User Guide of CKTSO

(Version 20240630)

Xiaoming Chen (chenxiaoming@ict.ac.cn)

Contents

1. Introduction

CKTSO is a high-performance parallel sparse direct solver specially designed for ***SPICE-based circuit simulations***. 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 features of CKTSO include

a) a new pivoting-reduction technique that significantly improves the performance and scalability of LU factorization with pivoting;
b) a new memory allocation strategy that reduces memory usage;
c) parallel forward and backward substitutions;
d) a novel nested dissection ordering method, which reduces about 10X floating-point operations for post-layout/mesh-style circuits, and it also produces fewer floating-point operations than METIS;
e) several novel minimum degree ordering variants, which reduce about 30-40% floating-point operations compared with the approximate minimum degree;
f) an adaptive numerical kernel selection method.

CKTSO solves a sparse linear system through three main steps: ***symbolic analysis, factorization, and solving***. Symbolic analysis orders the matrix to minimize fill-ins that will be generated in factorization. In circuit simulation, symbolic analysis is usually executed only once. 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 can be 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

Table 1 lists the basic software and hardware requirements. CKTSO libraries can be run on Windows or Linux. For Linux, libraries for CentOS and Ubuntu are provided, and they are compatible with many mainstream Linux distributions. CKTSO libraries are provided 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.

Table 1. Basic software and hardware requirements.

Operating system	Windows 7 or higher, or mainstream Linux distributions
CPU	x86_64, with AVX2 and FMA instructions supported

4. Matrix Format

The input matrix 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. 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.

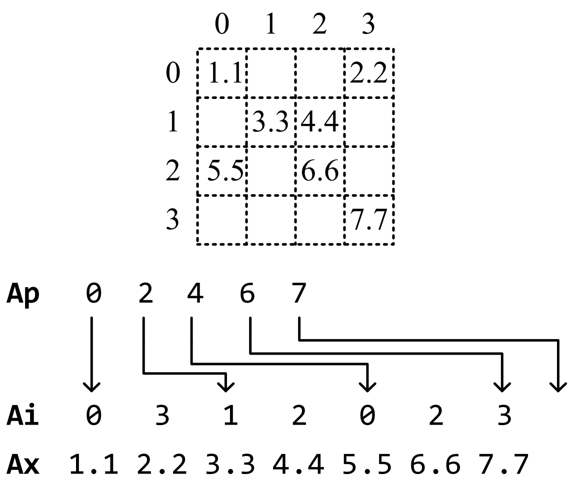

Figure 1: Example of CSR.

The column indexes in each row need NOT BE SORTED. From version 20231101, duplicated entries in CSR are allowed. However, the existence of duplicated entries will introduce additional latency and memory overheads. From version 20221214, ***CKTSO natively supports both real-number and complex-number matrices***. But the performance of complex-number computation has not been fully optimized.

5. C/C++ Functions

CKTSO provides both C and C++ interfaces. Both 32-bit and 64-bit integer versions are provided, and the latter involves an "`_L`" flag in the function name or object name.

The 64-bit interface uses 64-bit integers to represent the indexes of the input matrix. The internal data structures always use 64-bit integers to store the LU factors, even for the 32-bit interface.

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`, `CKTSO_L_CreateSolver`, `CKTSO_CreateSolverNoCheck`, or `CKTSO_L_CreateSolverNoCheck` 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`.

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 2.

Table 2: 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	`-13`	Resource leak
`-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
);
```

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

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

This function creates the solver instance. The created instance can be used in both C and C++ environments.

- The parameter `inst` returns the pointer to the created instance.
- 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. Do not free `iparm` and `oparm`. The arrays will be freed when the solver instance is destroyed.

`CKTSO_CreateSolver` or `CKTSO_L_CreateSolver` checks whether the CPU supports AVX2 and FMA instructions, and returns -10 if these instructions are not supported. `CKTSO_CreateSolverNoCheck` or `CKTSO_L_CreateSolverNoCheck` does not perform such check. This is useful on *virtual machines* which do not support these instructions but the CPU actually supports them.

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. Previously created threads will also exit. 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,
    bool is_complex,
    int n,
    const int ap[],
    const int ai[],
    const double ax[],
    int threads
);
```

