energies

MDPI

Article

Factorisation Path Based Refactorisation for High-Performance LU Decomposition in Real-Time Power System Simulation

Jan Dinkelbach [1], Lennart Schumacher [1,2], Lukas Razik [2,*], Andrea Benigni [2,3] and Antonello Monti [1]

[1] Institute for Automation of Complex Power Systems, RWTH Aachen University, 52062 Aachen, Germany; jdinkelbach@eonerc.rwth-aachen.de (J.D.); lschumacher@eonerc.rwth-aachen.de (L.S.); amonti@eonerc.rwth-aachen.de (A.M.)
[2] IEK-10: Energy Systems Engineering, Forschungszentrum Jülich, 52428 Jülich, Germany; a.benigni@fz-juelich.de
[3] Chair of Methods for Simulating Energy Systems, RWTH Aachen University, 52062 Aachen, Germany
* Correspondence: l.razik@fz-juelich.de

Abstract: The integration of renewable energy sources into modern power systems requires simulations with smaller step sizes, larger network models and the incorporation of complex nonlinear component models. These features make it more difficult to meet computation time requirements in real-time simulations and have motivated the development of high-performance LU decomposition methods. Since nonlinear component models cause numerical variations in the system matrix between simulation steps, this paper places a particular focus on the recomputation of LU decomposition, i.e., on the refactorisation step. The main contribution is the adoption of a factorisation path algorithm for *partial refactorisation*, which takes into account that only a subset of matrix entries change their values. The approach is integrated into the modern LU decomposition method NICSLU and benchmarked against the methods SuperLU and KLU. A performance analysis was carried out considering benchmark as well as real power systems. The results show the significant speedup of refactorisation computation times in use cases involving system matrices of different sizes, a variety of sparsity patterns and different ratios of numerically varying matrix entries. Consequently, the presented high-performance LU decomposition method can assist in meeting computation time requirements in real-time simulations of modern power systems.

Keywords: direct linear solvers; matrix decomposition; high-performance computing; power system simulation

Citation: Dinkelbach, J.; Schumacher, L.; Razik, L.; Benigni, A.; Monti, A. Factorisation Path Based Refactorisation for High-Performance LU Decomposition in Real-Time Power System Simulation. *Energies* **2021**, *14*, 7989. https://doi.org/10.3390/en14237989

Academic Editor: Frede Blaabjerg

Received: 29 October 2021
Accepted: 23 November 2021
Published: 30 November 2021

1. Introduction

1.1. Motivation

The ongoing integration of renewable energy sources into modern power systems has introduced challenges to the methods currently employed for power system simulation. New types of dynamic phenomena, e.g., those caused by a decrease in power system inertia, require smaller simulation step sizes and larger network models. Furthermore, new power electronics-based devices increase the complexity of the power system and its corresponding simulation model. These challenges make power system simulations more computationally demanding.

Real-time simulators applied in hardware-in-the-loop (HiL) setups also face these challenges. Real-time simulators should ensure deterministic computation times for fixed simulation step sizes in order to interact properly with real hardware devices. For a high level of accuracy, they perform electromagnetic transient (EMT) simulations that calculate instantaneous waveform values with small simulation step sizes. However, increasing simulation complexity makes it more difficult to keep computation times below the specified simulation step sizes.

To reduce the computational effort while maintaining a high level of accuracy, recent research has aimed to change the modelling paradigm behind the simulation by considering time-varying phasors that shift the original frequency spectrum and omit predominant carrier frequencies [1]. Thus, the calculation of time-varying phasors according to shifted frequency analysis (SFA) instead of instantaneous waveform values in the EMT domain can enable the use of larger simulation step sizes and alleviate restrictions on the computation times [2].

Complementary to a potential change of the modelling paradigm, our study aimed to speed up the computation times required in each simulation step by applying a high-performance LU decomposition method. Previous research works have shown that the application of high-performance LU decomposition methods, particularly those designed to operate on sparse matrices of power system simulations, can lead to a significant speedup of computation times [3]. Hence, in this work, we considered the application of different LU decomposition methods in real-time simulations and carried out a corresponding performance analysis.

Furthermore, we particularly focus on improving the computational efficiency when simulating power system components with nonlinear models. Nonlinear component models may cause numerically varying entries in the system matrix, i.e., either time-varying or iteratively varying matrix entries. This requires recomputing the LU decomposition of the system matrix once or multiple times during each simulation step. In this case, the symbolic pattern (i.e., the matrix sparsity structure) remains constant, whereas numerical variations occur for a subset of matrix entries. If subsequent recomputations of the matrix LU decomposition are required, repeating computation steps depending on a constant symbolic pattern can be omitted in the *preprocessing* and *factorisation* steps. The remaining computation steps account for numerical variations in the matrix and are combined in the so-called *refactorisation* step. Performing a refactorisation step after numerical variations in the matrix occur can significantly speed up computation times and lead to computationally efficient solutions of systems with nonlinear component models.

During the refactorisation step, we aim to exploit the fact that the number of varying matrix entries is usually small compared to the overall number of nonzero entries in the sparse system matrix. This is the motivation to refrain from a full refactorisation, which applies refactorisation operations to all nonzero entries of the matrix. Instead, a *partial refactorisation* is employed in order to take into account that only a small subset of nonzero matrix entries is numerically varied.

1.2. Related Work

Comparative analysis of the performance of LU decomposition methods for sparse matrices, which occur in power system simulations, has been carried out in previous studies [3,4]. Among others, the LU decomposition methods SuperLU, KLU and NICSLU have been benchmarked in these studies. In particular, it was shown in [3] that the application of NICSLU can yield improved refactorisation computation times, which is also a major motivation for employing NICSLU in this work. An advancement in this work compared to [3] is that refactorisation is sped up by means of partial refactorisation.

LU decompositions for sparse systems are not only applied by MNA based real-time solvers, as described in the present work in Section 2.1. Another potential application is shown in [5], where LU decompositions were used as subroutines of differential-algebraic system of equations (DAE) solvers. This applies, for instance, to SUNDIALS/IDA, a general-purpose DAE solver that is used by the variable time step solver of Dynaωo, an offline power system simulation suite [6,7]. Furthermore, Newton–Raphson (NR) method based solvers for nonlinear systems such as SUNDIALS/KINSOL, e.g., used by Dynaωo's fixed time step solver, also apply LU decompositions. Another application of such NR methods in power system simulation is the solution of load-flow problems based on Runge–Kutta methods [8], whose many Jacobian factorisations have motivated improvements of the applied LU factorisation methods.

However, the computational improvement of LU decomposition methods used by outer NR methods is not the only way to improve the overall computational performance. Instead, the computational performance of the overall calculation can be improved by reducing the total LU factorisations required to reach the final solution. For instance, ref. [9] presents two modern methods of this kind. The first is based on the third-order Newton-like method of Weerakoon, and the second is based on that of Özban. Both methods are modified in a way to benefit from the seventh order of convergence. Although both new methods require more function evaluations and the first method also has one more Jacobian evaluation than its original counterpart, they enable computational savings higher than 20% and 30%, as was shown with the aid of three power system operation scenarios.

However, in the present work, we focus on the improvement of the computational efficiency of the LU decomposition itself, particularly its refactorisation step. Other works have proposed approaches that target the speedup of the refactorisation step of LU decomposition methods. The authors in [10] suggested partition-based decompositions. These methods are based on the idea of partitioning matrices into sub-blocks, some of which contain numerically changing values. Subsequently, from this knowledge, a block-based updating algorithm can be developed for refactorisation. The drawback of this method is that it fully refactorises blocks that contain numerically changing values, without considering which parts of those that are refactorised need to be recomputed and which can be neglected. This leads to a potential unnecessary surplus of computations, especially if there are few numerically changing values. This motivates the improvement of the refactorisation approach by further reducing the number of computations, which is the main aspect of this work.

