Increasing computational efficiency of Gradyents heat network solver

Optimizing the Newton-Raphson method, applied to thermal networks

by

F. E. Kiewiet de Jonge

to obtain the degree of Bachelors of Science

at the Delft University of Technology,

to be defended publicly on Friday April 26th, 2024 at 9:00 AM.

Thesis Committee:	Prof. dr. ir. C. Vuik (TU Delft), dr. E. Lorist (TU Delft), ir. N. Lam (Gradyent)
Project Duration:	February, 2024 - April, 2024
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

TU Delft

Contents

Abstract

District heating leverages centralised, high efficiency combined heat and power (CHP) systems. It uses waste heat to lower energy consumption and reduce greenhouse emissions. The system also supports renewable energy sources like geothermal and biomass, providing a sustainable heating alternative.

This report examines a nonlinear network of pressure and flow challenges. It focuses on enhancing the Newton-Raphson method and refining direct solving techniques for a single time-step. As networks grow in complexity, efficient and effective solutions become crucial. Various strategies to speed up the Newton-Raphson algorithm in Gradyent's heat network solver are discussed, where derivative calculations are straightforward. Both direct and iterative methods to improve the algorithm's steps are explored. The effectiveness of these enhancements is tested and evaluated across networks of different sizes.

1

Introduction to district heating

The first chapter of this report presents the district heating model and algorithm with its assumptions, used by Gradyent. Chapter 2 presents the theory behind the Newton-Raphson method. The mathematical background is explored along with the demonstration of its convergence. Multiple linear solving methods are evaluated in chapter 3. In chapter 4, the various networks that are tested are presented and analysed. Following up on that comes chapter 5, explaining the backbone of this research. chapter 6 delves into the results applied to multiple networks. Initial conditions and potential thresholds for reusing previous computations are explored there. Finally, chapter 8 will provide a summary of the conclusions and offer recommendations.

For additional context, refer to the appendix. All Python codes are included in appendix A. More information and graphics are provided in appendix B, to defend the story more thoroughly. Lastly, while this study primarily addresses the resolution of the linear system, a chapter evaluating several methods to improve Newton-Raphson method is included in appendix C.

1.1. Why district heating

District heating is a promising solution for providing efficient heating and cooling services to urban areas, via a network of insulated pipes. These pipes extend from a centralised generation point to end users. Currently, traditional systems relying on fossil fuels leave a significant carbon footprint, hindering progress towards sustainability and climate goals (Gradyent, 2024).

To adapt to the evolving landscape of renewable and intelligent energy, district heating networks must become more flexible. These networks need to handle variable flow rates, diverse supply temperatures, and bidirectional flows to accommodate new sources of distributed energy production. Achieving this transformation requires the use of precise and efficient thermo-hydraulic models. These models offer accurate simulations of flow rates and temperature transients, serving as valuable tools for designing, managing, and optimising thermal distribution networks.

As urban centres grow and environmental concerns become more pressing, the demand for efficient and sustainable infrastructure intensifies. District heating systems, known for their potential to significantly reduce urban carbon footprints and enhance energy efficiency, stand at the forefront of this transition. This research delves into advancing the design and optimisation of district heating systems through the development and validation of sophisticated thermo-hydraulic models. These models are intricately designed to handle the complexities of energy demands in modern urban settings, with a particular focus on integrating fluctuating renewable energy sources and improving system flexibility to manage variable flow rates and bidirectional flows.

Gradyent's Digital Twin platform hence empowers businesses to be proactive players in combating climate change, for a greener and more sustainable future. Through their real-time Digital Twin platform, Gradyent facilitates the creation of a dynamic digital replica of the entire heating grid, allowing

users to optimise operations, simulate future scenarios, enhance performance, lower CO2 emissions, achieve cost savings, and make informed, environmentally conscious business decisions.

1.2. District heating model

The Digital Twin (DT) serves as a virtual counterpart to the physical district heating system, capturing real-time data and interactions to enable scenario exploration, behavior prediction, and operational optimisation. A robust computational framework is employed to address intricate heat transfer and fluid flow problems, bridging the gap between the physical and digital. This is based on theory from Frederiksen and Werner (2013). Typically, the network structure is divided into three main layers:

1. Individual elements within the network. District heating network elements are pairs of interconnected nodes where dynamic processes occur.
2. Integration of these elements into a cohesive network, applying conservation laws and boundary conditions. State variables such as pressure p and flow rate Q at each node define the network's state vector, facilitating the modeling of interactions between nodes.
3. Resolution of the network constraint function using root-finding techniques. To then solve the system of equations governing the network, variables are organized into a vector S (Equation 1.1),

$$S = \begin{bmatrix} p_1 \\ \vdots \\ p_n \\ Q_1 \\ \vdots \\ Q_n \end{bmatrix}, \quad f(S) = 0 \tag{1.1}$$

initiating a large-scale system seeking a root-form solution $f(S) = 0$ using a variant of the Newton-Raphson method. Newton's method forms the core of the solution approach, aiming to achieve a consistent solution the flow rates and pressures within the network.

A significant challenge stems from the network's pressure losses, which depend nonlinearly on flow rates. The equations are computed within the Gradyent software framework, implemented using Python. In the context of the Newton-Raphson method, the rapid and accurate computation of derivatives, particularly the Jacobian matrix, is important. Gradyent employs a software structure that effectively builds up the Jacobian, so for this report no explicit attention is given to computing that matrix efficiently.

Furthermore, the model is based on several assumptions. The Newton-Raphson method assumes smoothness and continuity, assured by initial simplifications. The model assumes uniform material properties and ideal fluid behaviour, neglecting temperature variations, viscosity, compressibility, and external factors like weather. Additionally, it assumes uniformly distributed flow, ignoring network irregularities and leakage. While simplifying, these assumptions may impact model accuracy across operational scenarios. They focus on key variables, making analysis manageable while providing valuable insights.

This research aims to uncover any underlying sparsity within the network's structure, which could potentially lead to efficiency improvements. Future chapters will delve into pertinent numerical theories and suggest approaches to enhance computational efficiency.

2

Numerical theory

Now that the model that Gradyent makes use of is clear, the numerical solution method can be proposed. In order to find the solution of the system from chapter 1, the Newton-Raphson method is used. The chapter explores the Newton-Raphson method, shedding light on its iterative nature and its effectiveness in refining solutions through successive approximations. In section 2.1 some necessary numerical definitions and theorems are stated about convergence, after which the Newton Rapshon is explored in section 2.2. In section 2.3, the practical implementation is explained. By delving into the convergence behavior and error considerations associated with the application of the Newton-Raphson algorithm, the chapter provides valuable insights into the robustness and reliability of Gradyent's modeling approach in accurately capturing the dynamic behavior of district heating systems.

2.1. Numerical background

Numerical methods offer practical approximations to complex mathematical problems, especially when exact solutions are impractical or computationally burdensome. Balancing accuracy and efficiency is key, and crucial for practical applications. These various methods involve formulating the problem, discretizing it, selecting appropriate algorithms, implementing them in code, iterating for convergence, and validating the results. In order to understand the methods and techniques used, a number of definitions and theorems from Vuik et al. (2023) will be posed in this section (Vuik et al., 2023).

To approach the desired solution, achieving convergence is essential. Let's define mathematically what that entails. Each numerical method generates a sequence $\{p_n\} = p_0, p_1, p_2, ...$ which should converge to p, i.e., $\lim_{n\to\infty} p_n = p$.

Definition 1 (Convergence)**.** *If there exist positive constants λ and α satisfying*

$$\lim_{n\to\infty} \left| \frac{p - p_{n+1}}{p - p_n} \right|^{\alpha} = \lambda \tag{2.1}$$

, then $\{p_n\}$ converges to p with order α and asymptotic constant λ. (definition 4.2.1 from Vuik et al. (2023)

In general, a higher-order method converges faster than a lower-order method. The value of the asymptotic constant, λ, is less important. There are two important cases:

- $\alpha = 1$: the process is linearly convergent. In this case, λ is called the asymptotic convergence factor.
- $\alpha = 2$: the process is quadratically convergent.

In order to show convergence, the following theorems are used;

Theorem 2.1.1. *Suppose that a sequence* $\{p_n\} = p_0, p_1, p_2, ...$ *satisfies* $|p - p_n| \leq k|p - p_{n-1}|$ *for* $n = 1, 2, ...,$ *where* $0 \leq k < 1$. *Then* $\lim_{n\to\infty} p_n = p$*: the sequence is convergent. (theorem 4.2.1 from Vuik et al. (2023))*

Proof. By induction,

$$\lim_{n\to\infty} |p - p_n| \leq \lim_{n\to\infty} k^n |p - p_0| = 0,$$

because $k < 1$. Hence p_n converges to p.

□

Theorem 2.1.2 (Fixed point convergence)**.** *Suppose* $g \in C[a, b]$, $g(x) \in [a, b]$ *for* $x \in [a, b]$, *and* $|g'(x)| \leq k < 1$ *for* $x \in [a, b]$. *Then the fixed-point iteration converges to* p *for each value* $p_0 \in [a, b]$. *(theorem 4.4.2 from Vuik et al. (2023))*

Proof. Under the provided conditions, g has a unique fixed point p. From the mean-value theorem it follows that

$$|p - p_n| = |g(p) - g(p_{n-1})| = |g'(\xi)||p - p_{n-1}| \leq k|p - p_{n-1}|,$$

where ξ is between p and p_{n-1}. From Theorem 2.1.1 it follows that p_n converges to p. □

2.2. Newton-Raphson

To discover solutions, numerous algorithms have been devised. Among them, the Newton-Raphson method stands out as particularly renowned and widely adopted. Known for its simplicity of implementation and ease of use, this method boasts robustness, making it a preferred choice for Gradyent in pursuit of system solutions. The Newton-Raphson method stands as one of the most well-established numerical techniques for resolving nonlinear equations $f(x) = 0$, with the point x termed as a zero or root of the function f (Chill, 2008).

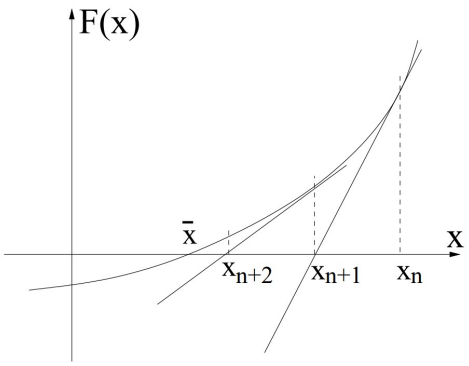


Figure 2.1: The Newton-Raphson algorithm

In figure 2.1, the iterative nature of the method is portrayed. It shows how each successive approximation is closer to the actual root than the previous one. The convergence is typically quite rapid, especially when the initial guess is close to the actual root.

The process starts with an initial guess X_n which is an estimate of the root of the function $F(x)$. This is represented by the furthest right vertical dotted line dropping down to the x-axis. The value of the function at this initial guess $F(X_n)$ is calculated, which corresponds to the height from the x-axis up to the function curve. A tangent line is then drawn at the point where the initial guess intersects the function. The tangent line represents the derivative of the function at X_n, which is $F'(X_n)$. The point where this tangent crosses the x-axis is taken as the next approximation to the root, denoted X_{n+1}. This is shown as the second vertical dotted line from the right.

This process is then repeated using X_{n+1} as the new guess. A new tangent line is drawn at the point where X_{n+1} intersects the function, and where this line crosses the x-axis becomes X_{n+2}, the next approximation. As the iterations continue, the values of X_n converge to the actual root $\bar{X}$, which is the point on the x-axis directly below the root of the function on the curve.

2.2.1. The algorithm

This section states the mathematical algorithm that Newton-Raphson is based on, explained using a Taylor polynomial. Suppose $f \in C^2[a,b]$. Let $\bar{x} \in [a,b]$ be an approximation of the root p such that $f'(\bar{x}) \neq 0$, and suppose that $|p - \bar{x}|$ is small. Consider the first-degree Taylor polynomial about $\bar{x}$:

$$f(x) = f(\bar{x}) + (x - \bar{x})f'(\bar{x}) + \frac{(x-\bar{x})^2}{2}f''(\xi(x)), \tag{2.2}$$

in which $\xi(x)$ is between x and $\bar{x}$. Using that $f(p) = 0$, equation 2.2 yields

$$0 = f(\bar{x}) + (p - \bar{x})f'(\bar{x}) + \frac{(p-\bar{x})^2}{2}f''(\xi(x)). \tag{2.3}$$

Because $|p - \bar{x}|$ is small, $(p - \bar{x})^2$ can be neglected, such that

$$0 \approx f(\bar{x}) + (p - \bar{x})f'(\bar{x}). \tag{2.4}$$

Note that the right-hand side is the formula for the tangent in $(\bar{x}, f(\bar{x}))$. Solving for p yields

$$p \approx \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}. \tag{2.5}$$

In the next theorem, the Newton-Raphson iteration is considered as a fixed-point method.

Theorem 2.2.1. *Let $f \in C^2[a,b]$. If $p \in [a,b]$ such that $f(p) = 0$ and $f'(p) \neq 0$, then there exists a $\delta > 0$, such that the Newton-Raphson method generates a sequence $\{p_n\}$ converging to p for each $p_0 \in [p-\delta, p+\delta]$.(theorem 4.5.1 from Vuik et al. (2023))*

Proof. In the proof, the Newton-Raphson method is considered as a fixed-point method $p_n = g(p_{n-1})$ with $g(x) = x - \frac{f(x)}{f'(x)}$. In order to use Theorem 2.1.2, it is necessary to show that there exists a $\delta > 0$, such that g satisfies the conditions of the theorem for $x \in [p-\delta, p+\delta]$.

1. Continuity of g: g is well-defined and continuous for all x in $[p-\delta_1, p+\delta_1]$. Since $f'(p) \neq 0$ and f' is continuous, there exists a $\delta_1 > 0$ such that $f'(x) \neq 0$ for all $x \in [p-\delta_1, p+\delta_1]$. Therefore,

2. Derivative of g: The derivative of g is

$$g'(x) = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2}. \tag{2.6}$$

   Since $f \in C^2[a,b]$, it follows that $g'(x)$ is continuous: $g(x) \in C^1[p-\delta_1, p+\delta_1]$. Note that $g'(p) = 0$ because $f(p) = 0$. Since g' is continuous, there exists a $\delta < \delta_1$ such that $|g'(x)| \leq k < 1$ for all $x \in [p-\delta, p+\delta]$.

