreads must also be called before.

5.6.9 NicsLU_Solve

```
int NicsLU_Solve
(
    _handle_t solver,
    _double_t *b,
    _double_t *x
);
```

This routine performs substitutions (i.e., $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$) to obtain the solution of $\mathbf{Ax} = \mathbf{b}$. It can be called after any factorization or re-factorization routine is called. Currently NICSLU only supports serial solving.

The argument b can be used for both input and output. On input, b is always the right-hand-side vector. If x is NULL, b will be overwritten by the solution on output. Otherwise b is not changed and x returns the solution on output.

If an application involves solving systems with identical $\mathbf{A}$ (for both symbolic pattern and numerical values) but different $\mathbf{b}$, then the factorization needs to be performed only once, and the substitutions are required for each system to compute the solutions. This happens in transient simulation of linear circuit with a fixed integration step length.

5.6.10 NicsLU_Refine

```
int NicsLU_Refine
(
    _handle_t solver,
    const _double_t *ax,
    const _double_t *b,
    _double_t *x
);
```

When necessary, this routine can be used to refine the solution to achieve a higher accuracy. However, it is not always successful.

The argument ax specifies the matrix data which must be identical to that is passed to the last call of NicsLU_Factorize or NicsLU_ReFactorize. The argument x should be the solution vector on input; on output, it will be updated by the refinement. The argument b is the right-hand-vector (input).

The refinement will not stop until the residual is less than a given tolerance, the number of iterations exceeds an allowed number, or the residual reaches the minimum. The refinement is implemented as follows:

compute residual $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$;
while $||\mathbf{r}||_2$ > eps && (iter++) < maxiter
    solve $\mathbf{Ad} = \mathbf{r}$;
    update solution $\mathbf{x} = \mathbf{x} - \mathbf{d}$;
    $r_0 = ||\mathbf{r}||_2$ and update residual $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$;
    if ($||\mathbf{r}||_2 > r_0$) break;

```
end while
```

`cfg[14]` is used to control the maximum number of iterations, and `cfg[15]` is used to control the precision requirement for the refinement.

5.6.11 NicsLU_Flops

```
int NicsLU_Flops
(
    _handle_t solver,
    int threads,
    _double_t *fflops,
    _double_t *sflops
);
```

This routine calculates the number of floating-point operations for a factorization and solving. The argument `threads` specifies the number of threads for factorizations (only used for calculating the workloads of each thread, regardless of the number of threads used for actual factorizations). The array `fflops` whose length must be larger than or equal to `threads` must be pre-allocated by users. On output, `fflops` stores the number of floating-point operations of each thread in a factorization. If `threads` is 1, the returned result is the total number of floating-point operations in a factorization. The argument `sflops` returns the number of floating-point operations in a solving. Call this routine after a factorization. Please note that `threads` passed to this routine can be different from that passed to `NicsLU_CreateThreads`. `NicsLU_CreateThreads` is not required to call before calling this routine.

5.6.12 NicsLU_GetFactors

```
int NicsLU_GetFactors
(
    _handle_t solver,
    _double_t *lx,
    _uint_t *li,
    _size_t *lp,
    _double_t *ux,
    _uint_t *ui,
    _size_t *up,
    _bool_t sort,
    _uint_t *row_perm_inv,
    _uint_t *col_perm_inv,
    _double_t *row_scale_inv,
    _double_t *col_scale_inv
);
```

This routine extracts the factorized LU factors and stores them in the CSR format. Users must pre-allocate memories for `lx`, `li`, `lp`, `ux`, `ui`, and `up`. The numbers of nonzeros in **L** and **U** can be obtained from `stat[9]` and `stat[10]`, which can be used to pre-allocate these arrays. The dumped CSR arrays contain the diagonals of **L** and **U**. The argument `sort` indicates whether to sort the indexes of **L** and **U**. The last four arrays retrieve the permutations arrays and scaling factors, if they are not `NULL`.

NOTE: Due to the limited precision of double-precision floating-point numbers, when you are pre-allocating memories for `lx`, `li`, `lp`, `ux`, `ui`, and `up` according to the values of `stat[9]` and `stat[10]`, it is recommended that you allocate a bit more memories because a floating-point may not hold all the significant digits.