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

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

```
virtual int __cktso_l_dummy::Analyze
(
    bool is_complex,
    long long n,
    const long long ap[],
    const long long ai[],
    const double ax[],
    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 can be called. In circuit simulation, it usually needs **only once** because the symbolic structure of the matrix is fixed during iterations.

- The parameter `is_complex` specifies whether the matrix is complex. For the complex case, all double-valued arrays should store complex numbers in an interleaved real-imaginary form. The complex-number data type can be defined as `double [2]`.
- The parameter `n` specifies the matrix dimension. Specifying a positive `n` will first destroy the previous matrix and perform analysis for the new matrix. Specifying `n=0` will only destroy the previous matrix, without creating the new matrix or doing analysis for the new matrix (the solver instance is still valid).
- The parameters `ap`, `ai`, and `ax` specify the CSR arrays of the matrix. `ax` can be `NULL` if the values are unavailable when the matrix is analyzed.

- 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 solver instance is destroyed or the matrix is destroyed. Specifying `threads=0` will use all physical CPU cores, and specifying `threads=-1` means using all logical CPU cores. The specified number of threads cannot exceed the number of logical CPU cores. 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. In addition, some old operating systems cannot detect the correct number of physical cores on CPUs with performance cores and efficient cores (12th Generation Intel Alder Lake CPUs or later). Therefore, it is highly recommended that the number of threads is explicitly specified. From version 20240630, CKTSO has a dynamic thread number adjustment method based on the system workload. It decides a proper number of threads for parallel processing to not over-consume CPUs, which can be smaller than the number of created threads. See Section 7 for details.

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. CKTSO has a thread control technique that can automatically judge whether using multiple threads is useful based on matrix features. 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. 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 9 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. It reuses the pivoting order and the symbolic structure of the LU factors obtained in the last factorization with pivoting. CKTSO has a thread control technique that can automatically judges whether using multiple threads is useful based on matrix features. 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 9 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,
    bool row0_column1
);
```

```
int CKTSO_L_Solve
(
    ICktSo_L inst,
    const double b[],
    double x[],
    bool force_seq,
    bool row0_column1
);
```

```
virtual int __cktso_dummy::Solve
(
    const double b[],
    double x[],
    bool force_seq,
    bool row0_column1
) = 0;
```

```
virtual int __cktso_l_dummy::Solve
(
    const double b[],
    double x[],
    bool force_seq,
    bool row0_column1
) = 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 based on matrix features.

- 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[]`. If so, the solution vector is overwritten on the right-hand-side vector.
- 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 based on matrix features. Please refer to Section 9 for more details.

- The parameter `row0_column1` specifies whether the matrix is transposed (i.e., whether to solve $\mathbf{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$). Please note that CKTSO uses the ***row-major CSR format*** by default. This means that in the transposed mode the matrix should be stored in the column-major format.

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

Table 3: 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.

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 usually has lower performance than row-mode parallel solving. Table 3 summarizes a general description of the performance of parallel solving, as well as our recommendations.

5.7 Solve Multiple Vectors

```
int CKTSO_SolveMV
(
    ICktSo inst,
    size_t nrhs,
    const double b[],
    size_t ld_b,
    double x[],
    size_t ld_x,
    bool row0_column1
);
```

`` int CKTSO_L_SolveMV (ICktSo_L inst,   size_t nrhs,   const double b[],   size_t ld_b,   double x[],   size_t ld_x,   bool row0_column1); ```
`` virtual int __cktso_dummy::SolveMV (size_t nrhs,   const double b[],   size_t ld_b,   double x[],   size_t ld_x,   bool row0_column1) = 0; ```
`` virtual int __cktso_l_dummy::SolveMV (size_t nrhs,   const double b[],   size_t ld_b,   double x[],   size_t ld_x,   bool row0_column1) = 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 array `b[]` is of length `n*ld_b`, and it stores the right-hand-vectors vector by vector on input. `ld_b` is the leading dimension of `b[]`. If `ld_b=0`, `ld_b=n`. In other cases, `ld_b` cannot be smaller than `n`.
- The array `x[]` is of length `n*ld_x`, and it stores the solution vectors vector by vector on output. `ld_x` is the leading dimension of `x[]`. If `ld_x=0`, `ld_x=n`. In other cases, `ld_x` cannot be smaller than `n`. The address of array `x[]` may be same as the address of array `b[]`. If so, the solution vectors are overwritten on the right-hand-side vectors.
- The parameter `row0_column1` specifies whether the matrix is transposed (i.e., whether to solve $\mathbf{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$).

The meaning of *leading dimension* is illustrated in Figure 2. When accessing a submatrix in a full matrix, using leading dimension is helpful. In short, leading dimension is the size of the first dimension of the full matrix, with respect to the storage order. If the matrix is stored in the row-major order, to get the next row head of the submatrix, one simply adds the leading dimension to the current row head. It is same for column-major stored matrices, if submatrices are accessed in the column order.

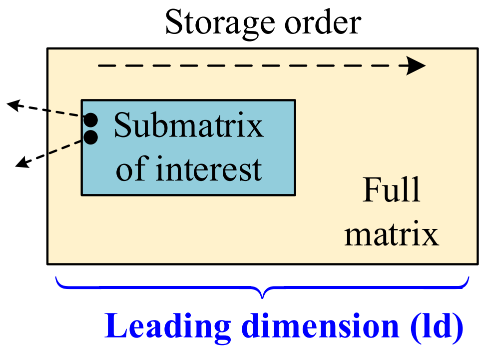

Figure 2: Leading dimension.

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 on CPU. Sorting factors is a necessary step for the GPU acceleration module. It is strongly recommended to sort the factors before initializing the GPU acceleration module.

- 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,
    bool row0_column1,
    char scaling,
    bool fuse_mac
);
```

```
int CKTSO_L_Statistics
(
    ICktSo_L inst,
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem,
    bool row0_column1,
    char scaling,
    bool fuse_mac
);
```

```
virtual int __cktso_dummy::Statistics
(
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem,
    bool row0_column1,
    char scaling,
    bool fuse_mac
) = 0;
```

```
virtual int __cktso_l_dummy::Statistics
(
    long long *factor_flops,
    long long *solve_flops,
    long long *factor_mem,
    long long *solve_mem,
    bool row0_column1,
    char scaling,
    bool fuse_mac
) = 0;
```

