Sparse Matrix Based Power Flow Solver For Real-Time Simulation of Power System

By

Sk Subrina Shawlin

A THESIS SUBMITTED TO

THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY

Toronto, Canada

September 2022

ABSTRACT

Analyzing a massive number of Power Flow (PF) equations even on almost identical or similar network topology is a highly time-consuming process for large-scale power systems. The major computation time is hoarded by the iterative linear solving process to solve nonlinear equations until convergence is achieved. This is a paramount concern for any PF analysis methods. This thesis presents a sparse matrix-based power flow solver that is fast enough to be implemented in the real-time analysis of largescale power systems. It uses KLU, a sparse matrix solver, for PF analysis. It also implements parallel processing of CPU and GPU which enables the simultaneous computation of multiple blocks in the algorithm leading to faster execution. It runs 1000 times and 200 times faster than newton raphson method for DC and AC power system respectively. On average, it is around 10 times faster than MATPOWER for both AC and DC power system.

Table of Contents

List of Tables

List of Figures

Acknowledgement

This thesis would not have been possible without the continuous guidance and help from my thesis supervisor, my family and my friends. I would like to start my expressing my deepest gratitude to my supervisor, Prof. Afshin Rezaei-Zare. It was his mentorship that has led to the successful completion of the thesis. Next, I would like to thank my parents and my sister for being there always and motivating me to achieve greater things. Thank you for being my support system. I would like to show my appreciation to my friends who made this difficult journey more bearable. I have had the privilege to meet some amazing souls who inspires me every day with their actions.

Finally, I would like to convey my gratitude towards Prof. Pirathayini Srikantha and Prof. Mojgan Jadidi to be part of my thesis committee. They have provided some great feedback on the thesis. In addition, I would like to acknowledge the contribution of my colleague, Dr. Fazel Mohammadi who helped me in my research.

Nomenclature

Abbreviations

PF	POWER FLOW
KLU	CLARK-KENT LU
LU	LOWER-UPPER
BTF	BLOCK TRIANGULAR FORMAT
AC	ALTERNATING CURRENT
DC	DIRECT CURRENT
RES	RENEWABLE ENERGY SOURCE
DER	DISTRIBUTED ENERGY RESOURCE
EV	ELECTRIC VEHICLE
NR	NEWTON RAPHSON
GS	GAUSE SEIDEL
AMD	APPROXIMATE MINIMUM DEGREE
COLAMD	COLUMN APPROXIMATE DEGREE
CPU	CENTRAL PROCESSING UNIT
GPU	GRAPHICS PROCESSING UNIT
MV	MEDIUM VOLTAGE
ACO	ANT COLONY OPTIMIZATION
SIMD	SINGLE INSTRUCTION MULTIPLE DATA
GC	GAUSS-JACOBIAN
MKL	MATH KERNEL LIBRARY

CUDA	COMPUTE UNIFIED DEVICE ARCHITECTURE
HPC	HIGH PERFORMANCE COMPUTING
CCS	COMPRESSED COLUMN STORAGE
CRS	COMPRESSED ROW Storage
FF	FULL FACTORIZATION
RF	RE-FACTORIZATION
DFS	DEPTH FIRST SEARCH
ANN	ARTIFICIAL NEURAL NETWORK

CONSTANTS

S	INJECTED POWER
V	VOLTAGE
I	CURRENT
Y	ADMITTANCE
G	CONDUCTANCE
B	SUSCEPTANCE
P	ACTIVE POWER
Q	REACTIVE POWER
Θ	ANGLE
Δ	ANGLE
J	JACOBIAN MATRIX
R	RESISTANCE
X	REACTANCE

Publications

Conference Paper:

1. S. S. Shawlin, F. Mohammadi and A. Rezaei-Zare, “GPU-Accelerated Sparse LU Factorization for Concurrent Analysis of Large-Scale Power Systems,” 2022 IEEE International Conference on Environment and Electrical Engineering and 2022 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe), 2022.

2. S. S. Shawlin, F. Mohammadi and A. Rezaei-Zare, “GPU-Based DC Power Flow Analysis Using KLU Solver,” 2022 IEEE International Conference on Environment and Electrical Engineering and 2022 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe), 2022.

CHAPTER 1

INTRODUCTION

1.1 Research Background

The development of any nation everywhere is directly linked with its power transmission capacity. A stable power transmission network ensures prosperity. To meet the expanding needs of energy in the world, the power system is experiencing higher penetration of renewable energy sources (RESS), distributed energy resources (DERs) and hybrid electrical vehicles (HEVs) into the power grids, which is causing higher uncertainties and complexity in the system. The adoption of the smart grid and advanced power electronics-based devices, as well as the continuous increase in power consumption, have led to the growth in the complexity of power systems over time [1] [2]. In order to ensure proper operation and stability of the system, power system solvers must be equipped to handle this expansion and frequent changes.

Power Flow (PF) is the key tool for power systems analysis. In an electrical system, the power flow or load flow analysis is a numerical analysis of the flow of electrical power in a grid or interconnected system. It determines the system's capability to supply the connected load and forecasted load demand adequately. Power flow or load flow analysis is crucial to evaluate the performance of the power system in case certain control measures need to be taken for emergencies [3] [4] .

The goal of a power flow analysis is to find out the voltage magnitude and angle for each bus in a power system under a specific condition. It is usually required to carry out multiple power flow analyses under a variety of conditions. The operating conditions depend on the constraints of generator capacities, load demand, and several other restraints.

After finding the voltage magnitude and angle at each bus, the real and reactive power flowing into each branch line along with the real and reactive power of the generators can be analytically determined. Additional information like, total system loss, and individual line losses can also be calculated.

Power flow analysis is necessary to plan for future expansion of the power system and determine the best-operating conditions. It allows us to run the power system at maximum capacity while minimizing that operating cost.

The power flow problem is formulated, assuming the power system network to be linear and balanced. However, this is not the case as the power and voltage constraints introduce nonlinearity in the power flow formulation. To solve a nonlinear system, iterative techniques are a must. There are multiple power flow solvers available all of which work in an iterative manner. Newton Raphson method, gauss seidel method, and fast decoupled load flow method are some of the mostly used ones. In all these methods, most of the computational time is spent on multiple iterations until convergence is met. The computational time varies largely with the number of Buses in the system and system complexity. In N-x contingency analysis, probabilistic PF analysis based on Monte-Carlo simulation, security constrained-based analysis, load forecasting, etc., a massive number of power flow solutions need to be solved on identical or similar grid topology [5]. This requires significant computational time to solve such power flow solutions. To ensure the smooth operation of the power system against unexpected and unprecedented changes, a PF solver with more efficient modeling is necessary to provide reduced computational time.

1.2 Thesis Objective and Contribution

This paper discusses a PF analysis method based on a sparse matrix solver. It implements the KLU algorithm as the linear solver with the goal of achieving a higher computational gain. The presented approach is based on Gilbert-Peierls' algorithm, and employs LU decomposition with no dense kernels. The unique sparsity pattern of power system matrices makes it perfect to use the proposed method.

Power system matrix has a sparse characteristic that makes them suitable to use KLU. They have only a few dense rows/columns originating from voltage/current sources which are effectively rearranged and removed by BTF permutation. Though the matrices are asymmetric, their non-zero pattern is roughly symmetric. After the BTF permutation, the non-zero pattern in the blocks is more symmetric than the original matrix. They also have zero-free or nearly zero-free diagonals. Due to the permutation, the sub-diagonal region in the matrix has zero work. Also, since the off-diagonal elements are not factorized, they don't cause any fill-in.

The linear equation in solving the power system comes from solving large non-linear equations using Newton Raphson method with multiple iterations until the result converges. These systems are typically of very large dimensions, leading to a larger amount of simulation time being spent on solving them. Solving a large-dimension system in an iterative process is often a bottleneck in a power flow analysis. Hence, sparse matrix-based solvers provide better solution time and are mostly preferred for circuit simulation.

The linear system looks like Ax=b, where A is the coefficient sparse matrix, x is the unknown vector, and b is the right-hand side. There are several methods available for solving such linear systems, like gaussian elimination, QR factorization, and Cholesky factorization. Gaussian elimination is the most widely used algorithm for solving linear systems. Cholesky factorization is typically used when A is symmetric positive definite.

The coefficient matrix A can be a dense matrix or sparse matrix. If most of the elements in a are nonzero, then it is considered a dense matrix. In a sparse matrix, there are only a few nonzero elements. Storing every element in a sparse matrix would be a waste of memory. An effective way to store a sparse matrix is to store only its nonzero elements value and position. However. It comes with its own challenges as well.

Saving only the non-zeros would lead to the positions of non-zeroes in the triangular matrices being unknown after factorization. We need to find an efficient algorithm that gives us a prominent gain over the dense system. That's where KLU comes in. Due to the unique characteristics of power system matrices and their amenability to BTF ordering, KLU Is a well-suited method to apply to power flow analysis.

This thesis proposes to use KLU, a sparse matrix-based solver, to solve linear equations in the process of solving the non-linear power flow system. Based on the characteristics of power system matrices and their amenability to KLU, KLU is chosen to be used in this thesis. On top of replacing the use of the dense matrix with the sparse matrix solver, this thesis proposes additional enhancements to the original KLU algorithm. It does so by implementing a column tracking method where the left-most column is found in each block of the BTF matrix after changes in values of the initial co-efficient matrix occurs. Since KLU follows a left-looking algorithm, the numerical values of L and U are updated for the subsequent column after the left-most changed column is identified.

The system performance is boosted by the implementation of parallel processing of CPU and GPU. Parallel processing allows simultaneous execution of the BTF blocks present in the system, which results in faster computation time. When tested, the proposed modified solver provides 1000 times faster computation time for DC system and 200 times for AC system when compared with the traditional newton-raphson method. It has been tested against MATPOWER as well which is a common power flow tool. The proposed system is 10 times faster than MATPOWER. It also shows higher performance in terms of memory usage.

The higher performance of the proposed modified KLU solver makes it a perfect choice when a large volume of power flow equations needs to be solved in a short time. It will be a perfect fit for the power system optimization problems where a large number of power flow equations need to be solved simultaneously with varying operating conditions. Besides, since it is a sparse matrix solver, it can be used in any physical modeling or system where a linear system, Ax=b, needs to be solved.

1.3 Thesis Structure

The thesis is designed as below:

Chapter 2: In this chapter, relevant works in the related field are reviewed for comparison. It provides a literature survey of power flow analysis using various approaches and their effect on the overall result.

Chapter 3: Chapter 3 discusses about how power flow analysis works and the theory behind them. It describes the traditional structure of the power grid and how that affects the power flow analysis itself. There are two kinds of power systems, AC and DC. Both approaches as been discussed in detail with the operating conditions and assumptions taken.

Chapter 4: This thesis is based on the sparse matrix. Chapter 4 starts with briefly describing what a sparse matrix is and how it works, followed by the data structure of the said matrix.

It describes the advantages of using the sparse matrix instead of the traditional dense matrix for power flow analysis.

Since the thesis uses KLU as its sparse matrix solver and provides some enhancement, the theory behind KLU is written in detail later. KLU is based on Gilbert-Peierls' algorithm, a non-supernodal algorithm, which is the predecessor to SuperLU, a supernodal algorithm. Before learning about KLU, SuperLU is discussed briefly to show its process flow.

KLU is divided into two stages, symbolic analysis, and numeric factorization. It also employs ordering mechanism and partial pivoting. Each of the stages in KLU and the detailed process can be found in this chapter which gives us a clear look into how KLU works and the steps it goes through.

Chapter 5: This chapter starts with a short summary of the KLU process. It then reiterates why KLU has been chosen for this thesis and how certain characteristics of the power system matrices make them so suitable to use KLU. Next, how KLU is implemented to be used in power flow analysis is shown with the use of a flow chart.

Chapter 6: Here, first, the enhancements made to the original KLU algorithm as part of this thesis are described in detail. Mainly the contribution is twofold. 1) A column tracking method has been put in place to find the left most changed column during the refactorization phase, which promises a significant boost in computation gain 2) Parallel processing of the blocks has been implemented on CPU and GPU, which allows simultaneous execution of the BTF blocks. For parallel processing, many tools are available, such as CUDA, OpenMP, OpenCL, etc. OpenMP has been chosen as the preferred parallel processing tool for this thesis. CUDA can also be used for thread-level programming. However, it requires a lot of changes to the sequential code. OpenMP is chosen for this purpose since it requires fewer changes to the sequential code.

The gain achieved from the modified KLU, and original KLU is then compared with the traditional power flow algorithm Newton-Raphson for both AC and DC power flow, where it has been proven that the solution proposed in this thesis is around 1000 times and 200 times faster for DC and AC solutions respectively. Later, the same comparison is drawn with MATPOWER, another power flow solver tool. The proposed method is around ten times faster than MATPOWER for the same test systems.

Chapter 7: Conclusion and future work is discussed in this chapter.

CHAPTER 2

LITERATURE REVIEW

There have been studies that employ data-driven machine learning methods like Artificial Neural Network (ANN) to find power system state estimation and approximation of power flow solutions [6] [7]. This approach requires the test system to be trained with a large number of power flow solutions of similar grid topology to fully gain the benefits shown for a medium voltage (MV) power system.

In [8], the authors present a parallel ant colony optimization (ACO) to be applied to multicore single-instruction, multiple-data (SIMD) CPU architecture where the construction of each ant is expedited by vector instructions. This approach resulted in 57.8 times faster computation than the CPU version.

The LU decomposition, which is frequently used in PF analysis, requires high computation time. Research has been done to expedite the process using parallel processing. In [9], faster computation is achieved with LU factorization using GPU-based parallelism and better memory-access efficiency. They have also ensured higher memory access efficiency by bundling a high number of LU factorization tasks and formulating a new larger-scale problem.

Paper [9] works with six different approaches with Newton-Raphson (NR) method running on a combination of both central processing unit (CPU) and graphical processing unit (GPU). They have used either LU decomposition or QR factorization to solve the linear equations for all six. Paper [10]compares the performance of NR and Gauss-Jacobian (GC) algorithm for power flow analysis. They have shown heightened performance by choosing the runtime environment on GPU units using Compute Unified Device Architecture (CUDA) over CPU versions. The CPU version was modified with Intel Math Kernel Library (Intel MKL), which contains a set of optimized, thread-parallel mathematical functions for solving the linear equations and the matrix operations.

In [11, 12, 13], parallel processing is used to solve the sparse matrix for circuit simulation. In [14], the sparsity in largescale power systems analysis is investigated. Furthermore, in [15, 16, 17, 18, 19], the total computation time of PF analysis is reduced using parallel processing to matrix calculations.

There have been several attempts to considerably reduce the computational burden of numerical analyses, particularly the computation time, using parallel and distributed processing, which require massive computing resources. For instance, in [20, 21, 22, 23, 24], High-Performance Computing (HPC) machines are used to reduce the computation time of PF calculations. Thereafter, the application of HPC for accelerating power systems analysis is investigated.

In addition, rapid PF computation using distributed computing is proposed in [21, 22], in which multiple computers connected via an Ethernet network are utilized. The development of HPC technologies, such as multi-core processors, clusters, and Graphics Processing Units (GPU), has led to an increased interest in conducting research on accelerating power systems analysis based on parallel and cloud computing and employing HPC technologies [20]. There also have been attempts to reduce the computation time for power systems analysis using GPU-based parallel computing. Particularly, in [25], the potential of general-purpose GPU for energy management systems is investigated. In [26], GPUs are used to reduce the PF computation time and resolve the issues related to physical connections between machines.

CHAPTER 3

POWER FLOW ANALYSIS

3.1 Introduction

PF problems are mainly solved using nodal analysis. Such analysis involves nonlinear equations and can be performed using iterative techniques, such as Newton-Raphson (NR). The main aim of the PF analysis is to obtain the voltage magnitude and angle at each bus and, accordingly, determine the active and reactive power flowing through each line and branch using the voltage magnitudes and angles at each bus [27].

There are three types of buses in the PF problem analysis, which are as follows:

Slack Bus: The slack bus is used to provide for system losses by emitting or absorbing active and/or reactive power to and from the system. The voltage magnitude and angle of the slack bus are known, while the active and reactive power should be determined. Since a slack bus is used as a reference bus, each power grid requires a slack bus.

Generation Bus (PV Bus): These buses maintain a constant power generation which is typically controlled by a prime mover and a constant bus voltage. The active power and voltage magnitude at each PV bus is known, while the voltage angle and reactive power should be determined.

Load Bus (PQ Bus): PQ buses do not have any voltage control devices such as a generator and are also not remotely controlled by a generator. The active and reactive power at each PQ bus is known, while the voltage magnitude and angle should be determined.

3.2 AC Power Flow

The net injected power at i^{th} Bus, S_i, can be determined using the corresponding bus voltage, V_i, the neighboring bus voltage, V_j, and the admittance between buses I and j $Y_{ij} = G_{ij} + jB_{ij}$, where G_{ij} And B_{ij}Are the conductance and susceptance between buses I and j. The PF equation at any bus can be written as follows:

$$S_i = P_i + jQ_i = V_i {I_i}^* \tag{1}$$

Where P_i snd Q_i denote the active and reactive power at bus I, and ${I_i}^*$ indicates the conjugate of the current at bus i.

Using Kirchhoff's current law, $I_i = \sum_{j=1}^{N} Y_{ij} V_j$, The PF equations can be derived as follows:

$$P_i = \sum_{j=1}^{N} |V_i||V_j||Y_{ij}| \operatorname{Cos}(\theta_{ij} + \delta_j - \delta_i) = P_i(|V|, \delta) \tag{2}$$

$$Q_i = -\sum_{j=1}^{N} |V_i||V_j||Y_{ij}| \operatorname{Sin}(\theta_{ij} + \delta_j - \delta_i) = Q_i(|V|, \delta)$$

Where N shows the total number of buses, $|V_i|$ and $|V_j|$ are respectively the magnitudes of the voltage at buses i and j , and δ_i And δ_j are the associated angles. In addition, $|Y_{ij}|$ denotes the magnitude of the bus admittance, Ybus, element between buses i and j, and θ_{ij} indicates the corresponding angle.

