User Guide of CKTSO

(version 20221123)

Xiaoming Chen (chenxiaoming@ict.ac.cn)

Contents

1. Introduction

CKTSO is a high-performance parallel sparse direct solver specially designed for SPICE-based circuit simulation. It solves $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A}$ is square and highly sparse. The software is written in pure C and provides both C and C++ interfaces. Dynamic-link libraries for Windows and Linux are provided.

CKTSO is the successor of our previous work, NICSLU (https://github.com/chenxm1986/nicslu). CKTSO uses many similar techniques to NICSLU. However, CKTSO integrates some novel techniques and shows higher performance, better scalability and less memory usage than NICSLU, while NICSLU provides more functionalities. The most important new features of CKTSO include a) a new pivoting-reduction technique that significantly improves the scalability of LU factorization with pivoting, b) a new memory allocation strategy that reduces memory usage, and c) parallel forward/backward substitutions.

CKTSO solves a sparse linear system through three main steps: symbolic analysis, factorization, and solving. Symbolic analysis tries to order the matrix to minimize fill-ins that will be generated in factorization. In circuit simulation, symbolic analysis is executed only once if the symbolic structure of the matrix is not changed. The factorization step factorizes the matrix into the product of a lower triangular matrix and an upper triangular matrix, i.e., $\mathbf{A} = \mathbf{LU}$. Partial pivoting is performed to ensure the numerical stability. The solving step finds the solution of the linear system through two triangular system solvers. i.e., $\mathbf{L}\boldsymbol{y} = \boldsymbol{b}$ and $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$.

Some algorithms of CKTSO are described in the following paper.

- Xiaoming Chen, "Numerically-Stable and Highly-Scalable Parallel LU Factorization for Circuit Simulation", in 2022 International Conference On Computer Aided Design (ICCAD'22).

2. License Key

Running CKTSO requires a valid license key file. The license key file must be named "`cktso.lic`" and put **together with the library file** (`*.dll` on Windows and `*.so` on Linux). It is a pure text file. Do not edit any content of the license key file, or it may be invalidated.

3. System Requirements

3.1 Software Requirements

CKTSO libraries can be run on Windows or Linux. The libraries compiled for systems of lower versions can be used on systems of higher versions. For Windows, Windows 7 is the minimum supported version. The libraries compiled for Windows 10 use some new Windows APIs which do not exist in

Windows 7. For Linux, libraries for CentOS and Ubuntu are provided, and they are compatible with many mainstream Linux distributions. The libraries compiled for Linux have different requirements in the `glibc` version, depending on the operating system and compiler versions. The libraries should be selected according to the operating systems and compilers which were used to compile them.

3.2 Hardware Requirements

CKTSO libraries are compiled as x64 dynamic-link libraries which can run on x64 CPUs. Pure x86 libraries are not provided. CKTSO uses AVX2 and FMA instructions. If the CPU does not support such instructions, an error will occur when running CKTSO libraries.

4. Matrix Format

The input format of CKTSO is the compressed sparse row (CSR) format. CSR uses an integer `n` and three arrays `ap[]`, `ai[]` and `ax[]` to represent a sparse matrix. `n` is the matrix dimension. Array `ap[]` of length `n+1` stores the row pointers. More specifically, `ap[]` stores the start position of every row in `ai[]` and `ax[]`. Array `ai[]` of length `ap[n]` stores the column indexes. Array `ax[]` of length `ap[n]` stores the matrix values (one-to-one corresponding to the elements of array `ai[]`). Figure 1 illustrates an example of CSR, in which the matrix is of dimension 4.