NOTE: The dumped factors are stored in the permuted and pivoted order, but not in the original order. The values are scaled if static or dynamic scaling is enabled. However, the retrieved permutation and scaling arrays are in the original (un-permuted) order.

5.6.13 NicsLU_ConditionNumber

```
int NicsLU_ConditionNumber
(
    _handle_t solver,
    const _double_t *ax,
    _double_t *cond
);
```

This routine estimates the condition number of the matrix. Call this routine after a factorization. The matrix values `ax` must be identical to those passed into the last factorization or re-factorization.

5.6.14 NicsLU_MemoryUsage

```
int NicsLU_MemoryUsage
(
    _handle_t solver,
    _size_t *mem
);
```

This routine estimates the virtual memory allocated by NICSLU. The actual physical memory usage is less than the reported value. The reported memory usage is estimated in bytes.

5.6.15 NicsLU_Performance

```
int NicsLU_Performance
(
    _handle_t solver,
    int tsync,
    int threads,
    _double_t *perf
);
```

This routines estimates the parallel performance of NICSLU for the last factorized matrix. The argument `tsync` specifies a hypothetical cost of inter-thread synchronization (in the unit of floating-point operation). Its typical value ranges from 10 to 100. The next argument `threads` specifies the number of threads, which has no relation with the actual used number of threads in parallel factorizations. The last argument `perf` returns the estimated performance. Its length should be at least 4, and it returns the following 4 performance estimations.

- `perf[0]`: theoretical ultimate speedup, regardless of the number of threads.

- `perf[1]`: maximum speedup when using the specified number of threads.
- `perf[2]`: ratio of waiting time.
- `perf[3]`: ratio of inter-thread synchronization time.

5.6.16 NicsLU_Determinant

```
int NicsLU_Determinant
(
    _handle_t solver,
    _double_t *coef,
    _double_t *expn
);
```

This routine estimates the determinant of the matrix. It can be called after the matrix is factorized. The arguments `coef` and `expn` return the coefficient and exponent parts of the determinant, respectively, which is expressed by the scientific notation, i.e.,

$$|\mathbf{A}| = \texttt{coef} \times 10^{\texttt{expn}}. \tag{3}$$

Note that for complex number matrices, the argument `coef` should be a `_complex_t` (cast it to a `_double_t` pointer when calling this routine).

5.7 High-Level Routines

In addition to the above routines, NICSLU also provides 2 high-level routines which are easier to use.

5.7.1 NicsLU_FactorizeMatrix

```
int NicsLU_FactorizeMatrix
(
    _handle_t solver,
    const _double_t *ax,
    int threads
);
```

This routine performs a numerical LU factorization or re-factorization. Whether to call factorization or re-factorization is determined by a built-in heuristic method. If a re-factorization fails, it will call a factorization automatically. The argument `threads` specifies the number of threads which will be used for an LU factorization. If `threads` is 0, this routine will use all the created threads.

`cfg[17]` is a threshold used to control whether to call factorization or re-factorization. Decreasing `cfg[17]` will increase the chance to call a factorization with partial pivoting. If `cfg[17]=0`, this routine always calls a factorization.

5.7.2 NicsLU_SolveAndRefine

```
int NicsLU_SolveAndRefine
(
    _handle_t solver,
    const _double_t *ax,
```

```
    const _double_t *b,
    _double_t *x
);
```

This routine is a combination of `NicsLU_Solve` and `NicsLU_Refine`. The difference between this routine and separately calling `NicsLU_Solve` and `NicsLU_Refine` is that this routine can automatically check whether an iterative refinement is required by a heuristic method.

`cfg[16]` is a threshold used to control whether an iterative refinement will be executed. Decreasing `cfg[16]` will increase the chance to do an iterative refinement.

5.8 Utility Routines

NICSLU also provides 5 utility routines. Utility routines are without the prefix `NicsLU_`. Calling the utility routines does not require a license.

5.8.1 SparseResidual

```
int SparseResidual
(
    _uint_t n,
    const _double_t *ax,
    const _uint_t *ai,
    const _uint_t *ap,
    const _double_t *b,
    const _double_t *x,
    _double_t *res,
    _matrix_type_t mtype
);
```

This routines calculates the residual error $\|\mathbf{Ax}-\mathbf{b}\|$ or $\left\|\mathbf{A}^{\mathrm{T}}\mathbf{x}-\mathbf{b}\right\|$. It supports both real number and complex number matrices. The last argument specifies the type of the matrix. Please refer to Section 5.6.3 to see its possible values. The argument `res` returns the residual error. Its length should be at least 4, and it returns the following four values.