From equation (2), it can be concluded that both active and reactive power are functions of (|V|,δ), where $|V| = (|V_1|, \dots \dots, |V_N|)^T$ *and* $\delta = (\delta_1, \dots \dots, \delta_N)^T$. In addition, $P_i(|V|, \delta) = P_i(x)$ and $Q_i(|V|, \delta) = Q_i(x)$, where $x = (\delta|V|)^T$.

Considering P_i (scheduled) and Q_i (scheduled) as the scheduled power at PQ buses, after a certain number of iterations, G should converge to the value making $P_i - P_i(x) = 0$ and $Q_i - Q_i(x) = 0$.

As a result, for all PQ buses, the following equation can be written.

$$f(x) = \begin{vmatrix} P(scheduled) - P(x) \\ Q(scheduled) - Q(x) \end{vmatrix} = \begin{vmatrix} \Delta\mathrm{P(x)} \\ \Delta\mathrm{Q(x)} \end{vmatrix} \cong 0 \tag{3}$$

As stated before, active and reactive power at the slack bus is unknown and cannot be used in equation (3). As a result, x is a 2 (N-1) vector.

Taking equations (2) and (3) into account, the following equation can be obtained by applying the NR method to an #-bus power system.

$$\begin{vmatrix} \Delta\mathrm{P} \\ \Delta\mathrm{Q} \end{vmatrix} = \begin{vmatrix} -J11 - J12 \\ -J21 - J22 \end{vmatrix} \begin{vmatrix} \Delta\delta \\ \Delta|\mathrm{V}| \end{vmatrix} \tag{4}$$

Where $|\Delta V| = (|\Delta V_2|, \dots \dots, |\Delta V_M|)^T$ *and* $\Delta\delta = (\Delta\delta_2, \dots \dots, \delta_N)^T$ with M being the total number of PQ buses and N being the total number of PQ and PV buses.

In addition, the Jacobian matrix $J(x)^{(N-1)*(N-1)}$ consists of J11, J12, J21 and J22.

$$J11 = \frac{\partial P_i(x)}{\partial \delta_j}, J12 = \frac{\partial P_i(x)}{\partial |V_j|},$$

$$J21 = \frac{\partial Q_i(x)}{\partial \delta_j}, J22 = \frac{\partial Q_i(x)}{\partial |V_j|},$$

For $i \neq j$

$$\left\{\begin{aligned} J11 &= \frac{\partial P_i(x)}{\partial \delta_j} = |V_i||V_j||Y_{ij}|\,\mathrm{Sin}(\theta_{ij} + \delta_j - \delta_i) \\ J12 &= \frac{\partial P_i(x)}{\partial |V_j|} = |V_i||Y_{ij}|\,\mathrm{Cos}(\theta_{ij} + \delta_j - \delta_i) \\ J21 &= \frac{\partial Q_i(x)}{\partial \delta_j} = |V_i||V_j||Y_{ij}|\,\mathrm{Cos}(\theta_{ij} + \delta_j - \delta_i) \\ J22 &= \frac{\partial Q_i(x)}{\partial |V_j|} = |V_i||Y_{ij}|\,\mathrm{Sin}(\theta_{ij} + \delta_j - \delta_i) \end{aligned}\right. \tag{5}$$

In addition, for i=j,

$$\left\{\begin{aligned} J11 &= \frac{\partial P_i(x)}{\partial \delta_j} = \sum_{\substack{j=1 \\ j\neq i}}^{N} |V_i||V_j||Y_{ij}|\,\mathrm{Sin}(\theta_{ij} + \delta_j - \delta_i) \\ J12 &= \frac{\partial P_i(x)}{\partial |V_j|} = 2|V_i||Y_{ii}|\,\mathrm{Cos}(\theta_{ii}) + \sum_{\substack{j=1 \\ j\neq i}}^{N} |V_i||Y_{ij}|\,\mathrm{Cos}(\theta_{ij} + \delta_j - \delta_i) \end{aligned}\right. \tag{6}$$

$$J21 = \frac{\partial Q_i(x)}{\partial \delta_j} = \sum_{\substack{j=1 \\ j \neq i}}^{N} |V_i||V_j||Y_{ij}| \operatorname{Cos}\left(\theta_{ij} + \delta_j - \delta_i\right)$$

$$J22 = \frac{\partial Q_i(x)}{\partial |V_j|} = 2|V_i||Y_{ii}| \operatorname{Sin}(\theta_{ii}) + \sum_{\substack{j=1 \\ j \neq i}}^{N} |V_i||Y_{ij}| \operatorname{Sin}\left(\theta_{ij} + \delta_j - \delta_i\right)$$

From the above-mentioned equations, it can be derived that if there is no connection between i^{th} And j^{th} Bus, then, $Y_{ij} = 0$. As a result, the same as the Ybus matrix, the Jacobian matrix is sparse. The voltage magnitude and angle is updated after each iteration k like equation (7) until power mismatch convergence is achieved. The iteration is repeated until each element of the mismatch matrix is below the tolerance level.

$$\begin{bmatrix} \Delta\delta^k \\ \Delta|V|^k \end{bmatrix} = \begin{bmatrix} -J11 - J12 \\ -J21 - J22 \end{bmatrix}^{-1} \begin{bmatrix} \Delta P^k \\ \Delta Q^k \end{bmatrix} \tag{7}$$

3.3 DC Power flow

In power transmission systems, G_{ij} and the difference in the voltage phase angles between buses i and j is small. This means $G_{ij} \cong 0$ and $\operatorname{Sin}\left(\delta_j - \delta_i\right) \cong \left(\delta_j - \delta_i\right)$. Therefore, Equation (2) can be sequentially solved in two stages, where the voltage magnitudes are constant, and the voltage phase angles are constant. Indeed, voltage magnitude and phase angle are the determining factors to solve the PF problem. Once the voltage phase angles are calculated, they can be used to calculate the reactive power mismatch.

After that, the reactive power mismatch can be used as & while calculating the voltage magnitudes. The updated voltage magnitudes and phase angles can be used to determine the active power mismatch which again can be used to update the voltage phase angles. This iterative process continues till the desired accuracy is achieved. Finally, the voltage magnitudes and phase angles are used to calculate the PF of all branches.

Since the ratio of reactance to the resistance of power transmission lines, i.e. $\frac{X}{R}$ ratio, is high; equation (2) can be written as follows:

$$P_i = \sum_{j=1}^{N} |V_i||V_j| \left(B_{ij}\mathrm{Sin}(\delta_j - \delta_i)\right) \tag{7}$$

$$Q_i = \sum_{j=1}^{N} |V_i||V_j| \left(-B_{ij}\mathrm{Cos}(\delta_j - \delta_i)\right)$$

As mentioned earlier, due to the fact that $G_{ij} \cong 0$ and $\mathrm{Sin}(\delta_j - \delta_i) \cong (\delta_j - \delta_i)$, Equation (7) can be simplified as follows:

$$P_i = \sum_{j=1}^{N} |V_i||V_j| \left(B_{ij}(\delta_j - \delta_i)\right) \tag{8}$$

$$Q_i = \sum_{j=1}^{N} |V_i||V_j|(-B_{ij})$$

For $i \neq j$, $B_{ij} = -b_{ij}$ indicating that the Ybus element in row i and column j is the negative of the susceptance of the circuit connecting bus i to bus j. In addition,

$$\text{For } i = j, B_{ii} = b_i + \sum_{j=1, j\neq i}^{N} b_{ij}. \tag{9}$$

The reactive power flow equation in Equation (7) can be written as follows:

$$Q_i = \sum_{j=1}^{N} |V_i| \left| V_j \right| (-B_{ij}) = -|V_i|^2 (B_{ii}) - \sum_{j=1, j\neq i}^{N} |V_i| \left| V_j \right| (B_{ij})$$

In addition,

$$Q_i = -|V_i|^2 (b_i + \sum_{j=1, j\neq i}^{N} b_{ij}) - \sum_{j=1, j\neq i}^{N} |V_i| \left| V_j \right| (-b_{ij})$$

$$Q_i = -|V_i|^2 b_i - |V_i|^2 \sum_{j=1, j\neq i}^{N} b_{ij} - \sum_{j=1, j\neq i}^{N} |V_i| \left| V_j \right| (-b_{ij}) \tag{10}$$

$$Q_i = -|V_i|^2 b_i - |V_i|^2 \sum_{j=1, j\neq i}^{N} b_{ij} + \sum_{j=1, j\neq i}^{N} |V_i| \left| V_j \right| (b_{ij})$$

As a result, Equation (10) can be written as follows:

$$Q_i = -|V_i|^2 b_i - (\sum_{j=1, j\neq i}^{N} |V_i|^2 b_{ij} + |V_i| \left| V_j \right| (b_{ij})) \tag{11}$$

$$Q_i = -|V_i|^2 b_i - \sum_{j=1, j\neq i}^{N} |V_i| b_{ij} \left(|V_i| - \left| V_j \right|\right)$$

As all circuits contain inductive elements in series, the numerical value of b_{ij} is negative. Therefore, Equation (11) can be written as follows:

$$Q_i = -|V_i|^2 b_i + \sum_{j=1, j\neq i}^{N} |V_i| b_{ij} \left(|V_i| - \left| V_j \right|\right) \tag{12}$$

The active power flow in Equation (7) can be written as follows

$$P_i = \sum_{j=1}^{N} |V_i| \left| V_j \right| (B_{ij} \mathrm{Sin}(\delta_j - \delta_i))$$

$$P = |V_i|^2 (B_{ii} (\delta_j - \delta_i)) + \sum_{j=1, j\neq i}^{N} |V_i| \left| V_j \right| (B_{ij} (\delta_j - \delta_i)) \tag{13}$$

Therefore, Equation (13) can be simplified as follows:

$$P_i = \sum_{j=1, j\neq i}^{N} |V_i||V_j| \left(B_{ij}(\delta_j - \delta_i)\right) \tag{14}$$

Taking Equations (12) and (14) into account, it is noted that the voltage phase angles are not included while calculating the reactive power. In addition, the voltage magnitudes are not included while calculating the active power.

In DC PF analysis, the above-mentioned iterative process is skipped, and the voltage phase angles can be calculated without considering reactive power and voltage magnitudes. In addition, it is assumed that $|V_i| = |V_j| = 1$, and transformer tap settings are ignored.

Making such an approximation leads to the following equations.

$$P_i = \sum_{j=1, j\neq i}^{N} \left(B_{ij}(\delta_j - \delta_i)\right) \tag{15}$$

$$Q_i = -b_i + \sum_{j=1, j\neq i}^{N} b_{ij} \left(|V_i| - |V_j|\right)$$

In DC PF analysis, the maximum difference in the voltage magnitudes of two buses is relatively small whereas the maximum difference in voltage phase angles is considerable. Therefore, the active power flow across power transmission lines tends to be significantly larger than the reactive power flow, which implies $P_{ij} \gg Q_{ij}$. As a result, Q_{ij} in Equation (15) is approximately equal to zero.

CHAPTER 4

POWER FLOW SOLVER BASED ON SPARSE MATRIX

4.1 Introduction

This thesis proposes a more efficient and faster power flow solver based on the sparse matrix. Instead of using the full dense matrix, the adaption of the sparse matrix for power flow analysis gives faster computation time as well as requires less memory usage.

This thesis proposes the use of KLU, a sparse matrix solver, to solve non-linear power flow systems. In this chapter, sparse matrices and their characteristics will be discussed in detail. It will be followed by an in-depth explanation of how KLU works and the steps associated with it.

4.2 Sparse Matrix

A sparse matrix is used to represent matrices where there is a high number of zeros present among its elements. The sparse matrix can have different percentages of sparsity. Unlike the regular dense matrix, most of the elements in a sparse matrix are zeros. They have different data analysis and storage protocols in contrast to a regular matrix. A sparse matrix is often seen in many applications of power systems and different types of physical modeling.

The use of sparse matrix offered some major benefits. One of the biggest advantages of a sparse matrix is the low memory storage requirement. Since most of the elements are zeroes, there is no need to assign memory for a high number of elements. This specific characteristic can be exploited when working with a very large system. Considering the sparsity of the matrix, the actual value of the sparse matrix will be stored only rather than storing a large number of elements with values of zero. Using a sparse matrix dramatically improves the computational speed of large-scale linear algebra problems since no calculation is needed for its zero elements. A sparse matrix has also been represented as a more "loosely integrated system," whereas a dense matrix implies more direct connections between data.

4.2.1 Sparse Matrix Representation

The representation of the sparse matrix depends heavily on the applications they are used in. Different types of applications have different ordering and memory storage requirements. Usually, the mostly used matrix representation is stored as a two-dimensional array. This is typically the preferred representation for dense matrix memory storage and applications. However, as previously mentioned, this is not the desired representation for a sparse matrix as it leads to wastage of memory storage since many zeros will be stored unnecessarily.

It is expected to use a format for the sparse matrix, which reduces its memory storage and saves only the nonzero elements in the matrix. Here two different formats of sparse matrix representations will be discussed that are often used in circuit analysis. These two techniques reduce the size of memory record to store matrices drastically.

4.2.1.1 Compressed Column Storage (CCS)

The compressed column storage (CCS) consists of three single-dimensional vectors that include the position and the values of the nonzero elements in the sparse matrix. For example, for a sparse matrix, A with the size of n by n and nnz nonzero elements, three vectors, namely Ap, Ai, and Ax represents the matrix.

Ap: this is the column pointer vector. It has a size of n + 1. It contains the index of the starting nonzero element of each column. The first element of this vector Ap(0) has to be zero and the last element Ap(n) is nnz.

Ai: this is the row indices vector. It has a size of nnz. It stores the row number of each nonzero element in A.

Ax: this is the nonzero value vector. Size of nnz. It stores the numerical value of all non-zero elements in A.

Let's take matrix A as an example. The matrix shown below expands on the use of the CSC format

$$A = \begin{bmatrix} 5 & 0 & 0 & -5 & 1 \\ 0 & 2 & -5 & 0 & 0 \\ 0 & -9 & 2 & 0 & 0 \\ -3 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The nonzero elements of this matrix have been numbered in a sequential manner in the following table:

Table 4.1

Position of Different Elements in Matrix A

Element number	Element position
0	A (0, 0)
1	A (3, 0)
2	A (4, 0)
3	A (1, 1)
4	A (2, 1)
5	A (1, 2)
6	(2, 2)
7	A (0, 3)
8	A (3,3)
9	A (0, 4)
10	A (0, 0)

The column pointer vector, Ap, is formed by including the first non-zero element number in each column sequentially. For instance, AP (1) is the first nonzero element of column two which is 3 according to the table above.

The Ap vector for the matrix is shown below

$$Ap=\begin{bmatrix}0\\3\\5\\7\\9\\10\end{bmatrix}$$

The first element of Ap starts with 0, and the last element will always be the same as the total number of nonzero elements in the original matrix, which in this case is 10. The size of Ap is n + 1, which is 6 here.

The row index vector Ai is formed by including the row index of every nonzero element in the original matrix in a sequential manner. For example, Ai (6) is the row number of the 7th element in position (2, 2).

The Ai vector for the matrix is shown below:

$$Ai=\begin{bmatrix}0\\3\\4\\1\\2\\1\\2\\0\\3\\0\end{bmatrix}$$

The size of a Ai is the same as the total number of non-zero elements in the original matrix, which is 10 in this case.

Vector Ax stores the numerical values of all the nonzero elements in the matrix. The order in which they are stored is the same as Ai.

The Ax vector for the matrix is shown below:

$$\text{Ax}=\begin{bmatrix} 5 \\ -3 \\ 1 \\ 2 \\ -9 \\ -5 \\ 2 \\ -5 \\ 1 \\ 1 \end{bmatrix}$$

The size of the Ax matrix is the same as the number of non-zero in the original matrix, which is 10 here.

4.2.1.2 Compressed Row Storage (CRS)

The compressed row storage (CRS) consists of three single-dimensional vectors that include the position and the values of nonzero elements in the sparse matrix. For example, or a sparse matrix A with the size of n by n and nnz nonzero elements, three vectors, namely Ap, Ai, and Ax represents the matrix.

Ap: this is the row pointer vector. It has a size of n + 1. It contains the index of the starting non-zero element of each row. The first element of this vector Ap(0) has to be zero, and the last element Ap(n) is nnz.

Ai: this is the column indices vector. It has a size of nnz. It stores the column number of each nonzero element in A.

Ax: this is the nonzero value vector. Size of nnz. It stores the numerical value of all non-zero elements in A.

The matrix shown below expands on the use of CRS format. Is the same matrix of the one used for a compressed column format.

$$A=\begin{bmatrix} 5 & 0 & 0 & -5 & 1 \\ 0 & 2 & -5 & 0 & 0 \\ 0 & -9 & 2 & 0 & 0 \\ -3 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The row pointer vector, Ap, is formed by including the first non-zero element number in each row sequentially. For instance, AP (1)=3 means that the fourth non zero element in the original matrix is the first non zero element of the second row.

The Ap vector for the matrix is shown below:

$$\mathrm{Ap}=\begin{bmatrix}0\\3\\5\\7\\9\\10\end{bmatrix}$$

The first element of Ap starts with 0, and the last element will always be the same as the total number of nonzero elements in the original matrix, which in this case is 10. The size of is Ap is n + 1, which is 6 here.

The Column index vector Ai is formed by including the column index of every non-zero element in the original matrix in a sequential manner. For example, Ai (6) is the column number of the 7th element in position (2, 2).

The Ai vector for the matrix is shown below:

$$Ai=\begin{bmatrix}0\\3\\4\\1\\2\\1\\2\\0\\3\\0\end{bmatrix}$$

The size of a Ai is the same as the total number of non-zero elements in the original matrix, which is 10 in this case.

Vector Ax stores the numerical values of all the nonzero elements in the matrix. The order in which they are stored is the same as Ai.

