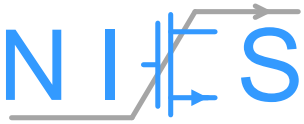
User Guide for NICSLU

—A Parallel Sparse Direct Solver for Circuit Simulation

(Version 202103)

Xiaoming Chen
chenxiaoming@ict.ac.cn

March 24, 2021



Nanoscale Integrated Circuit
and System (NICS) Laboratory

Department of Electronic Engineering
Tsinghua University

Contents

1 License

NICSLU, Copyrighted by NICS Laboratory, Tsinghua University and Xiaoming Chen. NICSLU is protected by three Chinese patents (CN201110337789.6, CN201110076027.5, and CN201110088086.4).

NICSLU is distributed 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 Key File

In most cases, running NICSLU requires a valid license key file. The license key file is of size 512 bytes. There are two alternative ways to set the license key file.

- Set the environment variable `NICSLU_LICENSE` to point to the license key file. The value of the environment variable must contain the full path with the file name, e.g., `NICSLU_LICENSE=/home/<USER NAME>/nicslu.lic`.
- (Simpler method) If `NICSLU_LICENSE` is not set, make sure that the license key file is named `nicslu.lic` and put it together with the executable (with the original target executable instead of any symbolic link) or with 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

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

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 proposed many advanced techniques to enhance the performance of NICSLU. NICSLU features the following advanced techniques.

- Adaptive numerical kernel selection [3];
- Static scaling [4, 5];
- Parallelization and scheduling for LU factorization [6–8];
- Pivoting reduction [9];
- Advanced heuristic ordering methods [10, 11].

Most techniques and algorithms used in NICSLU are described in detail in our book [12]. NICSLU supports both **real number** and **complex number** matrices. Native real number and complex number kernels are both integrated in NICSLU.

There are also some other popular software packages which do the same thing, such as SuperLU [13–15], PARDISO [16], 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 proven 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 intend to use NICSLU is a SPICE-style circuit simulator, please read Section 6 carefully.**

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` (Section 5.6.1)
  This routines first checks the license. If license check is passed, it creates the handle of the solver.
- `NicsLU_Analyze` (Section 5.6.3)
  This routine pre-orders the matrix to reduce fill-ins and then performs a symbolic factorization.
- `NicsLU_FactorizeMatrix` (Section 5.7.1)
  This routine performs the numerical factorization to compute the LU factors. One can also call `NicsLU_Factorize` (Section 5.6.7) or `NicsLU_ReFactorize` (Section 5.6.8) to perform the numerical factorization. For the first factorization, it is equivalent to `NicsLU_Factorize`.
- `NicsLU_Solve` (Section 5.6.9)
  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` (Section 5.6.2)
  This routine frees all objects and destroys the handle of the solver.

To solve multiple linear systems successively, the first routine (`NicsLU_Initialize`) and the last routine (`NicsLU_Free`) are not necessarily called for each linear system. Instead, a handle created by `NicsLU_Initialize` can be used for multiple linear systems.

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 key file according to the methods mentioned in Section 1.1. We also provide other demos in the `demo` folder of the release package.

```
1  #include <stdio.h>
2  #include "nicslu.h"
3
4  int main()
5  {
6      _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 };
7      _uint_t ai[13] = { 0, 3, 4, 1, 4, 1, 2, 3, 2, 4, 0, 3, 5 };
8      _uint_t ap[7] = { 0, 3, 5, 7, 8, 10, 13 };
9      _double_t b[6] = { 35.95, 53.9, 7.7, -17.6, 60.83, 98.18 };
10     _handle_t ctx = NULL;
11     _uint_t i;
12
13     if (__FAIL(NicsLU_Initialize(&ctx, NULL, NULL, NULL)))
14     {
15         printf("Failed to initialize\n");
```

```
16          return -1;
17      }
18
19      NicsLU_Analyze(ctx, 6, ax, ai, ap, MATRIX_ROW_REAL, NULL,
            NULL, NULL, NULL);
20      NicsLU_FactorizeMatrix(ctx, ax, 1);
21      NicsLU_Solve(ctx, b, NULL);
22      for (i = 0; i < 6; ++i) printf("x[%d] = %g\n", i, b[i]);
23
24      NicsLU_Free(ctx);
25      return 0;
26  }
```

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 $\mathbf{x} = [1, 2, 3, 4, 5, 6]^\mathsf{T}$. 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. NICSLU only accepts square matrices, i.e., `n`×`n`.
- `nnz`: an (unsigned) integer, 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,