- `res[0]`: root-mean-square error of the residual vector.
- `res[1]`: L_1-norm of the residual vector.
- `res[2]`: L_2-norm of the residual vector.
- `res[3]`: L_∞-norm of the residual vector.

5.8.2 SparseTranspose

```
int SparseTranspose
(
    _uint_t n,
    _double_t *ax,
    _uint_t *ai,
    _uint_t *ap,
    _transpose_t op
);
```

This routine transposes a matrix. It supports both real number and complex number matrices. The three arguments `ax`, `ai`, and `ap` are used for both input and output. The last argument `op` is an enumeration variable to specify the operation. It can be one of the following three values.

- `TRANSPOSE_REAL` (value 0): for real number matrices.
- `TRANSPOSE_COMPLEX` (value 1): for complex number matrices. This routines calculates the normal transposition.
- `TRANSPOSE_COMPLEX_CONJ` (value 2): for complex number matrices. This routines calculates the conjugate transposition.

5.8.3 ReadMatrixMarket

```
int ReadMatrixMarket
(
    const char *file,
    _uint_t *row,
    _uint_t *col,
    _uint_t *nnz,
    _double_t *ax,
    _uint_t *ai,
    _uint_t *ap,
    int is_complex
);
```

This routine reads a matrix from a matrix market formatted file. For details of the matrix market format, please refer to `http://math.nist.gov/MatrixMarket/formats.html`. Please follow the following two steps to use this routine, unless the lengths of `ax`, `ai` and `ap` are all known in advance.

- Set `ax`, `ai`, and `ap` to `NULL`, then this routine will return the numbers of rows, columns, and nonzeros, namely, `row`, `col`, and `nnz`.
- Allocate memories for `ax`, `ai`, and `ap` according to the returned numbers of rows, columns, and nonzeros, and then call this routine again to read the matrix data. If the matrix is a vector, `ai` and `ap` can be `NULL`, and `ax` must be a dense vector (regardless of whether the vector is stored in a sparse or dense format).

NOTE: The matrix market format stores sparse matrices in the column-major order. The resulting compressed storage is also in the column-major order, namely, CSC. This routine does not perform any transposition.

NOTE: The matrix market format is one-based indexed; however, the resulting arrays are converted to zero-based indexed.

5.8.4 SparseHalfToSymmetricFull

```
int SparseHalfToSymmetricFull
(
    _uint_t n,
    const _double_t *ax,
    const _uint_t *ai,
```

```
    const _uint_t *ap,
    _double_t *aax,
    _uint_t *aai,
    _uint_t *aap,
    int is_complex
);
```

If a matrix is symmetric, typically only half of the matrix (either the lower part or the upper part, including the diagonal) is stored. This routine is used to restore the full matrix in the CSR format. Before calling it, memories for `aax`, `aai`, and `aap` must be pre-allocated. After calling, they return the resulting full symmetric matrix. This routine does not check the validity of the input matrix.

Two typical examples of such matrices are G2_circuit and G3_circuit from the SuiteSparse matrix collection [19].

5.8.5 `PrintNicsLULicense`

```
int PrintNicsLULicense
(
    void (*fptr)(const char *)
);
```

This routine prints the license information. The argument `fptr` specifies a function pointer which is used for string output. If it is `NULL`, this routine uses the default `stdout`. In other words, it prints the license information on the screen.

6 Using NICSLU in Circuit Simulation

This section will illustrate the basic methodology to integrate NICSLU into a SPICE-style circuit simulator. NICSLU has two different usages in a SPICE-style circuit simulator, depending on whether low-level or high-level routines are used.

6.1 Using Low-Level Routines

In SPICE-style circuit simulation, the matrix is solved many times with an exactly identical symbolic pattern but different numerical values (i.e., in Newton-Raphson iterations and in TRAN iterations). Therefore, `NicsLU_Analyze`, `NicsLU_CreateThreads` **are needed ONLY ONCE**. Numerical LU factorizations and right-hand-solvings are performed many times, as shown in Figure 2.