3. Domain and range of g: Finally, it is to show that $g(x) \in [p-\delta, p+\delta]$ if $x \in [p-\delta, p+\delta]$. Using the mean-value theorem (for a continuous function on a closed interval, there exists at least one point where the instantaneous rate of change equals the average rate of change over that interval (Ross, 2013),

$$|g(p) - g(x)| = |g'(\xi)||p - x| \quad \text{for some } \xi \text{ between } x \text{ and } p. \tag{2.7}$$

   Since $x \in [p-\delta, p+\delta]$, it follows that $|p - x| < \delta$ and $|g'(\xi)| < 1$, and hence, $|g(p) - g(x)| < |p - x| < \delta$.

Using Theorem 2.1.2, the sequence $\{p_n\}$ converges to p for each $p_0 \in [p-\delta, p+\delta]$. This proves the theorem. □

2.2.2. Convergence (quadratic)

The convergence behavior of the Newton-Raphson method can be computed by making use of the following observation:

$$0 = f(p) = f(p_n) + (p - p_n)f'(p_n) + \frac{(p - p_n)^2}{2}f''(\xi_n) \quad \text{for } \xi_n \text{ between } p_n \text{ and } p. \tag{2.8}$$

The Newton-Raphson method is defined such that

$$0 = f(p_n) + (p_{n+1} - p_n)f'(p_n). \tag{2.9}$$

Subtracting expression 2.9 from 2.8 yields

$$(p - p_{n+1})f'(p_n) + \frac{(p - p_n)^2}{2}f''(\xi_n) = 0, \tag{2.10}$$

such that

$$\left|\frac{p - p_{n+1}}{p - p_n}\right|^2 = \frac{f''(\xi_n)}{2f'(p_n)}. \tag{2.11}$$

From equation 2.1 it follows that the Newton-Raphson method converges quadratically, with $\alpha = 2$ and

$$\lambda = \lim_{n\to\infty} \frac{f''(\xi_n)}{2f'(p_n)} = \frac{f''(p)}{2f'(p)}. \tag{2.12}$$

2.2.3. Newton-Raphson method in higher-dimensional systems

In this section, the theory will be extended to systems of nonlinear equations, which can be written either in fixed-point form

$$\begin{cases} \mathbf{g}(\mathbf{p}) = \mathbf{p}, \\ \mathbf{g}(\mathbf{p}) = \begin{bmatrix} g_1(p_1, \dots, p_m) \\ \vdots \\ g_m(p_1, \dots, p_m) \end{bmatrix}, \end{cases} \tag{2.13}$$

or in the general form

$$\begin{cases} \mathbf{f}(\mathbf{p}) = \mathbf{0}, \\ \mathbf{f}(\mathbf{p}) = \begin{bmatrix} f_1(p_1, \dots, p_m) \\ \vdots \\ f_m(p_1, \dots, p_m) \end{bmatrix}. \end{cases} \tag{2.14}$$

These systems can be represented in matrix-vector notation as $\mathbf{g}(\mathbf{p}) = \mathbf{p}$ and $\mathbf{f}(\mathbf{p}) = \mathbf{0}$, respectively, where

$$\mathbf{p} = \begin{bmatrix} p_1 \\ \vdots \\ p_m \end{bmatrix}, \quad \mathbf{g}(\mathbf{p}) = \begin{bmatrix} g_1(p_1, \dots, p_m) \\ \vdots \\ g_m(p_1, \dots, p_m) \end{bmatrix}, \quad \mathbf{f}(\mathbf{p}) = \begin{bmatrix} f_1(p_1, \dots, p_m) \\ \vdots \\ f_m(p_1, \dots, p_m) \end{bmatrix}. \tag{2.15}$$

Similar to the scalar case, an initial estimate for the solution is used to construct a sequence of successive approximations, $\{\mathbf{p}^{(n)}\}$, of the solution until a desired tolerance is reached. A possible stopping criterion is $\|\mathbf{p}^{(n)} - \mathbf{p}^{(n-1)}\| < \varepsilon$, where $\|\mathbf{x}\| = \sqrt{x_1^2 + \dots + x_m^2}$ denotes the Euclidean norm.

As in the scalar case of the Newton-Raphson method, the successive approximation is found by linearizing function $\mathbf{f}$ about iterate $\mathbf{p}^{(n-1)}$:

$$\begin{aligned} f_1(p) &\approx f_1(p^{(n-1)}) + \frac{\partial f_1}{\partial p_1}(p^{(n-1)})(p_1 - p_1^{(n-1)}) + \dots + \frac{\partial f_1}{\partial p_m}(p^{(n-1)})(p_m - p_m^{(n-1)}), \\ &\vdots \\ f_m(p) &\approx f_m(p^{(n-1)}) + \frac{\partial f_m}{\partial p_1}(p^{(n-1)})(p_1 - p_1^{(n-1)}) + \dots + \frac{\partial f_m}{\partial p_m}(p^{(n-1)})(p_m - p_m^{(n-1)}). \end{aligned} \tag{2.16}$$

Defining the Jacobian matrix of $\mathbf{f}(x)$ by

$$\mathbf{J}(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \dots & \frac{\partial f_1}{\partial x_m}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \dots & \frac{\partial f_m}{\partial x_m}(x) \end{bmatrix}, \tag{2.17}$$

the linearization can be written in the more compact form

$$\mathbf{f}(\mathbf{p}) \approx \mathbf{f}(\mathbf{p}^{(n-1)}) + \mathbf{J}(\mathbf{p}^{(n-1)})(\mathbf{p} - \mathbf{p}^{(n-1)}). \tag{2.18}$$

The next iterate, $\mathbf{p}^{(n)}$, is obtained by setting the linearization equal to zero:

$$\mathbf{f}(\mathbf{p}^{(n-1)}) + \mathbf{J}(\mathbf{p}^{(n-1)})(\mathbf{p}^{(n)} - \mathbf{p}^{(n-1)}) = 0, \tag{2.19}$$

which can be rewritten as

$$\mathbf{J}(\mathbf{p}^{(n-1)})s^{(n)} = -\mathbf{f}(\mathbf{p}^{(n-1)}), \tag{2.20}$$

where $s^{(n)} = \mathbf{p}^{(n)} - \mathbf{p}^{(n-1)}$. The new approximation equals $\mathbf{p}^{(n)} = \mathbf{p}^{(n-1)} + s^{(n)}$.

A fast way to compute $\mathbf{p}^{(n)}$ uses equation (2.18):

$$\mathbf{p}^{(n)} = \mathbf{p}^{(n-1)} - \mathbf{J}^{-1}(\mathbf{p}^{(n-1)})\mathbf{f}(\mathbf{p}^{(n-1)}). \tag{2.21}$$

Consider a system of n equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{x}$ is an m-dimensional vector and $\mathbf{F}(\mathbf{x})$ is a vector-valued function. The iteration formula becomes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n) \cdot \mathbf{F}(\mathbf{x}_n), \tag{2.22}$$

where $\mathbf{J}$ is the Jacobian matrix of $\mathbf{F}$, and $\mathbf{J}^{-1}$ is its inverse.

2.2.4. Order of the error

The order of the error in the Newton-Raphson method defines how quickly the method converges to the root of a given equation. The Newton-Raphson method exhibits quadratic convergence, which means that, with each iteration, the number of correct digits in the approximation roughly doubles. Specifically, the error $|e_{n+1}|$ at the $(n+1)$-th iteration is approximately proportional to the square of the error at the n-th iteration, represented as $|e_n|^2$. Mathematically, the quadratic convergence is expressed as $|e_{n+1}| \approx C|e_n|^2$, where C is a constant. This property makes the Newton-Raphson method highly efficient in rapidly approaching the true root. However, it is important to note that this quadratic convergence is asymptotic, and in practical scenarios, factors such as numerical precision and the behavior of the function can influence the actual convergence rate (Dennis and Schnabel, 1983).

Numerical errors can broadly be categorized into truncation errors and round-off errors, each affecting the convergence and stability of numerical solutions in distinct ways. Truncation errors arise when an infinite process is approximated by a finite one. This type of error is inherent in methods where continuous data is discretized or when infinite series are approximated by finite sums. For instance, the Newton-Raphson method uses the derivative's finite difference approximation, which can introduce truncation errors. The impact of these errors is especially pronounced when assessing the convergence behavior of the method:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n), \tag{2.23}$$

where $f'(x_n)$ is the derivative of f at x_n, and the higher order terms of the Taylor series are omitted. Truncation errors can lead to a significant deviation from the actual solution, particularly when the step size in the discretization is not sufficiently small.

Round-off errors occur due to the finite precision with which computers represent numbers. Operations involving very small or very large numerical values can worsen these errors, leading to significant inaccuracies in calculations. In iterative methods like the Newton-Raphson method, round-off errors can accumulate with each iteration, potentially affecting the stability and convergence of the solution. The iterative nature of the Newton-Raphson method often amplifies these errors, as shown in the iterative update formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{2.24}$$

where each computation of x_{n+1} involves finite-precision arithmetic that can introduce round-off errors. These errors are particularly problematic when the derivatives are small, which can cause the solution to diverge or stall. The convergence of a numerical method is critically dependent on managing both truncation and round-off errors. The balance between these errors often determines the choice of step size or discretization level. A smaller step size might reduce truncation errors but increase round-off errors and computational cost. Therefore, careful error analysis and step size optimization are essential for ensuring the robustness and accuracy of numerical solutions:

$$\text{Error}_{total} = \text{Error}_{truncation} + \text{Error}_{round-off}. \tag{2.25}$$

An awareness of these errors and their implications is fundamental to the development and validation of numerical models. The proper handling of these errors ensures that the solutions are not only accurate but also reliable, providing a solid foundation for decision-making based on the model outcomes.

2.2.5. Interval of convergence

The ability of Newton-Raphson to converge rapidly in higher-dimensional systems accelerates the solution-finding process, surpassing alternative methods. In systems behaving predictably, the quadratic convergence ensures a significant increase in accuracy with each iteration, promoting efficient local convergence. Moreover, when the initial guess closely approximates the actual solution, the method exhibits high precision, making it suitable for precise calculations.

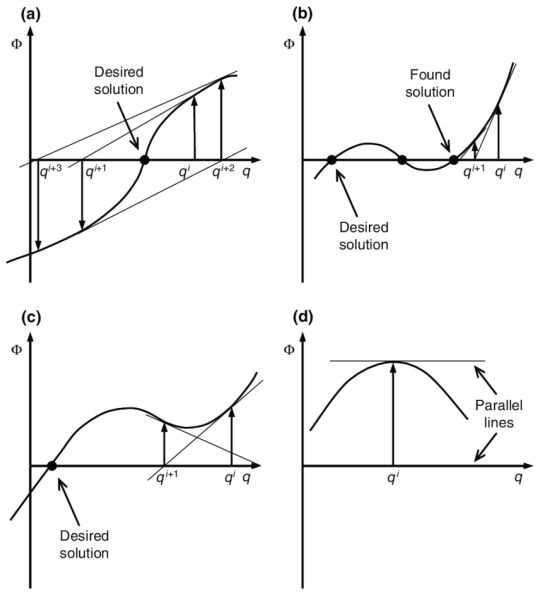


Figure 2.2: Newton-Raphson failing to converge (Flores, Paulo et al., 2016)

In the context of district heating networks, situations of non-convergence can arise. This method requires a good initial guess and relies on the system's Jacobian matrix being non-singular and well-conditioned throughout the iteration process. Non-convergence typically occurs under several scenarios. There could be multiple solution to the system, shown in figure 2.2 (b). Also, if the starting point

is too far from the true root, the method may diverge or oscillate indefinitely without converging to a solution, seen in (c) of figure 2.2. Additionally, an extreme value could imply a root, when it is not a root, seen in (d). Moreover, the method requires the inversion of the Jacobian matrix at each iteration. If the Jacobian becomes singular or nearly singular, the inverse becomes very large or undefined, leading to large or unbounded updates in the solution vector, which prevents convergence.

Also, the method assumes that the function whose roots are being sought is smooth and continuous. Discontinuities, sharp corners, or discontinuities in the function or its derivatives can lead to situations where the method fails to converge. Generally speaking for the dynamic interactions between multiple inputs and outputs of district heating networks, governed by complex boundary conditions and variable external conditions (such as flow rates), can lead to a highly non-linear system. These non-linearities, if not properly managed or approximated, can result in a Jacobian matrix with poor properties for convergence. Operational constraints like fixed temperatures or pressures at certain nodes introduce additional challenges, as these can make the system of equations stiff, another factor contributing to potential non-convergence (Frederiksen and Werner, 2013).

2.2.6. Newton-Raphson compared to other methods

In the field of numerical analysis, the Newton-Raphson method stands out for its fast convergence and reliability, especially when the initial estimate is close to the actual solution. It typically outperforms methods like the Bisection or Secant methods, which converge linearly (Dennis and Schnabel, 1983). However, its reliance on derivative calculations can be a drawback when derivatives are computationally expensive or hard to obtain analytically.

In contrast, methods like the Secant method and Broyden's method, a quasi-Newton method that iteratively approximates the Jacobian matrix, do not need explicit derivatives, potentially saving computation time in systems where derivative evaluations are costly. Because the Newton-Raphson method relies on derivatives, it is more sensitive to the initial estimate and prone to divergence if the function behaves poorly, such as near inflection points or where derivatives approach zero. Thus, while the Newton-Raphson method offers significant benefits in terms of speed and accuracy under favorable conditions, its use should be carefully weighed against these potential drawbacks, especially in systems with complex or poorly understood derivative structures.

2.3. Newton-Raphson's algorithm in practice

Understanding the theoretical basis of Newton's method, its practical implementation can now be evaluated. The version of Newton's method utilized is fairly straightforward: first calculate the Jacobian J_f at the current S, and solve (directly) for the step that will give a linear solution at that point. Then repeat. The stopping criterion is when the max-norm of the right hand side is below a certain threshold (e.g. 10^{-7}). There has been experimentation with calculating smart preconditioners and using an iterative linear solver (especially over multiple time-steps in a timeseries), but nothing stable yet. Gradyent does make use of the time-series continuity by using the previous time-step as an initial condition for the next.