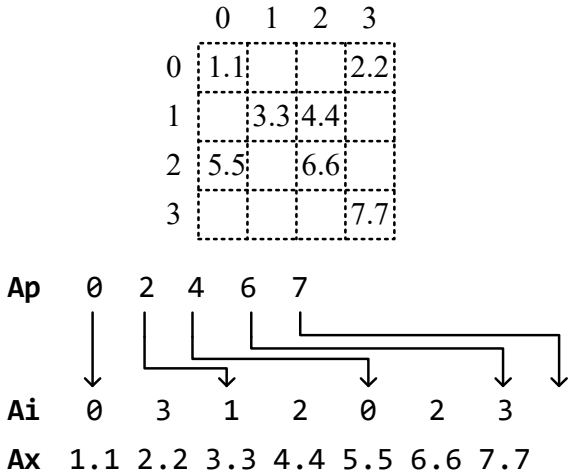

Figure 1: Example of CSR.

CKTSO does not need the column indexes in each row to be sorted, but duplicated entries are not allowed (a simple workaround is provided to handle inputs with duplicated entries). CKTSO only supports real-number matrices. For a complex-number linear system $(\mathbf{A} + \mathrm{j}\mathbf{B})(\boldsymbol{x} + \mathrm{j}\boldsymbol{y}) = \boldsymbol{b} + \mathrm{j}\boldsymbol{c}$, the following

real-number linear system of dimension `2n` is equivalent to the original problem

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix}.$$

By utilizing this conversion, CKTSO may also solve complex-number linear systems. However, solvers which natively support complex-number linear systems tend to have better performance on complex-number problems.

5. C/C++ Functions

CKTSO provides both C and C++ interfaces. The C interface has 10 global functions while the C++ interface provides an object that has 9 member functions. For the C interface, both 32-bit and 64-bit integer versions are provided, and the latter involves a "`_L`" flag in the function name. For the C++ interface, both 32-bit and 64-bit objects are provided, and the latter involves a "`_L`" flag in the object name. The 64-bit interface uses 64-bit large integers to represent the indexes of the input matrix. The interface type only affects the representation of the input matrix. The internal data structures always use 64-bit integers to store the LU factors.

Both the C and C++ interfaces involve an `ICktSo` (`struct __cktso_dummy *`) or `ICktSo_L` (`struct __cktso_l_dummy *`) object. An object instance is created by `CKTSO_CreateSolver` or `CKTSO_L_CreateSolver` and passed as the first argument to other C functions. Users may call its member functions in the C++ environment. The member functions of the object have one-to-one correspondence with the C functions, e.g., `__cktso_dummy::Analyze` corresponds to `CKTSO_Analyze`. The correspondence means that they have the same arguments except the first argument of the C function which is the instance and does not exist in the member functions of the object.

All CKTSO functions (including both C and C++ functions) return an integer to indicate the error. Zero means no error and negative values indicate errors. The meaning of the return code is listed in Table 1.

Table 1: Return code.

`0`	Successful	`-1`	Invalid instance handle
`-2`	Argument error	`-3`	Invalid matrix data
`-4`	Out of memory	`-5`	Structurally singular
`-6`	Numerically singular	`-7`	Threads-related error
`-8`	Matrix has not been analyzed	`-9`	Matrix has not been factorized
`-10`	Function is not supported	`-11`	File cannot be open
`-12`	Integer overflow	`-99`	License error
`-100`	Unknown error		

5.1 Create Solver

```
int CKTSO_CreateSolver
(
    ICktSo *inst,
    int **iparm,
    const long long **oparm
);
```

```
int CKTSO_L_CreateSolver
(
    ICktSo_L *inst,
    int **iparm,
    const long long **oparm
);
```

This function creates the solver instance that is returned by the first argument. The created instance can be used in both C and C++ environment.

The parameters `iparm` and `oparm` return the pointers to the internal input parameter and output parameter arrays, respectively. They can be `NULL` if not needed. If so, there will be no chance to change the configurations or retrieve the statistical information of the solver.

5.2 Destroy Solver

```
int CKTSO_DestroySolver
(
    ICktSo inst
);
```

```
int CKTSO_L_DestroySolver
(
    ICktSo_L inst
);
```

```
virtual int __cktso_dummy::DestroySolver
(
) = 0;
```

```
virtual int __cktso_l_dummy::DestroySolver
(
) = 0;
```

This function frees any data associated with the specified instance and also destroys the instance. After that, the instance is invalid. Any created instance should be destroyed when it will no longer be used. Do not destroy an instance more than once.