The research work presented in [11] also aimed to speed up refactorisation. In [11], the sparse matrix solver under investigation was KLU. The authors proposed a pivot validity test to ensure numerical stability as well as a partial refactorisation approach to accelerate the refactorisation process. The partial refactorisation approach discussed in [11] is based on the idea that the factorisation algorithm successively constructs the LU columns, starting from the leftmost column. It uses the previously computed columns for the factorisation. If an entry in the original matrix changes, it is possible to restart the factorisation at its corresponding column in the **LU** matrix and retain all columns to the left of it. Here, it should be mentioned that the **LU** matrix is a single matrix that stores the nonzero entries of the **L** and **U** matrices. A drawback of the algorithm in [11] is that it does not consider the column dependencies within the **LU** matrix. It refactorises all columns to the right of the first numerically varied one in the **LU** matrix, regardless of whether they need to be updated or not, which is a key difference to the approach presented in this paper.

Instead of completely refactorising all columns to the right, one can determine the so-called factorisation path [12,13], which contains the column-wise dependencies of the decomposed matrix. It is assumed that the symbolic pattern of the matrix does not change. Given an LU decomposition and a list of entries in the original matrix that holds information about which entries are numerically changing, one can determine the so-called factorisation path encompassing the columns to be refactorised. This leads to a smaller number of columns that have to be recomputed in order to factorise the numerically varying matrix. This concept of partial refactorisation is adopted within this paper and is described in more detail in Section 3.1. To the knowledge of the authors, no libraries currently implement a factorisation path based refactorisation approach.

1.3. Contribution and Outline

The main contribution of this work is the adoption of a factorisation path algorithm from the literature for the implementation of partial refactorisation in real-time simulations. We propose the integration of partial refactorisation into the modern LU decomposition method NICSLU [14] and provide evidence for its high performance based on experimental analysis. The analysis encompasses a performance comparison against the widely applied LU decomposition methods SuperLU [15] and KLU [16]. By interfacing the proposed LU

decomposition method with the commonly applied Eigen library [17], our implementation can be used not only by the open-source real-time simulator DPsim [18], as performed in this work, but also by other software projects making use of solvers for sparse linear systems.

In the following, Section 2 outlines the fundamentals essential for this work by describing a solution procedure commonly employed in real-time simulations and delineating the main concepts of relevant LU decomposition methods. Section 3 explains the proposed high-performance LU decomposition method, while its evaluation via use cases is covered in Section 4.

2. Fundamentals

2.1. Real-Time Simulation Solution Procedure

In real-time simulation, a common solution procedure to calculate the power system behaviour is modified nodal analysis (MNA), e.g., as employed by the commercial simulators *Hypersim*, *RTDS* [19] and the open-source simulator *DPsim* [18]. MNA provides a formalism to derive an equation system for the simulation of a given power system [20]:

$$\begin{bmatrix} \mathbf{Y} & \mathbf{B} \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{J} \\ \mathbf{F} \end{bmatrix}. \tag{1}$$

According to MNA, the shown LES incorporates the Kirchhoff current laws of all network nodes by means of the node admittance matrix **Y**, the node potentials **V** and the vector **J** of independent current sources. Independent voltage sources as well as either the current- or voltage-controlled ideal voltage and current sources are represented by means of matrices **B** and **C** together with vectors **I** and **F**. To represent the behaviour of dynamic components, the equation system can incorporate previous states to calculate a new system state by including equivalent admittances and sources that embed information on the past [21].

The formulation in Equation (1) is compatible with linear network and ideal source models. However, in simulations with a high level of detail, more complex nonlinear models of dynamic power system components, such as high-order synchronous generator models or models of power electronics devices, are included. In general, the dynamic behaviour of such components can be described by a first-order nonlinear ordinary differential equation (ODE) system:

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, t), \tag{2}$$

where **x** is the vector of the component's state variables, and **u** is the vector of the component's input signals [22]. The literature contains several approaches to incorporate a nonlinear power system component described by an ODE system, as in Equation (2), into the MNA system representation of Equation (1). A traditional approach referred to as delayed-current injection method (DCIM) [23] calculates the nonlinear component behaviour based on the input from the previous time step, i.e., with a delay of one time step in the input, and incorporates the solution as a current source injection into the MNA system formulation [24]. However, this approach, which solves the nonlinear component equations separately from the system equations, may lead to significant inaccuracies and instabilities for larger time steps [19]. Other approaches partially or fully integrate the corresponding nonlinear ODE system into the MNA system formulation [19,20,25,26].

Depending on the type of approach, this can lead to a variation in the entries in the MNA system matrix in two ways:

1. *Time-varying matrix entries*, e.g., time-varying conductance entries that correspond to a high-order nonlinear voltage-behind-reactance (VBR) synchronous machine model [25];
2. *Iteratively varying matrix entries* to run fixed-point algorithms, e.g., the NR method, to solve the nonlinearity by iterating multiple times within one time step. The matrix entries incorporate partial derivatives into the MNA system matrix [19,20,26].

Overall, depending on the considered power system components, the corresponding models and the chosen solution approach, a linear equation system (LES) is derived that can be written in generic matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3}$$

where $\mathbf{A}$ is the MNA based system matrix, $\mathbf{x}$ the solution vector of unknown current and voltage state variables, and $\mathbf{b}$ is the right-hand side (RHS) vector of known current and voltage quantities. Based on the MNA, this LES is built in a systematic manner for a given power system. It is composed by applying a dedicated matrix stamp for each power system component, which corresponds to its impact on the power system's equations.

In the following, as depicted in Figures 1 and 2, we distinguish between two phases in which the substantial computation steps of the MNA based solution procedure and the applied LU decomposition method can be performed. First, the simulator's *initialisation phase* takes place before the start of the simulation, prepares data structures and performs computations, the outcome of which can subsequently be reused. Second, in the simulator's *simulation phase*, the computations are performed for each simulation step in real time by scheduling them according to wall-clock time.

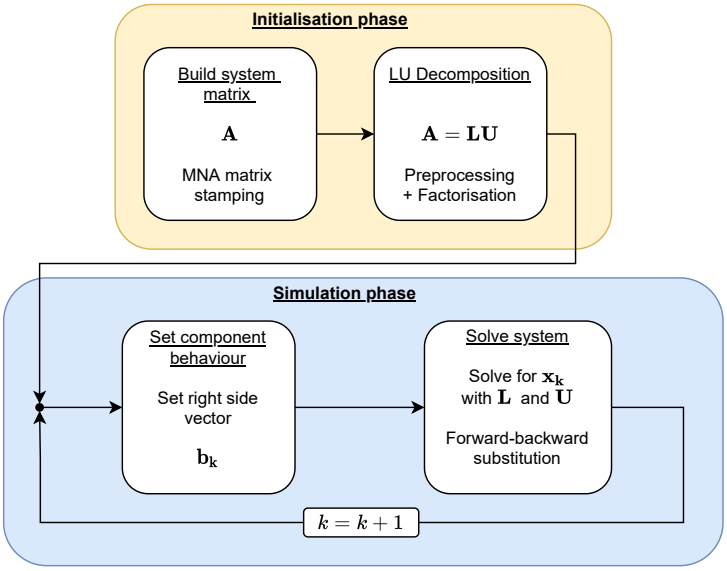

Figure 1. MNA based solution procedure for real-time simulation with a constant system matrix.

If neither time-varying nor iteratively varying matrix entries are present, the MNA system matrix remains constant over the entire simulation phase. Given this condition, as depicted in Figure 1, the MNA system matrix and its corresponding $\mathbf{L}$ and $\mathbf{U}$ decomposition matrices can be computed during the initialisation phase. During the simulation phase, only the RHS vector $\mathbf{b_k}$ is updated according to the components' behaviour at a discrete-time instant k. Based on that, the solution $\mathbf{x_k}$ can be determined by means of forward/backward substitution and reusing the $\mathbf{L}$ and $\mathbf{U}$ matrices, which have been precalculated in the initialisation phase. The main advantage is that the computationally demanding preprocessing and factorisation steps of the LU decomposition method can be performed during the initialisation phase, and their outcome can be reused during the simulation phase.