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

- Each of the first four parameters, `factor_flops`, `solve_flops`, `factor_mem` and `solve_mem`, can be `NULL` if not needed. No initialization for the four parameters is required. The memory access volumes are reported in bytes.
- The parameter `row0_column1` specifies whether the floating-point operations and memory access volumes are counted for solving $\mathbf{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$.
- The parameter `scaling` specifies whether to count floating-point operations and memory access volume of scaling. Zero means that they are not counted. A positive value means that they are counted. A negative value means that whether they are

counted depends on the solver settings.

- The parameter `fuse_mac` specifies whether a multiply-and-accumulate operation is counted for a single floating-point operation.

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, especially when Newton-Raphson iterations diverge, the memory usage may be more than required. This function can clean up such unnecessary memory consumption.

5.11 Calculate Determinant

```
int CKTSO_Determinant
(
    ICktSo inst,
    double *mantissa,
    double *exponent
);
```

```
int CKTSO_L_Determinant
(
    ICktSo_L inst,
    double *mantissa,
    double *exponent
);
```

```
virtual int __cktso_dummy::Determinant
(
    double *mantissa,
    double *exponent
) = 0;
```

```
virtual int __cktso_l_dummy::Determinant
(
    double *mantissa,
    double *exponent
) = 0;
```

This function calculates the determinant of the matrix, after the matrix has been factorized. The determinant is expressed in the form of $\mathrm{mantissa} \times 10^{\mathrm{exponent}}$, where $1 \le |\mathrm{mantissa}| < 10$.

- If the matrix is complex, the parameter `mantissa` points to a complex number, while `exponent` still points to a real value.

5.12 Factorize and Solve

```
int CKTSO_FactorizeAndSolve
(
    ICktSo inst,
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
);
```

```
int CKTSO_L_FactorizeAndSolve
(
    ICktSo_L inst,
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
);
```

```
virtual int __cktso_dummy::FactorizeAndSolve
(
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
) = 0;
```

```
virtual int __cktso_l_dummy::FactorizeAndSolve
(
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
) = 0;
```

This function factorizes the matrix and then solves the linear system. Whether to use fast factorization and parallel solving is determined by a heuristic method according to past factorizations.

5.13 Refactorize and Solve

```
int CKTSO_RefactorizeAndSolve
(
    ICktSo inst,
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
);
```

```
int CKTSO_L_RefactorizeAndSolve
(
    ICktSo_L inst,
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
);
```

```
virtual int __cktso_dummy::RefactorizeAndSolve
(
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
) = 0;
```

```
virtual int __cktso_l_dummy::RefactorizeAndSolve
(
    const double ax[],
    const double b[],
    double x[],
    bool row0_column1
) = 0;
```

This function refactorizes the matrix and then solves the linear system. Whether to use parallel solving is determined by a heuristic method according to past factorizations.