In order to compute the calculations, the first two terms of the Taylor series are utilized. The algorithm Gradyent uses consists of the following steps:

1. **Initialization:** This step typically involves setting initial values for variables, which is computationally trivial. Negligible time is spent on initialization. No specific optimization is needed for this step. Optimizing the initial guess is crucial for iterative equation-solving methods. It sets the starting point for iteration, affecting convergence and efficiency. Refining it iteratively can enhance optimization, leading to faster convergence, reduced computational load, and accurate solutions. In this study, a constant initial guess is used to avoid external influences.

2. **Iteration:** The iteration itself involves simple arithmetic operations and function evaluations, which are computationally straightforward. The time consumption depends on the number of iterations required for convergence and the complexity of the functions being evaluated. Employing efficient algorithms for function evaluation and arithmetic operations can help reduce time. Additionally, parallelisation techniques can be utilized to perform multiple iterations concurrently, speeding up the process. These distribute computational tasks across multiple processors or

cores to improve efficiency and speed up the execution of programs (Virtanen et al., 2020). Those techniques however exceed the scope of this report.

(a) **Jacobian and Function evaluation:** Evaluating the Jacobian matrix and the function involves computing partial derivatives and evaluating functions at given points, which can be computationally intensive for complex functions. Gradyent makes use of a software structure that ensures derivative evaluations are performed effectively, Nevertheless, this remains relatively hard to compute compared to the rest of the algorithm. Therefore, it is important to minimize the amount of Jacobian and function evaluations as much as possible.

(b) **Newton-Raphson Update:** Computing the update involves matrix inversion and matrix-vector multiplication can be computationally expensive. The time consumed in this step depends primarily on the size of the Jacobian matrix and the efficiency of the matrix inversion algorithm, but is significantly larger than all other steps. Implementing efficient matrix inversion algorithms, such as LU decomposition or iterative methods like Gauss-Seidel, can significantly reduce computational time (Trefenthen et al., 2022).

(c) **Residual Calculation:** Calculating the residual involves computing the norm of vectors, which is computationally straightforward. The time consumed in this step is typically negligible compared to other steps. The max norm, denoted as $\|x\|_\infty$, measures the maximum absolute value of the elements in a vector x. It is defined as $\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|$. In contrast, the Euclidean norm, denoted as $\|x\|_2$, measures the length of a vector in Euclidean space. It is defined as $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$.

The choice between max norm and Euclidean norm depends on the characteristics of the problem at hand. When dealing with systems of equations and iterative methods like Newton-Raphson, the max norm is often preferred due to its ability to provide a balanced measure of error across all elements of the Jacobian matrix.

Unlike the Euclidean norm, which may be dominated by the largest entries of the Jacobian matrix, the max norm considers the largest absolute value among all entries. This ensures that no component is neglected during the iteration process, particularly when the entries of the Jacobian matrix vary significantly in magnitude.

(d) **Convergence Check:** Comparing the residual to a specified tolerance level is computationally trivial. Gradyent makes use of a tolerance level of 10^{-7}, so this report will do so as well. This step incurs negligible time consumption, so no specific optimization is needed for this step.

3. **Update:** Updating the current guess involves simple arithmetic operations and assignment operations. Time consumption in this step is typically low compared to others. No specific optimization is needed for this step.

Furthermore, Newton-Raphson is robust but scales inefficiently. Improving computational efficiency becomes critical, particularly when the algorithm must be executed frequently or in real-time or on larger networks (Trefenthen et al., 2022).

Besides computational considerations, the Newton-Raphson method faces limitations in specific scenarios within one-dimensional problems (Vuik et al., 2023). Firstly, the algorithm fails when the initial guess (x_0) aligns with the inflection point of the function, indicated by $f''(x_0) = 0$. Secondly, it becomes ineffective when the initial guess (x_0) or any iterative value of the function reaches a horizontal extension, represented by $f'(x) = 0$. Lastly, if the initial guess (x_0) lies between a local maximum or minimum of the function, the algorithm faces challenges. Similar limitations in convergence and effectiveness persist in a multi-dimensional system. Consequently, strategies to mitigate these limitations remain pertinent in both one-dimensional and multi-dimensional applications of the Newton-Raphson method.

In the upcoming chapters, rapid techniques for addressing this root-finding problem will be introduced and extensively analyzed.

3

Methods for improving the linear solve

This chapter focuses on the solving linear systems within district heating network simulations, essential for accurate and efficient model predictions.

Direct solvers aim to solve the linear system within the Newton-Raphson. They typically involve decomposing the matrix into more manageable forms (e.g., LU, Cholesky, see section 3.1) from which the solution can be directly derived. Sparse direct solvers are a subtype of direct solvers, optimised for matrices where the majority of elements are zeros (sparse matrices). They exploit the sparsity to reduce computational cost and memory usage. Techniques include efficient storage formats and specialised algorithms for matrix decomposition that avoid operations on zero elements.

Unlike direct solvers, iterative methods approach the solution gradually by improving an initial guess based on a predefined procedure. These methods are valuable when the matrix is large and sparse, where direct methods become computationally expensive. Sparse iterative solvers are specifically designed to handle sparse matrices efficiently. They often require less memory and are faster for very large systems compared to their dense counterparts. They typically use preconditioners to improve convergence rates.

Dense solvers are used when the matrix is dense, i.e., most of the matrix elements are non-zero. They can be either direct or iterative but are not optimised for sparsity. Their use is limited to smaller matrices or those where the dense structure significantly simplifies the solution process. Understanding these distinctions is crucial for selecting the appropriate solver based on the matrix characteristics and the computational resources available. Each type has its own set of advantages and limitations, affecting their applicability to different types of problems.

Linear solving techniques are fundamental in iterative solutions of nonlinear equations. Efficiently solving these linear systems directly impacts computational load and solution stability. Section 3.1 proposes methods of decomposing the matrix in an efficient way in section 3.1, for example an LU decomposition. Subsequently, section 3.2 examines iterative methods, followed up by the Conjugate Gradient method, GMRES, and Bi-CGSTAB, crucial for large-scale district heating networks where direct methods may be impractical. These Krylov Subspace Methods approximate solutions iteratively, making them effective for sparse systems. The chapter concludes by providing an overview of linear solving techniques in section 3.3.

3.1. Direct methods for the linear solve

Most obvious in order to decrease the computational time and effort of solving a linear system are the direct methods. In this section the LU- and Cholesky decomposition are portrayed, as these methods are the cornerstones of sparse matrix solvers. They are extracted from Trefenthen et al. (2022).

One crucial concept in numerical linear algebra is the concept of sparsity, commonly found in matrix structures. It refers to the abundance of zero elements within a matrix. These matrices, characterised by their abundance of zero entries, offer advantages in terms of storage efficiency and computational complexity. Hence, considerable acceleration in direct solving methods for systems of linear equations

can be achieved. Most of the subsequent methods are, to some extent, founded on this sparsity.

3.1.1. LU Decomposition

LU decomposition is a method in numerical analysis to break down a matrix into the product of two other matrices, a lower triangular matrix (L) and an upper triangular matrix (U), such that $A = LU$. Consider a square matrix A of size $n \times n$. The objective of LU decomposition is to express A as:

$$A = LU$$

where:

- L is an $n \times n$ lower triangular matrix with ones on the diagonal.
- U is an $n \times n$ upper triangular matrix.

The procedure for LU decomposition proceeds as follows:

1. Initialization: Start with the first element of A, a_{11}, and assume an L and U such that:

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \tag{3.1}$$

and

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \tag{3.2}$$

2. First Column of L and First Row of U: Determine the first column of L and the first row of U using the first column and row of A.
3. Forward Elimination: Use the first column of A to eliminate all entries below a_{11}.
4. Continuing the Process: Apply the same process to the submatrix A' obtained by deleting the first row and column of A.
5. Iterating: Repeat the process for each submatrix A'', A''', etc., working your way along the diagonals of the matrix.
6. Continue this process until L and U have been fully populated.

Once A is decomposed into L and U, solving a linear system $Ax = b$ becomes a two-step process:

1. Solve $Ly = b$ for y using forward substitution.
2. Solve $Ux = y$ for x using backward substitution.

In practise, partial pivoting is often used to improve numerical stability. This involves interchanging rows of A to make sure that the largest element in each column of A gets placed on the diagonal of U. The LU decomposition converts a complex problem of matrix inversion or solution of linear systems into a much simpler one, exploiting the triangular structure of the matrices involved and allowing for efficient and numerically stable computations.

3.1.2. Cholesky Decomposition

Cholesky decomposition is a method to decompose a symmetric positive definite matrix A into the product of a lower triangular matrix L and its conjugate transpose, L^T, such that $A = LL^T$. The process is as follows;

1. Initialization: Start with the first element of A, a_{11}, and assume a lower triangular matrix L such that:

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \tag{3.3}$$

2. Diagonal Elements of L: Compute the diagonal elements of L using the square root of the corresponding diagonal elements of A:

$$l_{ii} = \sqrt{a_{ii}} \tag{3.4}$$

3. Off-diagonal Elements of L: Compute the off-diagonal elements of L using the formula:

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk}}{l_{ii}} \tag{3.5}$$

4. Completion: Continue this process until all elements of L have been computed.

Once A is decomposed into L, solving a linear system $Ax = b$ becomes a two-step process:

1. Solve $Ly = b$ for y using forward substitution.
2. Solve $L^T x = y$ for x using backward substitution.

Cholesky decomposition is more efficient than LU decomposition for solving systems of linear equations, especially when dealing with symmetric positive definite matrices. It provides a numerically stable and efficient method for solving linear least squares problems. If A is positive definite, the Cholesky decomposition is unique.

3.2. Iterative methods for linear system solving

Now that some direct methods are proposed and building blocks for iterative methods are given, this following section dives into iterative methods. See Axelsson (1996) for more details (Axelsson, 1996). Multigrid methods are presented first, after which preconditioning follows. Finally, Krylov subspace methods such as the Conjugate Gradient method and the Generalised Minimum Residual Method are proposed.

3.2.1. Multigrid methods

Multigrid methods are particularly effective for problems with grids or meshes, where the underlying structure can be exploited to accelerate convergence. Multigrid methods offer a hierarchical approach to solving linear systems, achieving efficiency by operating on multiple scales of the problem. They effectively capture both the fine and rough level details, accelerating the convergence rate significantly. By smoothing out errors at various levels, multigrid methods can often outperform traditional solvers, especially for problems with hierarchical or multiscale structures.

Despite their notable advantages, multigrid methods come with certain drawbacks. Their implementation can be complex, requiring substantial computational resources and memory. Moreover, challenges arise when dealing with nonsymmetric or indefinite matrices, which are central to this research. Hence, multigrid methods are not pursued as the primary focus of this study. Preconditioning however is an important building block for handling nonsymmetric matrices, and is covered next.

3.2.2. Preconditioning

Preconditioning is a technique employed to improve the convergence properties of iterative solvers, both for nonlinear systems like those addressed by the Newton-Raphson method and for linear systems solved by iterative linear solvers. The following section is based on information from Chen (2005). Preconditioning is applied to this linear system to improve the performance and convergence of the linear solver used at this step. It transforms the system into a form that simpler to solve. Hence, preconditioning contributes to a more efficient convergence by optimising the linear problem solved during each iteration.

It improves the condition number of the matrix, which is a measure of how well-suited a matrix is for numerical computations. A better-conditioned matrix leads to faster convergence of the iterative solver.

Diagonal preconditioning

Diagonal preconditioning is a basic preconditioning technique. It involves scaling each row of the coefficient matrix by a scalar, typically chosen as the reciprocal of the diagonal element in the corresponding row. This operation transforms the original linear system into a more well-conditioned form, thereby enhancing the convergence properties of iterative solvers. The process of diagonal preconditioning can be summarized as follows:

1. Scaling by Diagonal Elements: Given a linear system represented by the matrix A and vector b, the diagonal preconditioner matrix M is formed by taking the reciprocal of the diagonal elements of A. Thus, $M_{ii} = 1/A_{ii}$, where i represents the row and column index.
2. Preconditioned System: The preconditioned system is represented by $M^{-1}Ax = M^{-1}b$, where M is the diagonal preconditioner matrix.
3. Solution Process: Instead of solving the original linear system $Ax = b$, we solve the preconditioned system $M^{-1}Ax = M^{-1}b$ using an iterative solver such as Conjugate Gradient (CG) or Generalised Minimal Residual (GMRES).
4. Back Transformation: Once the solution x is obtained for the preconditioned system, it can be transformed back to the original solution x' of the original system using $x' = Mx$.

Diagonal preconditioning is computationally inexpensive as it involves only forming the diagonal of the coefficient matrix and scaling it. However, its effectiveness may vary depending on the problem characteristics and matrix properties. Several specific variants of diagonal preconditioning exist, including Jacobi preconditioning. Block Jacobi preconditioning extends the idea of Jacobi preconditioning to block matrices, where diagonal blocks are inverted separately to form the preconditioner.

ILU preconditioning

The Incomplete LU (ILU) factorization is another widely used technique in numerical linear algebra for solving large sparse linear systems. It involves decomposing a matrix into lower and upper triangular matrices, where certain elements are omitted or approximated to reduce fill-in and storage requirements while preserving the sparsity pattern of the original matrix.

ILU upholds the sparsity pattern of the original matrix, thus enabling efficient storage and computation for large sparse systems. Furthermore, ILU's capacity to diminish fill-in contributes to decreased computational expenses when solving linear systems, in contrast to direct methods such as LU decomposition. Additionally, ILU's flexibility is evident through its ability to tailor the level of fill-in and precision, rendering it adaptable to diverse problem properties and computational environments.
However, ILU has its limitations. The approximation errors that arise from incomplete factorization can affect solution accuracy compared to exact LU decomposition. Additionally, the effectiveness of ILU depends on choosing an appropriate preconditioner and understanding the specific characteristics of the problem in iterative methods. Finally, the complexity of implementing ILU effectively, particularly for systems with irregular sparsity patterns, presents a challenge. In incomplete Cholesky preconditioning, the preconditioner matrix is formed using an incomplete Cholesky factorization method, which is more sophisticated than simple diagonal scaling.

Alternative preconditioners
There exit many more preconditioning techniques. For instance, the Schur Complement Preconditioners operate on the Schur complement matrix of the original problem, which is often smaller and better conditioned than the original matrix A. Examples include the FETI (Finite Element Tearing and Interconnecting) preconditioner and the BDDC (Balancing Domain Decomposition by Constraints) preconditioner.