As explained above, the incorporation of nonlinear power system components into the system solution may lead to time-varying and/or iteratively varying matrix entries. Nonetheless, as shown in Figure 2, the preprocessing and initial factorisation can be performed during the simulator's initialisation phase. During the simulation phase, the matrix entries are updated according to the corresponding component behaviour that leads to an MNA system matrix $\mathbf{A}_\mathbf{k}^{(\mathbf{j})}$, where k is the discrete-time index, and j is the iteration index of

any fixed-point algorithm. Consequently, due to the numerical variation in matrix entries, the LU decomposition matrices $\mathbf{L}_\mathbf{k}^{(\mathbf{j})}$ and $\mathbf{U}_\mathbf{k}^{(\mathbf{j})}$ need to be recalculated; i.e., refactorisation needs to be performed. Based on that, the solution $\mathbf{x}_\mathbf{k}^{(\mathbf{j})}$ for each time k and iteration j can be determined by forward/backward substitution.

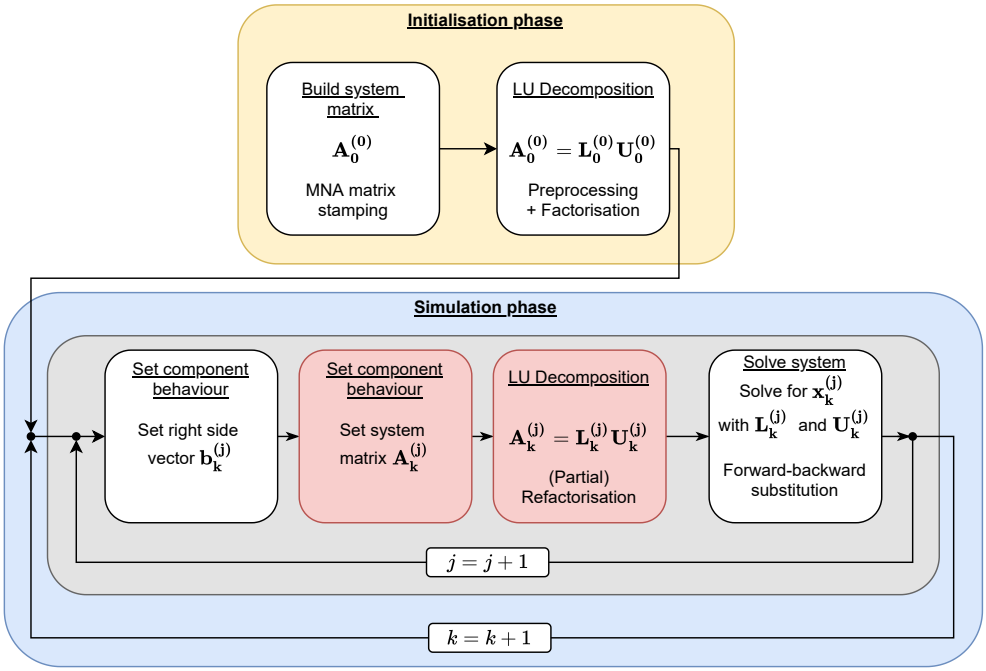

Figure 2. MNA based solution procedure for real-time simulation with a varying system matrix, with k as the discrete-time index and j as the iteration index of any fixed-point algorithm.

In addition to a full refactorisation of the system matrix, in this work, we consider a partial refactorisation. This is motivated by the fact that, usually, only a few nonzero entries undergo numerical variations. The partial refactorisation approach chosen in this work is explained in more detail in Section 3.1.

2.2. LU Decomposition Methods

In the following, we outline the main concepts behind the considered LU decomposition methods. This serves as a basis for understanding how the partial refactorisation approach is integrated into NICSLU and highlights key differences with respect to the methods SuperLU and KLU, against which we benchmark the proposed method. In this paper, we distinguish between four main steps of the considered LU decomposition methods:

1. Preprocessing: Preprocessing involves steps for the scaling and ordering of the matrix. This improves the numerical properties of the matrix and the performance of the LU decomposition by reducing the *fill-in* (i.e., additional nonzero entries arising during the factorisation).
2. Factorisation: The factorisation step performs the actual decomposition of the original matrix into its corresponding **L** and **U** matrices.
3. Refactorisation: Refactorisation implies the simplification of the factorisation procedure if subsequent LU decompositions are computed. Optionally, it may involve the use of a partial refactorisation approach.
4. Solution: The computation of the solution vector is commonly based on a forward/backward substitution approach that makes use of the decomposed **LU** matrix.

SuperLU [4,15,27] is a sparse LU decomposition method based on a matrix characteristic named supernodes. The aforementioned general steps are implemented by SuperLU as follows:

1. Preprocessing: Optionally, SuperLU can determine diagonal scaling matrices for stability. It then proceeds to calculate the column ordering of the input matrix, which is determined to preserve sparsity (fill-in reduction).
2. Factorisation: To compute a numeric factorisation with partial pivoting, the method aims to exploit the supernodal structure of the input matrices and calculates the LU decomposition using the scaling (optionally) and fill-in ordering matrices.
3. Solution: To solve for the RHS, SuperLU uses forward/backward substitution.

KLU [16] is also designed to operate on sparse matrices, but it does not rely on supernodes. In contrast, KLU performs additional preordering steps to yield a better-suited matrix for factorisation. The steps of KLU encompass the following:

1. Preprocessing: KLU utilises the block triangular form (BTF) transformation, which ensures a zero-free diagonal and orders the matrix to a form in which the resulting matrix has smaller block matrices on its diagonal and zero entries below it. Then, the Approximate Minimum Degree Ordering Algorithm (AMD) is performed for fill-in reduction on each of these blocks.
2. Factorisation: In the numerical procedure, KLU factorises each block independently with partial pivoting using the Gilbert–Peierls algorithm [28].
3. Refactorisation: If one needs another factorisation of a matrix with different numerical values but the same symbolic pattern, KLU can perform factorisation without recomputing the fill-in reduction or partial pivoting.
4. Solution: KLU successively solves for the unknown vector using forward/backward substitution starting at the lowest block.

The main focus of NICSLU [14,29] is on the parallelisation of sparse LU factorisations. This is conducted by analysing column dependency in the preprocessing step. Furthermore, NICSLU implements the main steps as follows:

1. Preprocessing: NICSLU performs MC64 scaling, which places large values on the diagonal and enhances numerical stability. As opposed to KLU, AMD is then used on the entire matrix. As a final preprocessing step, static symbolic factorisation is performed to determine parallelisability. The decision of whether to use parallelisation is up to the user.
2. Factorisation: In contrast to KLU, factorisation is performed on the entire matrix. The factorisation consists of two main steps: first, a symbolic factorisation to determine the resulting column structure; second, a numeric factorisation with partial pivoting using the Gilbert–Peierls algorithm.
3. Refactorisation: Numeric factorisation of the whole matrix without partial pivoting reusing the scaling, permutation and pivoting choices can be conducted for different numerical values, but only for the same symbolic pattern.
4. Solution: The RHS is solved using forward/backward substitution on the entire matrix.

3. High-Performance LU Decomposition

3.1. Partial Refactorisation Concept

In the following, we explain the concept of partial refactorisation, which was chosen in this work to improve the computation time of the refactorisation step. To begin with, having an LU factorisation, one can utilise it to determine the column dependencies in the *left-looking* factorisation algorithm. The factorisation algorithm is called left-looking, since it computes the LU decomposition column-wise from left to right. That is, it reads the data of the **LU** matrix computed in prior iterations, which are stored in the columns to the left of the current column. Here and throughout the entire paper, we refer to the **LU** matrix as the single matrix that stores the nonzero entries of the **L** and **U** matrices in a combined manner and which is equal to $\mathbf{L} - \mathbf{I} + \mathbf{U}$. The main idea is that if one knows the column dependencies of the **LU** matrix, any subsequent refactorisation can be expressed as a factorisation of a subset of the **LU** columns. This was discussed by Chan, Brandwajn and Tinney [12,13] and is referred to as the factorisation path algorithm. The algorithm

was adopted in this study for partial refactorisation and is described in more detail in the following.