For numerical factorization, the difference between `NicsLU_Factorize` and `NicsLU_ReFactorize` is that the former performs partial pivoting and the latter does not. During Newton-Raphson iterations, if matrix values change little, `NicsLU_ReFactorize` can be always used because it uses the pivoting order generated by the first `NicsLU_Factorize` and this will not lead to numerical instability. However, if the matrix values change much, `NicsLU_ReFactorize` may result in numerical instability. In this case, `NicsLU_Factorize` should be called. We suggest that NICSLU is used in the manner shown in Figure 2. When the Newton-Raphson iteration is converging, `NicsLU_ReFactorize` can be used, otherwise `NICSLU_Factorize` should be used. Generally speaking, `NicsLU_ReFactorize` is called much more times than `NicsLU_Factorize` in a transient simulation.

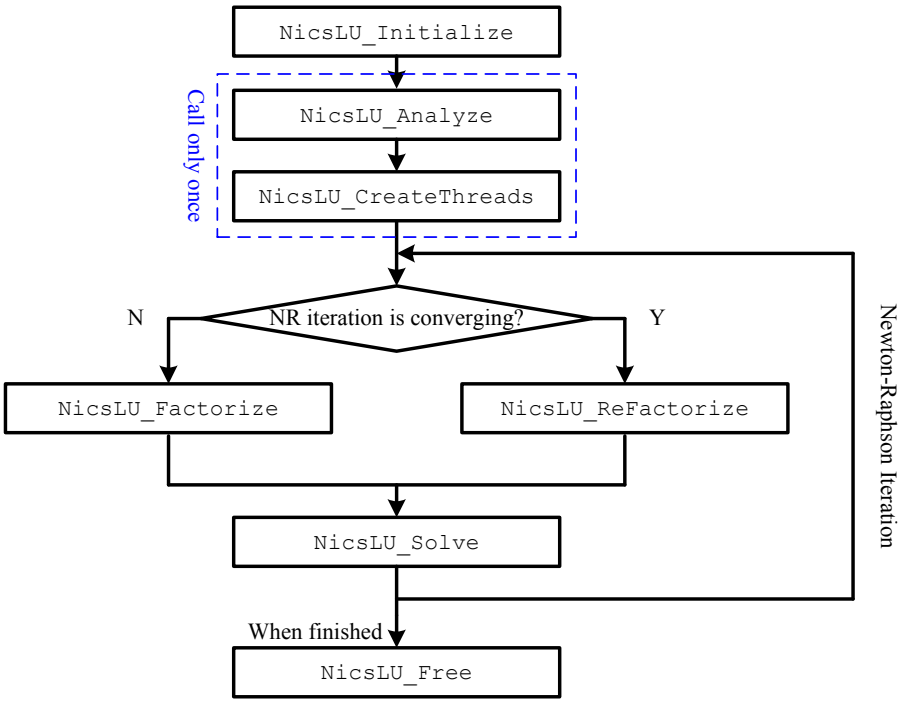


Figure 2: How to use NICSLU in circuit simulation (using low-level routines).

If you are using low-level routines, you need to judge whether to call `NicsLU_Factorize` or `NicsLU_ReFactorize` in each iteration by yourself. We provide a tricky method which utilizes the convergence check method in a SPICE-style circuit simulator. In SPICE-style circuit simulators, the following method is usually used to check whether the Newton-Raphson iteration is converged:

$$|x_k - x_{k-1}| < \text{AbsTol} + \text{RelTol} \times \min\{|x_k|, |x_{k-1}|\} \tag{4}$$

where AbsTol and RelTol are the given absolute and relative tolerances for checking convergence in SPICE. Since the Newton-Raphson iteration has a feature of quadratic convergence, we can simply relax the two tolerances to larger values to judge whether the Newton-Raphson iteration is converging:

$$|x_k - x_{k-1}| < \text{BigAbsTol} + \text{BigRelTol} \times \min\{|x_k|, |x_{k-1}|\} \tag{5}$$

where BigAbsTol >> AbsTol and BigRelTol >> RelTol. They can be determined empirically. Based on our experience, BigRelTol can be near 1.0. Equation (5) can be used to determine whether to call `NicsLU_Factorize` or `NicsLU_ReFactorize`. If Equation (5) holds, call `NicsLU_ReFactorize`, otherwise call `NicsLU_Factorize`.