Alternatively, Domain and Range Decomposition Preconditioners divide the problem into several subdomains and solve each subproblem separately, exchanging information between the subdomains to improve the solution. Examples include the Schwarz methods and the Additive Schwarz preconditioner. The matrix A is divided into blocks, then a preconditioner is applied to each block. This method is effective for problems with a natural block structure.

Preconditioners are a valuable tool for solving linear problems involving nonsymmetric sparse matrices. However, they come with some trade-offs. While convergence rates are enhanced, they require extra computations for their construction and application. Moreover, the effectiveness is closely tied to the complexity of their implementations.

3.2.3. Krylov Subspace methods

Krylov subspace methods, such as the Conjugate Gradient (CG) Method and the Generalised Minimum Residual Method (GMRES), can efficiently handle the nonsymmetry and indefiniteness that may occur in district heating system matrices (Axelsson, 1996). They consist of constructing a sequence of orthogonal vectors within the Krylov subspace that converges towards the solution. These methods can provide faster convergence compared to traditional iterative methods, especially for large-scale systems where direct factorization is impractical. Implementation of Krylov subspace methods requires mathematical understanding and careful selection of parameters, such as restart values and stopping criteria, to ensure numerical stability and convergence. Once these parameters are properly configured, the methods prove highly beneficial for solving linear systems.

Generalised Minimum Residual Method (GMRES) GMRES operates within a Krylov subspace, denoted as $K_n(A, r_0)$, solving the linear system $Ax = b$. r_0 denotes the initial residual vector $r_0 = b - Ax_0$, with x_0 being an initial estimate for the solution. GMRES orchestrates the construction of an orthonormal basis for the Krylov subspace via the Arnoldi iteration. This process creates a succession of n orthogonal vectors $q_1, q_2, \dots, q_n$ and an upper Hessenberg matrix H_n of dimensions $n \times n$,(a square matrix where all elements below the first subdiagonal are zero). The number n is the amount of edges and nodes added.

1. *Initialization*: Commence with an initial estimate x_0 and compute the initial residual $r_0 = b - Ax_0$.
2. *Arnoldi Iteration*: Generate $q_1, q_2, \dots, q_n$ and H_n via the Arnoldi iteration. This is an iterative method used to approximate eigenvalues and eigenvectors of large sparse matrices. It constructs an orthogonal basis for the Krylov subspace iteratively, providing an approximation to the dominant eigenpairs of the matrix.
3. *Minimization of Residual*: Determine the x within the subspace $K_n(A, r_0)$ that minimises the residual $||b - Ax||_2$ through the least squares method, akin to solving a small $n \times n$ linear least squares problem.
4. *Update of Solution*: Update the solution x utilising the minimiser identified in the preceding step.
5. *Iteration*: Recur the process until convergence criteria are satisfied or a maximum iteration limit is attained.

GMRES achieves convergence when the residual $||b - Ax||_2$ decreases adequately or when it reaches the predefined tolerance threshold. Convergence depends on factors such as the condition of the matrix A and the choice of the initial estimate. The computational burden of GMRES arises from the Arnoldi iteration, which requires matrix-vector multiplications with A and orthogonalisations.

Although each iteration adds computational overhead, GMRES often converges more quickly than direct methods for large, sparse matrices. Unlike direct methods, which often require extensive matrix factorization and storage, GMRES iteratively constructs a solution that minimizes the residual error. This iterative approach is advantageous for large sparse matrices because it avoids the computational overhead associated with dense factorizations. Additionally, GMRES can exploit the sparsity of the matrix, making it more efficient for large systems where direct methods may become impractical or even infeasible due to memory constraints.

Bi-Conjugate Gradient Method (BiCG)
The Bi-Conjugate Gradient Method (BiCG) extends the Conjugate Gradient method to handle non-symmetric linear systems, crucial for applications such as district heating systems. Like the Generalised Minimum Residual Method (GMRES), BiCG constructs solutions within a Krylov subspace, but it uniquely utilizes two such subspaces: $K_n(A, r_0)$ for the matrix A and $K_n(A^T, r_0^T)$ for its transpose A^T. This approach allows BiCG to manage the challenges of non-symmetry more effectively by minimizing the residuals $\|b - Ax\|_2$ and $\|b - A^T x\|_2$ simultaneously, enhancing both stability and convergence. The method iterates until convergence criteria are met, much like GMRES, but with the added complexity of maintaining two orthogonal projection processes.

3.3. Conclusion on linear solve improving methods

This chapter has investigated a range of advanced methods for enhancing the efficiency of linear solve steps in numerical solvers utilised in district heating network simulations, emphasizing a strategic balance between computational efficiency and solution accuracy. Direct solving methods, like LU and Cholesky decomposition, offer robust solutions by decomposing matrices into convenient forms for easier computations. Iterative methods are examined next.

This chapter has explored various methods for improving the efficiency and accuracy of linear system solving within district heating network simulations. Direct methods such as LU decomposition and Cholesky decomposition provide robust solutions by decomposing matrices into convenient forms for easier computations. However, their applicability may be limited by computational resources and the sparsity pattern of the matrix. In contrast, iterative methods offer flexibility and efficiency for large, sparse systems. Preconditioning techniques enhance the convergence properties of iterative solvers by transforming the original linear system into a more well-conditioned form. Krylov subspace methods such as GMRES and BiCG efficiently handle nonsymmetric matrices.

3.3.1. Linear solving techniques compared

Each linear solve technique discussed in this chapter offers distinct advantages for district heating network simulations, yet they also come with limitations that must be carefully managed to optimise performance. The LU Decomposition offers a reliable solution path for systems prioritizing stability and precision, particularly suitable for smaller matrices. There also exist various alternatives tailored for symmetric and positive definite matrices such as the Cholesky Decomposition, but these are not useful for this report.

The LU Decomposition faces challenges as system size increases, leading to significant computational and memory requirements, potentially limiting its applicability in large-scale settings due to the $O(n^3)$ complexity (Trefenthen et al., 2022). Additionally, this decomposition is less adaptable to changes in the system matrix.

Multigrid methods are extremely efficient at reducing all frequencies of the error and provide faster convergence than traditional iterative methods, making them ideal for multi-dimensional and multi-scale problems found in fluid dynamics and complex simulations. However, nonsymmetric matrices are not handled as well.

Preconditioning has become a fundamental method for enhancing the efficiency of iterative approaches. Techniques like Diagonal Preconditioning, Incomplete LU (ILU), and Algebraic Multigrid (AMG) aim to optimise the condition number of the system matrix, resulting in faster and more reliable

convergence. However, the effectiveness of these techniques can vary significantly depending on the properties of the matrix.

Other iterative methods, such as the Krylov subspace techniques, provide flexibility and efficiency, particularly suited for large-scale problems where direct methods are impractical. These are also more memory efficient than direct methods.

3.3.2. Future research in linear solving methods

While the methods discussed significantly enhance solver capabilities, they come with challenges as mentioned. Iterative methods, particularly multigrid and Krylov subspace techniques, demand complex implementation and robust preconditioning to effectively manage the intricacies of large-scale simulations. The success of iterative methods heavily relies on the effective application of preconditioners, which must be carefully matched to specific matrix characteristics to avoid performance degradation.

As computational demands increase with larger and more complex network models, optimising the linear solve step becomes even more crucial. It is evident that while improving the iterative algorithm can improve performance, significant gains come from focusing on optimising this critical step. Techniques such as preconditioning and certain decomposition methods can drastically enhance solver efficiency and robustness.

Considering the constraints observed, future research directions may include a combined solving methods that integrate the strengths of both direct and iterative approaches. Potentially even hybrid preconditioning approaches that combine multiple preconditioning strategies could be investigated to harness their collective strengths. This approach promises not only to enhance solver efficiency and scalability, but also to ensure the robustness necessary for modeling complex networks effectively. By understanding these strengths and limitations, one can more effectively choose and apply the appropriate linear solve techniques to meet the needs of district heating network simulations.

4

The networks

This chapter aims to offer a thorough analysis of structure of the networks. Section 4.1 presents and assesses the structure of all Jacobian matrices. Following up on that, section 4.2 elaborates on the eigenvalues and their mathematical implications. Section 4.3 delves into the condition numbers of the different networks. By synthesising these properties, a deliberate decision will be made in section 4.4, regarding the most promising method.

4.1. Structure of the networks

In the networks, nodes represent pivotal points where flow characteristics change, while edges symbolize the physical pipes connecting these nodes, enabling the transfer of heat and fluids. Due to their nature, edges can only link to two nodes, imposing a limit on their number. However, nodes are not subject to such constraints, allowing for potentially unlimited expansion (Frederiksen and Werner, 2013).

Through the analysis of networks with varying dimensions, insights are sought into the behaviour across different scales. The specific sizes are displayed in table 4.1. This facilitates the evaluation of the method's scalability, computational efficiency and effectiveness across a range of network sizes.

	S_0	S_1	S_2	M_1	M_2	L_1	L_2	XL_1
Nodes	34	81	105	474	524	1112	839	2190
Edges	37	82	105	506	654	1153	948	2554

Table 4.1: Number of nodes and edges

Figures 4.1, 4.3, 4.4 and 4.5 illustrate the Jacobian matrices. These matrices serve as fundamental descriptors of the network topology, necessary to understand the structural and connectivity characteristics.

4.1.1. Small networks S_0, S_1

The Jacobians of the small networks are presented in figure 4.1. To improve the visualisation of these plots, it is observed that many values approach zero, particularly in the lower-left region. To address this, a threshold close to zero (specifically, 10^{-2} yielded optimal results) is established. Subsequently, all values below this threshold are scaled by the reciprocal of the threshold. This approach effectively enhances the visibility of all data points, ensuring a more accurate representation of the plotted values. The plot of the Jacobian matrices S_0 and S_1 show a rather sparse structure. The majority of the matrix entries are zero or near zero and the non-zero entries appear to form a diagonal band from the top left to the bottom right of the matrix and another diagonal in the lower left. This pattern suggests that the system equations are mostly local, with each equation depending on a subset of all variables.

The bandwidth of the matrix — the width of the band around the diagonal where non-zero entries are located — has implications for the efficiency of solving linear systems. A smaller bandwidth typically

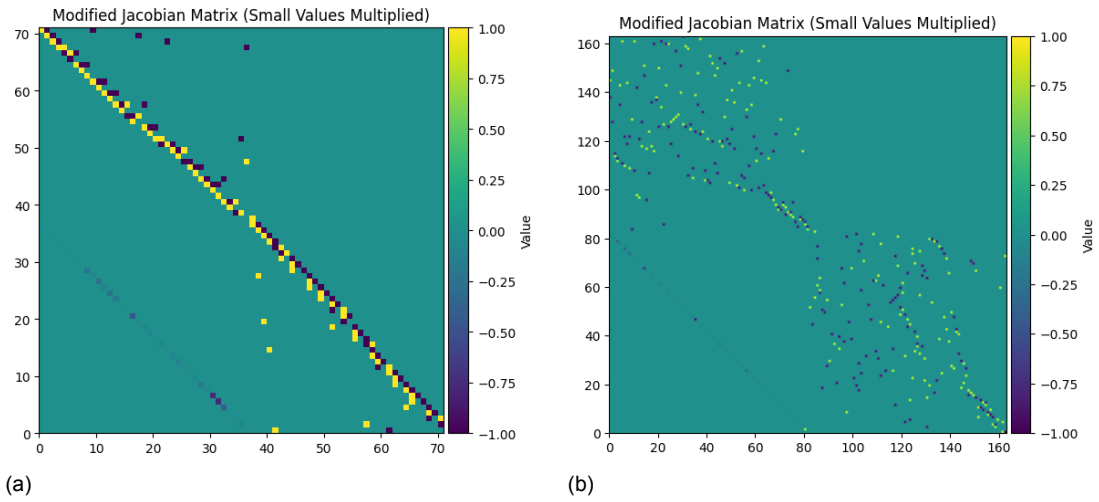


Figure 4.1: Jacobians of small networks S_0 and S_1.

leads to less computational effort in matrix factorization steps of direct solvers or in the multiplication steps of iterative solvers. From the plot, it is not possible to determine if the matrix is diagonally dominant, which would be the case if the absolute values of the diagonal entries are larger than the sums of the absolute values of the non-diagonal entries in each row. Diagonal dominance would imply that the system is well-conditioned and that iterative methods are likely to converge (Trefenthen et al., 2022).

Furthermore, the given Jacobian matrix can be thought of as being composed of four submatrices, which is denoted as A, B, C and D. This structure could well be of use for increasing computational speed. The sizes are determined by the number of edges and nodes. In this example there are 37 edges and 34 nodes. The amount of edges will always be greater than the number of nodes. Therefore, the upper left submatrix will always be of equal size as the lower right or larger.

Figure 4.2 displays the subdivision of the Jacobian. Each of these submatrices corresponds to different derivatives of the system functions with respect to the system variables:

- Submatrix A (Upper Left): This submatrix is associated with the derivatives of the flow equations with respect to flow variables. It is square in shape and corresponds to the interactions between edges in the network. The size of this block is determined by the number of edges in the network.
- Submatrix B (Upper Right): This submatrix represents the derivatives of the flow equations with respect to the pressure variables. This part of the Jacobian is almost filled with zeroes only. There is however always one point in this submatrix, since solving the system requires one set pressure value. That implies that the flow rates can never be independent of the pressures at the nodes.
- Submatrix C (Lower Left): This part captures the derivatives of the pressure equations with respect to flow variables. It often includes values calculated using Ohm's law for flow through the network's edges, similar to the current-voltage relationship in electrical circuits. This submatrix is supposably always diagonal, representing the direct relationship between pressure drop and flow in individual network elements.
- Submatrix D (Lower Right): This submatrix contains the derivatives of the pressure equations with respect to pressure variables. It's typically sparse, reflecting the relationship between pressures at different nodes. The non-zero entries indicate how the pressure at one node is affected by changes in pressure at another.