The Ax vector for the matrix is shown below. The size of Ax matrix is the same as the number of non-zero in the original matrix, which is 10 here.

$$Ax=\begin{bmatrix}5\\-5\\1\\2\\-5\\-9\\2\\-3\\1\\1\end{bmatrix}$$

4.2.2 Solving A Sparse Matrix

To solve the sparse matrix, the same process is followed as that of a dense matrix in terms of general steps and topology. The original matrix needs to be factorized into two factors; the lower and upper triangular factors, L and U, respectively. The product of these matrices is the same as the original matrix.

For a linear system of n equations, Ax=b

A is transformed into an upper and lower triangular matrix L and U such that A = L*U

$$
L=\begin{bmatrix}
L11 & 0 & & & & & 0 \\
L21 & L22 & 0 & & & & \\
L31 & L32 & L33 & & & & 0 \\
\ldots & \ldots & \ldots & \ldots & & & \\
\ldots & \ldots & \ldots & \ldots & \ldots & & \\
Ln1 & Ln2 & Ln3 & Ln4 & \ldots & \ldots & Lnn
\end{bmatrix}
\qquad
U=\begin{bmatrix}
U11 & U12 & U13 & \ldots & \ldots & u1n \\
0 & U22 & U23 & \ldots & \ldots & U2n \\
0 & 0 & U33 & \ldots & \ldots & u3n \\
 & & & \ldots & \ldots & \ldots \\
 & & & & \ldots & \ldots \\
0 & 0 & 0 & \ldots & \ldots & Unn
\end{bmatrix}
$$

It can be seen here the lower triangular matrix L has non-zeroes under the diagonal line, and the upper triangular matrix U has non-zero elements only above the diagonal line.

Mathematically, Ax=b written as

$$(LU)x=b \tag{16}$$

Solving this system requires two steps: forward and backward substitution. During the forward Question, one is solved. While in a backward substitution, equation two is solved.

$$L(Ux)=b \tag{17}$$

Substituting Ux=y in equation (17), we have

$$Ly=b \tag{18}$$

$$L=\begin{bmatrix} L11 & 0 & & & & 0 \\ L21 & L22 & 0 & & & \\ L31 & L32 & L33 & & & 0 \\ \dots & \dots & \dots & \dots & & \\ \dots & \dots & \dots & \dots & \dots & \\ Ln1 & Ln2 & Ln3 & Ln4 & \dots & Lnn \end{bmatrix} \begin{bmatrix} y1 \\ y2 \\ y3 \\ . \\ . \\ yn \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \\ b3 \\ . \\ . \\ bn \end{bmatrix}$$

After solving it for Y years in forward substitution, U is found using backward substitution from equation (17).

$$Ux=y \tag{19}$$

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & \dots & \dots & U_{1n} \\ 0 & U_{22} & U_{23} & \dots & \dots & U_{2n} \\ 0 & 0 & U_{33} & \dots & \dots & U_{3n} \\ & & & \dots & \dots & \dots \\ & & & & \dots & \dots \\ 0 & 0 & 0 & \dots & \dots & U_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ y_n \end{bmatrix}$$

4.3 Sparse Matrix Solver- SuperLU

SuperLU [28, 29, 30] is a sparse solver package known to be very efficient and reliable when working with the different types of sparse matrices in different applications. It is often used in fluid dynamics, structural mechanics, chemical process simulation, circuit simulation, electromagnetic fields and so on [30]. It is an open source software and readily available for everyone to use.

The first step in SuperLU is to minimize the number of fill-in elements in the lower triangular and upper triangular matrix L and U, respectively. This step is done to ensure that the number of nonzero elements in L and U factors is reduced as much as possible, which in turn reduces the overall solution time. There are many ordering algorithms integrated inside the SuperLU package, which promises reduced fill-in without affecting the solution quality or numerical stability in a quick manner.

After the fill-in ordering step, SuperLU Runs a symbolic algorithm to find out the non-zero pattern of the triangular factors. This process helps in allocating all fill-ins that are implemented in the L and U factors. In addition, it estimates the size of memory storage required for the problem before the next numerical step.

The nonzero pattern found in the symbolic analysis stage defines the numerical values of every column k of L and U. Necessary memory is allocated next for factorization work according to the memory estimation. SuperLU Package uses the compressed row storage CRS format to store sparse matrices. Additional memory is allocated to store L and U matrices as well.

Next comes the numerical factorization phase, where the coefficient matrix A is formatted into L and U matrices. This is the most time-consuming step out of the whole process. It starts with the symbolic analysis of the permuted A matrix (permuted in step 1). Next, the location of all supernodes is determined. The concept of supernodes will be explained later. Using supernodes allows dense nodes in the matrix so that packages like BLAS can be used, which are suitable for dense matrices. The different types of supernodes available. Once the supernodes are determined, they are considered as dense matrices for storage and computation.

To factorize the coefficient matrix A into L and U, SuperLU uses different types of left-looking algorithms. Standard dense matrix-vector multiplication kernels like BLAS level 2 and BLAS level 3 are used based on user selection or the degree of supernode's density.

BLAS Algorithm assumes supernodes and their corresponding columns and rows as a single element and expands them into the actual structure to find the actual L and U. This type of algorithm is known to be very efficient when working with dense matrices and sparse matrices with less than 90% sparsity.

SuperLU ends with doing a backward and forward substation to find the unknown vector. This step uses the traditional substitution techniques on the L and U factors found in the numerical factorization step and the right-hand side vector of the system.

There are many types of supernodes that take many forms. Fig. 4.1 shows different types of supernodes that may be found in a matrix.

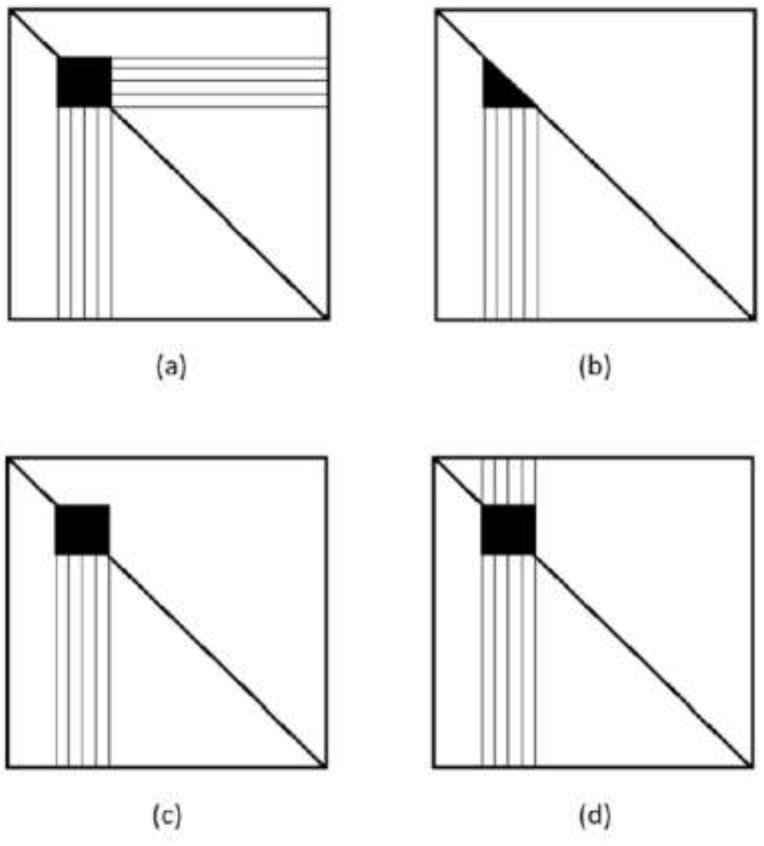


Fig. 4.1: Different types of supernodes in SuperLU a) T1 b) T2 c) T3 d) T4

The dense nodes shown in Fig. 4.1 represent supernodes. As can be seen from the figure, it may occur in different formats.

Supernode T1: A dense matrix where all the elements in the supernode are non-zero. There are nonzero elements along the columns of L and rows of U.

Supernode T2: A dense L matrix along the diagonal. All the elements in the supernode are non-zero. Nonzero elements scattered in the off-diagonal columns of L. There are no nonzero elements in the rows of U.

Supernode T3: A dense L matrix along the diagonal. All the elements in the supernode are non-zero. Nonzero elements scattered in the off-diagonal columns of L. A full U block with no off-diagonal elements in its rows.

Supernode T4: full L and U blocks along the diagonal. Nonzero elements scattered along the columns of L. Stretch of nonzero elements scattered in the columns associated with the full part of U.

Let's take a coefficient matrix A, for example, shown in Fig. 4.2.

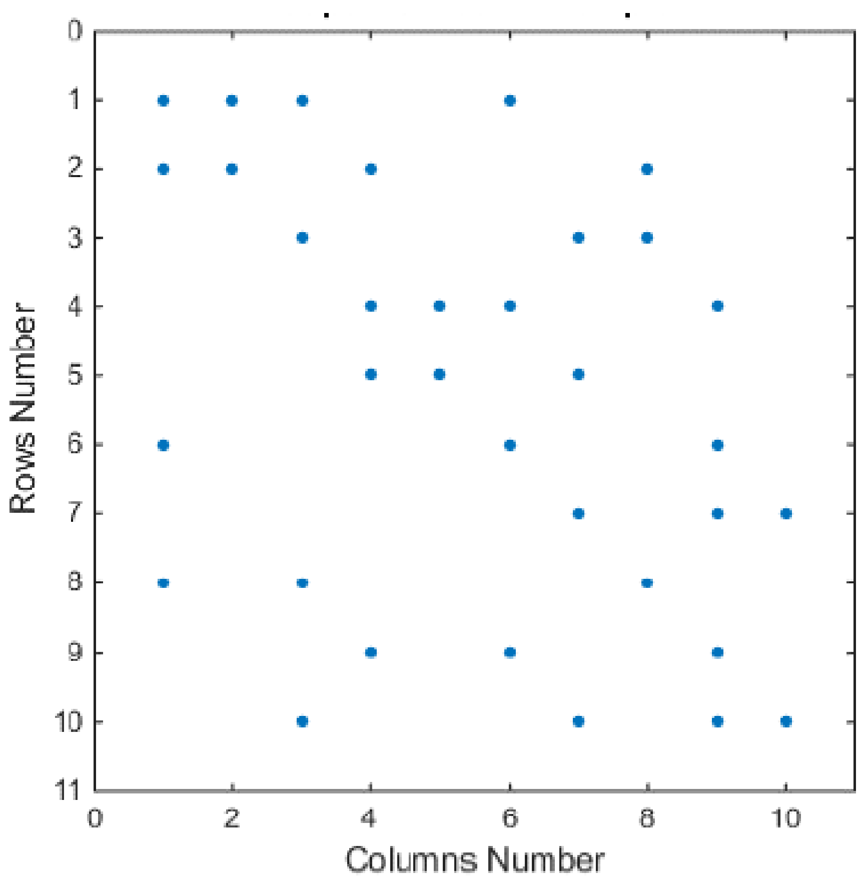
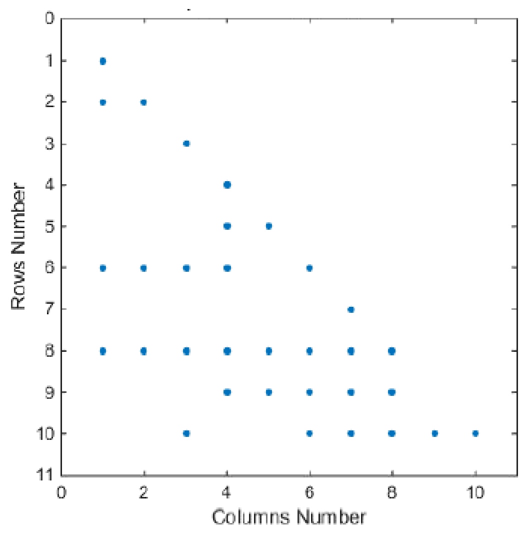
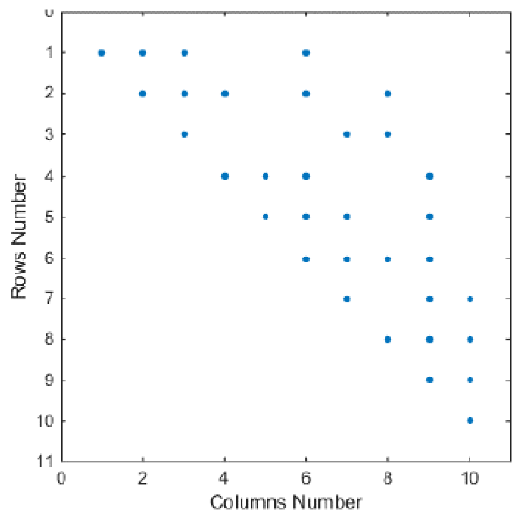


Fig. 4.2: Example of a Matrix to implement SuperLU

This matrix is in its original form. No ordering has been implemented yet. After the first step of SuperLU where a symbolic analysis is applied to the matrix that implements the fill-in reduction and finds the nonzero pattern of L and U, the triangular matrix is shown in fig. 4.3 is found.



(a)



(b)

Fig 4.3: Lower and upper triangular matrix L and U found in SuperLU

A special matrix called filled matrix is used to find all possible supernodes. According to [28], filled matrix can be found by

$$F=L+U-I \tag{20}$$

Where I is an identity matrix of size n by n. It is subtracted from L and U matrix is remove all elements along the diagonal. The SuperLU algorithm finds all possible super nodes in the matrix F based on the type of supernode selected. Fig. 4.4 show the sparsity pattern of the filled matrix F.

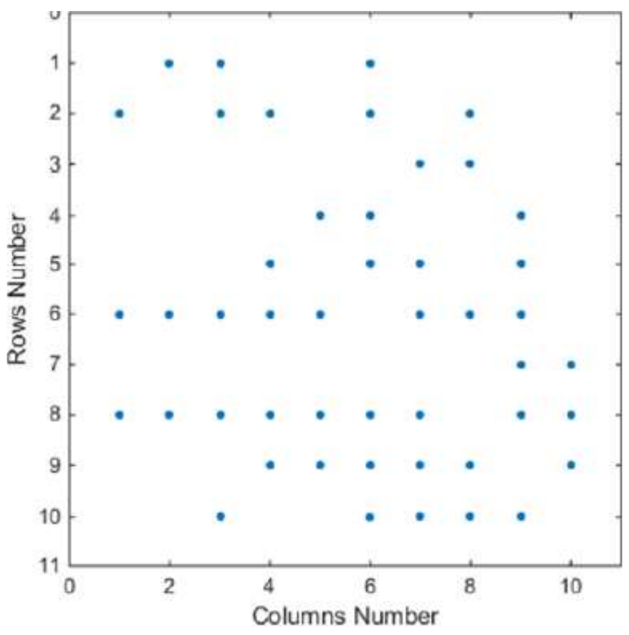


Fig 4.4: Sparsity pattern of filled matrix, F in SuperLU

SuperLU runs a search technique to define all possible supernodes based on the user's selection. For instance, if super node type T1 is selected, five supernodes can be found, as shown in fig. 4.5. The first one is a two-by-two node. It has nonzero elements scattered along the columns and rows corresponding to this full supernode.

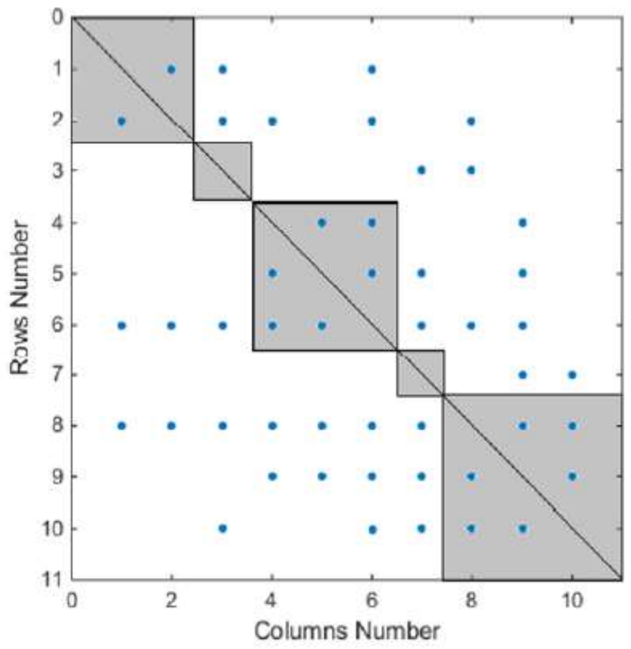


Fig 4.5: T1 Supernodes of matrix A using SuperLU

4.4 Proposed Sparse Matrix Solver- KLU

KLU [31] stands for Clark Kent LU. It is based on Gilbert-Peierls' algorithm, a non-supernodal algorithm, which is the predecessor to SuperLU, a supernodal algorithm [32]. KLU is a sparse, high-performance linear solver which uses hybrid ordering mechanisms and factorization and solves algorithms. It outperforms many traditional metric solvers in circuit simulation.

There are many kinds of gaussian elimination methods. One of them is a left-looking gaussian elimination which factorizes the metric starting from the left-most column. It computes columns of L and U one after another from left to right. Another way is using the right-looking gaussian elimination, which factors the matrix from top left to bottom right computing the column of L and row of U. Both methods have their pros and cons.

KLU uses a left-looking algorithm called Gilbert-Peierls' algorithm. It has two stages. The first stage is a graph theoretical symbolic analysis phase to identify the nonzero pattern of each column of L and U factors. The second stage comprises a left-looking numerical factorization with partial pivoting to calculate the numerical values of the triangular matrices.

KLU employs a hybrid ordering mechanism. Ordering is a way to permute the rows and columns of a matrix to ensure low fill-in in the L and U factors. A fill-in is a non-zero element in the L or U matrix when the corresponding element in A is zero. A good ordering algorithm ensures the minimum fill-in. Finding a good ordering algorithm that gives minimal fill-in is a complete problem itself. KLU offers multiple ordering algorithms like AMD, COLAMD, and user-defined.