The factorisation path algorithm stems from a symbolic view of the LU factorisation procedure, i.e., only considering the nonzero entries but not their numerical values. In each iteration k, the k-th column of A, denoted in the following as $\mathbf{a_k}$, is factorised and stored in the **LU** matrix. The k-th column factorisation depends on $\mathbf{a_k}$ and the factorised columns $\mathbf{y_j}$, with $j < k$, computed prior to iteration k, which have been stored in the **LU** matrix.

In order to track how numerical variations in the original matrix affect the factorisation, the values in the **LU** matrix that depend on these varying entries need to be identified. If any value of $\mathbf{a_k}$ changes, it affects the corresponding column $\mathbf{y_k}$ in the **LU** matrix and all subsequent columns to the right that depend on it. Since the left-looking algorithm treats entries above the diagonal, i.e., entries having a row index i with $i < k$, differently to those below it, one has to take into account the exact position of a changing entry when computing the factorisation path.

First, an example is presented to illustrate the factorisation path computation. In Figure 3, one can see a sample pattern of an **L** matrix. The decomposition here is assumed to be performed without permutations and to be symmetric, i.e., $\mathbf{U} = \mathbf{L}^T$, for simplicity. Further, we assume that the entries in $(2, 2)$ and $(4, 4)$ of the original matrix have been changed.

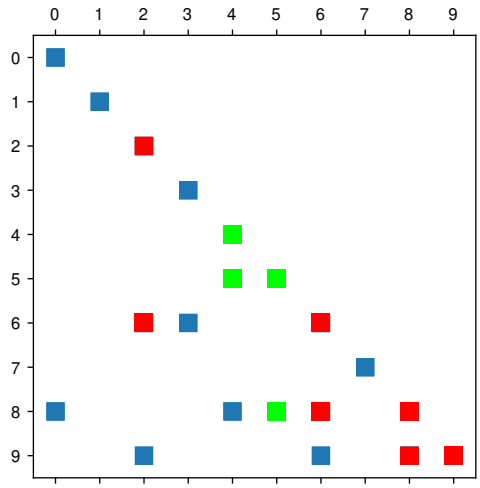

Figure 3. **L** matrix of a symmetric LU decomposition, with changing entries in $(2, 2)$ and $(4, 4)$. The singleton path is marked in red for $(2, 2)$ and in green for $(4, 4)$.

Starting from the first changed entry $(2, 2)$ in the **L** matrix, one finds the closest off-diagonal entry, which is $(6, 2)$. Since the value is below the diagonal, the next loop iteration starts at the corresponding diagonal entry $(6, 6)$. This procedure is repeated until reaching either the end of the matrix or an entry that is already part of the path.

After the first search loop for the initial column 2, the factorisation path is $\{2, 6, 8, 9\}$, consisting of all columns that have been found so far and need to be refactorised. Next, the main loop has to be repeated for the second changed entry $(4, 4)$. The affected entries are denoted in green. This time, only columns 4 and 5 are added to the path, since 8 and its subsequent columns are already in the path. The final factorisation path is $\{4, 5, 2, 6, 8, 9\}$.

In the following, the underlying algorithm is described in its general form. For any numerically varied entry (i, j) in the original matrix, one has to determine its corresponding entry $(\hat{i}, \hat{j})$ in the permuted **LU** matrix. In NICSLU, those permutations originate from AMD, MC64 and partial pivoting.

First, the column index $k = \min(\hat{i}, \hat{j})$ of the diagonal entry (k, k) is determined among the entries in the **LU** matrix and is added to the factorisation path. Then, beginning from the diagonal entry (k, k), the main loop of the algorithm starts. In each iteration, the algorithm finds the closest off-diagonal entry in the **LU** matrix. If the closest entry $(\tilde{i}, \tilde{j})$ is above the diagonal, it chooses $k = \tilde{j}$, and if the entry is below, it chooses $k = \tilde{i}$. Finally, the

column index k of the corresponding diagonal entry (k, k) is appended to the factorisation path. The loop finishes if either no further off-diagonal entry is found or $k = n$, where n is the number of rows/columns of the matrix.

The algorithm above considers a single changed value, and it delivers a so-called *singleton path*. The union of all singleton paths, starting from $(1, 1)$, is the full factorisation path of a matrix. The full factorisation path contains all columns of the **LU** matrix and will not achieve any performance improvement if used during refactorisation, since the iteration over the full factorisation path will result in the recomputation of all columns of the **LU** matrix. The benefit of this full path is that it encodes the inner column dependencies of the **LU** matrix; that is, it encodes which columns are dependent on one another and, thus, if any column changes numerically, which columns will be affected and need to be recomputed. It can be used to determine the columns to be refactorised if only a subset of values vary in the original matrix. This potentially causes a performance improvement, since the corresponding factorisation path no longer includes all of the columns.

The described algorithm has to be performed on the **LU** matrix, since this matrix contains, in addition to ordering permutations, the partial pivoting permutation and the fill-in values, which affect the inner column dependencies that the factorisation path algorithm determines. Therefore, the algorithm can only be performed after a first LU factorisation has been performed. The first full LU factorisation can be performed in the simulator's initialisation phase (see Figure 2), and then the computationally less expensive partial refactorisation is executed during the simulation phase.

3.2. Implementation

3.2.1. Partial Refactorisation in NICSLU

During the implementation of the partial refactorisation based on a factorisation path, it is important to carefully consider the data structures and ordering methods of NICSLU [30] in order to properly compute the factorisation path on the **LU** matrix. On the one hand, NICSLU uses the compressed sparse row (CSR) matrix format for internal storage. This means that the matrix's nonzero elements are only saved row-wise in a vector. Moreover, another vector stores the indices of all elements of the first vector that are the first element of a row. This saves memory space for nonzero elements and allows efficiently accessing the matrix in a row-wise manner. This has to be considered when computing the column-wise part of the algorithm.

On the other hand, the ordering methods applied by NICSLU have to be taken into account during the determination of the factorisation path. As mentioned in Section 2.2, NICSLU uses standard AMD and MC64 in the preprocessing step. In addition, NICSLU performs partial pivoting in the factorisation step. Given these reorderings during preprocessing and factorisation, an entry in the original matrix can be permuted to a different location in the **LU** matrix. Hence, to obtain the corresponding permutations, preprocessing and a regular factorisation have to be executed first. These permutations can be applied to the locations of numerically varying entries in the original matrix and, as a result, the locations of numerically varying entries in the **LU** matrix are obtained. Based on the locations in the **LU** matrix, the factorisation path for partial refactorisation is determined.

After the determination of the factorisation path, the routine that actually computes the partial refactorisation of the **LU** matrix can be called. According to the factorisation path, NICSLU iterates over the subset of columns and only refactors these columns. After this step, the implementation is the same as the original and can be used as such, e.g., calling the solve routine for RHS solving. Additionally, it should be remarked that NICSLU's parallel interface was not employed in this work, since it did not yield any significant performance improvements in the test cases.

If the partial pivoting choice with its corresponding permutation is kept while a subset of matrix entries varies numerically throughout the simulation, this may deteriorate the numerical stability of subsequent LU factorisations. To address this problem, the authors of [11] proposed a pivot validity test, which ensures that the pivoting choice remains

feasible. In the presented work, pivot validity testing was adopted by checking whether the pivot element remained above a certain threshold. This testing was incorporated into our NICSLU based refactorisation method. However, the pivot validity test was always passed, and consequently, it had no impact on the considered use cases. Nonetheless, the integrated pivot validity test shall ensure the numerical stability of our proposed method when applying it in other use cases.

3.2.2. Integration into DPsim Real-Time Simulator

The performance of the LU decomposition methods was analysed in real-time simulation experiments. We employed the open-source C++ real-time simulator DPsim [31]. DPsim is implemented based on the C++ library for linear algebra named Eigen [17]. The Eigen library offers support for custom linear solver libraries. By interfacing our implementations with the open-source Eigen library, they shall become reusable by other software projects. For the integration of a new linear solver, the three main steps, as presented in Section 2.2, must be interfaced:

- The preprocessing step using the `analyzePattern` routine,
- the factorisation step using the `factorize` routine and
- the solution step using the `_solve_impl` routine.