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

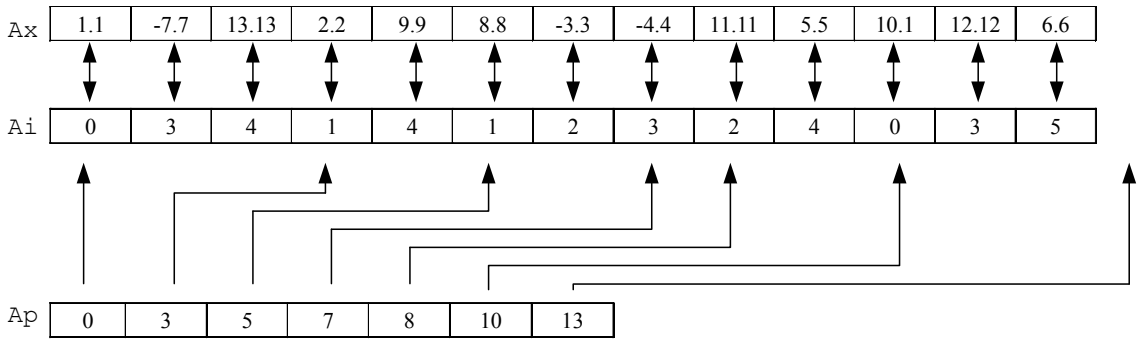


Figure 1: Example of the CSR format.

each element of `Ax[]` is a complex number, i.e., two consecutive 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. Elements of `Ax[]` and `Ai[]` must be one-to-one matched. `Ai[]` means "**A**'s indexes".
- `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 `Ap[n]=nnz` respectively. Values of the ith row of the matrix are stored in `Ax[Ap[i]], Ax[Ap[i]+1], ···, Ax[Ap[i+1]-1]`, and the corresponding column indexes of the nonzeros are stored in `Ai[Ap[i]], Ai[Ap[i]+1], ···, Ai[Ap[i+1]-1]`, number of nonzeros of the ith row is `Ap[i+1]-Ap[i]`. `Ap[]` means "**A**'s row pointers".

NICSLU uses CSR by default. However, 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. If your matrix is stored in CSC format, NICSLU can also directly handle it. In the CSC case, NICSLU solves $\mathbf{A}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$ 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 accept real number arrays, so for a complex number matrix, `Ax[]` should be casted to a real number pointer (i.e., `double *`).

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

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

NOTE: The CSR storage must not have symbolically 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

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

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 unsigned integer
_complex_t	double [2][b]	Complex number
_size_t	size_t	Returning type of sizeof

[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.

the default configurations. The 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 the 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.

5.2.1 Guidance on Adjusting Configurations

The optimal configurations for best performance depend on the hardware and characteristics of the matrix. Here the detailed functionalities of the configuration parameters are explained.

cfg[0]: timer control (default 0). A zero value disables the timer. A positive value enables a high-precision timer and a negative value enables a low-precision timer. The calling overhead of the built-in timer, especially the high-precision 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. If the timer is disabled, any runtime statistics (see Table 3) will report 0.

cfg[1]: partial pivoting threshold (default 0.001). During an LU factorization with partial pivoting, at row i, if $U_{ii} \geq \texttt{cfg[1]} \times \max\{\mathbf{U}_{i*}\}$, then the diagonal element is select as the pivot; otherwise the element with the largest absolute value in row i of $\mathbf{U}$ is selected as the pivot. Increasing cfg[1] introduces more off-diagonal pivots, which typically increase the number of factors, but at the same time, the accuracy of the solution tends to be higher.

cfg[2]: synchronization method (default -1). This parameters controls how the threads wait for tasks in a multi-thread execution. A zero or negative value enables blocked wait and a positive value enables spin wait. Spin wait has higher performance but the threads always consume CPU even when there is no task. On the contrary, blocked wait releases CPU when there is no task.

cfg[3]: ordering method (default 2). See Section 5.6.3 for details.

cfg[4]: threshold for detecting dense nodes in ordering (default 10.0). A row/column with more than $\texttt{cfg[4]} \times \sqrt{\texttt{n}}$ nonzeros is treated as a dense node. To save

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	2	0∼8	Ordering method. 0: no ordering; 1: user ordering; 2: selects the best one among all the built-in methods in parallel; 3: selects the best one among all the built-in methods in sequential; 4: AMD; 5: AMM; 6: AMO1; 7: AMO2; 8: AMO3; See Section 5.6.3 for more details
4	10.0	any	Threshold for dense row detection in ordering
5	0	0/1	Whether to do static scaling
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 for creating supernodes
9	4	4∼100	Number of initial rows for 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	Whether to 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 threshold 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
20	0	any	Garbage collection is performed per cfg[20] iterations. 0 means no garbage collection
21	0	0/1	Whether to use fast solving for very sparse right hand vector (only for $\mathbf{A}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$ mode)
22	1	any	Metric for selecting the best ordering method. ≥0: use # of flops; <0: use # of nonzeros

ordering time, all dense nodes are not ordered but are directly put at last, sorted by the number of nonzeros.