KLU implements another ordering stage to ensure the diagonal matrix only has non-zeros. Else the gaussian elimination would fail. KLU ensures a zero-free diagonal with an unsymmetric ordering and permutes the original matrix into a block upper triangular form (BTF) using a symmetric ordering.

In circuit simulation problems, matrix patterns are generally computed once and assumed to stay the same during the whole process. Only the numerical values of the matrix change. Hence, the matrix is permuted once to generate the ordering and nonzero patterns of L and

U factors. The same nonzero pattern is used for the subsequent matrices to update the value of L and U factors. This process is called refactorization, where the analysis and factorization phase is skipped after the first iteration. Refactorization leads to a significant computational time gain.

4.4.1 Gilbert-Peierls' Algorithm

It consists of two stages to determine the value of every column in L and U [32] [33]. The first stage is called a symbolic analysis stage which computes the nonzero pattern of the columns in the triangular factors. The second stage is the numerical factorization stage which calculates the numerical values of every column k of L and U.

The two stages are described in detail below:

4.4.1.1 Symbolic Analysis

The symbolic analysis stage is there to find out the nonzero structure of the L and U factors before the next stage, which is numerical factorization. Here, block A will be analyzed to find its non-zero pattern. The integration of partial pivoting makes this analysis stage more and more complex. Partial pivoting in any column k will add new non-zeros in the following columns starting from k + 1 to n. Hence, the nonzero pattern of L and U factors is hard to predict before numerical factorization due to the dynamic partial pivoting and needs to be updated if there is any change in the pivot of the original matrix A.

To find the nonzero pattern of L and U factors, Gilbert-Peierls' algorithm uses graph theory based on finding the reachability of any non-zero element of A. It starts with assuming the lower factor L to be a unity matrix. It processes each block sequentially in column order. For example, as shown in fig. 4.6, if there is a non-zero element at row j of column k in the A block and factor L has a nonzero element at (i,j) position, then there must be a non-zero present add row i of column k.

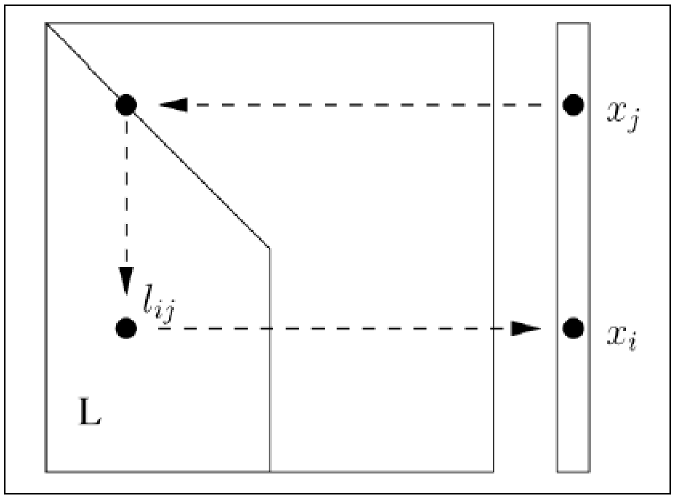


Fig 4.6: The nonzero pattern of x while solving Lx=b

Using this algorithm, we can find the nonzero elements in L and U factors for the next step, numerical factorization. The numerical values will only be calculated for the nonzero elements found in the symbolic stage.

If there is any change in the original matrix or a pivot for A is updated during the partial pivoting stage, the symbolic analysis stage must be updated as well.

4.4.1.2 Numerical Factorization

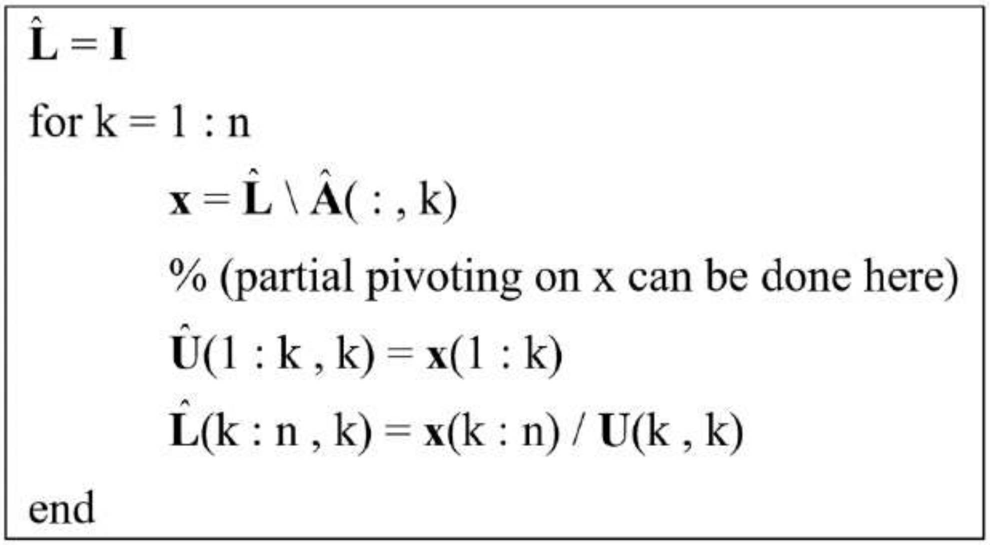
This stage consists of finding the numerical values of L and U factors after the nonzero pattern is found in the symbolic analysis stage. It consists of a left-looking at numerical factorization with the implementation of partial pivoting.

Typically to find the values of each column key of L and U factors, we would calculate to solve the unknown in increasing order of the row index. However, the row indices or the nonzero pattern computed by the depth-first search isn't always in an increasing order. In addition, the topological order of the row indexes can also be used to eliminate unknowns rather than using the increasing order of row indices. The value of X can be found if the values of all the other elements on which x is dependent are already known. Here, we will use our left-looking algorithm based on a depth-first search where a vertex i will be approached only after exploring vertices j, considering j appears before i.

After the new numerical factorization phase, matrix A becomes

$$A=LU \tag{21}$$

Here L and U are the lower and upper factors of the BTF diagonal blocks, respectively. The Gilbert-Peierls' algorithm starts with an identity matrix as the L matrix. The pseudocode for the entire left-looking algorithm is shown below in the fig. 4.7:



```
$\hat{\mathbf{L}} = \mathbf{I}$
for k = 1 : n
        $\mathbf{x} = \hat{\mathbf{L}} \backslash \hat{\mathbf{A}}( : , k)$
        % (partial pivoting on x can be done here)
        $\hat{\mathbf{U}}(1 : k , k) = \mathbf{x}(1 : k)$
        $\hat{\mathbf{L}}(k : n , k) = \mathbf{x}(k : n) / \mathbf{U}(k , k)$
end
```

Fig 4.7: Pseudocode for left-looking algorithm in KLU during Numerical Factorization

Where x=L/b, is the solution of the sparse lower triangular matrix. In this case, b is the kth column of A.

KLU has two types of factorization process namely the full factorization (KLU-FF) and the re-factorization (KLU- RF) process. During the former type, numeric analysis with partial pivoting is done following a symbolic analysis to find the nonzero pattern of the triangular matrices. The pivot of each column being factorized is selected during the partial pivoting stage.

In the refactorization stage, the nonzero pattern of L and U calculated during the previous iteration along with the pivoting order is assumed to be the same and used in the following iterations to find the nonzero values of each element in the L and U factors. This stage only updates the values of the L and U factors according to the change made in the original matrix A.

4.4.2 Block Triangular Format (BTF)

The block triangular form (BTF) transforms any matrix into a triangular matrix by putting as many non-zero elements of the matrix along the diagonal as possible [34]. It is similar to an upper triangular matrix. The only difference with the upper triangular matrix is that BTF matrix diagonals are in the shape of square blocks rather than scalar values.

Transforming a matrix into the BTF form provides many benefits. This reordering process enables partial decoupling of the matrix into different sub-matrices, which can then be solved independently. The diagonal blocks are independent of each other. Only the blocks need to be factorized. The non-diagonal non-zero elements do not contribute to any fill-in. The part of the matrix below the block diagonal does not need to be factorized at all.

The following figure shows a generic representation of a block triangular form of a matrix. The off-diagonal blocks or elements in the matrix are due to light links within a different part of the matrix. For example, blocks A11 and A33 are connected through the A13 block.

Block A22 and A55 are linked by the A25 block. All cases used in this thesis produce no non-zero off-diagonal elements because of the time domain decoupling by the transmission lines.

A=

A11	0	A13	0	0
0	A22	0	0	A25
0	0	A33	0	0
0	0	0	A44	0
0	0	0	0	A55

The permutation technique used to transform a matrix into its BTF form is based on Duff and Reid's algorithm. This algorithm involves finding all strongly connected vertices of a matrix to find the BTF matrix. Duff and Reid implement Tarjan's algorithm to determine the strongly connected components of a directed graph [34].

The first step is to prepare the matrix adjacency graph to guide the algorithm by moving from one graph vertex to another. Then starting from a random vertex depth-first search algorithm is launched to reach the maximum number of graph vertices of which there exists a path. It allocates a stack to keep track of all the visited and unvisited vertices, which also helps avoid many runtime errors like stack overflow and memory shortage.

The algorithm is based on a depth-first search (DFS) topology, which is a recursive algorithm to find all strongly connected vertices possible in a graph, keeping track of all visited and unvisited vertices at the same time [35]. Once the vertices are marked as visited, those form a block together. The algorithm is launched again to start from an arbitrary unvisited vertex.

When all the vertices in the path are explored, it generates strongly connected components from the top of the stack. [36]

Duff's algorithm differs from that of Cormen, Leiserson, Rivest, and Stein, which would suggest doing a depth-first search on the strongly connected component graph G [37]. This is followed by computing the transpose of G, G^T and running a depth-first search again on G^T. The following fig. (4.8-4.10) shows the BTF, a form of a matrix for the given circuit. The BTF method can automatically derive a triangular matrix with blocks in its diagonal line without any user intervention.

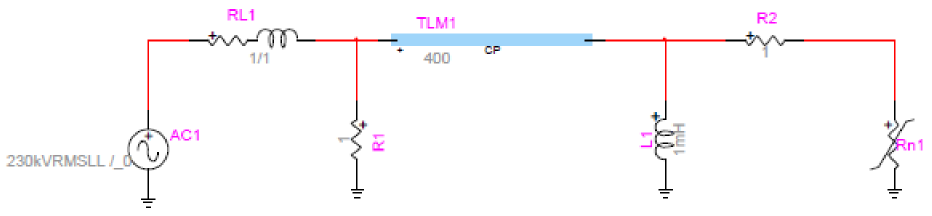


Fig 4.8: Test case for BTF ordering

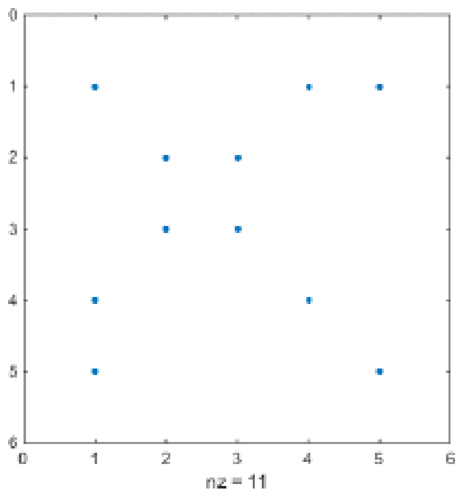


Fig 4.9: Sparsity pattern of A matrix derived from the test case circuit

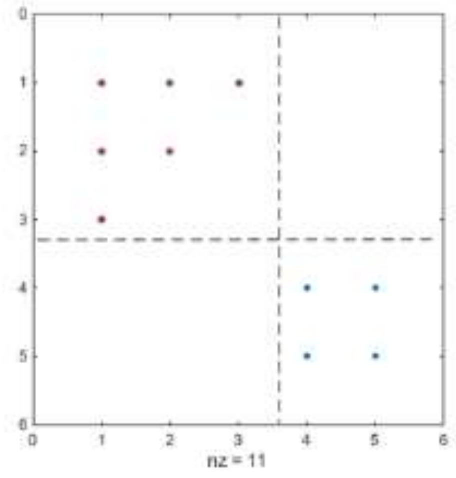


Fig 4.10: BTF form of matrix A

The BTF ordering is also similar to graph transversal ordering which follows Defer's algorithm to find all the decoupled subnetworks. The first step is to find that decoupled using the existing transmission lines and detect each subnetwork at the end of traversal. It applies a heuristic calculation at the same time for all the time costs of each component type in the subnetwork, for instance, resistance, inductance, and machine. It does this to make the simulation fit for real-time simulation. It joins several subnetworks into one based on the execution cost and puts them into one matrix if the resolution fits in one step.

To differentiate between an ordered matrix and non ordered matrix, the "hat" symbol is used to represent all BTF ordered matrices. Additionally, the second digit in the BTF block index is no longer needed since all the cases used here have no off-diagonal blocks, and both digits are used to refer to the same BTF block. For example, $A_i^{\wedge}$ Refers to block i in the BTF-ordered matrix $A^{\wedge}$.

4.4.3 Depth First Search (DFS)

As mentioned earlier, the pattern of any column of L depends on the reachability of the row indices of the said column of A in the graph of L. The reachability is found by a depth first search traversal of the graph of L. In addition, depth first search traversal also determines the topological order for the elimination of variables when solving the lower triangular system Lx=b.

Depth first search algorithm is a recursive algorithm. However, this implementation method of the depth first search creates a major problem in the form of stack overflow. After execution, each process is allocated as space in the stack. However, for a large number of recursive calls, this stack space runs out, resulting in the process being terminated abruptly. This is highly possible in the context of our depth first search algorithm in the case of a dense column of a matrix over a very high dimension.

The solution to the stack overflow problem due to recursion is it replaces it with iteration. In an iterative or non-recursive function, the entire depth first search happens in a single function stack. It uses an array of row indices called Pstack.

The row index of the next adjacent node is stored in the Pstack at the current position corresponding to the current node when descending to an adjacent node during the search. That way, the node stored in the Pstack is the node we need to descend into next after the search returns to the current node. Using this extra memory, the iterative version completes the depth first search in a single function stack.

This is an innovative way to avoid the stack overflow problem caused by the recursive process, which would have been a huge issue in solving high dimension system.

4.4.4 Maximum Transversal

An algorithm was proposed by Duff's to determine the maximum transversal of a directed graph [38, 39]. The aim is to find a row permutation that will minimize the number of zeros on the diagonal of the matrix. For non-singular matrices, the algorithm provides a zero-free diagonal. KLU ensures a zero-free diagonal implementing Duff's algorithm to find an unsymmetric permutation of the input matrix. It is not possible for a structurally singular matrix to be permitted to have a zero-free diagonal. A matrix is called structurally singular when there is no permutation of its nonzero pattern, making it numerically non-singular.

A transversal is a set of non-zeros that lies on the diagonal of the permuted matrix. The condition that the non-zeros have to follow is that they cannot be in the same row or column. A transversal of maximum length is called the maximum transversal.

In Duff's maximum transversal algorithm, the matrix is represented as a graph with each vertex corresponding to a row in the matrix. If there is a nonzero in A (i_k, j_{k+1}) and A $(i_{k+1} j_{k+1})$ is an element in the transversal set, then there is an edge between i_k And i_{k+1}. A path between vertices is i_0 And i_k consists of a set of non zeros (i_0, j_1) , (i_1, j_2), … … (i_{k-1}, j_k)

where the current transversal will include (i_1, j_1) , (i_2, j_2), … … (i_k, j_k) . If there is a nonzero in position (i_k, j_{k+1}) but no non-zero in row i_0 or column j_{k+1} is on the transversal currently, the transversal is increased by 1 by adding the non-zero (i_r, j_{r+1}), r= 0, 1, …, k to the transversal while removing the non-zeros (i_r, j_r) , r=1, 2, …..,k. This process is called augmenting path or reassignment chain, where non-zeros are added and removed to and from the transversal.

The process of appointing augmenting path is started by performing a depth first search from an unassigned row i_0 of the matrix. It stops when a vertex i_k is reached where the path is terminated upon finding a non-zero at A(i_k, j_{k+1}) and column j_{k+1} is unassigned. The search traces back to i_0 by adding and removing transversal elements as it goes. Thus the augmented path is created.

A vertex or row is called assigned if a non-zero in the row is part of the transversal set. Duff's maximum transversal algorithm has the worst-case time complexity of O(n*nnz), where n is the order of the matrix and nnz is the number of non-zeros in the matrix. In reality, the time complexity is close to O(n+nnz).

4.4.5 Ordering

Most of the time, the factorization step of the sparse linear systems are taken place following an ordering phase. The aim of this ordering phase is to find a permutation vector P that would reduce the fill-in in the next factorization phase. A fill-in is a nonzero value in a certain position of the factor that was previously zero in the original matrix. For instance, if L(i,j) is not 0 but A(j,j) is zero then we have a fill-in at (i,j) position.

After applying the required ordering algorithm, the created permitted matrix PAP^T Provides much less fill-in in the factorization phase compared with the unpermitted matrix A. To perform the ordering, numerical values are not needed. The ordering mechanism usually works with the structure of the input matrix without considering the numerical values. If partial pivoting takes place during the factorization phase, it may change the row permutation which will result in the potential increase of the number of fill-ins as opposed to the initial amount estimated by the ordering scheme.

For unsymmetric input matrix A, the formation of the matrix $A + A^T$ can be used as well. There are many minimum degree algorithms available for ordering. Some of the most widely used ordering schemes are approximate minimum degree (AMD) [40, 41] and column approximate minimum degree (COLAMD) [42].

After a matrix A is transformed into its BTF form using the maximum transversal and BTF orderings, KLU moves on to factorize each diagonal block. This is the time when the fill-reducing ordering algorithms are applied before factorizing it. KLU supports various minimum degree algorithms to be used for ordering. The user can also choose to opt for their own user-defined ordering algorithm. So, any given ordering algorithm can be implemented into KLU without much effort. KLU supports both approximate minimum degree and column approximate minimum degree algorithms for its ordering process.