6.2 Using High-Level Routines

If you are using high-level routines of NICSLU in circuit simulation, the usage is never so easy! Figure 3 illustrates the usage. There is no need to determine whether to call factorization or re-factorization by yourself.

NOTE: The heuristic method built in `NicsLU_FactorizeMatrix` may not work well under some extreme cases. Using low-level routines with carefully decided tolerances can be more robust than using high-level routines. Our suggestion is to use low-level routines in DC simulations and use high-level routines in TRAN simulations.

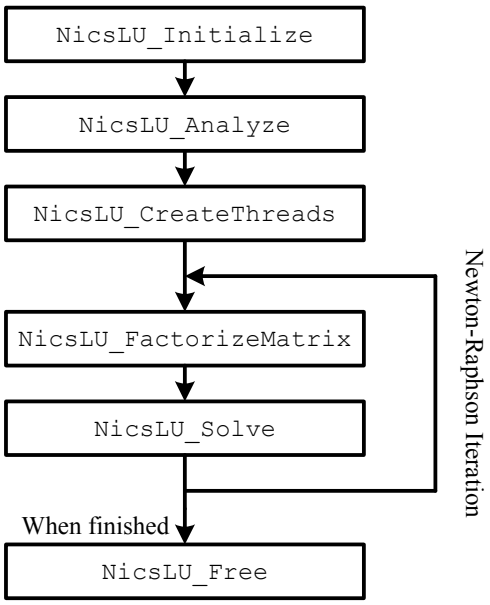


Figure 3: How to use NICSLU in circuit simulation (using high-level routines).

If a divergence happens when using high-level routines, users can try to decrease `cfg[16]`. Decreasing `cfg[16]` will increase the chance to call factorizations with partial pivoting.

7 Using NICSLU Libraries

This section introduces the system requirements and how to use the NICSLU libraries in your program.

7.1 System Requirements

NICSLU can be used on Intel x86 or AMD64 (x86_64) hardware platforms. Both Windows and GNU Linux are supported. For Linux, NICSLU relies on some gcc built-in functions so gcc or gcc-compatible compilers (e.g., icc) are required. In addition, any UNIX-like system which has gcc and supports the POSIX standard should also be supported; however, we have never tested NICSLU on any UNIX system. Table 5 shows the minimum system requirements of NICSLU.

Table 5: Minimum system requirements of NICSLU.

	Windows	GNU Linux
Architecture	x86, x86_64	
Operating system	Windows XP SP3	POSIX-compatible
Compiler	Visual C++ 2005 or 6.0[a]	gcc 4.1.2
Runtime	None	glibc 2.5[b]

[a] Visual C++ 6.0 only supports the 32-bit mode, so if you want to use 64-bit libraries, Visual C++ (Studio) 2005 or higher version is required.

[b] This is the theoretical minimum requirement. The detailed glibc version depends on the compilation platform.

7.2 Bit Width of Binaries and Integers

Basically, depending on the compilation options and whether the macro `__NICS_INT64` is defined, the NICSLU libraries have three different modes. Please note that when using NICSLU libraries, whether the macro `__NICS_INT64` should be defined or not must be consistent with the compilation option. This means that, we will tell customers whether `__NICS_INT64` should be defined when distributing NICSLU libraries. Under different modes, the routine names of NICSLU keep the same. The differences are in integer-related argument types, i.e., it affects the definition of `_size_t`, `_int_t` and `_uint_t`.

- 32-bit binary. In this mode, libraries are compiled into 32-bit binaries and can be used on both 32-bit and 64-bit machines. Integers can only be 32-bit (i.e., `__NICS_INT64` cannot be defined). This is because that 64-bit integer is not necessary in 32-bit binaries, as the memory usage of a matrix whose dimension or number of nonzeros is larger than `0xFFFFFFFF` will certainly exceed the 4G memory limit of a 32-bit process. Also for this reason, the matrix cannot be too large in this mode, otherwise the memory requirement may exceed the 4G limit.
- 64-bit binary and 32-bit integer. This is the most common mode of NICSLU we have distributed. Binaries can be used on 64-bit machines and do not have the 4G memory limit. In this mode, the dimension and the number of nonzeros of the original matrix **A** cannot exceed `0xFFFFFFFF` (the actual limit is smaller than this number). However, the size for LU factors does not have this limit. This mode uses 64-bit integers to store LU factors so the limit of the number of nonzeros in the factors only depends on the maximum virtual memory space. In other words, the 32-bit integer only limits the size of the original matrix **A**. This mode should work in most cases.
- 64-bit binary and 64-bit integer. `__NICS_INT64` is defined in this mode. All integers are 64-bit so the only limitation comes from the maximum virtual memory space. This mode is only required when the original matrix **A** is too large such that 32-bit integers cannot hold it.