The structure of the Jacobian matrix mirrors the network's complexity in practise. Diagonal dominance in submatrices A and D suggests that local parameters significantly affect nearby variables. Although sparser, submatrices B and C illustrate how different variables across the network are interrelated. When a pressure is fixed at a node, it adds non-zero entries to submatrix B, ensuring system

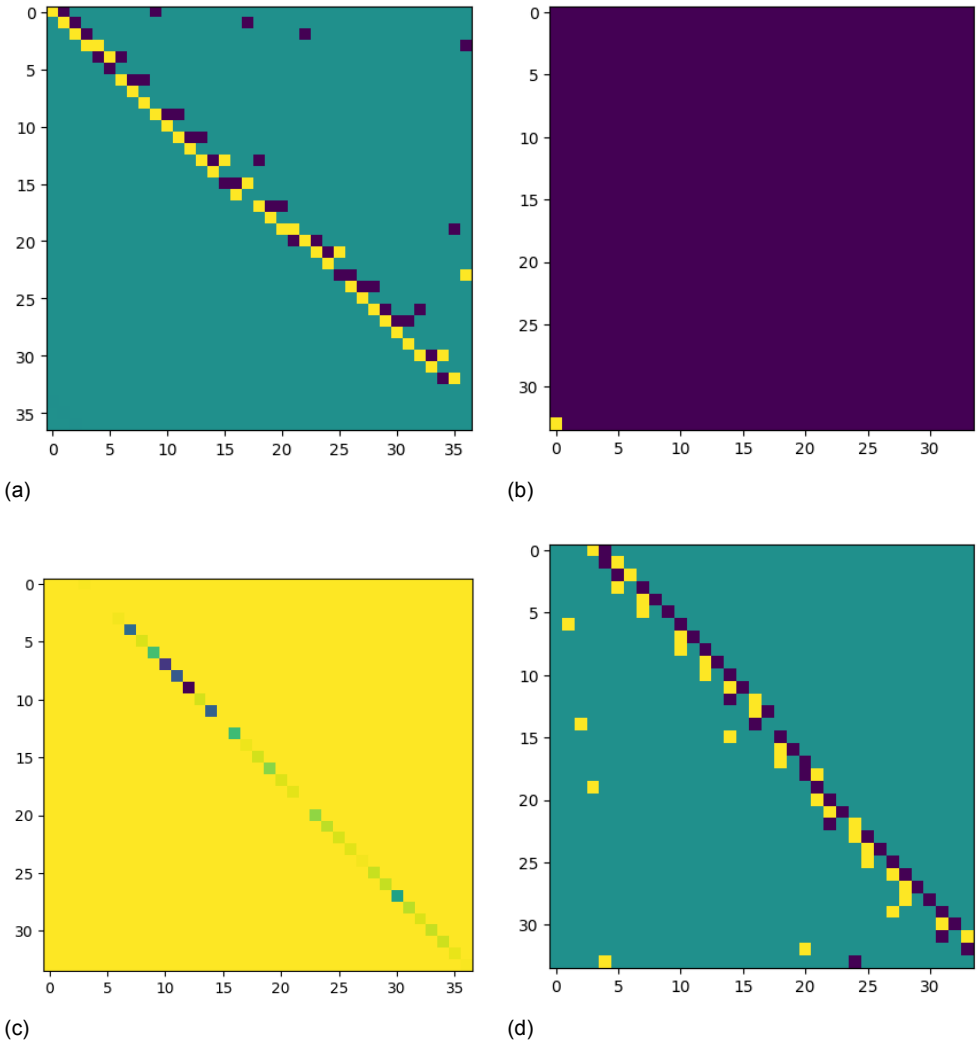


Figure 4.2: Block decomposition of the Jacobian of S_0.

determinacy. Similarly, entries in submatrix C show how edge values influence pressure equations. This breakdown helps understand system behaviour and makes it easier to apply numerical methods, particularly with sparse matrix techniques or parallel computation, improving solution efficiency.

Given that the upper right block contains essential data points necessary for determining pressure values, none of the matrices within the system can be considered empty. Nevertheless, an interesting avenue for further investigation lies in comparing the outcomes obtained when neglecting the values in the upper right matrix. One could explore solving the upper left matrix with the given vector and subsequently integrating this solution into the lower two submatrices. Such an analysis, however, falls beyond the current scope of this thesis.

4.1.2. Medium sized network M_1

In the case of M_1, the structure of the network does not display clearly when plotted (see appendix B). While some degree of sparsity is evident, However, for a more detailed understanding, a deeper examination of the underlying problem is necessary. Clipping is a useful method for improving visualisation by restricting value ranges within plots. This helps uncover the inherent structure of matrices. By setting lower and upper bounds, clipping focuses attention on the relevant value range, reducing the impact of extreme outliers and making the matrix structure clearer. This technique limits extreme values within specified intervals, such as $[-1, 1]$ or $[-10^{-7}, 10^{-7}]$, as shown in Figure 4.3.

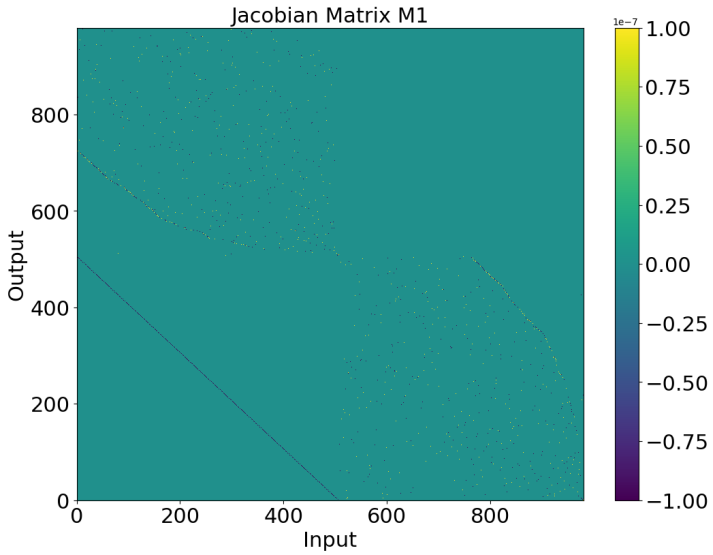


Figure 4.3: Jacobian of medium sized network M_1

Notably, while clipping is useful for the visualisation, the actual computations rely on unclipped, raw data.

4.1.3. Large networks L_1 and XL_1

The provided large network is clipped similarly in figure 4.4 and the extra large in figure 4.5. The Jacobian matrix L_1's sparse and banded structure suggests that computational efficiency is achievable with appropriate numerical techniques tailored to exploit these properties.

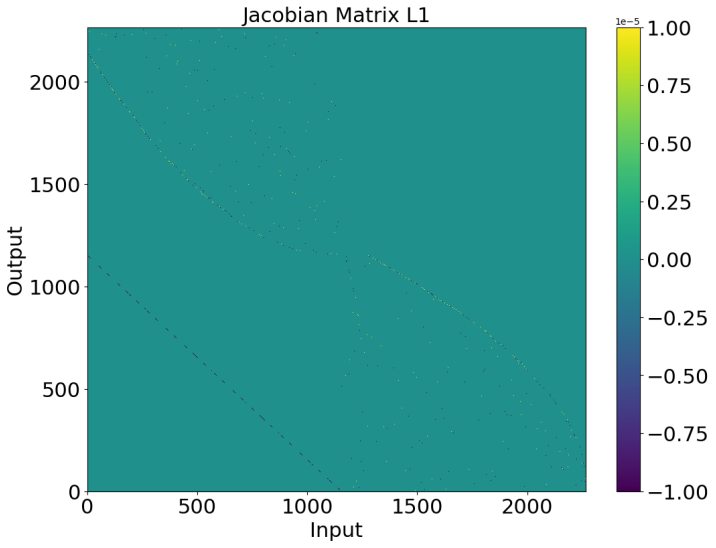


Figure 4.4: Jacobian of a large sized network, clipped

Figures 4.4 and 4.5 display that the structure of larger matrices remain similar.

4.1.4. Jacobians at the final Newton iteration

The Jacobian matrices exhibit similar structures when evaluated at the final iteration of the Newton method, as demonstrated in appendix B. However, absolute convergence may not be attained due to various factors associated with Newton's method.

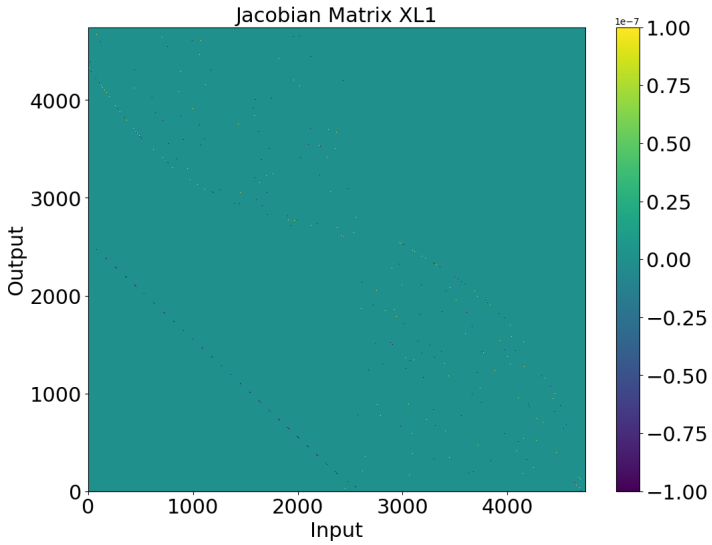


Figure 4.5: Jacobian of an extra large network, clipped

There are certain reasons for the Jacobian to be different at the final iteration versus the initial input. Newton's method iteratively refines root estimates until a convergence criterion is met, leading to varying convergence behaviour based on initial guesses and criteria. Consequently, Jacobian matrices evaluated at different converged roots may show slight differences. Additionally, numerical approximations in Newton's method introduce small perturbations during Jacobian computation, potentially accumulating over iterations and causing deviations from the original structure. Moreover, linearization around root estimates may compromise accuracy in highly nonlinear regions, contributing to differences between Jacobian matrices at converged roots and the original. Apart from these slight perturbations, the main structure of a Jacobian is likely to remain similar.

4.2. Eigenvalues

Eigenvalues play a pivotal role in various mathematical and computational contexts, particularly in linear algebra and numerical analysis. For systems of linear equations, eigenvalues can provide crucial insights into the behaviour and structure of the problem (Trefenthen et al., 2022). Specifically, the eigenvalues of a matrix can indicate its properties, such as its condition number and definiteness. This information guides the selection of appropriate numerical algorithms for solving the system efficiently. For instance, matrices with well-conditioned eigenvalues often lend themselves to accelerate direct solving methods, such as LU decomposition or Cholesky factorization. Thus, understanding the eigenvalue structure of a matrix is essential for devising and implementing effective direct solving strategies.

Figure 4.6 presents the eigenvalues of the network XL_1. Consequently, the convergence properties of the Newton-Raphson rootfinding method can be deduced. The eigenvalue plots of the other networks are supplied in appendix B.

The range of eigenvalue magnitudes indicates varying rates of change across different parts of the solution space. Smaller eigenvalues suggest slower transitions, while larger ones signify quicker changes along specific directions. The broad spectrum of eigenvalues signifies a system with intricate dynamics, which can complicate convergence patterns for numerical solution methods.

The real parts of eigenvalues play a crucial role in determining the stability and convergence behaviour of iterative methods. Negative real parts indicate stable directions, where convergence towards the solution is likely. Conversely, positive real parts suggest unstable directions, potentially leading to divergence unless effectively addressed. Eigenvalues with real parts close to zero often pose challenges to convergence, resulting in slower rates or stagnation.

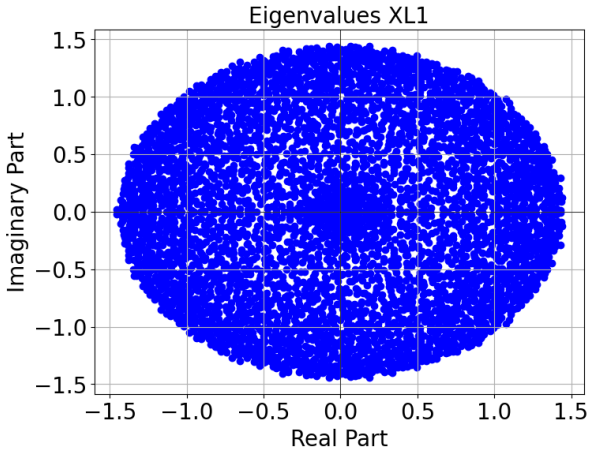


Figure 4.6: Eigenvalues of the XL_1 network.

Moreover, eigenvalues with non-zero imaginary parts introduce oscillatory dynamics into the system. These oscillations can alternate between convergence and divergence during iteration. The magnitude of the imaginary parts determines the frequency of oscillations, with larger values leading to faster oscillations. In extreme cases, purely imaginary eigenvalues may cause persistent oscillations or even chaotic behaviour within the system.

The eigenvalues of all other networks follow similar patterns (and hence behaviour), but with fewer values since the Jacobians are smaller. The influence of eigenvalues, particularly in relation to their real and imaginary parts, is mostly a concern for iterative solvers. These solvers rely on the properties of the system's matrix to guide their convergence. The eigenvalue distribution directly impacts their performance. Direct solvers, on the other hand, generally do not depend on these dynamics as they typically involve decomposing the matrix directly to find solutions, irrespective of the eigenvalue characteristics. Thus, understanding and managing the spectrum of eigenvalues is essential primarily for the effective application of iterative solvers in handling time-dependent complexities within large-scale and dynamic simulations.

4.3. Condition numbers

The condition number, given in equation 4.1 measures how sensitive the solution of a linear system is to changes in the matrix, while singularity refers to a matrix's lack of invertibility. A high condition number often indicates ill-conditioning, where small changes in the matrix can lead to large changes in the solution. This ill-conditioning can result in singularity, where the matrix is not invertible and lacks a unique solution. Hence, the condition number and singularity are related through their indication of the stability and uniqueness of solutions in linear systems (Trefenthen et al., 2022).

$$\kappa(A) = \frac{\sigma_{\text{max}}}{\sigma_{\text{min}}} \tag{4.1}$$

The analysis of the networks reveals high condition numbers for all systems, as presented in table 4.2. They are however far from infinite, so not singular. Networks S_0 and S_1 have relatively low condition numbers, indicating better numerical stability compared to the other networks. Networks L_1 and XL_1 have high condition numbers, indicating potential numerical instability and slower convergence for iterative methods compared to the other networks. The condition number shoul not scale,

The number of nodes and edges in each network provides insight into the complexity and connectivity of the network structure. Networks L_1 and XL_1 have significantly more nodes and edges compared to networks S_0 and S_1, indicating higher complexity and potentially more interconnectedness. The higher complexity and connectivity of networks L_1 and XL_1 may contribute to their higher condition numbers

	S_0	S_1	M_1	L_1	XL_1
Condition number	6.1×10^7	2.8×10^6	5.7×10^{10}	2.6×10^{10}	1.4×10^{16}

Table 4.2: Condition number of the various matrices

due to increased potential for ill-conditioning and numerical instability.