cfg[5]: static scaling control (default 0). Static scaling scales the matrix such that the diagonal elements become ±1 and the absolute values of all off-diagonal elements are not larger than 1. This functionality is generally good for ill-conditioned matrices (with a large condition number) but has little effect for well-conditioned matrices. Note that the scaling vectors are generated in the pre-processing stage based on the matrix values specified to NicsLU_Analyze, so for different matrix values specified to NicsLU_Factorize, NicsLU_ReFactorize or NicsLU_FactorizeMatrix, static scaling cannot achieve the same goal. This option is turned off by default, as circuit matrices are always changing the values during Newton-Raphson iterations.

`cfg[6]`: **parameter for pre-allocating memory (default 4)**. If this parameter is larger, NICSLU will pre-allocate more memory for the factors, so that the possibility of run-time memory growth (by `realloc`) will be reduced, but there may be more redundant memory.

`cfg[7]`: **dynamic memory growth factor (default 1.5)**. Once the memory for the factors is insufficient, NICSLU grows the memory at run-time (by `realloc`) by increasing the memory size to `cfg[7]` times of the currently allocated size.

`cfg[8]`: **minimum number of columns for creating supernodes (default 80)**. A supernode must have at least `cfg[8]` columns. Otherwise even it satisfies the conditions of supernode it will not be treated as a supernode.

`cfg[9]`: **number of initial rows for supernodes (default 4)**. Once a new supernode is created, NICSLU allocates `cfg[9]` rows of memory for the supernode. The significance of this parameter is similar to that of `cfg[6]`.

`cfg[10]`: **pipeline scheduling control (default 8)**. A level in the scheduling graph with less than `cfg[10]`$\times$(`# of threads`) nodes indicates that this level has low intra-level parallelism. Instead, the level is scheduled in a pipelined fashion by exploiting inter-level parallelism.

`cfg[11]`: **load balance factor for pipeline (default 0.95)**. This parameter is used to control the workloads for threads in a parallel factorization or re-factorization. Typically a value close to 1 is good.

`cfg[12]`: **dynamic scaling control (default 0)**. NICSLU can scale all columns in every factorization or re-factorization based on the per-factorization matrix values. Similar to KLU, NICSLU also provides two kinds of dynamic scaling, sum-scaling and max-scaling. A zero value (default) disables dynamic scaling. A positive value enables max-scaling which uses the maximum value of each column to scale the corresponding column, and a negative value enables sum-scaling which uses the absolute sum of each column to scale the corresponding column. This functionality only has effect for a small portion of matrices but has no effect for most matrices.

`cfg[13]`: **automatic sequential/parallel factorization selection (default 1)**. In SPICE-based circuit simulations, matrices created by modified nodal analysis are typically highly sparse and involve a very low computation-to-communication ratio. As a result, not every matrix is suitable for parallel factorization or re-factorization. NICSLU can automatically determine whether a matrix can get speedup from parallel factorization or re-factorization. This feature is strongly recommended.

`cfg[14]`: **maximum number of iterations for iterative refinement (default 3)**. The iterative refinement (if performed) will stop if the residual is below `cfg[15]` or the number of iterations reaches `cfg[14]`. Iterative refinement is not recommended for good-conditioned matrices.

`cfg[15]`: **residual threshold for iterative refinement (default 10^{-10})**. The iterative refinement (if performed) will stop if the residual is below `cfg[15]` or the number of iterations reaches `cfg[14]`.

`cfg[16]`: **Pseudo condition number threshold for doing refinement (default 10^{12})**. NICSLU can automatically determine whether an iterative refinement is needed or can improve the accuracy of the solution. The determination is based on an estimated pseudo condition number. Decreasing `cfg[16]` will increase the chance to do an iterative refinement.

`cfg[17]`: **Threshold to determine whether to call re-factorization or fac-**

torization in `NicsLU_FactorizeMatrix` (default 5). In `NicsLU_FactorizeMatrix`, NICSLU automatically determines to call factorization with partial pivoting or re-factorization without partial pivoting. Decreasing `cfg[17]` will increase the chance to call a factorization with partial pivoting. If `cfg[17]=0`, `NicsLU_FactorizeMatrix` always calls factorization with partial pivoting.

`cfg[18]`**: Pivot perturbation threshold (default 0)**. If the absolute value of a pivot is smaller than `cfg[18]`, NICSLU will set it to `cfg[18]` or `-cfg[18]` according to its original sign. This feature is called "pivot perturbation". A zero or negative value disables pivot perturbation. This feature is not recommended as pivot perturbation impairs the accuracy of the solution and an iterative refinement is needed.