Eigen's standard implementation for the LU decomposition of sparse matrices is based on the SuperLU package [32] and named *EigenSparse*. To date, EigenSparse has been employed with the real-time simulator DPsim. Here, we extended the Eigen library by two new solver interfaces named *EigenNICSLU* and *EigenPartialNICSLU*.

For EigenNICSLU, the preprocessing routine `analyzePattern` not only determines the permutations but also performs the allocation and initialisation of matrix- and NICSLU-specific data structures. In addition, the key difference between EigenSparse and EigenNICSLU is the refactorisation capability of the latter, which was implemented as an additional, distinct routine named `refactorize`.

Furthermore, EigenPartialNICSLU interfaces and implements the factorisation path algorithm. EigenPartialNICSLU uses the same `analyzePattern` routine as EigenNICSLU. In addition to this, it modifies the `factorize` routine and adds the `refactorize` routine. The factorisation path computation is performed after the first factorisation as part of `factorize`. It receives a list of tuples of numerically varied entries as input. Using these entries, it computes the factorisation path with respect to NICSLU's permutations. It is assumed that the pattern of numerically varying entries stays the same. Therefore, the factorisation path is only computed once. Then, in each subsequent call of `refactorize`, the partial refactorisation implementation for NICSLU is called, which iterates over the columns in the factorisation path.

Apart from NICSLU interfaces, there has been a previous implementation of an interface for KLU in Eigen. However, at present, the KLU interface does not provide a refactorisation implementation. The integration of the factorisation path based partial refactorisation into Eigen's KLU interface could be part of further research work, but it is beyond the scope of this paper.

4. Evaluation via Use Cases

4.1. Factorisation in Real-Time Simulation

The following performance analysis of the NICSLU method was carried out with the real-time simulator DPsim. In the first use case, we exclusively investigated the factorisation and solution performance of NICSLU. We benchmarked the two LU decomposition steps and intended to highlight potential performance gains that can be achieved in real-time simulation scenarios, in which the underlying system matrix remains constant and in which no refactorisation is required.

For this purpose, we considered the simulation of the WSCC 9 bus benchmark system by means of time-varying phasors according to the SFA. The benchmark system incorporates $K = 9$ network nodes, which leads to a matrix size of $N = 48$, and the number of

nonzero matrix entries is $NNZ = 204$. Here, the benchmark system uses a linear network representation and models the synchronous generation in a simplified manner by means of ideal voltage sources.

As explained in Section 2.1, such a model is compatible with the MNA formulation in Equation (1) and yields a constant MNA system matrix. Given a constant system matrix, the LU factorisation has to be performed only once in the simulator's initialisation phase, as depicted in Figure 1, and refactorisation of the matrix is not required. Instead, the solution vector has to be recalculated in each simulation step.

We compared the EigenNICSLU implementation, as described in Section 3.2.2, with the previously available EigenSparse implementation based on SuperLU. In addition, to indicate the impact of the underlying matrix size on the performance analysis, we synthesised larger systems by interconnecting copies of the original benchmark system through additional transmission lines between the load buses. By interconnecting up to 20 system copies, matrix sizes of up to $N = 1134$ were considered.

The computer system used has an AMD Ryzen 7 2700X 3.7 GHz (4.3 GHz Turbo) and 8 cores/16 threads running Fedora Linux with kernel version `5.13.19-200.fc34.x86_64`. GCC `v11.2.1` with a global `-O2` optimisation flag was employed for compilation. The wall-clock times for factorisation and RHS solving were measured. Before and after each call, monotonic clock values were measured using the `chrono` library from the C++ standard libraries. The elapsed time was calculated. For both the factorisation and solution, the average computation time per single step was determined. To diminish the effect of variance, the measurements were repeated 100 times, and the average was computed.

The measured computation times of the factorisation are shown in Figure 4. Here, the speedup is defined as the improvement of the computation speed of EigenNICSLU compared to EigenSparse in relative terms and is calculated as the computation time of EigenSparse divided by that of EigenNICSLU. As can be seen, EigenNICSLU leads to a speedup of between 3.7 and 5.9 compared to EigenSparse. Overall, the speedup slightly increases for larger matrix sizes, while fluctuations of this trend occur depending on the underlying sparsity pattern. It should be noted that the speedup of the factorisation improves the computation time of the initialisation phase only and has no impact on the computation times during the simulation phase.

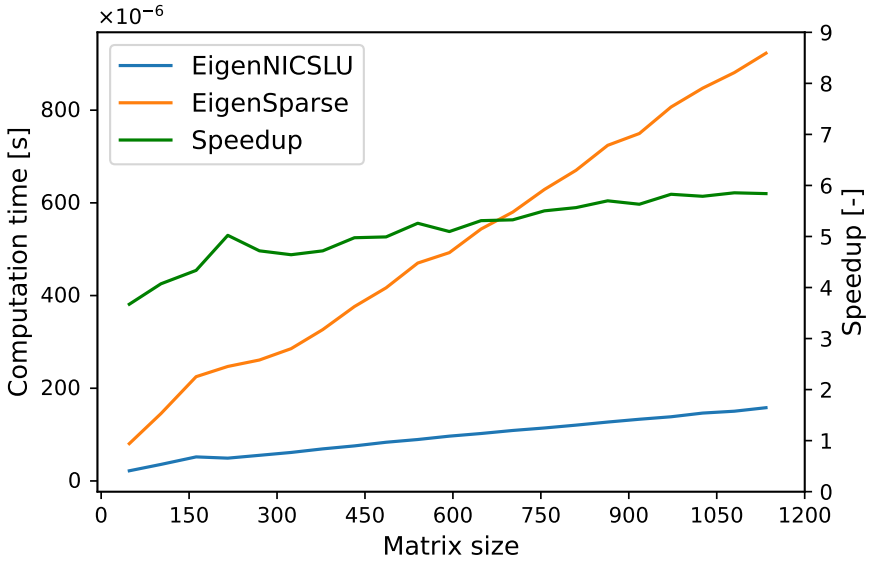

Figure 4. Factorisation computation times for an increasing number of system copies (the speedup is defined as the computation time of EigenSparse divided by that of EigenNICSLU).

During the simulation phase, the time for the computation of the solution vector is of particular relevance, i.e., the time for the computations according to the specific forward/backward substitution approach applied by an LU decomposition method using

its previously factorised **LU** matrix. As depicted in Figure 5, the computation of the solution vector with EigenNISCLU is between 2.9 and 3.7 times as fast as with EigenSparse. Here, the speedup of EigenNISCLU slightly decreases for larger matrix sizes.

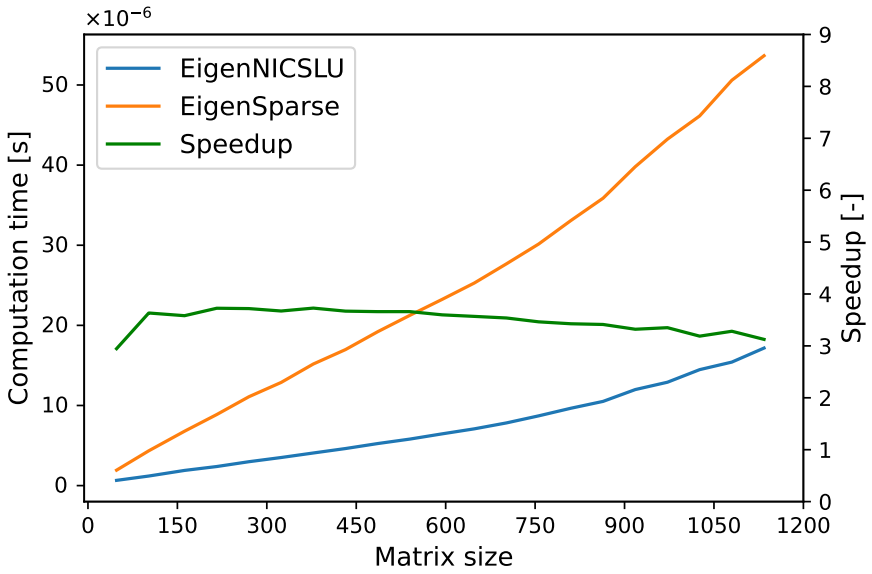

Figure 5. Solution computation times for an increasing number of system copies (the speedup is defined as the computation time of EigenSparse divided by that of EigenNICSLU).