This suggests that networks S_0 and S_1 are relatively simpler and more numerically stable compared to the larger networks. Networks L_1 and XL_1, with higher complexity and connectivity, exhibit higher condition numbers, indicating potential numerical instability and slower convergence for iterative methods. XL_1 stands out with the highest condition number, indicating potential challenges in numerical stability and convergence when solving linear systems associated with this network.

4.4. Conclusion on Jacobian properties

In conclusion, this chapter established a foundation for numerical method selection. Since the matrices are non-symmetric and non-positive definite and their eigenvalues are widely divergent, many direct and iterative methods are not applicable. Therefore, the preference goes to a direct method that leverages sparsity without the complexities associated with iterative solver convergence. The analysis reveals that despite scalability challenges with large matrices, direct solving methods remain feasible and effective for network sizes not exceeding around one million entries in the Jacobian (del hoyo Arce, 2018).

The strategic selection of direct and iterative solving methods, as outlined, will be further explored in subsequent chapters. Robust Python software packages are evaluated from chapter 5 onwards. These methods offer directly applicable sophisticated methods. Chapter 6 compares the efficiency of them, along with their accuracy benefit.

5

Implementation Details

Now focusing on the linear solve, the question of its implementing arises. This chapter delves into the practical aspects of implementing numerical methods for solving district heating network problems. Specifically, it explores strategies for efficiently storing sparse matrices, selecting fast direct solvers in Python, and optimising computational processes. Additionally, considerations for reusing evaluations, ensuring scientific comparability, and evaluating performance metrics are discussed. The only set threshold by Gradyent is an accuracy of 10^{-7}, since their product should be reliable. There is no bound on iterations and timing apart from that.

Section 5.1 highlights the disparity between CPU time and wall clock time to clarify the metrics employed in this study. Before delving into actual computations, the potential memory overhead from matrix storage is addressed in section 5.2, offering insights into efficient handling techniques. Subsequently, a survey of various software packages pertinent to rootfinding methods is conducted in section 5.3. Furthermore, to ensure equitable comparison of all outcomes, section 5.4 outlines the methodology employed for measurements to promote fairness.

5.1. CPU time versus wall clock time

The evaluation of these various rootffinding algorithms often relies on two types of time measurements: CPU times and wall clock times, especially in parallel computing contexts. CPU time measures the duration of processor utilisation, while wall clock time encompasses all elapsed time, including waiting and overhead. Wall clock time is preferred for its comprehensive view of performance, incorporating factors like parallelization overhead (the additional time and resources required to execute parallel tasks) and communication latency (the delay in transmitting data between computing units). It provides a realistic assessment of algorithmic scalability and efficiency across different hardware setups. Since this research is conducted within one environment that remains consistent, the wall clock times are valid for comparison.

5.2. Storing the sparse matrix

Apart from optimising the solving process, it is also important to store the data wisely. Zeroes take up unnecessary space and can be avoided. Sparse matrices can be represented in various formats, including Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate (COO) formats. These formats offer efficient storage and manipulation of sparse matrices, each with its unique benefits. Only the non-zero elements are stored along with their indices, resulting in reduced memory usage compared to storing all elements of the matrix. Matrix-vector multiplication can be performed more efficiently, as only the non-zero elements need to be accessed and computed (Shanaz, 2005).

- **CSR (Compressed Sparse Row) Format:** In CSR format, the matrix is stored row-wise. It consists of three arrays: the values array containing the non-zero elements of the matrix, the column indices array indicating the column indices of the non-zero elements, and the row pointer

array specifying the starting index of each row in the values and column indices arrays. CSR format is efficient for matrix-vector multiplication and row-wise operations.

- **CSC (Compressed Sparse Column) Format:** In CSC format, the matrix is stored column-wise. Similar to CSR format, it comprises three arrays: the values array containing the non-zero elements, the row indices array indicating the row indices of the non-zero elements, and the column pointer array specifying the starting index of each column in the values and row indices arrays. CSC format is advantageous for column-wise operations and matrix-vector multiplication.
- **COO (Coordinate) Format:** In COO format, each non-zero element is represented by its coordinates (row index, column index, value). This format does not require any compression, making it straightforward for insertion and manipulation of individual elements. However, it may consume more memory compared to CSR and CSC formats, especially for matrices with many non-zero elements.

Below is an example illustrating the representation of a sparse matrix in CSR, CSC and COO formats:

$$\text{Matrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 5 & 0 & 6 \end{bmatrix}$$

CSR Format:

$$\text{Values} = \{1, 2, 3, 4, 5, 6\}$$

$$\text{Column Indices} = \{0, 1, 2, 3, 1, 3\}$$

$$\text{Row Pointer} = \{0, 1, 3, 3, 5\}$$

All non-zero values of the matrix are listed row-wise, resulting in the array $\{1, 2, 3, 4, 5, 6\}$. The column indices of each non-zero value are recorded, also row-wise. For example, the first non-zero value 1 is in column 0, the second non-zero value 2 is in column 1, and so on. This yields the array $\{0, 1, 2, 3, 1, 3\}$. An array indicating the starting index of each row in the Values and Column Indices arrays is maintained. For instance, row 0 starts at index 0, row 1 starts at index 1, row 2 starts at index 3, and row 3 starts at index 3. This array is $\{0, 1, 3, 3, 5\}$.

CSC Format:

$$\text{Values} = \{1, 2, 5, 3, 6, 4\}$$

$$\text{Row Indices} = \{0, 1, 3, 1, 3, 2\}$$

$$\text{Column Pointer} = \{0, 1, 4, 5, 6\}$$

All non-zero values of the matrix are listed column-wise, resulting in the array $\{1, 2, 5, 3, 6, 4\}$. The row indices of each non-zero value are recorded, also column-wise. For example, the first non-zero value 1 is in row 0, the second non-zero value 2 is in row 1, and so on. This yields the array $\{0, 1, 3, 1, 3, 2\}$. An array indicating the starting index of each column in the Values and Row Indices arrays is maintained. For instance, column 0 starts at index 0, column 1 starts at index 1, column 2 starts at index 4, and column 3 starts at index 5. This array is $\{0, 1, 4, 5, 6\}$.

COO Format:

$$\text{Values} = \{1, 2, 5, 3, 6, 4\}$$

$$\text{Row Indices} = \{0, 1, 1, 2, 3, 3\}$$

$$\text{Column Indices} = \{0, 1, 2, 3, 1, 3\}$$

All non-zero values of the matrix are listed, along with their respective row and column indices. For example, the first non-zero value 1 is located at row 0, column 0, resulting in the entry $(0, 0, 1)$ in the COO format. Similarly, the second non-zero value 2 is at row 1, column 1, leading to the entry $(1, 1, 2)$. This process continues for all non-zero values in the matrix.

Sparse matrix formats provide significant memory savings by storing only the non-zero elements along with their indices, reducing memory usage compared to dense matrices. This benefit is particularly valuable for large-scale problems and applications with memory constraints.

Additionally, exploiting the sparsity pattern of matrices leads to improved computational efficiency. Sparse matrix representations enable faster matrix-vector multiplication and other operations, making iterative solvers and numerical algorithms more efficient, especially when repeated matrix operations are performed. The packages discussed in section 5.3 make use of these formats in their computations as well. See appendix A for the Python code.

Comparison

In this research, all main focus lies on the Jacobians, which are square matrices. The choice between Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats then may not significantly impact the performance of solvers, such as the `spsolve` function from `pypardiso` library. For square matrices, the primary difference between CSR and CSC formats lies in their data organisation regarding row and column indices. Both CSR and CSC formats offer distinct advantages depending on the sparsity pattern and specific operations performed on the matrices. However, for square matrices, the differences in performance between the two formats might be minimal. In practise, it is common to choose the format that the solver requires.

5.3. Fast direct solvers in Python

In addition to the efficient storage and computation of matrices, it is important to consider the choice of solver for sparse systems. With a Jacobian matrix size far under one million entries, using a direct solver remains practical and effective. As matrix sizes increase, it may become necessary to explore preconditioning techniques to maintain computational efficiency. This section is based on Johansson (2015).

5.3.1. Python Packages; SciPy, PARDISO, SuperLU

The decomposition methods discussed in chapter 3 serve as the foundation for numerous software packages. These packages are developed with considerable expertise, evident in their efficiency and ease to use.

Scipy's: One example is SciPy. This package relies on sparse direct solvers and iterative methods to tackle large-scale sparse linear systems effectively. Through direct solvers like LU or Cholesky decomposition, the `scipy.spsolve` function breaks down the matrix into triangular components (lower and upper for LU, lower for Cholesky), solving efficiently through forward and backward substitution. Moreover, for larger systems, SPSolve may utilise iterative methods like Conjugate Gradient (CG) or Bi-CGSTAB to enhance computational efficiency (Virtanen et al., 2020).

PARDISO: Another method involves PARDISO (Parallel Direct Sparse Solver), designed for high-performance solving of large-scale sparse linear systems. Leveraging parallelism and multithreading, PARDISO spreads the computational load across multiple processors, substantially reducing solution times for sizable problems. Utilising direct factorization methods such as LU or Cholesky decomposition, PARDISO adeptly decomposes the sparse matrix into triangular components, ensuring numerical stability and accuracy (Schenk, 2024)

SuperLU: Additionally, SuperLU is examined, proficient in factoring sparse matrices via LU decomposition with partial pivoting. Employing techniques like supernodal factorization, SuperLU optimises computational efficiency and memory usage. By capitalising on sparse matrix representation and employing a sparse direct solver strategy, SuperLU accurately computes solutions without iterative refinement, proving beneficial for applications requiring precision (Li, 2004).

Each solver —SPSolve, PARDISO, and SuperLU—has its strengths. SPSolve is well-integrated within SciPy, making it user-friendly and readily accessible for Python users. PARDISO employs advanced algorithms and parallel computing techniques, making it particularly well-suited for sparse ma-

trices and high-performance computing environments. SuperLU complements these by offering a balance between ease of use and efficiency, particularly in parallel environments. The choice among these depends on the specific requirements of the problem, available computational resources, and ease of integration into the existing workflow. As the size of the problem scales, the selection criteria might shift towards more advanced features like parallel execution and sophisticated memory management offered by PARDISO and SuperLU.

5.3.2. Iterative Linear Solvers options

In the context of enhancing the Newton-Raphson method for solving district heating network problems, the integration of iterative linear solvers such as Conjugate Gradient (CG), BiConjugate Gradient Stabilised (Bi-CGSTAB), and Generalized Minimal Residual (GMRES) within a Python environment offers a robust solution framework. These solvers are particularly adept at handling the large, sparse linear systems often encountered in such applications (Johansson, 2015).

BiConjugate Gradient Stabilised (Bi-CGSTAB) Method: The BiConjugate Gradient Stabilised Method includes an incomplete LU factorization (`scipy.sparse.linalg.spilu`) as a preconditioner, which approximates the matrix inverse and thus enhances its efficiency. This method is well-suited for non-symmetric matrices. The preconditioner, using Scipy's function `LinearOperator`, significantly improves the convergence rate, particularly in systems with complex sparsity patterns (SciPy, 2022).

Generalized Minimal Residual (GMRES) Method: Similar to Bi-CGSTAB, this method employs preconditioning, for instance through an ILu. GMRES works by minimising the residual over a Krylov subspace formed by the matrix and the right-hand side, making it highly effective for diverse matrix properties. The inclusion of the same preconditioner as Bi-CGSTAB facilitates a direct comparison of how each algorithm handles the Krylov subspace differently (SciPy, 2020).

All three methods convert the Jacobian to a sparse CSC format, which is efficient for matrix-vector operations crucial in iterative solvers. Both Bi-CGSTAB and GMRES can utilise preconditioning, crucial for improving computational efficiency and convergence rates in large-scale problems. This approach is beneficial in managing the computational complexity inherent in district heating networks.

However, CG is inherently different in its approach, targeting symmetric positive definite matrices, while Bi-CGSTAB and GMRES do not have this limitation and are suitable for a broader range of problems. Bi-CGSTAB stabilises the bi-conjugate gradient method, potentially offering faster convergence for certain types of non-symmetric matrices. GMRES, on the other hand, minimises the residual across the entire Krylov subspace, which can be computationally intensive but robust over a wider range of problems.

5.3.3. Library options; tolerance, fill factor, permutations

In the context of numerical methods such as Newton-Raphson, which involve nested solvers, the choice of tolerance levels for both the main solver and the nested iterative linear solvers such as BiConjugate Gradient Stabilised (Bi-CGSTAB) and Generalized Minimal Residual (GMRES) is crucial. This selection significantly impacts the accuracy and computational efficiency of the overall solution process. Setting the tolerance for iterative linear solvers at 10^{-9}, much smaller than the 10^{-7} tolerance for the Newton-Raphson method, is strategic. This differential ensures that the errors introduced by the linear solvers are significantly less than the tolerance to which the overall Newton method converges, preventing these inaccuracies from impeding or limiting the convergence of the Newton-Raphson method. If the tolerance of the linear solver were higher or comparable to that of Newton's method, it could result in the premature termination of the Newton-Raphson iteration due to inaccuracies propagated from the linear solver, thus compromising the overall solution accuracy (Virtanen et al., 2020). Moreover, certain parameters used in the construction of preconditioners like ILU (Incomplete LU) significantly impact the performance and effectiveness of the preconditioning process:

Drop Tolerance: This parameter controls the threshold at which smaller elements of the matrix are discarded in the ILU factorization process. A higher drop tolerance leads to more elements being

dropped, reducing memory usage and computational time but potentially decreasing the accuracy and stability of the preconditioner. Conversely, a lower drop tolerance retains more elements, enhancing the effectiveness and cost of the preconditioner.

Fill Factor: Determines the maximum allowed fill-in elements in the ILU factorization relative to the original matrix. It controls how much additional memory space is allocated for elements that fill in during the factorization process. A higher fill factor allows more fill-in, enhancing the accuracy and robustness of the preconditioner but at the cost of increased memory usage.

Permutation Specification: Controls the strategy for reordering the matrix before factorization to improve matrix stability and factorization efficiency. Different strategies, such as natural ordering or minimum degree ordering, can significantly impact the effectiveness of the factorization, particularly in terms of reducing fill-in and enhancing numerical stability.