7.3 Using NICSLU on Windows

Add `nicslu.lib` to "Additional Dependencies" of the configuration of Visual Studio, or add the code

```
#pragma comment(lib, "nicslu.lib")
```

to any position of your codes. Put `nicslu.lib` into the project directory when compiling and linking. When running the program, put `nicslu.dll` along with the binary. `nicslu.lib` is only required in linking but not required when running the executable.

7.4 Using NICSLU on Linux

Link with the option `-L<path of NICSLU libraries> -lnicslu`. Also add the following three system libraries when linking: `-lpthread`, `-lm`, and `-ldl`. Some systems may need another library: `-lrt`. When running the program, set the environment variable `LD_LIBRARY_PATH` to contain the path where the NICSLU libraries locate. If you cannot link NICSLU successfully on Linux, the typical reason is that the version of your gcc and/or glibc is too low. Please consider to update your gcc and/or glibc.

7.5 Using Vendor Optimized BLAS

NICSLU can explore higher performance by using vendor optimized BLAS when factorizing denser matrices. For an alternative choice when there is no available vendor optimized BLAS, NICSLU provides a built-in BLAS. The built-in BLAS is slightly slower than highly optimized BLAS. When NICSLU requires an external BLAS which is not statically linked into the NICSLU library, the BLAS library must be provided in the link process. For example, if you are using Intel MKL BLAS, three MKL libraries should be added: `-lmkl_intel_lp64` (or `-lmkl_intel` on 32-bit platforms, and `-lmkl_intel_ilp64` when `__NICS_INT64` is defined on 64-bit platforms), `-lmkl_sequential` and `-lmkl_core`. We will tell whether NICSLU requires a vendor optimized BLAS when distributing it. Typically, NICSLU libraries are statically linked with Intel MKL BLAS so no explicit BLAS is required. If an explicit BLAS is required, please see the following notes. We strongly recommend Intel MKL BLAS for both Windows and Linux. Other BLAS libraries, like OpenBlas, perform poorly on Windows.

NOTE: NICSLU uses the Fortran-interface BLAS. CBLAS is not supported.

NOTE: Only serial BLAS can be used. NICSLU performs all the parallelization. The BLAS must be thread-safe.

NOTE: If NICSLU is compiled with `__NICS_INT64` defined and vendor optimized BLAS is used, the integer used in BLAS must also be 64-bit (i.e., using the ILP64 model).

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] X. Chen, L. Xia, Y. Wang, and H. Yang. Sparsity-Oriented Sparse Solver Design for Circuit Simulation. In *Design, Automation, and Test in Europe (DATE) 2016*, pages 1580–1585, 14-18 March 2016.

[4] 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.

[5] 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.

[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. NICSLU: 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.

[8] 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.

[9] X. Chen, Y. Wang, and H. Yang. A Fast Parallel Sparse Solver for SPICE-based Circuit Simulators. In *Design, Automation, and Test in Europe (DATE)*, pages 205–210, 9-13 march 2015.

[10] X. Chen, Y. Wang, and H. Yang. *Parallel Sparse Direct Solver for Integrated Circuit Simulation*. Springer Publishing, 1st edition, 2017.

[11] 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.

[12] 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.

[13] 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.

[14] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, April 2004.

[15] P. R. Amestoy, T. A. Davis, and I. S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, October 1996.

[16] C. Mészáros. The "Inexact" Minimum Local Fill-In Ordering Algorithm. Technical report, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, 1995.

[17] T. A. Davis, j. R. Gilbert, S. I. Larimore, and E. G. Ng. A Column Approximate Minimum Degree Ordering Algorithm. *ACM Trans. Math. Softw.*, 30(3):353–376, September 2004.

[18] G. Karypis and V. Kumar. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University of Minnesota, August 1995.

[19] SuiteSparse Matrix Collection. [Online] https://sparse.tamu.edu/.