4.2. Refactorisation in Real-Time Simulation

In this use case, we analysed the performance of the refactorisation and solution steps of the NICSLU method. As explained in Section 2.1, the computational efficiency of both refactorisation and solution steps are of particular relevance in real-time simulation scenarios, in which the integration of nonlinear component models leads to varying matrix entries. In addition, we considered the proposed partial refactorisation implementation and comparatively analysed its potential performance improvements.

As a use case, we performed an EMT simulation with a high level of detail of the aforementioned WSCC 9 bus benchmark system. Instead of incorporating ideal voltage source models to represent the synchronous generation, we employed a detailed model that describes the component behaviour by a ninth-order nonlinear ODE system. The model is a VBR model [25], which involves time-varying conductances. These conductances are updated within the MNA system matrix once per simulation step and, hence, require matrix refactorisation. The number of network nodes of the EMT benchmark system is $K = 9$, the matrix size is $N = 63$ with $NNZ = 198$, and the number of varying matrix entries, which originate from the included VBR models, is $NVE = 27$. In this use case, the measurements of the average computation time for refactorisation and solution steps were repeated 20 times to reduce the effect of variance.

Considering the full refactorisation of the system matrix in each time step, Figure 6 demonstrates that EigenNICSLU results in a speedup of 18.9 compared to EigenSparse for the computation of the refactorisation. Moreover, the computation times required by EigenNICSLU for determining the solution vector are 4.3 times as fast as with EigenSparse.

For a better understanding of the impact of partial refactorisation, Figure 7a shows the original input matrix pattern in this use case. The numerically varying entries are marked in red. This matrix was permuted and factorised in NICSLU to yield the pattern visible in Figure 7b. Here, the entries changing between subsequent LU factorisations are marked in red. It is observed that only a few columns have varying entries. Those columns are $\{36, 55, 56, 57, 58, 59, 60, 61, 62\}$, which make up the resulting factorisation path when using the adopted algorithm. The path contains nine columns, meaning that only 14.3% of all columns have to be refactorised.

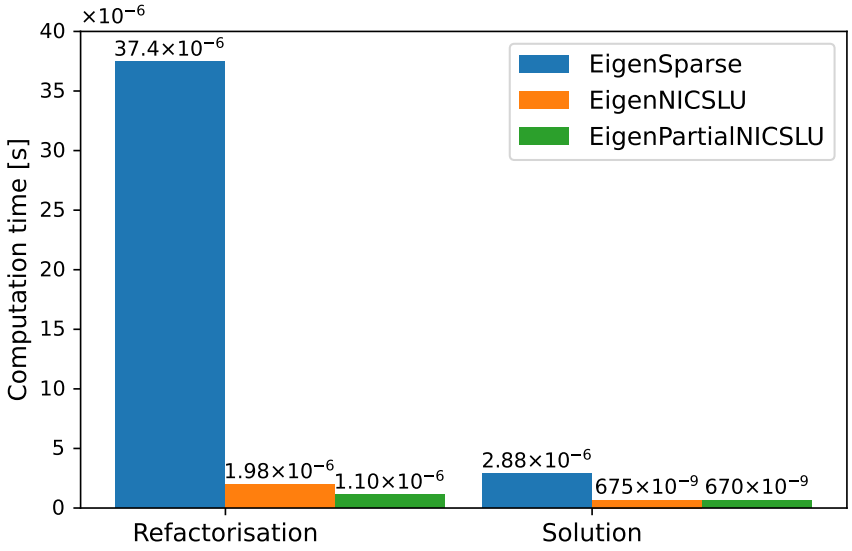

Figure 6. Refactorisation and solution computation time per simulation step.

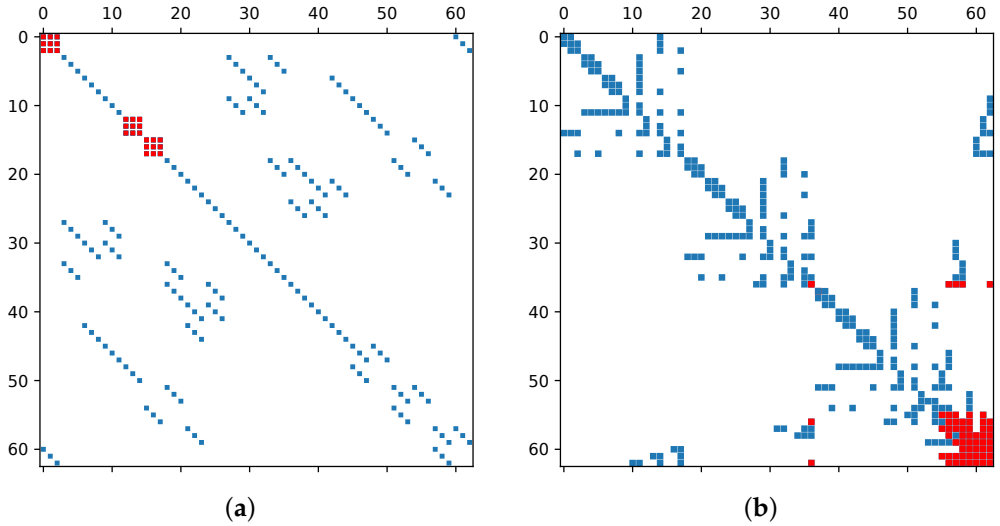

Figure 7. Matrix patterns with varying elements. (**a**) Original input matrix **A** with the numerically changing entries marked in red; (**b**) decomposition matrix **LU** with changing entries between subsequent factorisations marked in red.

When the factorisation path is considered during refactorisation by employing EigenPartialNICSLU, the refactorisation computation time is further improved, as shown in Figure 6. The corresponding speedup of EigenPartialNICSLU compared to EigenNICSLU was 1.8. The solution computation time remained the same, as the resulting LU decomposition matrix did not differ between EigenPartialNICSLU and EigenNICSLU. Furthermore, no modifications to the RHS solving routine were performed in EigenPartialNICSLU, so in both cases, the same computations were performed in the solution step. However, developing and analysing a partial RHS solving routine that uses a suitable dependency path algorithm, similar to what is performed for partial refactorisation, can be part of a future investigation.

4.3. Refactorisation of Large-Scale Matrices

In the following use case, for a thorough analysis of the refactorisation performance, we extended the scope of our study by applying the proposed NICSLU method to matrices with additional characteristics. We considered matrices that, on the one hand, are of very

large size and, on the other hand, exhibit a variety of sparsity patterns that correspond to real networks. Furthermore, we investigated the impact of the number of numerically varying matrix entries on the performance of the partial refactorisation approach. The analyses indicate the performance improvements that can be achieved in simulations of large real networks and take the impact of the number of nonlinear component models into account.

We chose a set of 10 sample matrices with the characteristics shown in Table 1. The analysis involves the consideration of large-scale matrices corresponding to networks with up to 7500 nodes and matrix sizes of up to 220,000. The matrices belong to real networks of the French transmission system. They were extracted from the offline simulation tool suite Dynaωo [6,33], which builds the mathematical problem for time-domain simulation in the form of a DAE from C++ and Modelica models and is capable of conducting steady-state, short-circuit and a variety of network stability studies. Even though the underlying mathematical problem formulation of the matrices in this case differs from the MNA based formulation in the real-time simulation described in Section 2.1, the matrices can serve as a basis for a systematic analysis of the performance of NICSLU and the implemented partial refactorisation approach, because they still exhibit sparsity patterns as they can be found in power system simulations of real networks.