`cfg[19]`**: garbage collection threshold (default 2)**. During Newton-Raphson iterations of a circuit simulation, due to the change of the matrix values, and in turn, the increase of the LU factors, the memory for the factors may be grown. NICSLU can free such redundant memory, which is called "garbage collection". If the allocated memory size is larger than `cfg[19]` times of the required memory size, NICSLU can perform garbage collection to free redundant memory every `cfg[20]` iterations.

`cfg[20]`**: period of garbage collection (default 0)**. NICSLU performs garbage collection every `cfg[20]` iterations. A zero value disables garbage collection.

`cfg[21]`**: fast solving control (default 0)**. NICSLU supports fast solving for very sparse right-hand-side vector $\mathbf{b}$ (only for $\mathbf{A}^{\mathsf{T}}\mathbf{x} = \mathbf{b}$ mode). Based on our experience, if the number of nonzeros in $\mathbf{b}$ is less than `n/10`, fast solving can improve the performance. Otherwise fast solving can even degrade the performance.

`cfg[22]`**: metric for selecting the best ordering method (default 1)**. NICSLU can use estimated number of nonzeros or floating-point operations to select the best ordering method. A zero or positive value makes NICSLU to use the number of floating-point operations and a negative value makes NICSLU to use the number of nonzeros. Note that in both cases, the estimated number of nonzeros or floating-point operations may not be very accurate so that NICSLU may select an ordering method that is actually not the best. This option generally has a small impact on the ordering performance.

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 up to microsecond (μs).

`stat[0]`: runtime of `NicsLU_Analyze`. Only set when the function succeeds.

`stat[1]`: runtime of `NicsLU_Factorize`, `NicsLU_ReFactorize` or `NicsLU_FactorizeMatrix`. Only set when the function succeeds.

`stat[2]`: runtime of `NicsLU_Solve` or `NicsLU_SolveAndRefine`. Only set when the function succeeds. For `NicsLU_SolveAndRefine`, the runtime statistics includes refinement.

`stat[3]`: number of floating-point operations estimated by `NicsLU_Analyze`. Only set when the function succeeds.

`stat[4]`: number of LU factors estimated by `NicsLU_Analyze`. Only set when the function succeeds.

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

Index	Description
0	Runtime of the last pre-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	Height of ETree
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
16	Selected best ordering method (4 to 8 correspond to AMD, AMM, AMO1, AMO2 and AMO3, respectively)
17	Selected algorithm (1: row-row; 2: partial supernode; 3: full supernode)
18	Singular row index (in original order)
29	License expiration date (format: YYYYMMDD)
30	Compilation time (format: YYYYMMDD.HHMMSS)
31	Version

`stat[5]`: height of elimination tree reported by `NicsLU_CreateThreads`. Only set when the function succeeds.

`stat[6]`: runtime of iterative refinement performed by `NicsLU_Refine`. Only set when the function succeeds. `NicsLU_SolveAndRefine` does not set this parameter.

`stat[7]`: number of iterations performed by iterative refinement. Set by `NicsLU_Refine` or `NicsLU_SolveAndRefine`. Only set when the function succeeds.

`stat[8]`: number of LU factors reported by `NicsLU_Factorize` or `NicsLU_FactorizeMatrix`. Only set when the function succeeds.

`stat[9]`: number of L factors reported by `NicsLU_Factorize` or `NicsLU_FactorizeMatrix`. Only set when the function succeeds.

`stat[10]`: number of U factors reported by `NicsLU_Factorize` or `NicsLU_FactorizeMatrix`. Only set when the function succeeds.

`stat[11]`: number of off-diagonal pivots reported by `NicsLU_Factorize` or `NicsLU_FactorizeMatrix`. Only set when the function succeeds.

`stat[12]`: number of supernodes reported by `NicsLU_Factorize` or `NicsLU_FactorizeMatrix` when the selected algorithm is supernode. Only set when the function succeeds.

`stat[13]`: number of perturbed pivots reported by `NicsLU_Factorize`, `NicsLU_ReFactorize` or `NicsLU_FactorizeMatrix` when pivot perturbation is enabled. Only set when the function succeeds.

`stat[14]`: number of factorizations with partial pivoting performed.

`stat[15]`: number of re-factorizations without partial pivoting performed.