5.3 Matrix Analysis

```
int CKTSO_Analyze
(
    ICktSo inst,
    int n,
    const int ap[],
    const int ai[],
    const double ax[],
    int row_col_tran,
    int threads
);
```

```
int CKTSO_L_Analyze
(
    ICktSo_L inst,
    long long n,
    const long long ap[],
    const long long ai[],
    const double ax[],
    int row_col_tran,
    int threads
);
```

```
virtual int __cktso_dummy::Analyze
(
    int n,
    const int ap[],
    const int ai[],
    const double ax[],
    int row_col_tran,
    int threads
) = 0;
```

```
virtual int __cktso_l_dummy::Analyze
(
    long long n,
    const long long ap[],
    const long long ai[],
    const double ax[],
    int row_col_tran,
    int threads
) = 0;
```

This function creates internal data for the matrix, reorders the matrix to minimize fill-ins, performs static symbolic factorization and also creates threads. It must be called before any factorization is called. In circuit simulation, it usually needs only once because the symbolic structure of the matrix is fixed

during iterations.

The parameter `n` specifies the matrix dimension. Specifying `n=0` will destroy the previous matrix and free any data associated with the previous matrix, while the instance is still valid. The parameters `ap`, `ai` and `ax` specify the CSR arrays of the matrix. Please note that CKTSO uses the **row-major CSR format** by default. The parameter `row_col_tran` specifies whether the matrix is transposed (i.e., whether to solve $\mathbf{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$). CKTSO provides three modes. Zero means the ***row mode***. A positive value means the ***column mode***, which factorizes the original matrix but uses a column-order substitution process. A negative value means the ***transposed mode***, which factorizes the transposed matrix and uses the same row-order substitution process as the row mode. Both the column and transposed modes can solve $\mathbf{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, while the transposed mode is primarily designed for the GPU acceleration module. The parameter `threads` specifies the number of threads. The created threads will be used for matrix analysis, factorization, refactorization, sorting factors, and solving. They are managed as a thread pool and will not exit until the instance is destroyed or the matrix is destroyed. Specifying `threads=0` will use all physical CPU cores, while `-1` means using all logical CPU cores. The specified number of threads cannot exceed the number of logical CPU cores. The maximum number of threads is suggested to be the number of physical CPU cores (specifying `threads=0`). It is not suggested to use more threads than the number of physical cores. Depending on the hardware and operating system, CKTSO may not retrieve the correct number of physical cores, so it is highly recommended that the number of threads is explicitly specified.

5.4 Factorize with Pivoting

```
int CKTSO_Factorize
(
    ICktSo inst,
    const double ax[],
    bool fast
);
```

```
int CKTSO_L_Factorize
(
    ICktSo_L inst,
    const double ax[],
    bool fast
);
```

```
virtual int __cktso_dummy::Factorize
(
    const double ax[],
    bool fast
) = 0;
```

```
virtual int __cktso_l_dummy::Factorize
(
    const double ax[],
    bool fast
) = 0;
```

This function factorizes the reordered matrix into LU factors with partial pivoting to ensure the numerical stability. The factorization uses the threads created in matrix analysis. CKTSO has a thread control technique that can automatically judges whether using multiple threads is useful. If not, it will automatically use a single thread, even if multiple threads have been created. Except for the first factorization, CKTSO employs a fast pivoting-reduction technique which can significantly improve the performance of matrix factorization but the numerical stability is still maintained. CKTSO also utilizes a sparsity-oriented algorithm selection method to enhance the factorization performance for different sparsities.

The parameter `ax` specifies the array storing the matrix values, which is of length `ap[n]` specified in matrix analysis. The parameter `fast` specifies whether fast factorization based on pivoting reduction is enabled.