Table 1. Characteristics of matrices from the French transmission system (K = number of nodes, N = matrix size, NNZ = number of nonzeros, d = density factor according to $\frac{NNZ}{N \cdot N}$).

No.	Power Grid	K	N	NNZ	d [%]
(1)	Regional EHV/HV with SL	1000	11,003	44,623	0.036
(2)	French EHV with SL	2000	26,432	92,718	0.013
(3)	French EHV/Regional HV with SL	4000	56,564	183,278	0.0057
(4)	French EHV with VDL	2000	60,236	188,666	0.0051
(5)	F. + one neighbour EHV, SL	3000	47,900	20,5663	0.0089
(6)	F. + one neighbour EHV, VDL	3000	75,300	266,958	0.0047
(7)	F. + neighb. countries EHV, SL	7500	70,434	267,116	0.0054
(8)	F. EHV + regional HV, SL	4000	90,940	316,280	0.0038
(9)	F. EHV + regional HV, VDL	4000	197,288	586,745	0.0015
(10)	F. + neighb. countries EHV, VDL	7500	220,828	693,442	0.0014

In the analysis carried out, the matrices were not refactorised during actual simulations. Instead, the matrices were extracted from the offline simulation tool and then refactorised after systematically randomised numerical variations were introduced. In particular, this enables the systematic analysis of the impact of different ratios of varying matrix entries. In addition, in this analysis, we compared the performance of NICSLU with that of KLU, since previous studies have shown that the latter performs well, as outlined in Section 1.2.

To compare KLU, NICSLU and PartialNICSLU, a measurement environment was set up independently from Eigen. For each solver, a driver (i.e., a program with a `main` routine utilising the solver) that measures the elapsed time of refactorisation was implemented. To impose a certain ratio of varying matrix entries, we randomly varied a subset of rows after the factorisation of those entries. The size of the varied subset is an input parameter set by the user and is indicated here by percentage values.

We analysed the performance of the refactorisation procedures of KLU and NICSLU, as described in Section 2.2, as well as that of PartialNICSLU, as described in Section 3.2.1. First, we considered a fixed percentage of varying matrix entries of 0.1%. The measured computation times for the refactorisation of the 10 sample matrices are depicted in Figure 8. It can be seen that the computation times of NICSLU are lower than those of KLU for all matrices, with computation time speedups in NICSLU ranging from 2.64 to 8.55 compared to KLU. This is in line with previous findings, e.g., in [3]. Furthermore, it can be seen that a further speedup of the refactorisation times for all matrices is accomplished when

PartialNICSLU performs a partial refactorisation based on the factorisation path, which results in speedups from 7.34 to 49.01 compared to KLU.

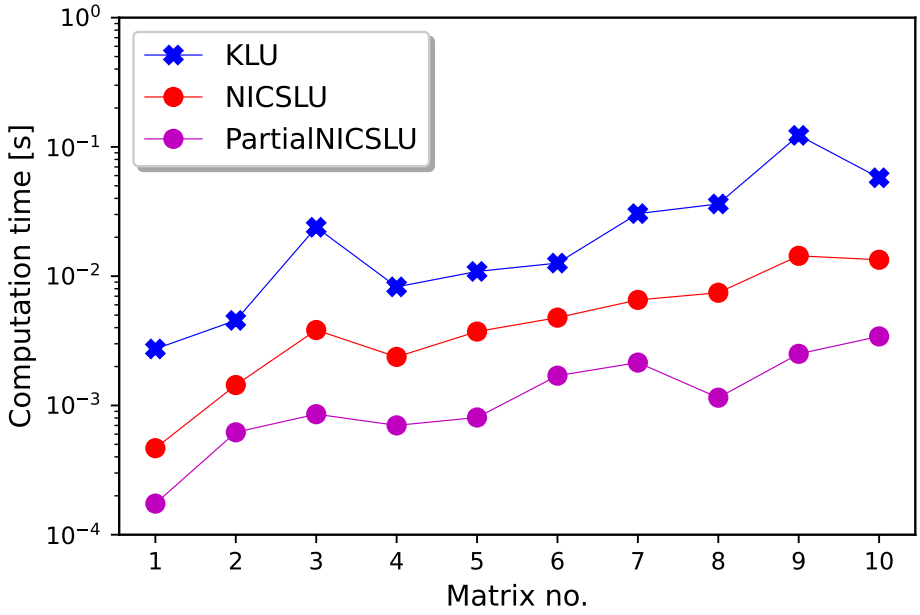

Figure 8. Refactorisation times for large-scale matrices assuming a ratio of varying matrix entries of 0.1%.

Next, we investigated the impact of an increasing ratio of varying matrix entries. Figure 9 demonstrates that the computation time increases for all matrices with an increasing ratio of varying matrix entries and that, for the largest ratio of 10%, the computation time of PartialNICSLU becomes similar to that of NICSLU. This is due to the fact that a variation of 10% of the rows means that the factorisation path already includes 48% of the entries in the **LU** matrix.

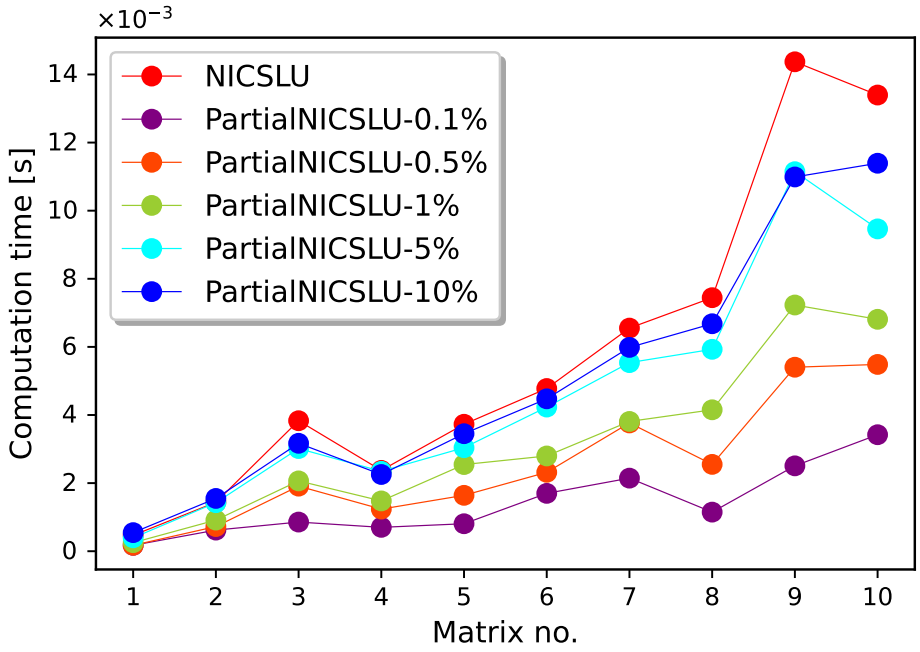

Figure 9. Impact of ratio of varying matrix entries on partial refactorisation times.

5. Conclusions and Outlook

In real-time simulations with nonlinear component models, numerically varying system matrices occur and require the recomputation of the LU decomposition in each simulation step. It is shown that the employment of the NICSLU method accelerates the refactorisation computation in comparison with both the SuperLU and KLU methods. In addition, we propose the integration of a partial refactorisation approach into the NICSLU method by adopting a factorisation path algorithm and show that an additional speedup was achieved in all considered use cases.

In the presented performance analysis, we cover use cases involving system matrices of different sizes, a variety of sparsity patterns and different ratios of numerically varying matrix entries. This allows us to conclude that the proposed high-performance LU decomposition method can help to meet computation time requirements in a variety of real-time simulation scenarios.

Furthermore, the application of the NICSLU method leads to a significant speedup of the solution vector computation in comparison to SuperLU when applying it in a scenario within the real-time simulator DPsim. Such speedup of the solution step is of particular relevance in real-time simulation scenarios in which the underlying system matrix remains constant.