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	Invalid pointer
-3	Out of memory, no enough virtual memory (`malloc` or `realloc` fails)
-4	Structurally singular
-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	Failed to create threads
-10	Matrix has not been analyzed
-11	Matrix has not been 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
-15	Functionality not supported
-255	An unknown failure has occurred. Please contact the author
+1	Setting thread schedule failed
+2	Static scaling is invalid
+3	The same number of threads have already been created
+4	Incorrect file name

`stat[16]`: selected best ordering method reported by `NicsLU_Analyze`. Only set when the function succeeds.

`stat[17]`: selected algorithm reported by `NicsLU_Analyze`. Only set when the function succeeds.

`stat[18]`: singular row index (in original order) reported by `NicsLU_Factorize`, `NicsLU_ReFactorize` or `NicsLU_FactorizeMatrix`. Only set when the function fails with error code `-4`, `-5` or `-12`.

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 warnings 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. 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 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 27 user-callable routines, including 18 low-level routines, 2 high-level routines, and 7 utility routines. The `C` header file `nicslu.h` defines the interface of the NICSLU routines. NICSLU uses the same 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. 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, implying 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 a `NULL` pointer is

passed to `cfg`, `stat`, or `last_err`, there will be no chance to obtain the corresponding pointer later.

NOTE: From version 202006, we provide NICSLU libraries that use or do not use fused-multiply-add (FMA) intrinsic instructions. FMA is supported only by CPU architectures not older than Intel's Haswell (the 4th Core family) or AMD's Piledriver. This routine checks whether the CPU supports FMA. It returns -15 (functionality not supported) if the CPU does not support FMA and the FMA-enabled library is used. In this case, please use the library without enabling FMA and the cost is a tiny performance degradation.

Usage example:

```
    _handle_t ctx = NULL;
    _double_t *cfg = NULL;
    const _double_t *stat = NULL;
    const char *last_err = NULL;
    ...
    int err = NicsLU_Initialize(&ctx, &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(ctx);
    ctx = 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. 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 in 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).

If this routine is called more than once, all memories associated with the previous matrix as well as the existing threads are first freed/destroyed, and then the new matrix is created and analyzed. NICSLU does not provide a separate function to free the matrix. Instead, specifying `n = -1` in this routine frees the matrix (only leaving the handle valid).

NICSLU provides 5 different ordering methods: AMD [17], approximate minimum Markowitz (AMM), and 3 approximate minimum operation (AMO) methods (AMO1, AMO2 and AMO3). `cfg[3]` is used to select the ordering method. By default, NICSLU selects the best one from the 5 built-in ordering methods. Generally speaking, AMM and AMOs are better than AMD in most cases, and AMM and AMOs are almost as fast as AMD. There are 9 pre-defined ordering methods, depending on the value of `cfg[3]`.

- `cfg[3] = 0`: the natural order is used. `row_perm[]` and `col_perm[]` are not used.
- `cfg[3] = 1`: a user-defined ordering is used. `row_perm[]` and `col_perm[]` specify the ordering.
- `cfg[3] = 2` (**default**): the best one in AMD, AMM, AMO1, AMO2 and AMO3 is selected **in parallel**. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 3`: the best one in AMD, AMM, AMO1, AMO2 and AMO3 is selected **in sequential**. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 4`: AMD is used. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 5`: AMM is used. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 6`: AMO1 is used. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 7`: AMO2 is used. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.
- `cfg[3] = 8`: AMO3 is used. `row_perm[]` and `col_perm[]` retrieve the ordering if they are not `NULL`.

Specifying `cfg[3] = 2` enables a **parallel** selection of the best ordering method. In this case, the number of threads is automatically controlled by NICSLU and users do not need to control the threads. The created threads will exit after finishing the ordering selection. If the CPU has only one core, then NICSLU switches to run a sequential selection if one specifies `cfg[3] = 2`.

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

If an application involves successively solving multiple linear systems with identical nonzero pattern of $\mathbf{A}$ 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 uses 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.

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 which 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 (the same number of threads have already been created) 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$0 (default is -1) indicates using the blocked waiting method and `cfg[2]`$>$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.

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.

Parallel selection of the ordering method performed by `NicsLU_Analyze` does not rely on the threads created by this routine. Instead, `NicsLU_Analyze` must be called before calling this routine.

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 created by `NicsLU_CreateThreads` 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,
    const int param[]
);
```

This routines changes threads' scheduling policy by binding threads to cores. The second argument `op` is an enumeration variable to specify the operation. The last argument `param[]` specifies the parameters for the corresponding operation. `op` can be one of the following four values.