Results have shown that out of the various ordering schemes applicable, approximate minimum degree ordering (AMD) gives the best result for the power system and circuit matrices. The goal of AMD is to reduce an optimistic estimate of fill-in. It assumes no numerical pivoting during its process. AMD calculates the permutation vector P to reduce fill-in for the Cholesky factorization of PAP^T. If the input matrix A is unsymmetric, then it finds a permutation P for the factorization of $P(A + A)^T P^T$.

COLAMD is an unsymmetric ordering scheme that finds a column permutation Q with an aim to reduce the fill-in for Cholesky factorization of $(AQ)^T AQ$. Unlike AMD, COLAMD tries to reduce a pessimistic estimate of fill-in.

Another ordering scheme that creates permutation in such a way that the input matrix can be transformed into block diagonal form with the vertex separators is called nested dissection. Although a popular choice, it is unsuitable to use in circuit matrices when applied to the matrix as such. It can be used on the blocks generated by BTF pre-ordering.

The following figures are given to expand upon how the minimum degree algorithm works. A structurally symmetric matrix can be represented by an equivalent undirected graph with vertices that resemble the row and column indices. If there is a non-zero at A(i,j) position in A, then there is an edge from i to j.

Let’s consider a symmetric matrix, and its graph representation is shown in fig. 4.11. The matrix is factorized considering vertex 1 as the pivot; then it would transform to fig. 4.12 after the first gaussian elimination step. The first step of elimination is the same as removing node 1 and all its edges from the graph. It adds new edges to connect all the nodes adjacent to 1. This step is equivalent to creating a clique of the nodes adjacent to the eliminated node. Note that there are as many fill-ins in the transformed matrix as there are edges add it in the clique formation.

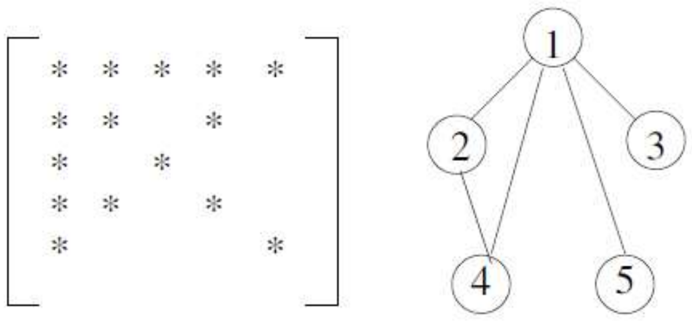


Fig. 4.11: A symmetric matrix and its graph representation

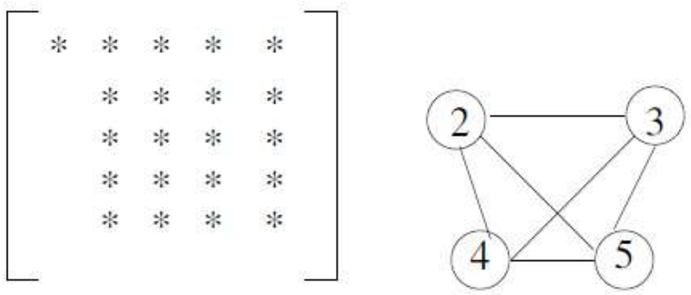


Fig 4.12: The matrix and its graph after the first step of gaussian elimination

It is a good idea to choose an index with minimum degree as the pivot. In this example, It was wrong to choose node one as pivot since it has the maximum degree. Instead, node 3 or 5 should have been chosen as they both have the minimum degree. This would have resulted in zero fill-ins after the elimination since they both have degree 1.

This is the main idea behind the minimum degree algorithm. It chooses a permutation in a way so that a node with minimum degree is eliminated in each step of the elimination process in shooting a minimal fill-in. It doesn't take the numerical values in the node into consideration. It only works with a nonzero pattern. If partial pivoting is implemented in the later stage of numerical factorization, a different node other than the one suggested by the minimum degree algorithm may be chosen as the pivot because of its numerical magnitude. This is why the fill-in estimate suggested by the ordering algorithm could be different from that found in the factorization phase.

4.4.6 Pivoting

One issue with Gaussian elimination is that it fails if any diagonal element in the input matrix is zero. Considering two by two matrix,

$$A = \begin{bmatrix} 0 & a12 \\ a21 & a22 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \end{bmatrix} \tag{22}$$

To solve the system, the elimination process first computes the multiplier -a21/a11 and eliminates the coefficient element a21 from the matrix by multiplying row one with the multiplier and adding it to row 2. For this case, this step fails since a11 is 0.

Let's look at another scenario where the diagonal element is nonzero but close to 0.

$$A = \begin{bmatrix} 0.0001 & 1 \\ 1 & 1 \end{bmatrix} \tag{23}$$

The multiplier is -a21/a11=-1/.0001=-10^4.

The factors L and U are

$$L = \begin{bmatrix} 1 & 0 \\ 10^4 & 1 \end{bmatrix} \tag{24}$$

$$U = \begin{bmatrix} 0.0001 & 1 \\ 0 & -10^4 \end{bmatrix} \tag{25}$$

Here the element u22 has the value 1-10^4 But it was rounded off too -10^4.

The product of L and U is

$$L * U = \begin{bmatrix} 0.0001 & 1 \\ 1 & 0 \end{bmatrix} \tag{26}$$

which is different from the original input matrix A. It so happens because the multiplier is so large that when added with the small element a22 with the value 1, it obscures the tiny value present in a22. This issue is tackled by pivoting. We could solve this problem for the two examples mentioned above if rows 2 and 1 are interchanged.

Pivoting is a mechanism where rows and columns are interchanged to pick a large element as the diagonal, which inevitably avoids numerical failures or in accuracies. To pick the pivot element, either the element at the current column is considered or the entire submatrix across both rows and columns. The former is called partial pivoting, and the latter is called complete pivoting.

Comparing the performance between these two, complete pivoting seems to be more expensive and adds a higher time complexity. Hence it is typically avoided with the exception of special cases. KLU implements partial pivoting with diagonal preference. If the diagonal element times a constant threshold is bigger than the largest element in the column, it is chosen as the pivot. This constant threshold is called pivot tolerance.

Pivot tolerance*Adiag > Ahighest

4.4.7 Scaling

The pivoting process cannot completely overcome the issue of small elements in the matrix getting obscure during the elimination process and the accuracy of the results getting skewed because of numerical addition. Let's take an example of a 2* 2 matrix to expand upon this problem:

$$A = \begin{bmatrix} 10 & 10^5 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} 10^5 \\ 2 \end{bmatrix} \tag{27}$$

If gaussian elimination with partial pivoting is applied to the above system, the entry a11 being the largest in the first column, would be considered the pivot. After the first step of elimination, we would have

$$A = \begin{bmatrix} 10 & 10^5 \\ 0 & -10^4 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} 10^5 \\ -10^4 \end{bmatrix} \tag{28}$$

Solving the above system, we getx1=1, x2=0. However, the correct solution is x1=1, x2=1.

If you divide each row of the matrix and the corresponding element in the right-hand side by the largest element in that row before gaussian elimination, we would have

$$A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{29}$$

After applying partial pivoting,

$$A = \begin{bmatrix} 1 & 1 \\ 10^{-4} & 1 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{30}$$

Ending after an elimination step, the result will be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 - 10^{-4} \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 - 10^{-4} \end{bmatrix} \tag{31}$$

which gives us the correct solution x1=1, x2=1.

The process of balancing out the numerical enormity or obscurity on each row or column is called scaling. In the above example, we will use row scaling, where scaling is done with respect to the maximum value in a row. There is another way to scale, which considers the sum of the absolute value of all elements across a row rather than the maximum value. There is also column scaling, where scaling is done with respect to either the maximum value in a column or the sum of absolute values of all elements in a column.

Row scaling is the same as finding an invertible diagonal matrix D1 such that all the rows in the matrix $D^{-1}A$ have equally large numerical values. Once such a diagonal matrix is obtained, the solution of the original system Ax=b is equivalent to solving the system A'x=b' where $A' = D^{-1}A$ and b'=$D^{-1}b$. Scaling is also referred to as equilibration.

In KLU, the diagonal elements of the diagonal matrix D1 are either the largest elements in the rows of the original matrix or the sum of the absolute values of the elements in the rows. Scaling can be used at users' discretion. Although it provides better numerical results when solving systems, Scaling is not mandatory. If the values are already balanced, scaling might not be necessary.

4.4.8 Left-looking Gaussian Elimination

We will expand on how the left-looking version of Gaussian elimination work. For an input matrix

A of n*n order can be represented as a product of 2 triangular matrices, L and U.

Let,

$$\begin{bmatrix} A11 & \boldsymbol{a12} & A13 \\ \boldsymbol{a21} & a22 & \boldsymbol{a23} \\ A31 & \boldsymbol{a32} & A33 \end{bmatrix} = \begin{bmatrix} L11 & 0 & 0 \\ \boldsymbol{l21} & 1 & 0 \\ L31 & \boldsymbol{l32} & L33 \end{bmatrix} * \begin{bmatrix} U11 & \boldsymbol{u12} & U13 \\ 0 & u22 & \boldsymbol{u23} \\ 0 & 0 & U33 \end{bmatrix} \tag{32}$$

Where Aij is a block, **aij** is it vector and aij is a scalar. The dimension of these different elements in the matrices are as follows:

A11, L11, U111 are k*k blocks

a12, u12 are k*1 vectors.

A13, U13 are k*n-(k+1) blocks

a21, l21 are 1*k row vectors.

a22, u22 are scalars.

a23, u23 are 1*n-(k+1) row vectors.

A31, L31 are n-(k+1)*k blocks

a32, l32 are n-(k+1)*1 vectors.

A33, L33, U33 are n-(k+1)* n-(k+1) blocks.

From equation (32) the following set of equations can be derived.

$$L11 * U11 = A11 \tag{33}$$

$$L11 * \mathbf{u11} = \mathbf{a12} \tag{34}$$

$$L11 * U13 = A13 \tag{35}$$

$$\mathbf{l21} * U11 = \mathbf{a21} \tag{36}$$

$$\mathbf{l21} * \mathbf{u12} + u22 = a22 \tag{37}$$

$$\mathbf{l21} * U13 + \mathbf{u23} = \mathbf{a23} \tag{38}$$

$$L31 * U11 = A31 \tag{39}$$

$$L31 * \mathbf{u12} + \mathbf{l32} * u22 = \mathbf{a32} \tag{40}$$

$$L31 * U13 + \mathbf{l32} * \mathbf{u23} + L33 * U33 = A33 \tag{41}$$

From (34), (37), and (40) , we can compute the second column of L and U, assuming we have already found L11, **l21,** and L31. First, the lower triangular system (34) is solved for **u12.** Then u22 is solved using (37)

$$u22 = a22 - \mathbf{l21} * \mathbf{u12} \tag{42}$$

Finally, **l32** is solved from (40)

$$\boldsymbol{l32} = \frac{a32 - L31 * \boldsymbol{u12}}{u22} \tag{43}$$

This process of computing the second column of L and U is the same as solving a lower triangular system as follows

$$\begin{bmatrix} L11 & 0 & 0 \\ \boldsymbol{l21} & 1 & 0 \\ L31 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \boldsymbol{u12} \\ u22 \\ \boldsymbol{l32} * u22 \end{bmatrix} = \begin{bmatrix} \boldsymbol{a12} \\ a22 \\ \boldsymbol{a32} \end{bmatrix} \tag{44}$$

This mechanism of computing column k of L and U when solving a lower triangular system Lx=b is the key step in a left-looking factorization algorithm. Gilbert-Peierls' Algorithm used in KLU is based on solving this lower triangular system. It is called a left-looking algorithm because column K of L and U are computed by using the already calculated columns in the left 1…..k-1 of L. That means to compute column K of L and U, the column to the left of the currently computed column k has to be calculated already.

Chapter 5

Implementation of KLU in Power Flow Analysis

5.1 Overview of KLU

The KLU algorithm follows the following steps

1. The matrix is permuted to a block upper triangular form (BTF) consisting of two stages:
    a. An unsymmetric permutation to ensure zero free diagonal using maximum transversal
    b. A symmetric permutation to create block triangular form by finding the strongly connected components of the graph.
2. Each block is ordered with a fill-in-reducing ordering scheme. Symmetric permutation using AMD on $A + A^T$ Shows the best result for electrical simulations. Users can also provide any customized ordering algorithm. If needed, an unsymmetric permutation of each block using COLAMD on AA^T can also be done.
3. Each block is factorized by the left-looking Gilbert-Peierls' algorithm with partial pivoting
4. The system is solved with block-back substitution accounting for the off-diagonal elements. The solution is re-permuted to bring it back to the original order.

Let's consider the original system to solve as

$$Ax=b \tag{45}$$

Let R be the diagonal matrix with the scale factors for each row. Applying scaling, we have

$$RAx=Rb \tag{46}$$

Let P’ and Q’ be the row and column permutation matrices that combine the permutations for maximum transversal and block upper triangular form together.

Applying these permutations together, we have

$$P'RAQ'Q'Q'^{T}x = P'Rb \tag{47}$$

After symmetric permutation produced by AMD and partial pivoting row permutation produced by factorization, we get row and column permutation matrix P and Q.

$$PRAQQ^{T}X = PRb \tag{48}$$

$$(PRAQ)Q^{T}X = PRb$$

The matrix (PRAQ) consists of two parts.

1. The diagonal blocks that are factorized
2. The off-diagonal elements that are not factorized

We get

$$(PRAQ)=LU+F \tag{49}$$

Where LU represents the factors of all blocks altogether, F represents the entire off-diagonal region. So, equation (48) can be rewritten as

$$(\mathrm{LU} + \mathrm{F})Q^T x = PRb$$

$$\mathrm{X} = \mathrm{Q}(\mathrm{LU} + \mathrm{F})^{-1}(PRb) \quad (50)$$

Equation (50) consists of two steps. The first one is applying block back substitution where $(\mathrm{LU} + \mathrm{F})^{-1}(PRb)$ Is computed. The second step is to apply the column permutation Q.

For a 3*3 system, the block back substitution method can be explained as follows:

$$\begin{bmatrix} L11U11 & F12 & F13 \\ 0 & L22U22 & F23 \\ 0 & 0 & L33U33 \end{bmatrix} * \begin{bmatrix} X1 \\ X2 \\ X3 \end{bmatrix} = \begin{bmatrix} B1 \\ B2 \\ B3 \end{bmatrix} \quad (51)$$

Solving the above system gives us the following equations

$$\mathrm{L11U11*X1+F12*X2+F13*X3=B1} \quad (52)$$

$$\mathrm{L22U22*X2+F23*X3=B2} \quad (53)$$

$$\mathrm{L33U33*X3=B3} \quad (54)$$

In block back-substitution, equation (54) is first solved for X3. Now, X3 is eliminated from equation (52) and (53) using the off-diagonal entries. Next, equation (53) is solved for X2 and X2 is eliminated from equation (52). Finally, we can solve equation (52) for X1.

5.1 Characteristic of Circuit Matrices

There are certain unique characteristics of circuit matrices that make KLU so suitable for them. They are listed below:

- Circuit matrices are usually very sparse.
- BLAS Kernels cannot be applied to them due to their high sparsity.
- They often have a few dense rows/columns which originate from the voltage or current sources. However, BTF permutation effectively removes these dense rows or columns.
- Although circuit matrices are asymmetric, the nonzero pattern is roughly symmetric.
- They are also easily permittable to block upper triangular form.
- Most circuit matrices have zero free or nearly zero free diagonal.
- A strange characteristic of the circuit matrix is that the nonzero pattern of each block after BTF permutation is more symmetric than the original matrix.
- When applied to the original matrix, typical ordering strategies cause high fill-in. But when applied to BTF blocks. The result is less fill-in.
- The whole sub diagonal region in the matrix has zero work.
- The off-diagonal elements in the BTF blocks do not cause any fill-in.

5.2 Applying KLU to Non-Linear Power Flow Analysis

Power system solvers typically work with nonlinear systems. The linear systems in the circuit simulation process are part of solving a large system of nonlinear equations. The linear systems usually include a coefficient matrix A, an unknown vector x, and right-hand side b.

The non-linear system is solved by an iterative process of solving the said linear system. During iterations, the coefficient matrix A maintains its non-zero pattern. The only changes that occur are in its numerical values. Hence in KLU, the initial symbolic analysis phase in the factorization phase is needed to be completed only once at the start. During the symbolic analysis phase, the initial system is permitted to ensure a zero free block diagonal form and minimum degree ordering on blocks. It is followed by the factorization phase, where the lower and upper triangular matrices are formed.

In the following iterations, A'x=b is solved where A' differ from A only in numerical values. Hence the sparsity pattern remains the same, this matrix can be solved using a mechanism called refactorization. In the re-factorization process, it is assumed that the row and column permutations formed by the analysis phase and partial pivoting remain constant for the remainder of the simulation process. Only the numerical values changes in the subsequent systems. Refactorization decreases the computational time significantly since the time to perform symbolic analysis, factorization, and partial pivoting is avoided. The nonzero pattern of the upper and lower triangular matrix U and L are the same as for the initial system. During re-factorization, only the numerical values of the triangular matrices are updated based on the changes in the original system. This step is followed by the solve step. KLU is able to solve up to four right-hand sides in a single solve step.

For AC power system, using Newton Raphson method, the linear system looks like Jx=b. Here, J is the Jacobian matrix derived from the admittance matrix Y, b is the active (ΔP) and reactive power (ΔQ) mismatch with the given Voltage (V), the result x is used to update the V. x has both the angle of the PQ and PV buses and magnitude of the PQ buses as can be seen in equation (4).

$$J_{1....n}.x_{1...n} = b_{1....n} \tag{55}$$

$$x = [\,\Delta\delta(pv,pq), \Delta|V|(pq)] \tag{56}$$

$$b = [\Delta P, \Delta Q] \tag{57}$$

Over the course of the simulation, the sparsity pattern and structure of the J matrix do not alter. Only its numerical values change due to the presence of time-varying and non-linear elements in the system.