5.14 Matrix Analysis with Outputs

```
int CKTSO_Analyze2
(
    ICktSo inst,
    bool is_complex,
    int n,
    const int ap[],
    const int ai[],
    const double ax[],
    int threads,
    int rperm[],
    int cperm[],
    double rscale[],
    double cscale[]
);
```

```
int CKTSO_L_Analyze2
(
    ICktSo_L inst,
    bool is_complex,
    long long n,
    const long long ap[],
    const long long ai[],
    const double ax[],
    int threads,
    long long rperm[],
    long long cperm[],
    double rscale[],
    double cscale[]
);
```

```
virtual int __cktso_dummy::Analyze2
(
    bool is_complex,
    int n,
    const int ap[],
    const int ai[],
    const double ax[],
    int threads,
    int rperm[],
    int cperm[],
    double rscale[],
    double cscale[]
) = 0;
```

```
virtual int __cktso_l_dummy::Analyze2
(
    bool is_complex,
    long long n,
    const long long ap[],
    const long long ai[],
    const double ax[],
    int threads,
    long long rperm[],
    long long cperm[],
    double rscale[],
    double cscale[]
) = 0;
```

This function is similar to `CKTSO_Analyze` or `CKTSO_L_Analyze`, with the only difference that this function can output the permutation and scaling vectors. This functionality can be used to get the pre-processing results of CKTSO for subsequent custom processing.

- The vector `rperm[]` returns the row permutation vector for minimizing fill-ins, if it is not `NULL`. The memory space should be pre-allocated by the user, which is of length `n`. `rperm[i]=j` means that row `i` in the permuted matrix is row `j` in the original matrix. The similar vector `cperm[]` returns the column permutation vector for minimizing fill-ins, if it is not `NULL`.
- The vector `rscale[]` returns the row scaling vector, if it is not `NULL`. The memory space should be pre-allocated by the user, which is of length `n`. The scaling vector corresponds to the original (i.e., unpermuted) matrix. `rscale[i]=s` means that row `i` of the original matrix should be multiplied with `s`. The similar vector `cscale[]` returns the column scaling vector, if it is not `NULL`. Please note that whether scaling is actually performed in CKTSO depends on `iparm[7]`. Even if scaling is disabled, the scaling vector values are still calculated in the pre-processing stage and will not change later.

5.15 Extract LU Factors

```
int CKTSO_ExtractFactors
(
    ICktSo inst,
    size_t lp[],
    int li[],
    double lx[],
    size_t up[],
    int ui[],
    double ux[],
    int rperm[],
    int cperm[],
    double rscale[],
    double cscale[]
);
```

```
int CKTSO_L_ExtractFactors
(
    ICktSo_L inst,
    size_t lp[],
    long long li[],
    double lx[],
    size_t up[],
    long long ui[],
    double ux[],
    long long rperm[],
    long long cperm[],
    double rscale[],
    double cscale[]
);
```

```
virtual int __cktso_dummy::ExtractFactors
(
    size_t lp[],
    int li[],
    double lx[],
    size_t up[],
    int ui[],
    double ux[],
    int rperm[],
    int cperm[],
    double rscale[],
    double cscale[]
) = 0;
```

```
virtual int __cktso_l_dummy::ExtractFactors
(
    size_t lp[],
    long long li[],
    double lx[],
    size_t up[],
    long long ui[],
    double ux[],
    long long rperm[],
    long long cperm[],
    double rscale[],
    double cscale[]
) = 0;
```

This function extracts the LU factors in the CSR format for separated $\mathbf{L}$ and $\mathbf{U}-\mathbf{I}$, where $\mathbf{L}$ is with the diagonal and $\mathbf{U}-\mathbf{I}$ is without the diagonal (the diagonal elements of **U** are all 1.0). Please note that the extracted matrices are permuted, pivoted, and scaled (if scaling is enabled, depending on `iparm[7]`). The permutation and scaling vectors can also be extracted.