- `THREAD_BINDING_ALL` (value 0): binding all threads to cores. Binding threads to cores may increase the performance for very sparse matrices. The array `param[]` should have at least two elements, where `param[0]` specifies the minimum core ID that will be bound and `param[1]` specifies the number of cores that each thread will be bound to. Threads are bound to consecutive cores from `param[0]`. More specifically, thread 0 (i.e., the calling thread) is bound to cores numbered `param[0]`, `param[0]+1`, ..., `param[0]+param[1]-1`, thread 1 is bound to cores numbered

param[0]+param[1], ..., param[0]+2*param[1]-1, and so on. If this routine is called after NicsLU_CreateThreads is called, then all the created threads are bound to the specified cores; otherwise only thread 0 (i.e., the calling thread) is bound to the specified cores.

- THREAD_BINDING_ONE (value 1): binding one thread to specified cores. param[0] specifies the thread ID. param[1] specifies the number of cores that thread param[0] will be bound to. The parameters from param[2] with length at least param[1], i.e., the array {param[2], param[3], ..., param[2+param[1]-1]}, specify the core IDs.
- THREAD_UNBINDING_ALL (value 2): unbinding all threads. This means setting the affinities of all threads except thread 0 (i.e., the calling thread) to the default affinity, and setting the affinity of thread 0 (i.e., the calling thread) to its affinity recorded before initializing NICSLU. param[] is not used so it can be NULL.
- THREAD_UNBINDING_ONE (value 3): unbinding one thread. param[0] specifies the thread ID that will be unbound from cores. If param[0]=0, this routine sets the affinity of the calling thread to its affinity recorded before initializing NICSLU. If param[0]>0, this routine sets the affinity of thread param[0] to the default affinity.

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 dependency and dynamic update of the dependency. 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.

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

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.

NICSLU has a feature of pivot perturbation for `NicsLU_ReFactorize`, although it is not recommended. When re-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. However, pivot perturbation cannot be applied to complex number matrices.

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 circuits with a fixed integration step length.

NICSLU supports fast solving if the right hand vector $\mathbf{b}$ is very sparse by setting `cfg[21]=1` (only for $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ mode). This option is turned off by default. Based on our experience, if the number of nonzeros in $\mathbf{b}$ is less than `n/10`, fast solving may improve the performance.

5.6.10 NicsLU_Refine

```
int NicsLU_Refine
(
    _handle_t solver,
    const _double_t ax[],
```

```
5       const _double_t b[],
6       _double_t x[]
7   );
```

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 r = Ax − b;
while ||r||_2 > eps && (iter++) < maxiter
    solve Ad = r;
    update solution x = x − d;
    r_0 = ||r||_2 and update residual r = Ax − b;
    if (||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 threshold for the refinement.

5.6.11 NicsLU_Flops

```
1   int NicsLU_Flops
2   (
3       _handle_t solver,
4       int threads,
5       _double_t fflops[],
6       _double_t *sflops
7   );
```

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`. In fact, `NicsLU_CreateThreads` is not required 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, instead of in the original order. The values are scaled if static or normalization 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,
```

```
4       _size_t *mem
5   );
```

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

5.6.15 NicsLU_Performance

```
1   int NicsLU_Performance
2   (
3       _handle_t solver,
4       int tsync,
5       int threads,
6       _double_t perf[4]
7   );
```

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[4]` returns the estimated performance. Its length should be at least 4, and it returns the following 4 performance estimations.

- `perf[0]`: theoretical speedup upper bound, 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

```
1   int NicsLU_Determinant
2   (
3       _handle_t solver,
4       _double_t *coef,
5       _double_t *expn
6   );
```

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).

5.6.17 NicsLU_MemoryTraffic

```
int NicsLU_MemoryTraffic
(
    _handle_t solver,
    _double_t *fcomm,
    _double_t *scomm
);
```

This routine estimates the volume of memory traffic for the factorization and solving phases. Call this routine after a factorization. The argument `fcomm` and `scomm` return the volume of memory traffic for factorization and solving, respectively. The estimated memory traffic volume is reported in bytes.

5.6.18 NicsLU_DrawFactors