The flow chart showing the entire algorithm is given below in fig. 5.1.

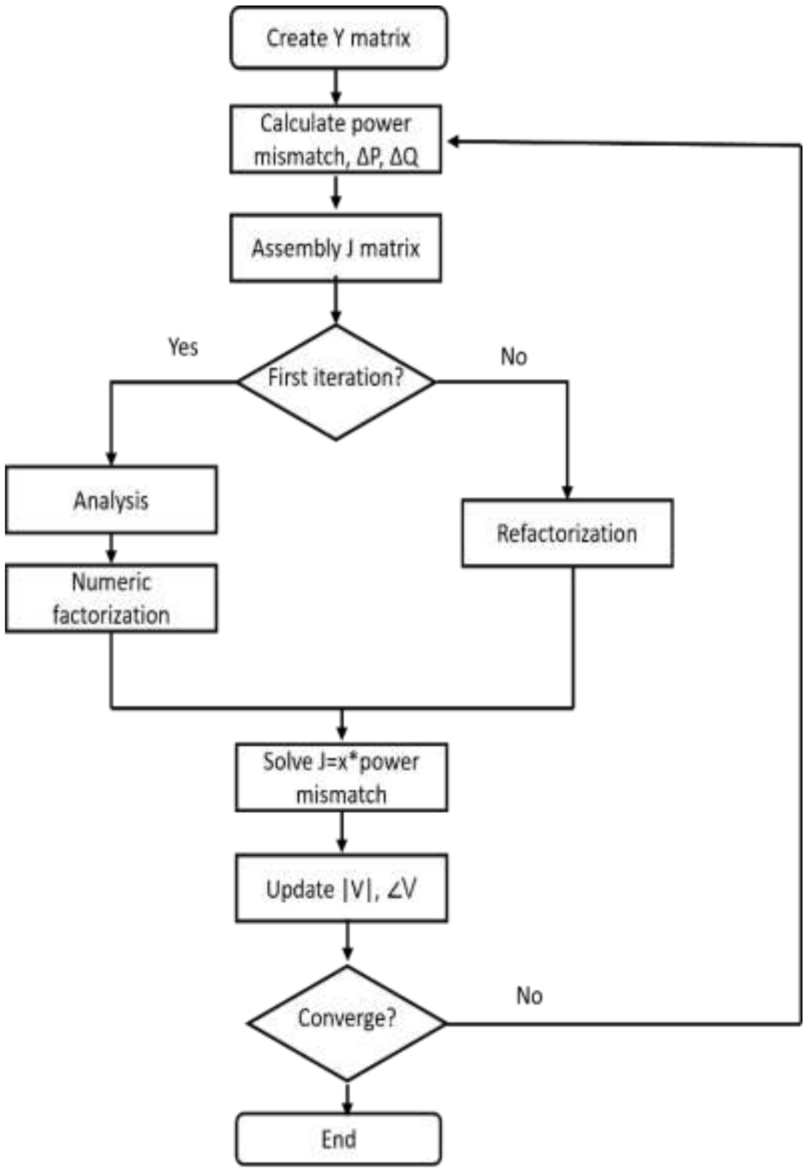


Fig 5.1: Flow chart to implement KLU in PF analysis

CHAPTER 6

COMPUTATION RESULTS AND CONTRIBUTIONS

6.1 Contributions

To reduce the computational time, KLU refactorization is used, which is implemented from the second iteration. This thesis provides a further computational gain in the refactorization process.

The blocks in the BTF matrix are independent of each other. It is possible to solve these blocks simultaneously. As KLU follows a left-looking gaussian algorithm, any value in the columns of the original coefficient matrix after factorization depends on all the columns left to that. Hence, if there is any change in the matrix, all the values to the right of that element will be changed leaving the left columns as it is.

This characteristic is used to provide higher computational gain in this thesis. For any change in the original matrix, only the values to the right of that elements are updated in the L and U matrix during the refactorization process. Usually, in KLU, during refactorization, the values in the L and U matrix are updated from the first column. If we follow the suggested modification, the gain will depend on the position of the change. If the change happens in the column to the furthest right, it will give the highest gain. However, if the change happens in the first column, there will not be any significant gain compared with the typical refactorization.

To make this modification, first, the mapping between the original coefficient matrix and the BTF matrix needs to be prepared. Using the row permutation and column permutation matricies P and Q, a mapping can be drawn to determine the relation between the position of each element in the original matrix A and BTF transformed form A.

f: A->A~

Let *f* be a set containing the mapping between the original matrix A and BTF matrix A~.

During the first phase of KLU, the symbolic analysis computes the column permutation matrix P, which reflects the relationship of each column between the original and BTF matrix. P[0]=2 means that the first column of the BTF matrix contains the 3rd column of the original matrix. Block matrix R in KLU contains the starting and ending columns of each block in BTF. For example, R[n] gives the starting column number of n number block. The difference between R[n] and R[n+1] provides the size of n block.

To find the corresponding column number in a BTF matrix of the changed elements in A, the column permutation vector P has to be inversed first. From Pinv, one can convert the column indices of the original matrix to the corresponding BTF column indices. Using the starting and ending column number of n block and the column permutation matrix P, first, the block number of the changed elements are found.

When there is a change in the original matrix, the first mapping is done to find the corresponding column in the BTF matrix. Next, it is found which block that matrix belongs to. Once all the changed columns and blocks are identified, the left-most column is used for further calculation of the L and U matrix. Everything left to that column is left alone. For example, in a 50 by 50 matrix, let’s assume the changed elements are in columns 9, 15, 30, 40, 44, and 46 in the BTF form of the same block. Then the calculation to find the numerical values of the triangular matrix L and U is done starting from column 9 since it is the leftmost column. There is no computation needed for columns 1 to 8.

Due to the use of the left-looking algorithm, it is possible to significantly reduce the computational time depending on the position of the changed column. From the above example, if the changed element resides in column 50th column, it gives the highest gain since only one column is updated in the L and U matrix.

The efficiency of the algorithm is heightened with the implementation of parallel programming. The parallel execution of the code is ensured by using OpenMP, a thread programming tool in the Windows computing environment. It is a high-level threading technique that requires the user to define certain segments of the code where parallel processing is expected. Compared with the other available resource for parallel programming in the market, the implementation of OpenMP requires minimum changes to the sequential code.

Open MP has different directives to control the environment of its parallel operation. In the KLU first stage, symbolic analysis is done sequentially since the computation time required for this is not too big as it only runs once at the beginning of the simulation. However, the other steps of KLU, factorization, re-factorization, and the forward and backward substitution can be made to execute parallelly. The BTF permutation rearranges the initial J matrix into n number of blocks that are mutually exclusive. Hence, KLU factorization and re-factorization can be done in parallel by allowing the execution of different blocks simultaneously. In the final stage of KLU, the backward and forward substitution can also be similarly done in parallel.

The solver is implemented on distributed memory model where each thread has thread-specific variables and some variables they share. It is important to distinguish among each type of variables if they are shared or thread-specific to avoid any potential race conditions. Blocks can be assigned to threads in either a dynamic or static manner. However, it is seen that choosing the dynamic operation guarantees higher gain, especially for multicore processors.

6.2 Computation Results

6.2.1 Comparison with Newton-Raphson (NR) Method

For testing, a computer with Intel ® Core(TM) i7-1165G7 processor and 16 GB RAM is used with Windows 11 operating system. CPU parallelization is implemented using OpenMP.

The test systems are as follows:

1. IEEE 6: 6 bus, 3 gen case from Wood & Wollenberg.
2. IEEE 9: WSCC System. Contains 9 bus, 3 generators.
3. IEEE 30: American Electric Power system in December 1961
4. IEEE 39: 10-machine New-England Power System. Has 10 generators and 46 lines.
5. IEEE 57: American Electric Power system in early 1960s. Has 57 buses, 7 generators, and 42 loads
6. IEEE 118: 19 generators, 35 synchronous condensers, 177 lines, 9 transformers, and 91 loads
7. IEEE 300- 69 generators, 60 LTCs, 304 transmission lines, and 195 loads.
8. IEEE 2383: Polish System data in Winter 1999-2000 peak.
9. IEEE 2736: Polish 400, 220 and 110 kV networks during summer 2004 peak conditions
10. IEEE 2746: Polish system - winter 2003-04 evening peak.
11. IEEE 3012: bus- Polish system - winter 2007-08 evening peak

12. IEEE 3120: bus- Polish system - summer 2008 morning peak

13. IEEE 6468: bus- French VHV+HV grid in 2013

14. IEEE 6495: bus- French VHV+HV grid in 2013

15. IEEE 6515: bus- French VHV+HV grid in 2013

16. IEEE 10000: bus- U.S. portion of the Western Electricity Coordinating Council (WECC)

The performance is compared using the original KLU solver, modified KLU solver, and Newton Raphson model for the AC power flow.

TABLE 6.1

Computation time of each test system using KLU, Modified KLU, and N-R in AC power flow

Test System	**Original KLU Solver**	**Modified KLU Solver**	**Newton-Raphson**
IEEE 6 bus	.0056	.003	.033
IEEE 9 bus	0.015	0.01	0.029
IEEE 30 bus	0.004	0.005	0.018
IEEE 39 bus	0.002	0.001	0.015
IEEE 57 bus	0.006	0.006	0.019
IEEE 118 bus	0.008	0.007	0.026

IEEE 300 bus	0.034	0.033	0.065
IEEE 2383 bus	0.254	0.234	6.007
IEEE 2736 bus	0.192	0.177	3.914
IEEE 2746 bus	0.183	0.176	3.742
IEEE 3012 bus	0.093	0.084	2.774
IEEE 3120 bus	0.327	0.301	8.646
IEEE 6468 bus	0.388	0.33	34.229
IEEE 6495 bus	0.386	0.376	32.408
IEEE 6515 bus	0.378	0.333	33.473
IEEE 10000 bus	0.748	.665	134.358

Fig 6.1 lists the time needed for each test system by original KLU, modified KLU, and the newton-raphson solve for AC power flow. As seen above, the original KLU and modified KLU outperforms the newton Raphson model to a great extent.

Since this is for AC power flow, each system is solved by doing a certain number of iterations. To better understand the efficiency of the solvers, the computation time needed per iteration is shown below in table 6.2.

TABLE 6.2

Computation time of each test system per iteration using KLU, Modified KLU, and N-R in AC power flow

Test System	**Original KLU Solver**	**Modified KLU Solver**	**Newton-Raphson**
IEEE 6 bus	.002	.001	.011
IEEE 9 bus	.004	.003	.007
IEEE 30 bus	.001	.002	.006
IEEE 39 bus	.002	.001	0.015
IEEE 57 bus	.002	.002	.006
IEEE 118 bus	.003	.002	.009
IEEE 300 bus	.007	.007	.013
IEEE 2383 bus	.042	.039	1.001
IEEE 2736 bus	.048	.044	.979
IEEE 2746 bus	.046	.044	.936
IEEE 3012 bus	.047	0.042	1.387
IEEE 3120 bus	.055	.050	1.441
IEEE 6468 bus	.129	.110	11.410
IEEE 6495 bus	.129	.125	10.803

IEEE 6515 bus	.126	.111	11.158
IEEE 10000 bus	.183	.166	35.590

As seen from table 6.2, modified KLU needs the least time to execute one iteration, followed by the original KLU solver. Newton-raphson method requires the highest time out of these three for all the test cases.

The computational gain is given below for each of the test cases in fig 6.1:

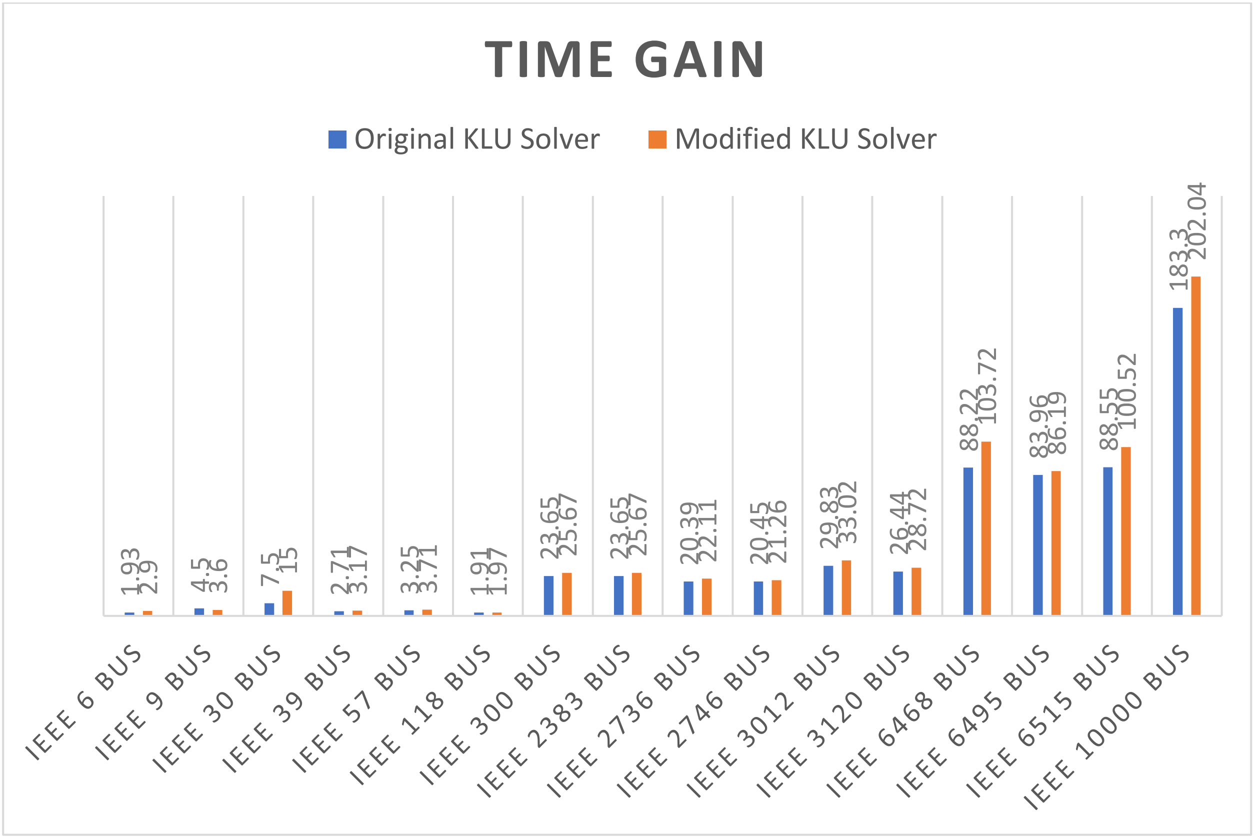


Fig 6.1: Computational gain of each test system comparing NR with KLU and Modified KLU in AC power flow

The computational gain increases with the increase in the size of the system. For the smallest power system of 6 buses, it is 1.93 and 2.9 respectively for original KLU and modified KLU when compared with the newton-raphson model. It is as high as 200 times for 10K bus system using the proposed algorithm, as seen in fig 6.1. This means that the proposed solver can solve a power flow equation 200 times faster than the traditional newton-raphson model.

The same performance analysis is shown for a DC power flow below in table 6.3

TABLE 6.3

Computation time of each test system using KLU, Modified KLU, and N-R

in DC power flow

Test System	**Original KLU Solver**	**Modified KLU Solver**	**Newton-Raphson**
IEEE 6 bus	.000	.000	0.212
IEEE 9 bus	0.002	.000	.038
IEEE 30 bus	0.002	0	0.028
IEEE 39 bus	0.002	.000	.052
IEEE 57 bus	.002	.000	.062
IEEE 118 bus	.003	.0005	.071
IEEE 300 bus	.004	.000	.047

IEEE 2383 bus	.023	.006	.368
IEEE 2736 bus	.021	.008	.598
IEEE 2746 bus	.002	.011	.708
IEEE 3012 bus	.006	.003	.434
IEEE 3120 bus	.005	.003	.476
IEEE 6468 bus	.011	.007	3.076
IEEE 6495 bus	.012	.007	3.13
IEEE 6515 bus	.011	.007	3.302
IEEE 10000 bus	.019	.01	9.96

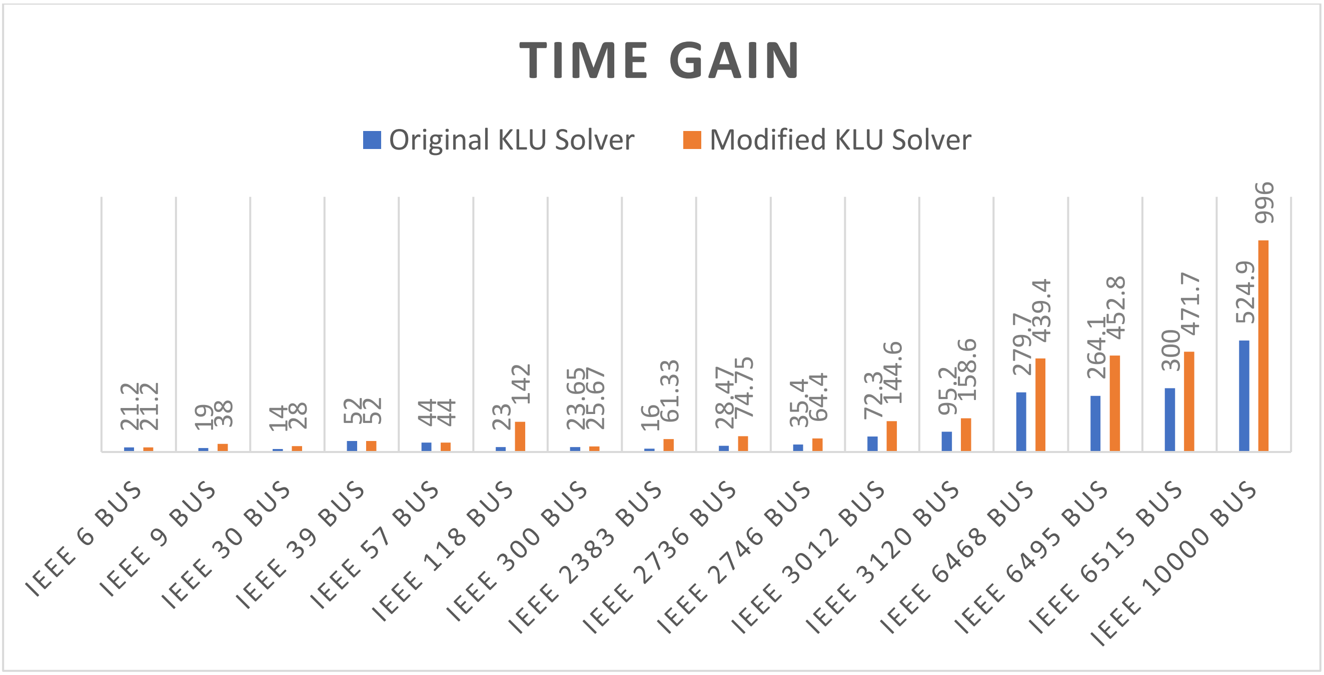


Fig 6.2: Computational gain of each test system comparing NR with KLU and Modified KLU in DC power flow

The computational gain is even higher for the DC power flow. It increases as the total number of buses increases. For a 10k bus system, as seen in fig 6.2, it is around 1000 times faster when compared with the traditional newton Raphson model.