Please note that **subsequent factorizations are generally much faster than the first-time factorization, if fast factorization is enabled**. In the best case, the performance and scalability of fast factorization are almost same as those of refactorization without pivoting. This is the benefit of the pivoting-reduction technique. However, in some extreme cases, the fast factorization technique may lead to more fill-ins. It is suggested that **the first factorization of each independent simulation method (e.g., a DC simulation method like GMIN stepping or pseudo-transient, the entire transient simulation, etc.) should disable fast factorization**. Please refer to Section 8 for more details.

5.5 Refactorize without Pivoting

```
int CKTSO_Refactorize
(
    ICktSo inst,
    const double ax[]
);
```

```
int CKTSO_L_Refactorize
(
    ICktSo_L inst,
    const double ax[]
);
```

```
virtual int __cktso_dummy::Refactorize
(
    const double ax[]
) = 0;
```

```
virtual int __cktso_l_dummy::Refactorize
(
    const double ax[]
) = 0;
```

This function refactorizes the matrix without pivoting. It should be called after factorization with pivoting has been called at least once. It reuses the pivoting order and the symbolic structure of the LU factors obtained in the last factorization with pivoting. The refactorization uses the threads created in matrix analysis. CKTSO has a thread control technique that can automatically judges whether using multiple threads is useful. If not, it will automatically use a single thread, even if multiple threads have been created.

In circuit simulation, there are usually many Newton-Raphson iterations. When Newton-Raphson iterations are converging, the matrix values tend to change slowly. In this situation, factorizing the matrix without pivoting is generally numerically stable. This function provides an opportunity to improve the solver performance in circuit simulation by utilizing the features of Newton-Raphson iterations. To judge whether Newton-Raphson iterations are converging, one can simply check the difference between the solutions of two adjacent iterations. See Section 7 for details.

The parameter `ax` specifies the array storing the matrix values, which is of length `ap[n]` specified in matrix analysis.

Every time after the symbolic pattern of the LU factors has been changed (factorization with pivoting has been called), the first call of this function has an additional scheduler initialization step, which causes some additional time.

5.6 Solve

```
int CKTSO_Solve
(
    ICktSo inst,
    const double b[],
    double x[],
    bool force_seq
);
```

```
int CKTSO_L_Solve
(
    ICktSo_64 inst,
    const double b[],
    double x[],
    bool force_seq
);
```

```
virtual int __cktso_dummy::Solve
(
    const double b[],
    double x[],
    bool force_seq
) = 0;
```

```
virtual int __cktso_l_dummy::Solve
(
    const double b[],
    double x[],
    bool force_seq
) = 0;
```

This function performs forward and backward substitutions to solve the linear system, after the matrix has been factorized. CKTSO supports sequential or parallel solving. CKTSO has a thread control technique that can automatically judges the best number of threads for parallel solving.

The parameter `b` specifies the right-hand-side vector. The parameter `x` specifies the solution vector. The address of array `x[]` may be same as the address of array `b[]`. The parameter `force_seq` specifies whether to force CKTSO to perform sequential solving. However, even if forced sequential solving is not enabled, CKTSO does not necessarily perform parallel solving. Instead, CKTSO has a mechanism to determine whether to perform parallel solving and also the optimal number of threads automatically. Please refer to Section 8 for more details.

Every time after the symbolic pattern of the LU factors has been changed (factorization with pivoting has been called), the first call of this function has an additional scheduler initialization step, which causes about 2-5X time of the sequential solving time.

The performance of parallel solving varies on different operating systems and for different matrix storage formats. Due to the much stronger dependency in parallel solving for matrices stored in the column-major format, column-mode parallel solving has lower performance than row-mode parallel solving. Table 2 summarizes a general description of the performance of parallel solving, as well as our recommendations.

Table 2: Performance of parallel solving.

	Windows	Linux
Row-major order	Generally most matrices have speedups but a few do not. Parallel solving is **recommended** in this scenario.	Almost all matrices have speedups. Parallel solving is **recommended** in this scenario.
Column-major order	Some matrices have speedups but some do not. Parallel solving is **not recommended** in this scenario.	Generally most matrices have speedups but a few do not. Parallel solving is **recommended** in this scenario.