```
int NicsLU_DrawFactors
(
    _handle_t solver,
    const char file[],
    int size
);
```

This routine draws the LU factors into a bitmap image file. The bit depth of the generated bitmap file is 1, meaning that there are only two colors (white and black) on the image. Figure 2 shows an example of the generated factor profile. The last argument `size` specifies the width/height of the square image, in pixel. The file name should have a `.bmp` suffix; otherwise this routine will return a +4 warning.

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 factorization.

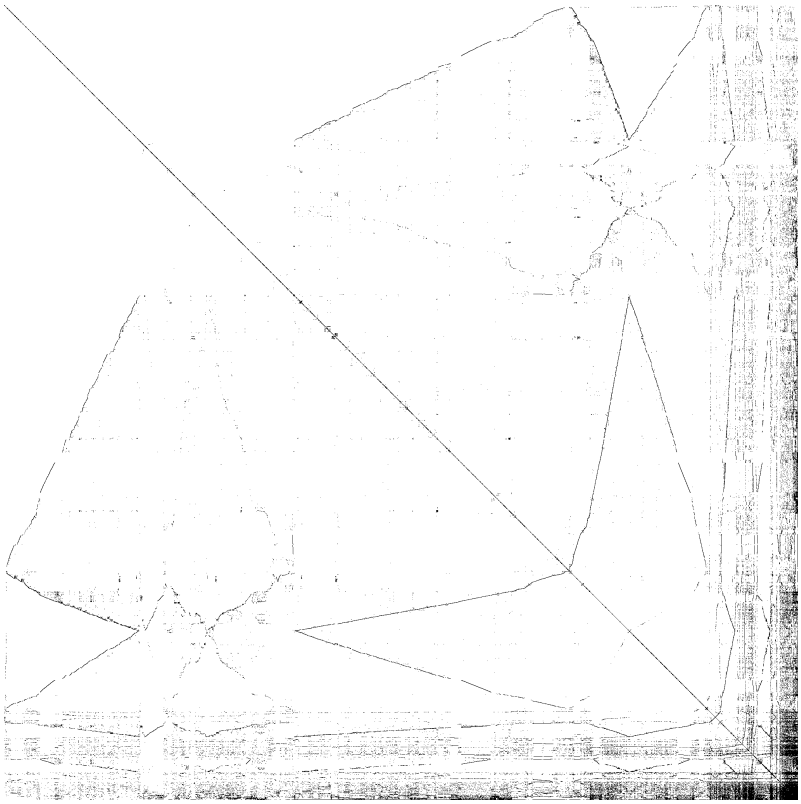


Figure 2: Example of a factor profile.

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 7 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[4],
    _matrix_type_t mtype
);
```

This routines calculates the residual error $\|\mathbf{Ax} - \mathbf{b}\|$ or $\left\|\mathbf{A}^{\mathsf{T}}\mathbf{x} - \mathbf{b}\right\|$. 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[4]` 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 ReadMatrixMarketFile

```
int ReadMatrixMarketFile
(
    const char file[],
    _uint_t *row,
```

```
    _uint_t *col,
    _uint_t *nnz,
    _double_t ax[],
    _uint_t ai[],
    _uint_t ap[],
    _bool_t *is_dense,
    _bool_t *is_complex,
    int *is_symmetric
);
```

This routine reads a matrix from a matrix market formatted file. It supports a subset of the matrix market format. 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 dense, `ai[]` and `ap[]` can be `NULL`, and the length of `ax[]` must be at least `row`×`col` for a real number matrix or 2×`row`×`col` for a complex number matrix.

The last three arguments return the properties of the matrix. For `is_dense` and `is_complex`, they are Boolean variables; while for `is_symmetric`, a positive value denotes a symmetric matrix, a negative value denotes a Hermitian matrix (only for complex number matrices), and a zero denotes a non-symmetric matrix.

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 `WriteMatrixMarketFile`

```
int WriteMatrixMarketFile
(
    const char file[],
    _uint_t row,
    _uint_t col,
    _uint_t nnz,
    const _double_t ax[],
    const _uint_t ai[],
    const _uint_t ap[],
    _bool_t is_dense,
    _bool_t is_complex,
    int is_symmetric
);
```

This routine writes a matrix to a file formatted by the matrix market format. For sparse matrices represented by CSC, `ax[]`, `ai[]`, and `ap[]` are all used; while for dense matrices, only `ax[]` is used. The last three arguments specify the properties of the matrix. For `is_dense` and `is_complex`, they are Boolean variables; while for `is_symmetric`, a positive value denotes a symmetric matrix, a negative value denotes a Hermitian matrix (only for complex number matrices), and a zero denotes a non-symmetric matrix.

5.8.5 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[],
    _transpose_t op
);
```

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/CSC format. The last argument `op` specifies the matrix type and how to generate the missing part, namely, the other half matrix. See Section 5.8.2 for its possible values. Before calling this routine, the memories for `aax[]`, `aai[]`, and `aap[]` must be pre-allocated. After calling it, they return the resulting symmetric/Hermitian full 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 [18].

5.8.6 SparseDraw