In future work, the proposed method can be extended by a partial RHS solving routine, which uses a suitable dependency path algorithm similarly as for partial refactorisation to speed up the solution vector computation. Furthermore, the method can be applied to various use cases and scenarios for a broader analysis of its benefits and limitations. The knowledge gained from the analysis can be used for further conceptual and software-technical improvements of the method and its implementation.

Author Contributions: Conceptualisation, methodology, investigation, J.D., L.S. and L.R.; software, visualisation and validation, J.D. and L.S.; data curation, J.D. and L.S.; writing—original draft preparation, J.D., L.S. and L.R.; writing—review and editing, J.D., L.S. and L.R.; resources, supervision and project administration, J.D. and L.R.; funding acquisition, A.B. and A.M. All authors have read and agreed to the published version of the manuscript.

Funding: The work of L. Schumacher, L. Razik and A. Benigni is supported by the Helmholtz Association under the Joint Initiative Energy Systems Integration.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to unpublished results in this project.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AMD	Approximate Minimum Degree Ordering Algorithm
BTF	Block triangular form
CSR	Compressed sparse row
DAE	Differential-algebraic system of equations
DCIM	Delayed-current injection method
EMT	Electromagnetic transient
HiL	Hardware-in-the-loop
LES	Linear equation system
MNA	Modified nodal analysis
NR	Newton–Raphson

ODE	Ordinary differential equation
RHS	Right-hand side
SFA	Shifted frequency analysis
VBR	Voltage-behind-reactance

References

1. Zhang, P.; Marti, J.R.; Dommel, H.W. Shifted-Frequency Analysis for EMTP Simulation of Power-System Dynamics. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2010**, *57*, 2564–2574. [CrossRef]
2. Dinkelbach, J.; Nakti, G.; Mirz, M.; Monti, A. Simulation of Low Inertia Power Systems Based on Shifted Frequency Analysis. *Energies* **2021**, *14*, 1860. [CrossRef]
3. Razik, L.; Schumacher, L.; Monti, A.; Guironnet, A.; Bureau, G. A comparative analysis of LU decomposition methods for power system simulations. In Proceedings of the 2019 IEEE Milan PowerTech, Milan, Italy, 23–27 June 2019; pp. 1–6.
4. PEGASE Consortium. D4.1 Algorithmic requirements for simulation of large network extreme scenarios. 2011.
5. Razik, L. High-Performance Computing Methods in Large-Scale Power System Simulation. Ph.D. Thesis, RWTH Aachen University, Aachen, Germany, 2020.
6. Guironnet, A.; Saugier, M.; Petitrenaud, S.; Xavier, F.; Panciatici, P. Towards an Open-Source Solution using Modelica for Time-Domain Simulation of Power Systems. In Proceedings of the 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), Sarajevo, Bosnia and Herzegovina, 21–25 October 2018. [CrossRef]
7. Hindmarsh, A.C.; Brown, P.N.; Grant, K.E.; Lee, S.L.; Serban, R.; Shumaker, D.E.; Woodward, C.S. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans. Math. Softw.* **2005**, *31*, 363–396. [CrossRef]
8. Tostado-Véliz, M.; Kamel, S.; Jurado, F. Comparison of various robust and efficient load-flow techniques based on Runge–Kutta formulas. *Electr. Power Syst. Res.* **2019**, *174*, 105881. [CrossRef]
9. Tostado-Véliz, M.; Kamel, S.; Jurado, F. Two Efficient and Reliable Power-Flow Methods With Seventh Order of Convergence. *IEEE Syst. J.* **2021**, *15*, 1026–1035. [CrossRef]
10. Quintana-Ortí, E.S.; Van De Geijn, R.A. Updating an LU Factorization with Pivoting. *ACM Trans. Math. Softw.* **2008**, *35*. [CrossRef]
11. Abusalah, A.; Saad, O.; Mahseredjian, J.; Karaagac, U.; Kocar, I. Accelerated Sparse Matrix-Based Computation of Electromagnetic Transients. *IEEE Open Access J. Power Energy* **2020**, *7*, 13–21. [CrossRef]
12. Chan, S.M.; Brandwajn, V. Partial Matrix Refactorization. *IEEE Trans. Power Syst.* **1986**, *1*, 193–199. [CrossRef]
13. Tinney, W.F.; Brandwajn, V.; Chan, S.M. Sparse Vector Methods. *IEEE Trans. Power Appar. Syst.* **1985**, *PAS-104*, 295–301. [CrossRef]
14. Chen, X.; Wang, Y.; Yang, H. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *Comput.-Aided Des. Integr. Circuits Syst. IEEE Trans.* **2013**, *32*, 261 –274. [CrossRef]
15. Demmel, J.W.; Eisenstat, S.C.; Gilbert, J.R.; Li, X.S.; Liu, J.W.H. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.* **1999**, *20*, 720–755. [CrossRef]
16. Davis, T.A.; Natarajan, E.P. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.* **2010**, *37*, 1–17. [CrossRef]
17. Eigen Developers. Eigen. Available online: https://eigen.tuxfamily.org/ (accessed on 15 October 2021).
18. Mirz, M.; Dinkelbach, J.; Monti, A. DPsim—Advancements in Power Electronics Modelling Using Shifted Frequency Analysis and in Real-Time Simulation Capability by Parallelization. *Energies* **2020**, *13*, 3879. [CrossRef]
19. Dufour, C.; Jalili-Marandi, V.; Bélanger, J. Real-Time Simulation Using Transient Stability, ElectroMagnetic Transient and FPGA-Based High-Resolution Solvers. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; pp. 283–288. [CrossRef]
20. Ho, C.W.; Ruehli, A.; Brennan, P. The modified nodal approach to network analysis. *IEEE Trans. Circuits Syst.* **1975**, *22*, 504–509. [CrossRef]
21. Dommel, H.W. Digital Computer Solution of Electromagnetic Transients in Single-and Multiphase Networks. *IEEE Trans. Power Appar. Syst.* **1969**, *PAS-88*, 388–399. [CrossRef]
22. Kundur, P.; Balu, N.J.; Lauby, M.G. *Power System Stability And Control*; McGraw-Hill: New York, NY, USA, 1994.
23. Dufour, C. Highly Stable Rotating Machine Models Using the State-Space-Nodal Real-Time Solver. In Proceedings of the 2018 IEEE Workshop on Complexity in Engineering (COMPENG), Florence, Italy, 10–12 October 2018; pp. 1–10. [CrossRef]
24. Dommel, H.W. *EMTP Theory Book*; B.P.A: Portland, OR, USA, 1986.
25. Wang, L.; Jatskevich, J. A Voltage-Behind-Reactance Synchronous Machine Model for the EMTP-Type Solution. *IEEE Trans. Power Syst.* **2006**, *21*, 1539–1549. [CrossRef]
26. Chua, L.O. *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, 1st ed.; Prentice Hall: Englewood Cliffs, NJ, USA, 1975.
27. Li, X.S.; Demmel, J.W.; Gilbert, J.R.; Grigori, L.; Shao, M.; Yamazaki, I. *SuperLU User' Guide*; Lawrence Berkeley National Laboratory: Berkeley, CA, USA, 2011.
28. Gilbert, J.R.; Peierls, T. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM J. Sci. Stat. Comput.* **1988**, *9*, 862–874. [CrossRef]
29. Chen, X.; Wang, Y.; Yang, H. An adaptive LU factorization algorithm for parallel circuit simulation. In Proceedings of the 17th Asia and South Pacific Design Automation Conference, Sydney, NSW, Australia, 30 January–2 February 2012; pp. 359 –364.

30. NICSLU Developers. NICSLU—Parallel Sparse Solver for Circuit Simulation. Available online: https://github.com/chenxm1986/nicslu (accessed on 15 November 2021).
31. DPsim Developers. DPsim: A Real-Time Power System Simulator. Available online: https://dpsim.fein-aachen.org (accessed on 15 October 2021).
32. Li, X.S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* **2005**, *31*, 302–325. [CrossRef]
33. Dynaωo Developers. Dynaωo. Available online: https://dynawo.github.io (accessed on 12 November 2021).