5.7 Solve Multiple Vectors

```
int CKTSO_SolveMV
(
    ICktSo inst,
    size_t nrhs,
    const double b[],
    double x[]
);
```

```
int CKTSO_L_SolveMV
(
    ICktSo_64 inst,
    size_t nrhs,
    const double b[],
    double x[]
);
```

```
virtual int __cktso_dummy::SolveMV
(
    size_t nrhs,
    const double b[],
    double x[]
) = 0;
```

```
virtual int __cktso_l_dummy::SolveMV
(
    size_t nrhs,
    const double b[],
    double x[]
) = 0;
```

This function performs forward and backward substitutions for solving multiple vectors (i.e., solving $\mathbf{A}\boldsymbol{x}_i = \boldsymbol{b}_i$, where $i = 1,2,\cdots,nrhs$), after the matrix has been factorized. If multiple threads have been created when analyzing the matrix, each thread will solve a vector independently; otherwise CKTSO just solves all vectors sequentially. The parameter `nrhs` specifies the number of vectors to be solved. The arrays `b[]` and `x[]` are both of length `n*nrhs`, and they store the right-hand-vectors and the solution vectors vector by vector.

5.8 Sort Factors

```
int CKTSO_SortFactors
(
    ICktSo inst,
    bool sort_values
);
```

```
int CKTSO_L_SortFactors
(
    ICktSo_L inst,
    bool sort_values
);
```

```
virtual int __cktso_dummy::SortFactors
(
    bool sort_values
) = 0;
```

```
virtual int __cktso_l_dummy::SortFactors
(
    bool sort_values
) = 0;
```

This function sorts each row of the LU factors. It is typically used after factorization. Due to the factorization algorithm, the indexes of the LU factors are not guaranteed to be in order. With sorted factors, the cache efficiency of subsequent refactorization and solving processes may be improved, but the improvement is typically small.

The parameter `sort_values` specifies whether to sort values as well. If not, only indexes are sorted. If values are not sorted, a subsequent solving will return an error unless a factorization or refactorization has been called before solving.

5.9 Calculate Workloads

```
int CKTSO_Statistics
(
    ICktSo inst,
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem
);
```

```
int CKTSO_L_Statistics
(
    ICktSo_L inst,
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem
);
```

```
virtual int __cktso_dummy::Statistics
(
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem
) = 0;
```

```
virtual int __cktso_l_dummy::Statistics
(
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem
) = 0;
```

This function calculates the number of floating-operations and memory access volumes of factorization and solving. It should be called after factorization with partial pivoting has been called.

Each of the four parameters, `factor_flops`, `solve_flops`, `factor_mem` and `solve_mem`, can be `NULL` if not required. No initialization for the four parameters is required. The memory access volumes are reported in bytes.

5.10 Clean up Garbage

```
int CKTSO_CleanupGarbage
(
    ICktSo inst
);
```

```
int CKTSO_L_CleanupGarbage
(
    ICktSo_L inst
);
```

```
virtual int __cktso_dummy::CleanupGarbage
(
) = 0;
```

```
virtual int __cktso_l_dummy::CleanupGarbage
(
) = 0;
```

This function cleans up redundant memory. If factorization with partial pivoting has been called many times, the memory usage may be more than required. This function can clean up such unnecessary memory usage.

6. Parameters

CKTSO maintains an input parameter array for behavior configurations and an output parameter array for information statistics. Input parameters are initialized to default values in `CKTSO_CreateSolver` or `CKTSO_L_CreateSolver`. When calling `CKTSO_CreateSolver` or `CKTSO_L_CreateSolver`, one has a chance to retrieve the pointers to the two internal arrays, which are `in **iparm` and `const long long **oparm`. By setting the input parameter array, the behavior of CKTSO may be configured. Most input parameters are effective if they are set before matrix analysis, and changing some input parameters after matrix analysis is ineffective. By reading the output parameters, one can get some statistical runtime information of CKTSO.