```
int SparseDraw
(
    _uint_t n,
    const _uint_t ai[],
    const _uint_t ap[],
    const char file[],
    int size
);
```

Similar to `NicsLU_DrawFactors`, this routine draws a sparse matrix represented in CSR into a bitmap image file. The last argument `size` specifies the width/height of the square image, in pixel. The file name should have a `.bmp` suffix; otherwise this routine will return a +4 warning.

5.8.7 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 in a console/terminal.

6 Using NICSLU in Circuit Simulation

This section illustrates 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 3.

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

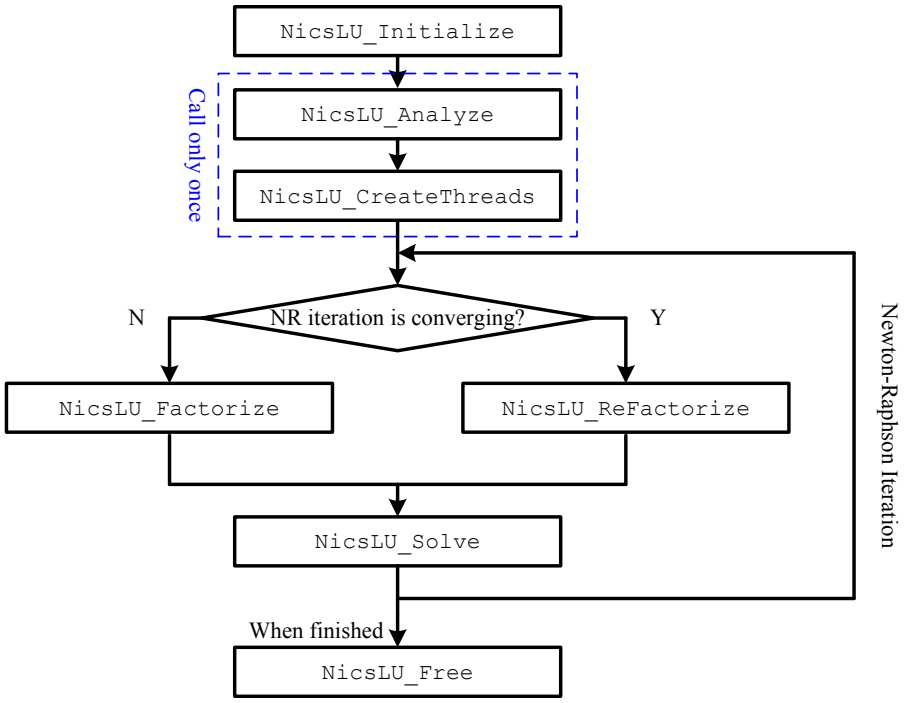


Figure 3: How to use NICSLU in circuit simulation (using low-level routines).

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 3. 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.

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 $\gg$ AbsTol and BigRelTol $\gg$ 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 4 illustrates the usage. There is no need to determine whether to call factorization or re-factorization by yourself.

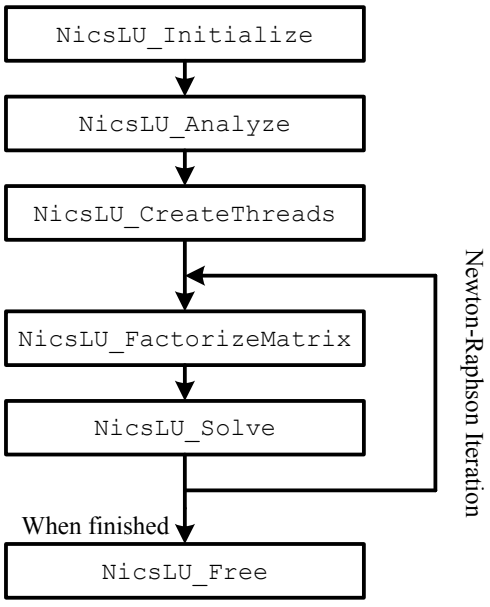


Figure 4: How to use NICSLU in circuit simulation (using high-level routines).

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.

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 7	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.

From version 202006, we provide NICSLU libraries that use or do not use fused-multiply-add (FMA) intrinsic instructions. FMA is supported only by CPU architectures not older than Intel's Haswell (the 4th Core family) or AMD's Piledriver. `NicsLU_Initialize` checks whether your CPU supports such intrinsics and returns -15 if not. For the case of nonsupport, please use the FMA-disabled libraries and the cost is a small performance degradation.

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. In other words, the bit widths of `_size_t`, `_int_t` and `_uint_t` will be affected.

- 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/systems. 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

We use Microsoft Visual Studio as an example to illustrate how to link the NICSLU library. For other `C/C++` development platforms, the setting is similar.

To link NICSLU, 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`. Systems with a low version glibc 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.

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] E. Rothberg and S. C. Eisenstat. Node Selection Strategies for Bottom-Up Sparse Matrix Ordering. *SIAM J. Matrix Anal. Appl.*, 19(3):682–695, July 1998.

[11] G. Reiβig. Local Fill Reduction Techniques for Sparse Symmetric Linear Systems. *Electrical Engineering*, 89(8):639–652, 2007.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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