6.2.2 Comparison with MATPOWER

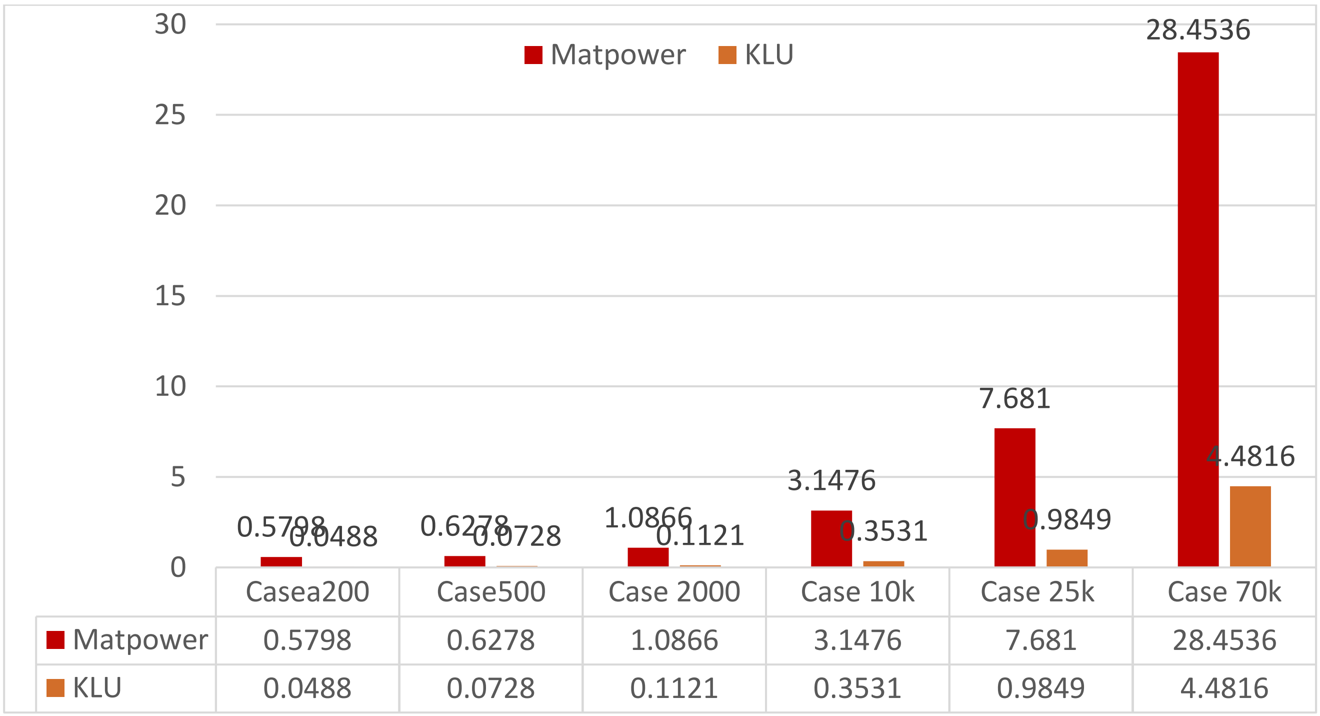
MATPOWER is another popular software used to solve power systems. KLU outperforms MATPOWER also in every case.

The test cases used to compare the performance of MATPOWER are:

1. ACTIVSg200 (200-bus synthetic power grid geolocated in Illinois, USA)
2. ACTIVSg500 (500-bus synthetic power grid geolocated in South Carolina, USA)
3. ACTIVSg2000 (2000-bus synthetic power grid geolocated in Texas, USA)
4. ACTIVSg10k (10,000-bus synthetic grid geolocated in the Western USA)
5. ACTIVSg25k (25,000-bus synthetic power grid geolocated in the Mid-Atlantic USA)
6. ACTIVSg70k (70,000-bus synthetic grid geolocated in the Eastern USA)

The test platform for the KLU solver is a computer equipped with an NVIDIA GeForce RTX® 2080 GPU, one Intel(R) Core(TM) i9-9900 Central Processing Unit (CPU). The operating system is Microsoft Windows 10, and the CUDA version is 7.5.

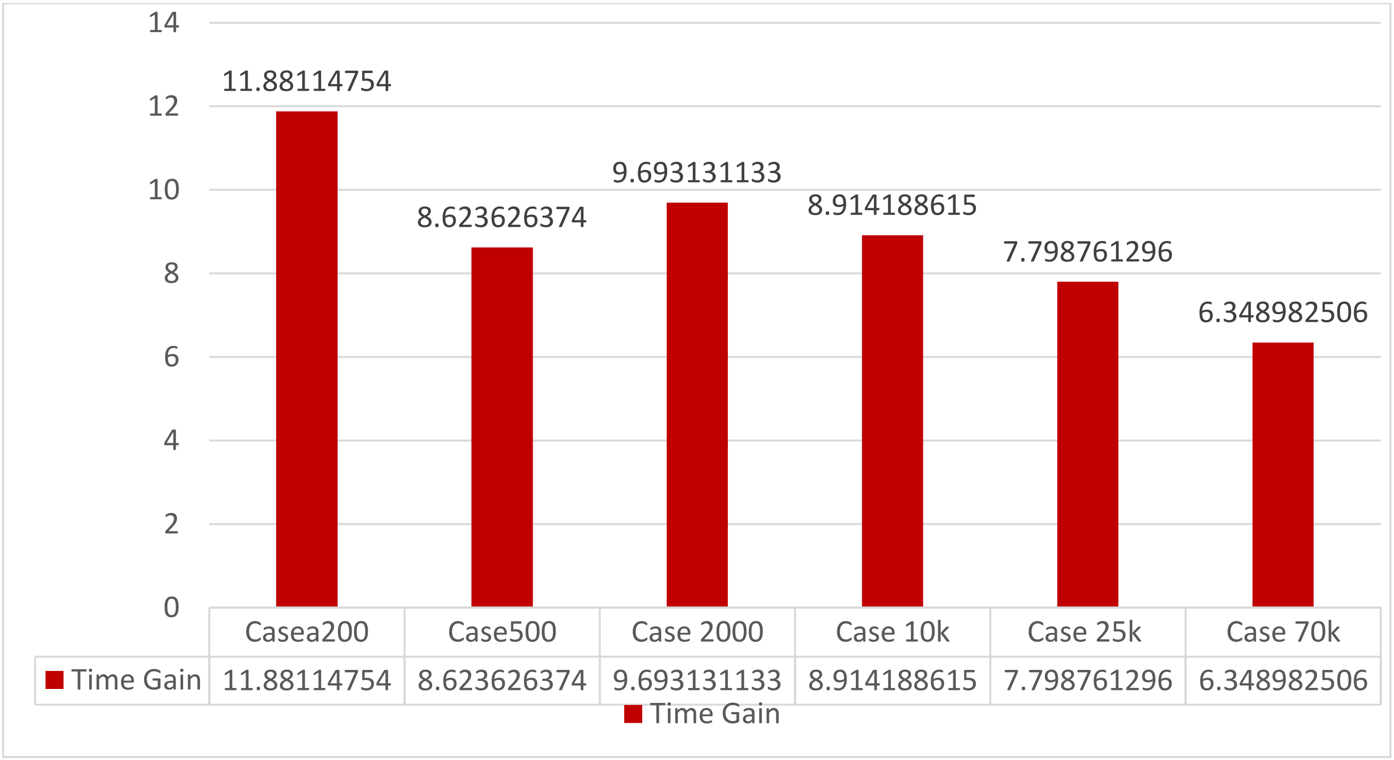
GPU parallelization is used in this case to ensure a further gain. Compute Unified Device Architecture (CUDA) is a parallel computing platform that enables programming on NVIDIA GPUs. Mainly, the execution workhorse in a CUDA-capable GPU is formed using an array of Streaming Multiprocessors (SMs). Each SM contains execution resources, such as streaming processors, double-precision units, special function units, and load/store units. Each subunit contains a register file, enabling rapid context-switching of threads. Threads are the fundamental building blocks of parallel programs. Indeed, multiple threads are grouped into blocks and assigned to SMs for execution. Therefore, GPU threads are utilized to factorize the executable columns and accelerate the factorization process.



	Casea200	Case500	Case 2000	Case 10k	Case 25k	Case 70k
🟥 Matpower	0.5798	0.6278	1.0866	3.1476	7.681	28.4536
🟧 KLU	0.0488	0.0728	0.1121	0.3531	0.9849	4.4816

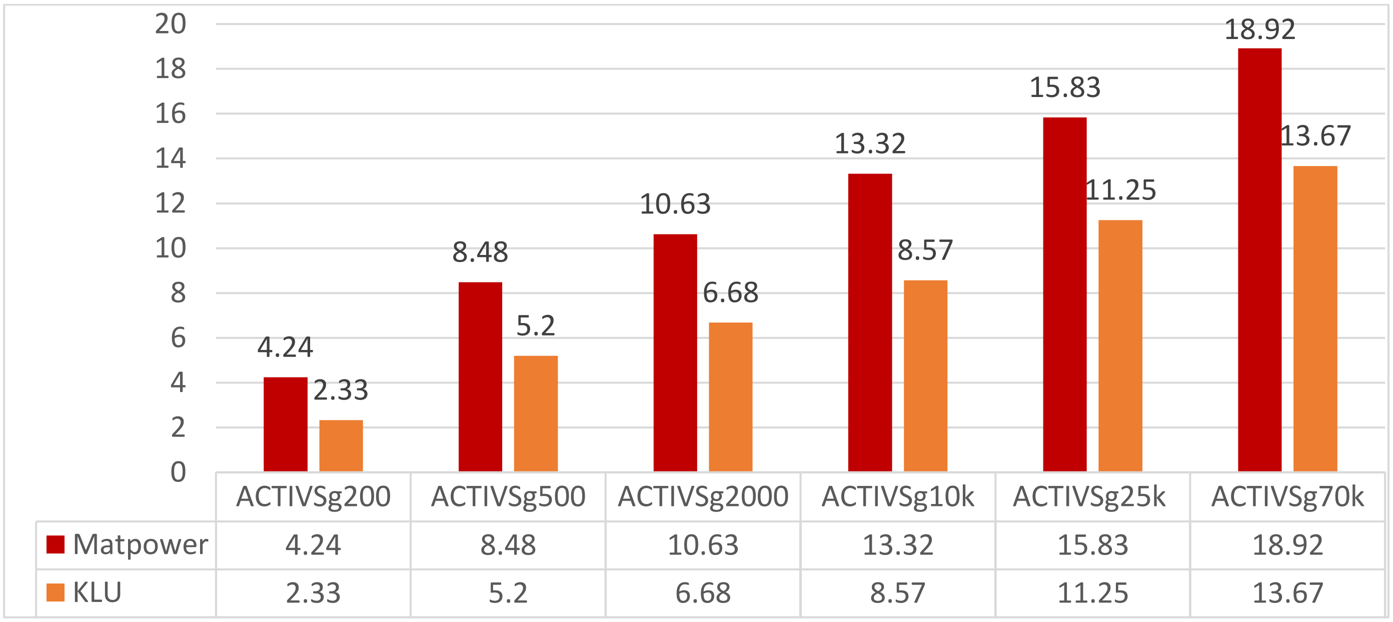
Fig 6.3: Average Computational time of each test case using MATPOWER and KLU in AC power flow

Fig 6.3 shows the average time needed for each test system using MATPOWER and KLU for the AC power system. Considering the number of buses in each case study, MATPOWER requires more computation time to solve AC PF problems. Comparing the obtained results in fig 6.4, it is revealed the ratio of average computation time for cases ACTIVSg200, ACTIVSg500, ACTIVSg2000, ACTIVSg10k, ACTIVSg25k, and ACTIVSg70k are 11.88, 9.99, 9.69, 8.91, 7.73, and 6.34, respectively. These show that the KLU solver is over ten times faster than MATPOWER in solving AC PF problems for smaller bus systems. As the number of buses increases, the time gain is slightly reduced. However, it is still 6 times faster than MATPOWR for a 70k bus system.



	Casea200	Case500	Case 2000	Case 10k	Case 25k	Case 70k
🟥 Time Gain	11.88114754	8.623626374	9.693131133	8.914188615	7.798761296	6.348982506

Fig 6.4: Time gain for each test case comparing MATPOWER with KLU for AC power flow



	ACTIVSg200	ACTIVSg500	ACTIVSg2000	ACTIVSg10k	ACTIVSg25k	ACTIVSg70k
🟥 Matpower	4.24	8.48	10.63	13.32	15.83	18.92
🟧 KLU	2.33	5.2	6.68	8.57	11.25	13.67

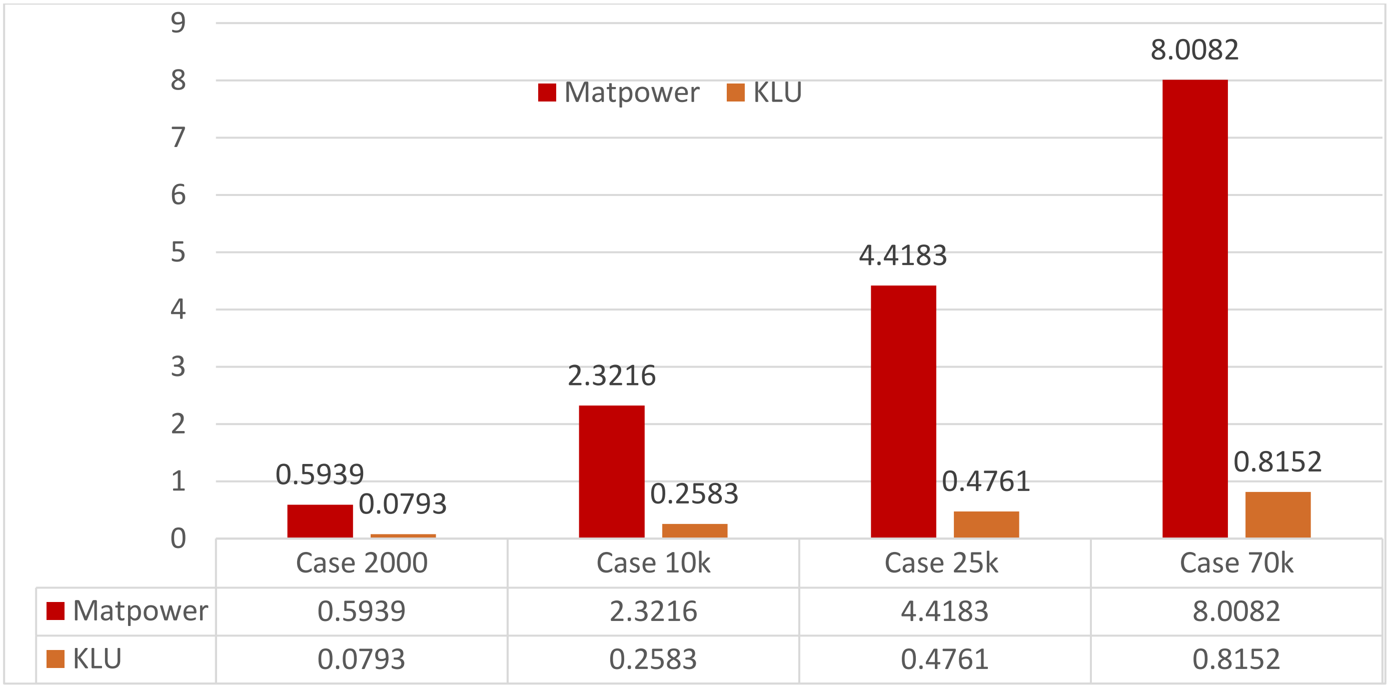
Fig 6.5: Average GPU memory usage by MATPOWER and KLU for AC system

A sparse matrix-based solver needs less memory space than the traditional dense matrix. Fig 6.5 shows the memory usage needed for each method in percentage. It is seen that MATPOWER uses over 1.35 times more GPU memory compared with the KLU solver to solve AC PF problems for all case studies.