The detailed settings of parameters like drop tolerance, fill factor, and permutation specification in preconditioners are crucial for optimising the performance of numerical solvers in tackling complex systems such as district heating networks. These settings tailor the solver's behaviour to balance computational efficiency and solution accuracy effectively.

In determining whether to use Bi-CGSTAB or GMRES, consideration should be given to specific system characteristics, such as the degree of matrix non-symmetry and the effectiveness of the preconditioner. Integration of these iterative solvers into the Newton-Raphson method results in a flexible and powerful toolkit for addressing the complexities of district heating network simulations.

5.4. Measurements, scientific comparability

In order to ensure meaningful comparisons among the results obtained from various optimization techniques aimed at enhancing the performance of solvers for district heating networks, it is essential to establish a controlled and consistent experimental framework. This framework should cover all aspects of the computational process, from initialisation to execution and final data analysis. In this section, the critical components of such a framework are evaluated. The theory is based on Johansson (2015).

To start off with, it is important to ensure that all algorithms begin with identical starting values to guarantee comparability. This uniformity includes initial guesses for system states, fixed parameters, boundary conditions, and any other input data affecting outcomes. If not stated differently, the initial values are randomised for every iteration with values between zero and one. The same seed value is utilised, to increase robustness even more. Seed randomisation is the process of initialising a random number generator with a specific value to ensure reproducibility in research, as using the same seed value guarantees identical random sequences, enabling consistent results across different runs.

Furthermore, integrity and reproducibility are wanted. For this, maintaining hardware consistency is necessary. All computational experiments should be conducted on the same hardware setup, comprising identical processors, memory configurations, and storage systems. This practice eliminates variations arising from differences in computational power and data handling speeds. This entire research has been deducted on one computer, ensuring the hardware remained the same. Secondly, consistency in the software environment, including operating systems, programming languages, and numerical libraries, is crucial across all experiments. The software environment of this thesis has been established before, and not changed during this research.

Additionally, single-tasking operations should be adopted to mitigate discrepancies caused by varying CPU and memory usage. This involves dedicating the computational environment solely to the solver each run and closing unnecessary programs and background processes that may interfere with computational performance. In practise, that means turning off all applications outside this program. Also, if possible, remove any unnecessary connections such as internet. These measures collectively contribute to the reliability and consistency of computational experiments. For this research, that involved shutting down all other applications on the computer. However, disabling internet connectivity

and exclusively running the code is not possible, since internet is necessary for entering the virtual computer that Gradyent works with.

Ensuring consistency in the core computational code across all experimental runs is crucial for reliable research outcomes. One should properly document any variations introduced in the functions. Version control systems could be of use for managing code changes. For the functions examined in this report, the code is optimised before any results reported in this document were obtained.

Lastly, establishing a predefined set of performance metrics for comparison is imperative. These performance metrics may include solution accuracy, computational time, memory usage, and the number of iterations to convergence. Applying these metrics uniformly across all methods facilitates comprehensive evaluation and comparison. These practises collectively contribute to the reliability and consistency of experimental results.

This chapter addressed the implementation of numerical methods for effectively solving district heating network problems using Python. It provides insights into handling sparse matrices, selecting optimal solvers, and optimising computational efficiency. The exploration of time metrics, storage techniques, and robust software packages lays a solid groundwork for executing numerical models essential for network optimisation. In the subsequent chapter, these methods will be applied to the networks, enabling comparison of their effectiveness and practical implications.

6

Results and analysis of various functions

This chapter delves into the various Newton methods, elucidating the theoretical foundations upon which they are predicated and exploring their computational efficiencies. Initially, the most straightforward method, involving the inversion of the Jacobian matrix, is evaluated in light of its theoretical underpinnings. Subsequently, an alternative approach, which entails solving the linear system without explicitly computing the inverse, is presented. Building upon this, the chapter scrutinises different sparse matrix solvers, including Scipy's SPSolve, PARDISO's SPSolve, and SuperLU, conducting a comparative analysis to discern their respective strengths and weaknesses.

The direct methods are evaluated in sections 6.1, 6.2, 6.3, 6.4, 6.5 and compared in section 6.6. Next, the iterative methods for the linear solve are proposed in sections 6.7 and 6.8, with a comparison in section 6.9. Section 6.10 compares all methods to each other, and evaluates their scaling properties. Appendix B contains additional plots for enhanced visualization. Specifically, it includes residual plots versus iterations for all methods, including the extra networks S_2, M_2, and L_2.

6.1. Newton's Method; Inverting the Jacobian

The first function computes the inverse of the Jacobian matrix using and then performs matrix-vector multiplication. While conceptually straightforward, this method may suffer from numerical instability and increased computational complexity due to the explicit inversion of the Jacobian matrix, especially for large matrices or those with high condition numbers.

The plots in 6.1 and results in table B.1 represent the performance and convergence of a more efficient Newton-Raphson method applied to solve systems of equations characterised by the matrices.

	S_1	M_1	L_1	XL_1
#iter	5	12	10	13
CPU time (ms)	14.7×10^2	6.17×10^3	34.5×10^3	3.62×10^4
Wall clock time (ms)	2.80×10^2	1.22×10^3	5.80×10^3	5.86×10^4

Table 6.1: Computational time Inverse Jacobian method

The results plotted in figure 6.1 can be analysed as;

- S_1: Convergence within 5 iterations with a steady decline in the residual, demonstrating efficient performance.
- M_1: Takes 12 iterations to converge, fairly smooth.
- L_1: Requires 10 iterations to converge, with the residual plot showing a consistent downward trend. This suggests that despite the larger size or complexity, the system remains numerically stable under the Newton method which inverses the Jacobian.

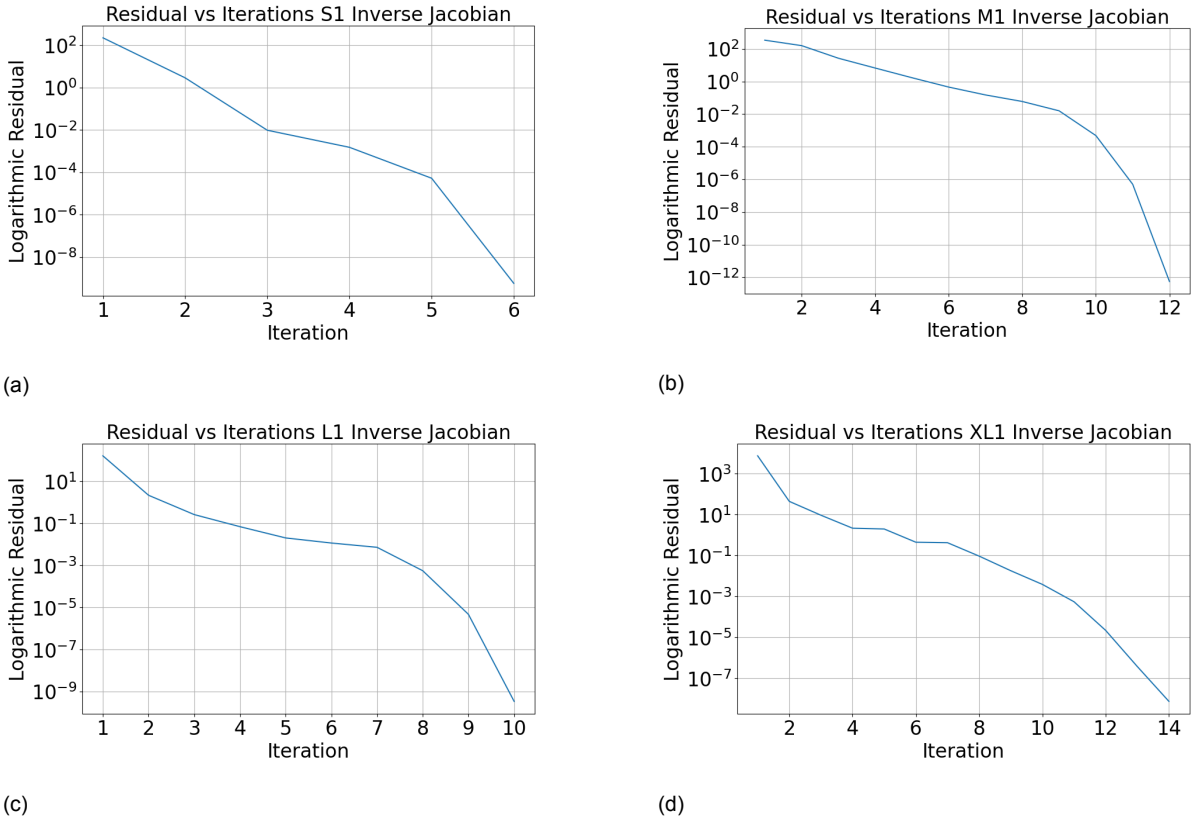


Figure 6.1: Convergence of Newton -inverting the Jacobian- applied directly, to different networks.

- XL_1: The residual indicates a convergence over 13 iterations. Notably, the rate of decrease in the logarithmic residual is less steep compared to the smaller networks, hinting at the increased computational challenge presented by larger systems.

Table B.1 indicates the number of iterations and both CPU and wall clock times for computational performance evaluation. The iteration count is directly correlated with the matrix size and complexity. S_1 and M_1, being smaller or simpler, converge faster than L_1 and XL_1. This suggests that the larger matrices may have more complex dynamics or may not be as well-conditioned as the smaller matrices.

- CPU time: An $O(n^3)$ increase in CPU time with the size and complexity of the network matrices, which is expected due to the increased operations required for larger matrix inversions.

- Wall clock time: The table shows a significant disparity from CPU time, especially for L_1. This could indicate that the computation for L_1 is efficiently parallelised, making good use of multi-threading or distributed computing resources. The significant difference between CPU and wall clock times for XL_1 is even more pronounced than for L_1, potentially indicating efficient use of multi-threading or distributed computing environments for this complex computation.

As the size of the network (matrix) increases, so does the computational effort, as seen from the increase in CPU and wall clock times. This reflects the larger number of floating-point operations required. A faster convergence rate, as seen in S_1 and M_1, leads to lower computational times. The fewer the iterations, the less work is required. Moreover, the residual plots show that the method is numerically stable for all network sizes, but the rate of convergence slows for larger networks.

In conclusion, the Inverse Jacobian Newton-Raphson method exhibits stable and efficient convergence for smaller systems. However, as the system size increases, there is a notable increase in computational effort.

6.2. Newton's method; Linear Solve

To facilitate a fair and thorough comparison of upcoming methods like PARDISO and Scipy's SPSolve, an intermediate step is proposed here. Instead of relying on direct matrix inversion, this solver shifts towards utilizing functions that linearly solve the system. It establishes a robust foundation for fair and insightful evaluations of the efficiency.

The linear solving function utilises NumPy to directly solve the linear system represented by the Jacobian. By leveraging NumPy's computational power, known for its proficiency in solving linear systems, the solver ensures smooth compatibility with both PARDISO and Scipy (Johansson, 2015). Considering these factors, the linear solving method is expected to outperform the first method in terms of computational efficiency and numerical stability. The residual plots for the Linear Solve method, displayed in appendix B, show a rapid decrease in residuals, indicating a fast rate of convergence. Linear Solve maintains the efficient convergence properties of the Newton-Raphson method.

	S_1	M_1	L_1	XL_1
#iter	5	12	10	13
CPU time (ms)	13.6×10^2	9.44×10^3	27.2×10^3	2.49×10^4
Wall clock time (ms)	2.87×10^2	1.70×10^3	4.62×10^3	3.88×10^4

Table 6.2: Computational time Linear Solve method

The iteration counts for the Linear Solve method, as seen in table 6.2, are the same as those for the Inverse Jacobian method for the networks of size up to XL_1, indicating that the method's efficiency in reducing residuals is consistent irrespective of the approach taken for the Newton step. Notably, the CPU time for XL_1 is less than that for L_1, which might indicate a more efficient handling of larger systems or a difference in system conditioning.

The scalability of the Linear Solve method seems to be comparable to the first method. However, given the reduced risk of numerical instability, Linear Solve may offer better performance for even larger network sizes beyond those tested. The differences between CPU and wall clock times suggest that the Linear Solve method is able to effectively utilise parallel computing resources, which becomes increasingly important as the network size grows.

Based on the provided data, the Linear Solve method should be considered over direct Jacobian inversion, especially for large-scale problems. The convergence behaviour of the Linear Solve method in the XL_1 network underscores its suitability for larger and more intricate systems. It is recommended that future work in solving such systems considers the use of the Linear Solve method, particularly when dealing with large-scale networks that may benefit from enhanced numerical stability.

6.3. SciPy's SPSolve

SPSolve from SciPy's library is a function built for solving sparse linear systems. It converts the Jacobian matrix into a Compressed Sparse Column (CSC) format, which is particularly efficient for matrix-vector operations as seen in chapter 5. This method is expected to be more efficient than dense solvers when dealing with sparse systems due to the optimised use of storage and computational resources.

The residual plots, in appendix B, indicate that the SPSolve method converges for all network sizes within a comparable number of iterations to the previous methods, which implies that the change in solving technique does again not affect the convergence properties of the Newton-Raphson method. The plots exhibits a gradual but consistent decrease in the logarithmic residual. The XL_1 network's CPU and wall clock times for SPSolve suggest that while SPSolve maintains an advantage in computational efficiency for sparse matrices, this may diminish as the matrix size and complexity increase.

The CPU time required by SPSolve is significantly reduced, particularly noticeable for S_1 and M_1, demonstrating the efficiency of sparse solvers for smaller or simpler network matrices. Appendix B contains a table of all computational times tables combined. The other CPU times also remain lower than those of the Linear Solve Newton method, highlighting SPSolve's advantage for larger networks.

	S_1	M_1	L_1	XL_1
#iter	5	12	10	13
CPU time (ms)	3.12×10^1	20.9×10^2	18.3×10^3	2.04×10^4
Wall clock time (ms)	2.56×10^1	5.57×10^2	3.42×10^3	3.29×10^4

Table 6.3: Computational time Linear Scipy's SPSolve method

However, scaling from smaller to larger networks doesn't show notably improved efficiency compared to previous methods. The increasing gap in wall clock time for the XL_1 network indicates a growing challenge in maintaining the same level of efficiency for very large and complex networks.