6.1 Input Parameters

`iparm[0]`: timer control. Zero means no timer. A positive value means using a high-precision timer while a negative value means using a low-precision timer. High-precision timer also involves higher overhead than low-precision timer. Only when `iparm[0]` is nonzero, `oparm[0]` to `oparm[3]` are effective. Default is zero (timer disabled).

`iparm[1]`: pivoting tolerance. It is in millionth (1/1000000). Default is 1000 (0.001 actually).

`iparm[2]`: ordering method. CKTSO has 8 different ordering methods. Zero

means selecting the best from the 8 methods. 1 to 8 mean using the corresponding single ordering method. A double-digit number in decimalism `x0` (20 to 80) means selecting the best ordering method from the first `x` methods. For example, specifying `iparm[2]=50` means selecting the best from the ordering methods 1 to 5. A negative value means no ordering (using the natural order). Default is zero (selecting best from all 8 methods).

`iparm[3]`: threshold for dense node detection. It is in hundredth. A row with more than $\text{iparm}[3]/100 \times \sqrt{n}$ nonzeros is treated as a dense node and is removed before ordering to save ordering time. Default is 1000 (10 actually).

`iparm[4]`: metric for ordering method selection. Zero or a positive value means using estimated number of floating-point operations to select the best ordering method, while a negative value means using estimated number of factors. Default is zero (using floating-point operation number).

`iparm[5]`: maximum supernode size. Each supernode can have at most `iparm[5]` rows. A supernode that has more than `iparm[5]` rows will be split into multiple smaller supernodes. Zero means that CKTSO determines the maximum supernode size according to the cache size. Negative one means no limitation for supernode size. Default is -1 (infinite).

`iparm[6]`: minimum number of columns for supernode detection. A supernode must have at least `iparm[6]` columns. Default is 64.

`iparm[7]`: whether to perform scaling. Scaling may improve the solution accuracy for some ill-conditioned matrices with the overhead of a small performance drop. Default is zero (scaling disabled).

`iparm[8]`: whether the right-hand-side vector is highly sparse. Typically if the right-hand-side vector has less than 10% nonzeros, it can be regarded as "highly sparse". This parameter is used to control whether to use a faster substitution method. It is only effective in sequential column-mode solving. Default is zero (not highly sparse).

`iparm[9]`: whether to control the number of threads for parallel factorization, refactorization, and solving. If enabled and the matrix is too small or too sparse, factorization and refactorization may use a single thread even if multiple threads have been created, and solving may use fewer threads than created. Default is 1 (thread number control enabled).

`iparm[10]`: dynamic memory growth factor. It is in percentage. A larger value helps reduce reallocations at runtime but costs more memory usage. Default is 150 (1.5 actually).

`iparm[11]`: initial number of rows for supernode creation. CKTSO allocates memory of `iparm[14]` rows when a new supernode is created. A larger value helps reduce reallocations at runtime but costs more memory usage. Default is 16.

`iparm[12]`: static pivoting method. Zero means using the conventional matching-based static pivoting. A positive values means using a fill-in aware method based on the conventional method. A negative value means using a column size aware method based on the conventional method. The non-conventional methods help reduce fill-ins when there are many elements with identical numerical values. Default is 1 (using the fill-in aware method).

`iparm[13]`: synchronization method for threads. Zero or a positive value means using blocked wait, while a negative value means using busy wait. The latter reduces threads' synchronization cost but consumes CPUs. Default is zero (using blocked wait).

Please do not change other input parameters (after `iparm[13]`) which are not listed above.

6.2 Output Parameters

`oparm[0]`: time of matrix analysis, in microsecond (us).

`oparm[1]`: time of factorization or refactorization, in microsecond (us).

`oparm[2]`: time of solving, in microsecond (us).

`oparm[3]`: time of sorting factors, in microsecond (us).

`oparm[4]`: number of off-diagonal pivots.

`oparm[5]`: number of nonzeros of L, including diagonal.