- The vectors `lp[]`, `li[]`, and `lx[]` return the CSR storage of matrix **L**. The memory spaces of them should be pre-allocated by the user, where `lp[]` is of length `n+1`, `li[]` is of length `oparm[5]`, and `lx[]` is of length `oparm[5]` for double or `oparm[5]`*2 for complex. `lx[]` can be `NULL` if the values are not needed.
- The vectors `up[]`, `ui[]`, and `ux[]` are similar to `lp[]`, `li[]`, and `lx[]`, respectively, with the only difference that their lengths are related to `oparm[6]`.
- The vector `rperm[]` returns the row permutation vector for minimizing fill-ins, if it is not `NULL`. The memory space should be pre-allocated by the user, which is of length `n`. `rperm[i]=j` means that row `i` in the permuted matrix is row `j` in the original matrix.
- The vector `cperm[]` returns the final column permutation vector, if it is not `NULL`. The memory space should be pre-allocated by the user, which is of length `n`. `cperm[i]=j` means that column `i` in the permuted matrix is column `j` in the original matrix. The final column permutation includes both the fill-in reduction column permutation and the numerical pivoting order (this is different from the vector `cperm[]` of `CKTSO_Analyze2` or `CKTSO_L_Analyze2`, which only returns the fill-in reduction column permutation).
- The vector `rscale[]` returns the row scaling vector, if it is not `NULL`. The memory space should be pre-allocated by the user, which is of length `n`. The scaling vector corresponds to the original (i.e., unpermuted) matrix. `rscale[i]=s` means that row `i` of the original matrix should be multiplied with `s`. The similar vector `cscale[]` returns the column scaling vector, if it is not `NULL`. Please note that whether scaling is actually performed in CKTSO depends on `iparm[7]`. Even if scaling is disabled, the scaling vector values are still calculated in the pre-processing stage and can be extracted here.

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.

6.1 Input Parameters

Parameter	Default value	Description

`iparm[0]`: timer control	0	0: no timer
		>0: using a high-precision timer
		<0: using a low-precision timer
`iparm[1]`: pivoting tolerance	1000	In millionth (1/1000000). Pivoting tolerance=`iparm[1]`/1000000
`iparm[2]`: ordering method	0	0: selecting best from all 10 ordering methods
		1-8: using corresponding single minimum degree ordering method
		9-10: using corresponding single nested dissection ordering method
		11: selecting best from 2 nested dissection ordering methods
		12-18: selecting best from `iparm[2]`-10 minimum degree variants (for example, `iparm[2]`=15 means selecting best from minimum degree ordering methods 1 to 5)
		<0: no ordering (using natural order)
`iparm[3]`: threshold for dense node detection	1000	In hundredth. A row with more than $\mathrm{iparm[3]}/100\times\sqrt{\mathrm{n}}$ nonzeros is treated as a dense node and is removed before ordering to save ordering time
`iparm[4]`: metric for ordering method selection	0	>=0: using estimated number of floating-point operations to select best ordering method
		<0: using estimated number of factors to select best ordering method
`iparm[5]`: maximum supernode size	-1	>0: 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
		0: CKTSO determines the maximum supernode size according to the cache size
		-1: no limitation for supernode size
`iparm[6]`: minimum number of columns for supernode detection	64	A supernode must have at least `iparm[6]` columns
`iparm[7]`: whether to perform scaling	0 (Boolean)	Scaling may improve the solution accuracy for some ill-conditioned matrices with the overhead of a small performance drop
`iparm[8]`: whether right-hand-side vector is highly sparse	0 (Boolean)	Typically if 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 for highly sparse right-hand-side vector. It is only effective in sequential column-mode solving
`iparm[9]`: whether to control of thread number for parallel factorization, refactorization, and solving, based on matrix features	1 (Boolean)	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
`iparm[10]`: dynamic memory growth factor	150	In percentage. Growth factor=`iparm[10]`/100. A larger value helps reduce reallocations at runtime but costs more memory usage

`iparm[11]`: initial number of rows for supernode creation	16	CKTSO allocates memory of `iparm[11]` rows when a new supernode is created. A larger value helps reduce reallocations at runtime but costs more memory usage.
`iparm[12]`: static pivoting method	-1	0: using conventional matching-based static pivoting
		>0: using a fill-in aware variant of conventional method
		<0: using a diagonal-first variant of conventional method
`iparm[13]`: synchronization and dynamic adjustment method for threads	0	0: using blocked wait
		<0: using busy wait (busy wait reduces threads' synchronization cost but consumes CPUs)
		>0: using blocked wait and also enables dynamic thread number adjustment based on system workload, while the value means minimum time interval in 100 milliseconds of the adjustment (for example, `iparm[13]=2` means that CKTSO adjusts the thread number at an interval of at least 200 milliseconds). See Section 7 for more details
`iparm[14]`: timeout value for waiting for slave threads to exit	-1	In milliseconds. When calling `CKTSO_DestroySolver` or `CKTSO_L_DestroySolver`, if one or more slave threads do not exit after `iparm[14]` milliseconds, the calling thread will continue. In this case, -13 (resource leak) will be returned
		0:calling thread continues without waiting for threads to exit
		>0: timeout value
		-1: waiting until all slave threads have exited

Please do not change other input parameters (after `iparm[14]`) which are not listed above.