The test cases used to compare the performance of DC power flow between MATPOWER and KLU are:

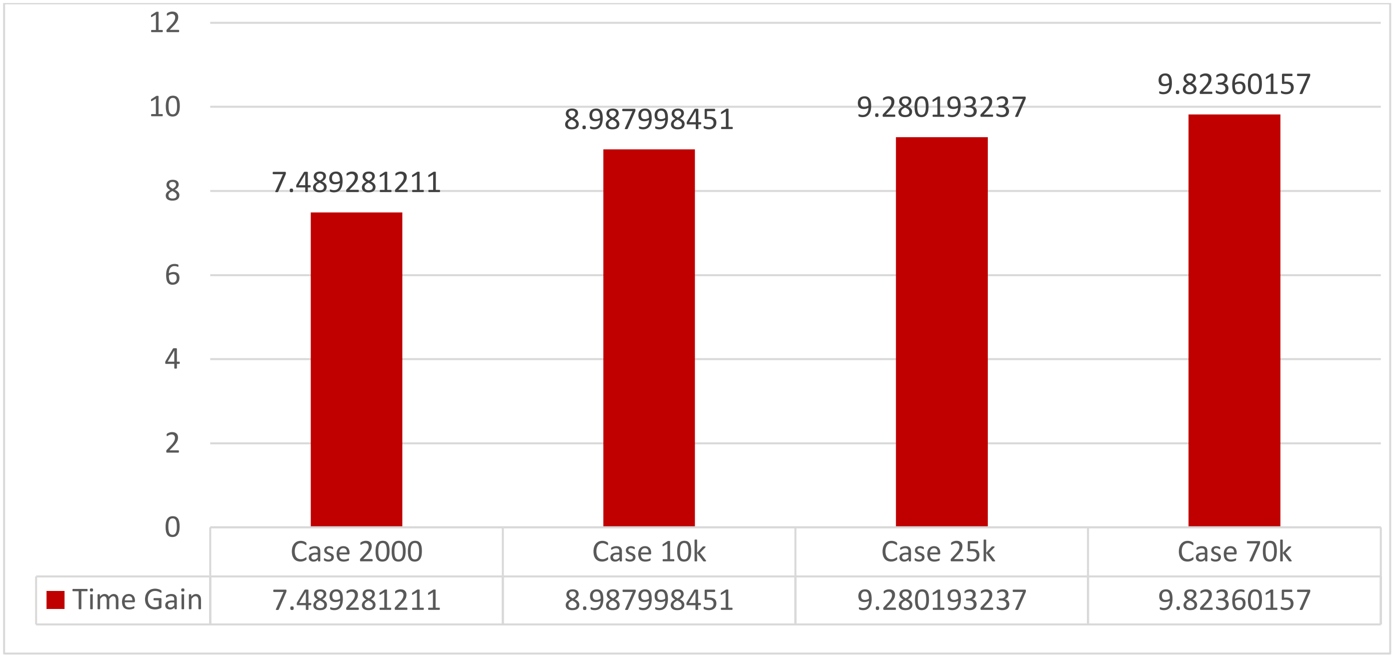
1. ACTIVSg2000 (2000-bus synthetic power grid geolocated in Texas, USA)
2. ACTIVSg10k (10,000-bus synthetic grid geolocated in the Western USA)
3. ACTIVSg25k (25,000-bus synthetic power grid geolocated in the Mid-Atlantic USA)
4. ACTIVSg70k (70,000-bus synthetic grid geolocated in the Eastern USA)

The test platform is the same as AC power flow, and GPU parallelization is also used for DC pf analysis.



	Case 2000	Case 10k	Case 25k	Case 70k
🟥 Matpower	0.5939	2.3216	4.4183	8.0082
🟧 KLU	0.0793	0.2583	0.4761	0.8152

Fig 6.6: Average Computational time of each test case using MATPOWER and KLU in DC power flow

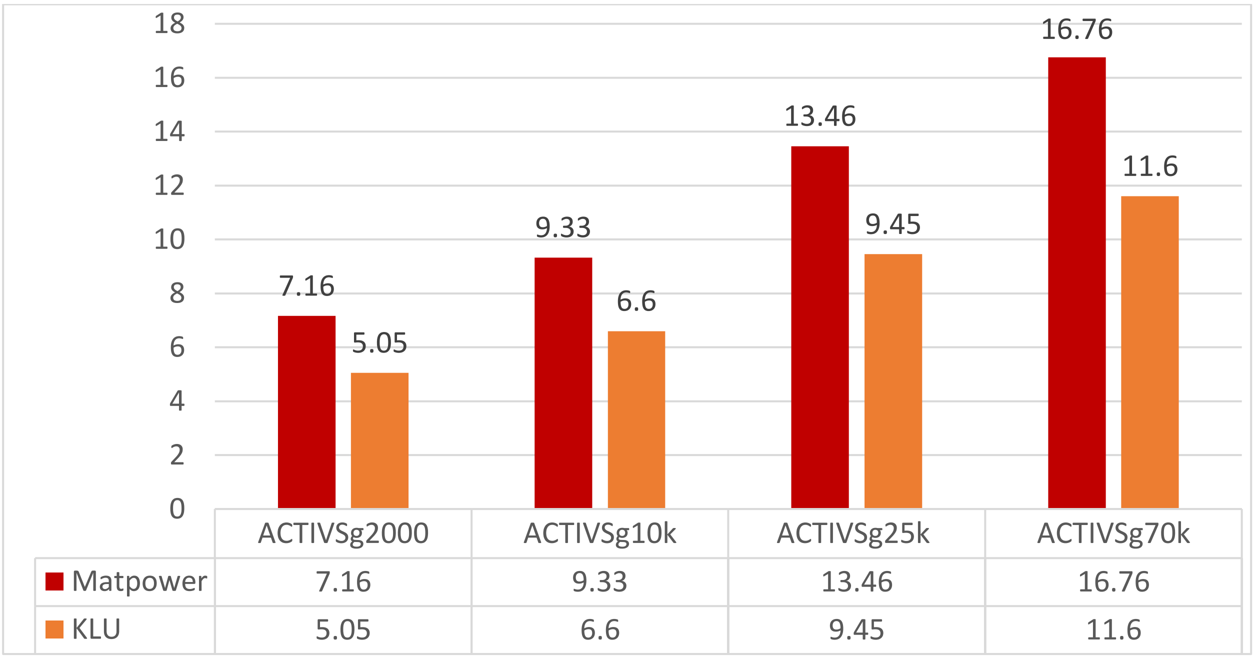


	Case 2000	Case 10k	Case 25k	Case 70k
🟥 Time Gain	7.489281211	8.987998451	9.280193237	9.82360157

Fig 6.7: Time gain for each test case comparing MATPOWER with KLU for DC power flow

Considering the number of buses in each case study, as seen from fig 6.6 MATPOWER requires more computation time to solve DC PF problems. Fig 6.7 gives the time gain with respect to MATPOWER for each of the test cases. It is revealed that the ratio of average computation time for cases ACTIVSg2000, ACTIVSg10k, ACTIVSg25k, and ACTIVSg70k are 7.4896, 8.9879, 9.2802, and 9.8236, respectively.

These show that the KLU solver is approximately 10x faster than MATPOWER in solving DC PF problems. In addition, as shown in fig. 6.8, MATPOWER uses approximately 1.4 more GPU memory than the KLU solver to solve DC PF problems for all case studies.



	ACTIVSg2000	ACTIVSg10k	ACTIVSg25k	ACTIVSg70k
🟥 Matpower	7.16	9.33	13.46	16.76
🟧 KLU	5.05	6.6	9.45	11.6

Fig 6.8: Average GPU memory usage by MATPOWER and KLU in DC power flow

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Power flow calculation is one of the crucial methods of power system analysis. It is needed for the smooth operation of the power system as well as power system planning and stability. Considering the high penetration of renewable energy sources, distributed energy sources, and electric vehicles nowadays, the power system is more prone to instability and complexity. High performance and fast power system computation algorithms and solvers are required to ensure timely control and efficient operation of power systems with rapid load and generation changes and configuration changes.

Power flow analysis is a numerical analysis to find the magnitude and angles of the bus voltages under a specific condition and thereby calculate the flow of power in the lines and transformers. This operating condition changes based on the load demand and generation capacity. With the known bus voltages, the active and reactive powers in each line or branch can be calculated. This process is highly complex, especially when dealing with a large number of bus systems which is typically the case. Including variable power generation components, different loads, and power electronics makes the analysis more complex.

Power flow analysis constitutes a non-linear set of equations to solve. The main computation time for solving a non-linear system is spent on the iterative process of solving the associated linear system resulting from the linearization of the equations. For traditional power flow solvers like Newton-Raphson and Gauss-Seidel, the majority of the computation time is hoarded by this iterative process. The computation time increases significantly with the increase in the system's size and complexity.

Chapter 2 discusses the recent development that has been done in the related field. It provides a literature review of the different approaches to solving the issue and their performance. Chapter 3 explains how power flow analysis works for both AC and DC power flow solutions. The mathematical work is shown in detail which provides a clearer understanding of the theory behind the analysis.

The paper proposes a fast-computing sparse matrix based solver for power flow analysis for real-time simulations. KLU is used as the sparse matrix solver which has proven to be highly efficient when solving electric circuit problems due to the unique sparsity characteristics of power system matrices and their amenability with the ordering techniques used. Chapter 4 details how sparse matrices work and the process of solving them. It also describes the KLU process in depth.

Chapter 5 is a quick summary of the things explained so far. It shows how to implement KLU into the power flow analysis process. Chapter 6 is where the additional contributions made as part of the thesis are explained in detail. When the circuit matrix value changes due to its elements, a higher computational gain is promised with refactorization. Its performance is enhanced with the use of GPU. It also shows better memory access efficiency. The computation results prove the hypothesis when tested with data from multiple real-life power system cases. Compared with different power system sizes, the proposed algorithm shows a speedup of around 10x than achieved in MATPOWER. Compared with the traditional Newton-Raphson method, it is more than 200 times faster. The result is even more promising for DC power flow solution showing an astounding 1000x gain for 10k test system.

7.2 Future Work

Future work will focus on analyzing the impact of the percentage of sparsity on the accuracy of the solver and its efficiency. For this thesis, the initial point has been taken from the operating state of a real power grid at a certain time. The changes in the initial assumptions for the voltage magnitude and angles may impact the convergence of the iterative solution. Another area of focus will be the development of an ordering mechanism, which will provide the option to modify the columns of the varying elements in the matrix and move them to the end. Since the enhancements proposed in the algorithm depend on the position of the changed elements in the original matrix, having a customized ordering algorithm to control the position of the time-varying elements will ensure a higher gain.

In addition, depth first search method is used for the first step of KLU, symbolic analysis. There are other algorithms in the graph theory which may present better results. One of the focus areas will be exploring other possible algorithms during the symbolic analysis phase to find the non-zero pattern of the triangular matrices.

References

[1] A. Moradzadeh, S. Zakeri, M. Shoaran, B. Mohammadi-Ivatloo and F. Mohammadi, "Short-term load forecasting of microgrid via hybrid support vector regression and long short-term memory algorithms," Sustainability, vol. 12, pp. 7076, 2020.

[2] F. Mohammadi, B. Mohammadi-Ivatloo, G. B. Gharehpetian, M. H. Ali, W. Wei, O. Erdinç and M. Shirkhani, "Robust Control Strategies for Microgrids: A Review," IEEE Systems Journal, pp. 1-12, 2021.

[3] F. Mohammadi, R. Bok and M. Hajian, "Real-Time Controller Hardware-in-the-Loop Testing of Power Converters," 2022.

[4] A. Abdollahi, A.A. Ghadimi, M.R. Miveh, F. Mohammadi and F. Jurado, "Optimal power flow incorporating FACTS devices and stochastic wind power generation using krill herd algorithm," Electronics, vol. 9, pp. 1043, 2020.

[5] Z. Wang, S. Wende-von Berg and M. Braun, "Fast parallel Newton–Raphson power flow solver for large number of system calculations with CPU and GPU," Sustainable Energy, Grids and Networks, vol. 27, pp. 100483, 2021.

[6] Y. Liu, N. Zhang, Y. Wang, J. Yang and C. Kang, "Data-driven power flow linearization: A regression approach," IEEE Transactions on Smart Grid, vol. 10, pp. 2569-2580, 2018.

[7] K.R. Mestav, J. Luengo-Rozas and L. Tong, "Bayesian state estimation for unobservable distribution systems via deep learning," IEEE Trans.Power Syst., vol. 34, pp. 4910-4920, 2019.

[8] Y. Zhou, F. He, N. Hou and Y. Qiu, "Parallel ant colony optimization on multi-core SIMD CPUs," Future Generation Comput.Syst., vol. 79, pp. 473-487, 2018.

[9] M. D'orto, S. Sjöblom, L.S. Chien, L. Axner and J. Gong, "Comparing Different Approaches for Solving Large Scale Power-Flow Problems With the Newton-Raphson Method," IEEE Access, vol. 9, pp. 56604-56615, 2021.

[10] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in 2010 Annual IEEE India Conference (INDICON), pp. 1-5, 2010.

[11] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li and J.W. Liu, "A supernodal approach to sparse partial pivoting," SIAM Journal on Matrix Analysis and Applications, vol. 20, pp. 720-755, 1999.

[12] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," Future Generation Comput.Syst., vol. 20, pp. 475-487, 2004.

[13] M. Christen, O. Schenk and H. Burkhart, "General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform," in First workshop on general purpose processing on graphics processing units, pp. 32, 2007.

[14] F.L. Alvarado, W.F. Tinney and M.K. Enns, "Sparsity in large-scale network computation," Advances in Electric Power and Energy Conversion System Dynamics and Control, vol. 41, pp. 207-272, 1991.

[15] A.A. El-Keib, H. Ding and D. Maratukulam, "A parallel load flow algorithm," Electr.Power Syst.Res., vol. 30, pp. 203-208, 1994.

[16] Y. Fukuyama, Y. Nakanishi and H. Chiang, "Parallel power flow calculation in electric distribution networks," in 1996 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 669-672, 1996.

[17] M. Amano, A.I. Zecevic and D.D. Siljak, "An improved block-parallel Newton method via epsilon decompositions for load-flow calculations," IEEE Trans.Power Syst., vol. 11, pp. 1519-1527, 1996.

[18] K. Lau, D.J. Tylavsky and A. Bose, "Coarse grain scheduling in parallel triangular factorization and solution of power system matrices," IEEE Trans.Power Syst., vol. 6, pp. 708-714, 1991.

[19] J.Q. Wu and A. Bose, "Parallel solution of large sparse matrix equations and parallel power flow," IEEE Trans.Power Syst., vol. 10, pp. 1343-1349, 1995.

[20] R.C. Green, L. Wang and M. Alam, "High performance computing for electric power systems: Applications and trends," in 2011 IEEE Power and Energy Society general meeting, pp. 1-8, 2011.

[21] F. Li and R.P. Broadwater, "Distributed algorithms with theoretic scalability analysis of radial and looped load flows for power distribution systems," Electr.Power Syst.Res., vol. 65, pp. 169-177, 2003.

[22] R. Baldick, B.H. Kim, C. Chase and Y. Luo, "A fast distributed implementation of optimal power flow," IEEE Trans.Power Syst., vol. 14, pp. 858-864, 1999.

[23] V.C. Ramesh, "On distributed computing for on-line power system applications," International Journal of Electrical Power & Energy Systems, vol. 18, pp. 527-533, 1996.

[24] D.M. Falcão, "High performance computing in power system applications," in International Conference on Vector and Parallel Processing, pp. 1-23, 1996.

[25] J. Tournier, V. Donde and Z. Li, "Potential of general purpose graphic processing unit for energy management system," in 2011 Sixth International Symposium on Parallel Computing in Electrical Engineering, pp. 50-55, 2011.

[26] C. Guo, B. Jiang, H. Yuan, Z. Yang, L. Wang and S. Ren, "Performance comparisons of parallel power flow solvers on GPU system," in 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 232-239, 2012.

[27] F. Mohammadi, G. Nazri and M. Saif, "An improved mixed AC/DC power flow algorithm in hybrid AC/DC grids with MT-HVDC systems," Applied Sciences, vol. 10, pp. 297, 2019.

[28] J.W. Demmel, "SuperLU users' guide," 1999.

[29] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li and J.W. Liu, "A supernodal approach to sparse partial pivoting," SIAM Journal on Matrix Analysis and Applications, vol. 20, pp. 720-755, 1999.

[30] X.S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," ACM Transactions on Mathematical Software (TOMS), vol. 31, pp. 302-325, 2005.

[31] T.A. Davis and E.P. Natarajan, "Algorithm 8xx: Klu, a direct sparse solver for circuit simulation problems," ACM Transactions on Mathematical Software, vol. 5, pp. 1-17, 2011.

[32] J.R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," SIAM Journal on Scientific and Statistical Computing, vol. 9, pp. 862-874, 1988.

[33] A. George and E. Ng, "An implementation of Gaussian elimination with partial pivoting for sparse systems," SIAM Journal on Scientific and Statistical Computing, vol. 6, pp. 390-409, 1985.

[34] I.S. Duff and J.K. Reid, "An implementation of Tarjan's algorithm for the block triangularization of a matrix," ACM Transactions on Mathematical Software (TOMS), vol. 4, pp. 137-147, 1978.

[35] D. Paré, G. Turmel, J.C. Soumagne, V.Q. Do, S. Casoria, M. Bissonnette, B. Marcoux and D. McNabb, "Validation tests of the hypersim digital real time simulator with a large AC-DC network," in Proc. Int. Conf. Power System Transients, pp. 577-582, 2003.

[36] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM Journal on Computing, vol. 1, pp. 146-160, 1972.

[37] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to algorithms, MIT press, 2022, .

[38] I.S. Duff, "Algorithm 575: Permutations for a zero-free diagonal [F1]," ACM Transactions on Mathematical Software (TOMS), vol. 7, pp. 387-390, 1981.

[39] I.S. Duff, "On algorithms for obtaining a maximum transversal," ACM Transactions on Mathematical Software (TOMS), vol. 7, pp. 315-330, 1981.

[40] P.R. Amestoy, T.A. Davis and I.S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," ACM Transactions on Mathematical Software (TOMS), vol. 30, pp. 381-388, 2004.

[41] P.R. Amestoy, T.A. Davis and I.S. Duff, "An approximate minimum degree ordering algorithm," SIAM Journal on Matrix Analysis and Applications, vol. 17, pp. 886-905, 1996.

[42] T.A. Davis, J.R. Gilbert, S.I. Larimore and E.G. Ng, "Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm," ACM Transactions on Mathematical Software (TOMS), vol. 30, pp. 377-380, 2004.