`oparm[6]`: number of nonzeros of U, excluding diagonal (U diagonal is one). `oparm[5] + oparm[6]` is the number of the LU factors.

`oparm[7]`: number of supernodes.

`oparm[8]`: selected ordering method, from 1 to 8, corresponding to `iparm[2]`.

`oparm[9]`: singular row index when -6 is returned.

`oparm[10]`: number of memory reallocations invoked, only reported in factorization.

`oparm[11]`: memory requirement in bytes when -4 is returned.

`oparm[12]`: current memory usage in bytes.

`oparm[13]`: maximum memory usage in bytes.

`oparm[14]`: number of rows completed with pivoting reduction, only reported in factorization.

7. Using CKTSO Libraries

CKTSO is provided in the format of x64 dynamic-link libraries. Two individual libraries for 32-bit integers and 64-bit integers are provided. Link the correct

library to the user's executable.

On Linux, link CKTSO with "`-L<library path> -lcktso`" or "`-L<library path> -lcktso_l`". Some additional libraries may also be needed. Specifically, add "`-lpthread -lm -ldl`" to the linking command if linking is not successful. On some Linux systems, "`-lrt`" is also required. Before running the executable, run "`export LD_LBRARY_PATH=<library path>`" to set the library path if necessary.

For Visual Studio on Windows, add "`#pragma comment(lib, "cktso.lib")`" or "`#pragma comment(lib, "cktso_l.lib")`" to any place of the user's source code. `cktso.lib` or `cktso_l.lib` is only required in linking. When running the executable, only `cktso.dll` or `cktso_l.dll` is required. The `*.dll` file should be put together with the executable or in the system directory (i.e., `C:\Windows\` or `C:\Windows\System32\`).

Remember that the license key file should be put together with the library file, rather than with the executable file.

8. Using CKTSO in Circuit Simulators

First, call `CKTSO_CreateSolver` to create a solver instance. After the matrix is filled for the first time, call `CKTSO_Analyze`. Please note that besides the symbolic pattern of the matrix, the matrix values should also be provided when calling `CKTSO_Analyze`, since `CKTSO_Analyze` determines a matrix ordering based on both the symbolic pattern and numerical values. The values specified to `CKTSO_Analyze` should have similar features of the matrix values during the iterations of the entire simulation process, or fill-ins may be dramatically increased. If matrix values are invalid before entering the simulation iterations, a simple workaround is to make a wrapper for `CKTSO_Factorize`, and call `CKTSO_Analyze` just before the first `CKTSO_Factorize`. In this case, please make sure to call `CKTSO_Analyze` only once, like the following code.

```
int lu_factor(...) //a wrapper for CKTSO_Factorize
{
   static bool analysis_done = false;
   if (!analysis_done)
   {
      int err = CKTSO_Analyze(...);
      if (err < 0) ...;
      else analysis_done = true;
   }

   int err = CKTSO_Factorize(...);
   if (err < 0) ...;
}
```

During the iterations of circuit simulation, call `CKTSO_Factorize` or

`CKTSO_Refactorize` and `CKTSO_Solve` to solve a linear system in each iteration. It is crucial to correctly select `CKTSO_Factorize` and `CKTSO_Refactorize`. It is suggested that in DC simulation, `CKTSO_Factorize` should always be used. In transient simulation, the two functions can be selected based on the convergence situation. Intuitively, matrix values change more slowly when Newton-Raphson iterations are nearer convergence. Based on this observation, when Newton-Raphson iterations are converging, using `CKTSO_Refactorize` tends to be numerically stable. Conventional SPICE-style simulators use the following method to judge whether Newton-Raphson iterations are **converged**

$$\left|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\right| < \varepsilon_a + \varepsilon_r \times \min\{||\boldsymbol{x}^{(k)}||, ||\boldsymbol{x}^{(k-1)}||\}$$

where ε_a and ε_r are given absolute and relative thresholds, respectively. One can simply use the same method **with larger thresholds** to judge whether Newton-Raphson iterations are **converging**. If so, it means that the matrix values are changing slowly and `CKTSO_Refactorize` can be used; otherwise `CKTSO_Factorize` should be used. The thresholds for judging whether Newton-Raphson iterations are converging may be empirically determined. Note that the first factorization must call `CKTSO_Factorize`.