6.2 Output Parameters

Parameter	Description
`oparm[0]`	Time of matrix analysis, in microsecond (us), reported by `CKTSO_Analyze` and `CKTSO_L_Analyze`
`oparm[1]`	Time of factorization or refactorization, in microsecond (us), reported by `CKTSO_Factorize`, `CKTSO_L_Factorize`, `CKTSO_Refactorize`, and `CKTSO_L_Refactorize`
`oparm[2]`	Time of solving, in microsecond (us), reported by `CKTSO_Solve`, `CKTSO_L_Solve`, `CKTSO_SolveMV`, and `CKTSO_L_SolveMV`
`oparm[3]`	Time of sorting factors, in microsecond (us), reported by `CKTSO_SortFactors` and `CKTSO_L_SortFactors`
`oparm[4]`	Number of off-diagonal pivots
`oparm[5]`	Number of nonzeros of $\mathbf{L}$, including diagonal
`oparm[6]`	Number of nonzeros of $\mathbf{U}-\mathbf{I}$, excluding diagonal ($\mathbf{U}$ has the unit diagonal). `oparm[5] + oparm[6]` is the number of LU factors

`oparm[7]`	Number of supernodes
`oparm[8]`	Selected ordering method, from 1 to 10, 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 after `CKTSO_Factorize` or `CKTSO_L_Factorize` with fast factorization enabled. `oparm[14]=n` means that the entire matrix is completed with pivoting reduction, implying no change in the matrix structure
`oparm[15]`	Time of factorization-solve or refactorization-solve, in microsecond (us), reported by `CKTSO_FactorizeAndSolve`, `CKTSO_L_FactorizeAndSolve`, `CKTSO_RefactorizeAndSolve`, and `CKTSO_L_RefactorizeAndSolve`
`oparm[16]`	Predicted number of factors (i.e., number of nonzero elements of $\mathbf{L}+\mathbf{U}-\mathbf{I}$) by symbolic analysis
`oparm[17]`	Predicted number of floating-point operations by symbolic analysis (multiply-and-accumulate is counted as 2 operations)

7. Dynamic Thread Number Adjustment

From version 20240630, CKTSO supports dynamic thread number adjustment based on the system workload. This feature is useful when the system has heavy workloads. In such cases, if the number of created threads equals to the number of cores and all created threads are enabled, it may cause heavy block as different applications can scramble for CPUs. CKTSO can dynamically adjust the number of threads used for parallel processing, based on the system workload of a time interval that has just passed. The created threads may not be all enabled for a specific parallel operation. The decided number of used threads can be smaller than the number of created threads, and the unused threads will be blocked and not consume CPU.

Setting `iparm[13]>0` enables this feature. When enabled, the value of `iparm[13]` means the minimum time interval in 100 milliseconds for the adjustment. All parallel operations excluding matrix analysis will be affected by this feature, if it is enabled. An exception is solving. If CKTSO decides to use some number of threads which does not equal to the number of created threads, solving will be sequential.

For example, if the time interval of two successive calls to `CKTSO_Refactorize` is longer than `iparm[13]*100` milliseconds, each call will decide the number of threads used for refactorization based on the system workload since the last adjustment. If the time interval is shorter than `iparm[13]*100` milliseconds, then the number of used threads will not change from the last adjustment.

Note that dynamic thread number adjustment will introduce some overhead. On Windows, the overhead is about 20 microseconds per adjustment, which is negligible.

On Linux, the overhead is from 100 to several hundred microseconds per adjustment.

8. 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 old Linux systems, "`-lrt`" is also needed. 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 needed in linking. When running the executable, only `cktso.dll` or `cktso_l.dll` is needed. The `*.dll` file should be put together with the executable or in the system directory (i.e., `C:\Windows\` or `C:\Windows\System32\`).

9. Using CKTSO in Circuit Simulators

First, call `CKTSO_CreateSolver` to create a solver instance. After the matrix structure is constructed, call `CKTSO_Analyze`. Please note that `ax` can be `NULL` when calling `CKTSO_Analyze`. However, if `ax` is provided, 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, since `CKTSO_Analyze` determines a matrix ordering based on both the symbolic pattern and numerical values if `ax` is provided.

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, `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.

`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. 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 if `oparm[14]<n` (matrix structure changes), while for all other iterations, 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.