`CKTSO_Factorize` has an argument to specify whether fast factorization based on pivoting reduction is enabled. Pivoting reduction may boost the performance and scalability of parallel factorization in most cases. However, in some extreme cases, it may degrade the overall circuit simulation performance. To avoid bad situations, fast factorization should not be always enabled. In a circuit simulation process, there are some independent simulation "methods". In DC simulation, direct Newton-Raphson iterations, GMIN stepping, source stepping, and pseudo-transient are popular DC simulation "methods". The entire transient simulation can be regarded as a single "method". The usage of fast factorization is recommended as follows. The first factorization of each method should disable fast factorization, while the other factorizations may enable fast factorization. The purpose of this strategy is to avoid inter-method influences. To be more conservative, for each different stepping value (e.g., each different GMIN value) in stepping-based methods (e.g., GMIN stepping), the first factorization may also disable fast factorization. In transient simulation, each time after the time node is rolled back due to divergence, the next factorization should also disable fast factorization. In both DC and transient simulations, once factorization fails due to numerical singularity, the next factorization should disable fast factorization.

`CKTSO_Solve` has an argument to specify whether a forced sequential solving will be used. It should also be carefully specified. Parallel solving needs a one-time scheduler initialization step which takes about 2-5X time of the sequential solving time. The scheduler initialization step needs to be called

every time once the symbolic pattern of the LU factors changes. If the symbolic pattern of the LU factors keeps unchanged, the scheduler initialization step is skipped. Thus, the following usage is recommended. In DC simulation, `CKTSO_Factorize` and `CKTSO_Solve` with forced sequential solving are recommended, as `CKTSO_Factorize` changes the symbolic pattern of the LU factors frequently. In transient simulation, `CKTSO_Factorize` and `CKTSO_Refactorize` are selected based on the above-mentioned strategy, and whether to use forced sequential solving depends on the selection of `CKTSO_Factorize` and `CKTSO_Refactorize`. For the iterations which call `CKTSO_Factorize`, solving should be forced sequential, while for the iterations which call `CKTSO_Refactorize`, solving can be parallel. Please note that even if forced sequential solving is disabled, CKTSO may still use sequential solving, according to some internal strategies.

9. Frequently Asked Questions

Q: Why does `CKTSO_CreateSolver` return -10 (not supported)?
A: CKTSO uses AVX2 and FMA instructions. `CKTSO_CreateSolver` checks whether the CPU supports such instructions. If not, a -10 code is returned.

Q: Why is the first call of `CKTSO_Factorize` much slower than subsequent calls?
A: CKTSO integrates a pivoting-reduction technique which can significantly improve the performance and parallel scalability from the second factorization. In circuit simulation, since matrix values change smoothly during the Newton-Raphson iterations, in most cases the pivoting order is not changed so symbolic-related operations may be skipped from the second factorization. However, the first factorization needs to perform all necessary operations.

Q: For parallel solving, why is the first call of `CKTSO_Solve` much slower than subsequent calls?
A: Before the first parallel solving process, an initialization step for the scheduler is needed. The scheduler depends on the symbolic structure of the LU factors. The initialization step is needed every time when the symbolic structure of the LU factors is changed (factorization with pivoting is called). If the symbolic structure of the LU factors keep unchanged, the initialization step is skipped and the solving process may get its benefit from parallelism.

Q: What are the differences between NICSLU and CKTSO?
A: They are both sparse parallel solvers for circuit simulation and provide similar interface and usage. CKTSO is the successor of NICSLU. CKTSO uses some new techniques and has better performance and scalability, while the memory usage is smaller. However, NICSLU provides more functionalities.

Users may also refer to the user guide and FAQ of NICSLU (https://github.com/chenxm1986/nicslu) to find more notes for using sparse direct solvers in circuit simulators.