SPSolve maintains the stability observed in the Linear Solve method while potentially offering better performance. There is a significant performance gain in using SPSolve for sparse matrices, as shown by the low CPU and wall clock times, making it suitable for real-world applications with large-scale systems. The SPSolve method is highly recommended for solving sparse systems compared to the other methods seen up to this point, due to its efficient use of computational resources.

6.4. PARDISO

The PARDISO solver is designed to handle sparse matrices (symmetric and nonsymmetric) matrices with high efficiency. It is particularly suited for the types of systems often encountered in finite element simulations and other applications involving large-scale scientific computations. It is therefore expected that this solver outperforms the previous method of section 6.3.

	S_1	M_1	L_1	XL_1
#iter	5	12	10	-
CPU time (ms)	14.1×10^1	35.1×10^2	20.2×10^4	-
Wall clock time (ms)	3.72×10^1	6.68×10^2	3.59×10^4	-

Table 6.4: Computational time PARDISO method

Table 6.4 and figure 6.2 display that S_1 and M_1 exhibit consistent convergence, indicating PARDISO's efficiency in handling smaller and less complex systems. However, the plot for the L_1 network displays irregularities with sharp spikes in residuals, suggesting numerical instability or an ill-conditioned matrix. Possible causes include insufficient solver parameters or preconditioning. L_2 shows similar, nonconverging behaviour, with residuals varying across several orders of magnitude. Convertion to CSR format have similar looking plots to those of the initial, CSC, format (CSC is the preferred format for PyPardiso). Plots are included in appendix B. Attempts to plot the residual function for the XL_1 network were unsuccessful, as the program halted before reaching a solution.

The lack of convergence for the XL_1 network with the PARDISO solver underlines the need for further evaluation of solver capabilities and limitations. Possible problems could consist of multiple reasons. PARDISO relies on LU factorization, and as problem size grows, stability can diminish, especially without effective management of scaling issues in pivoting strategies. Initialization is also considered as a potential cause. Testing with various input vectors, including randomised and constant ones, yields similar results. Although PARDISO can be sensitive to initial guesses, it seems unlikely to be the problematic point of the oscillatory behaviour. Alternatively, precision issues may arise in large networks, leading to underflow or overflow, causing solver convergence failures. This could potentially explain oscillating or diverging residuals.

To diagnose where in these possibilities the actual problem lies within PARDISO, the obtained roots are examined. By employing the same initial value vector and solving the system with an alternative method, comparison data is obtained. Given that the problem arises at L_1, the focus is on solving its Jacobian. Subtracting these vectors and computing their norm allows assessment of whether it constitutes a valid solution. Since both the two-norm and the infinity norm are notably nonzero (rounded to 126 and 25, respectively), the solution provided by the PARDISO solver is not actually a root. Consequently, certain settings within the solver may not be optimal yet.

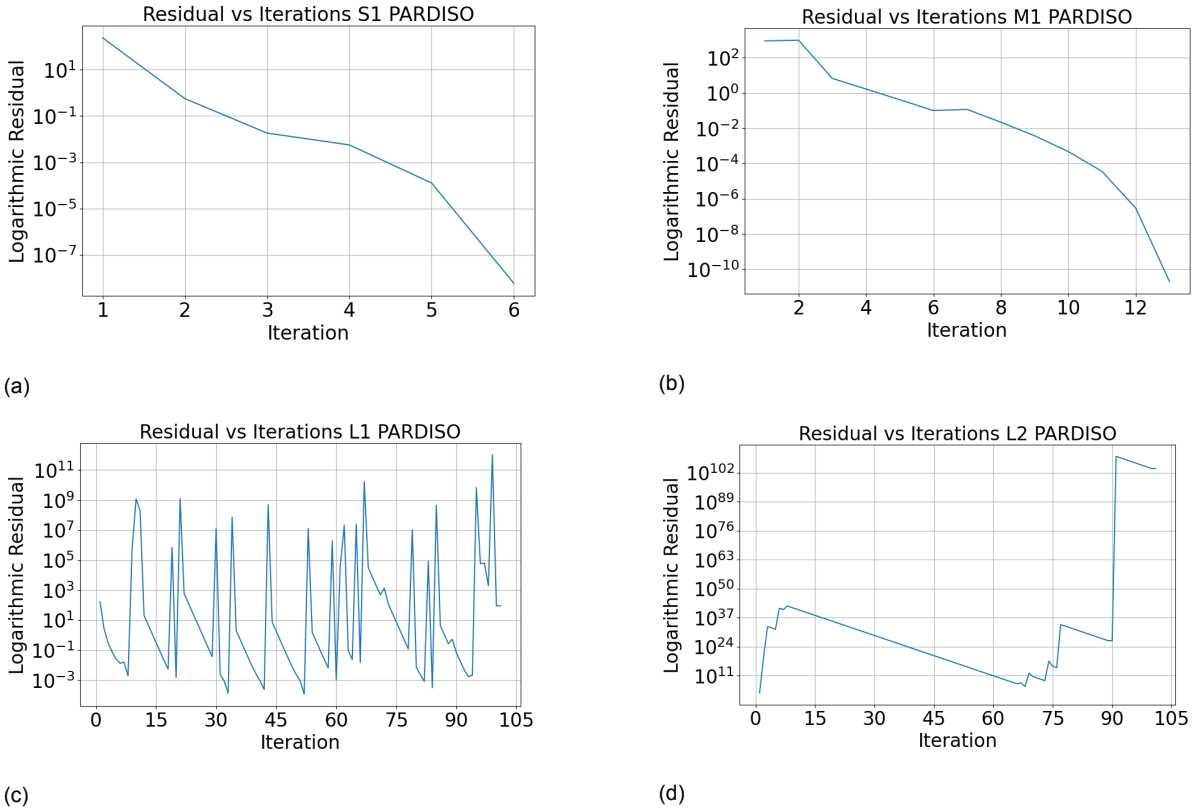


Figure 6.2: Convergence of the PARDISO Newton applied directly, to different networks.

PARDISO library option; UMFpack

The observed nonconvergence behaviour for larger networks when using PARDISO might be mitigated by switching to UMFPACK, which is better suited for nonsymmetric and sparse systems. This function implements an Unsymmetric MultiFrontal method, which is designed to handle sparse nonsymmetric matrices efficiently. Theoretically, it should not exhibit the same nonconvergence behaviour as PARDISO (Schenk, 2002).

Next to that, UMFPACK uses different strategies for reducing fill-in, such as column pre-ordering to minimise the work during numerical factorization. It also applies partial pivoting to maintain stability during the factorization, which can be crucial for larger systems that might lead to instability.

When UMFpack is implemented to be 'False' within the PARDISO library, a standard LU decomposition solver is used within the PARDISO spsolve function, which may not have the same optimization for handling sparsity and nonsymmetry as UMFPACK. In the plots provided in appendix B, it can be seen that the behaviour with and without UMFpack remains similar.

Further investigation is necessary to understand and potentially improve its stability. However, for the scope of this research, PARDISO will not be taken into account any further.

6.5. SuperLU

SuperLU tackles sparse linear systems by decomposing the sparse matrix into lower and upper triangular matrices. The residual plots, shown in appendix B, suggest a consistent decrease in residuals, indicating that the method converges to the solution without any abnormal behaviour. The absence of spikes in the residuals, as seen with the PARDISO method for the L_1 network, suggests better numerical stability in this context.

The number of iterations for SuperLU is generally in line with or slightly higher than the other methods discussed, which may reflect the overhead of decomposing the matrix in each iteration. The CPU

	S_1	M_1	L_1	XL_1
#iter	6	14	12	13
CPU time (ms)	7.81×10^1	26.4×10^2	21.7×10^3	22.3×10^3
Wall clock time (ms)	3.01×10^1	6.73×10^2	4.12×10^3	3.30×10^3

Table 6.5: Computational time SuperLU method

times vary, with S_1 showing a fast resolution. For M_1, L_1 and XL_1 the times are again reasonable given the complexity of the networks. The wall clock times remain low. SuperLU shows good performance scaling with increased network complexity, as indicated by the relatively low wall clock times for larger networks (L_1 and XL_1). The low wall clock times across the board suggest that SuperLU is very efficient when solving sparse systems, making full use of the sparse matrix structure to optimise computation.

The SuperLU solver demonstrates effective handling of the Newton-Raphson method across different network sizes. SuperLU shows promise as a solver that can handle a wide range of network sizes without significant stability issues. The provided data suggests SuperLU as a robust choice for solving sparse systems efficiently and stably, even as the size and complexity of the system increase.

6.6. Comparison of various direct methods

This section offers insights into the comparative effectiveness across various network configurations. While the preceding sections examined running the functions once, in this section, they are executed ten times to adequately showcase their characteristics. Their mean times are presented along with their standard deviation, a lower standard deviation indicates more predictable performance. All durations are presented in seconds.

Solver Method	S_1 Mean Time	S_1 Std. Dev.	M_1 Mean Time	M_1 Std. Dev.
Inverse Jacobian	4.25×10^{-2}	49×10^{-4}	11.5×10^{-1}	19×10^{-2}
Linear Solve	4.13×10^{-2}	79×10^{-4}	8.54×10^{-1}	8.0×10^{-2}
Scipy's SPSolve	$\mathbf{2.18 \times 10^{-2}}$	$\mathbf{3.0 \times 10^{-4}}$	$\mathbf{6.23 \times 10^{-1}}$	$\mathbf{1.0 \times 10^{-2}}$
SuperLU	2.19×10^{-2}	5.0×10^{-4}	6.32×10^{-1}	2.1×10^{-2}
Solver Method	L_1 Mean Time	L_1 Std. Dev.	XL_1 Mean Time	XL_1 Std. Dev.
Inverse Jacobian	5.44×10^0	16×10^{-2}	5.35×10^1	2.7×10^0
Linear Solve	3.83×10^0	13×10^{-2}	3.68×10^1	1.9×10^0
Scipy's SPSolve	$\mathbf{3.28 \times 10^0}$	$\mathbf{4.8 \times 10^{-2}}$	3.05×10^1	1.5×10^0
SuperLU	3.31×10^0	9.3×10^{-2}	$\mathbf{3.05 \times 10^1}$	$\mathbf{1.4 \times 10^0}$

Table 6.6: Computational times for networks S_1, M_1, L_1, and XL_1, bold implies fastest method for that network

Both Scipy's SPSolve and SuperLU demonstrate strong performance. The Inverse Jacobian method generally lags behind other methods in terms of both speed and reliability across all network sizes. Standard deviations tend to increase with network size, indicating greater variability in computational times for larger networks. Both the Inverse Jacobian and the Linear Solving method generally show higher mean times, especially as the network size increases. This indicates that larger networks require more computational resources and time to solve. The increase in mean time is more pronounced for these methods compared to Scipy's SPSolve and SuperLU, suggesting that the latter two methods are more efficient in handling larger networks. Furthermore, the standard deviation is relatively low for all solver methods. This implies consistent performance in terms of computation time.

6.7. GMRES

With the analysis of direct iterative methods now known, more insights could be given by the iterative linear system solvers. Therefore, the following sections examine the functioning of GMRES and Bi-CGSTAB, both methods that iteratively solve the system. They are designed for ill-conditioned matrices, so the expectation is that they work efficiently. All upcoming residual plots are included in the

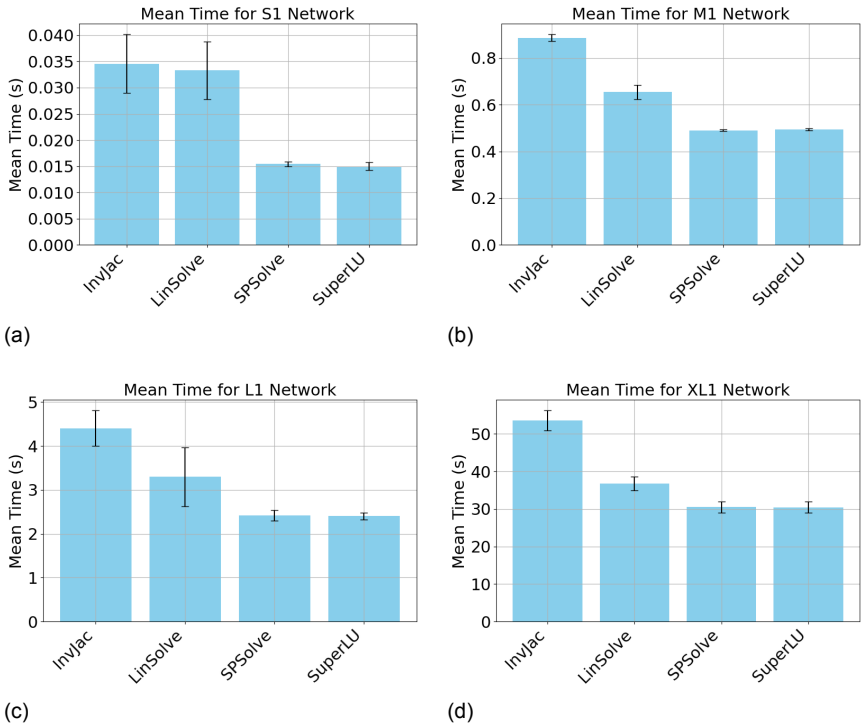


Figure 6.3: Computational times for various networks

appendix B.

	S_1	M_1	L_1	XL_1
#iter	4	11	10	13
CPU time (ms)	4.69×10^1	23.4×10^2	18.7×10^3	20.8×10^4
Wall clock time (ms)	3.01×10^1	6.52×10^2	4.07×10^3	3.58×10^4

Table 6.7: Computational time GMRES method

The GMRES convergence profiles presented for the model systems S_1, M_1, L_1 and XL_1 demonstrate a consistent decrease in the logarithmic residual norm, slightly slower than the SuperLU and the Scipy. GMRES is however using slightly less iterations.

6.8. Bi-CGSTAB

Examining the convergence profiles of the Bi-CGSTAB algorithm across diverse model systems provides insights into its iterative precision and efficiency, compared to that of the GMRES method. This shows how Bi-CGSTAB adapts to different system characteristics.

	S_1	M_1	L_1	XL_1
#iter	4	11	10	13
CPU time (ms)	7.81×10^{-2}	20.2×10^{-1}	18.6×10^0	20.8×10^1
Wall clock time (ms)	3.16×10^{-2}	6.62×10^{-1}	3.97×10^0	3.56×10^1

Table 6.8: Computational time Bi-CGSTAB method

In the Bi-CGSTAB convergence profiles, it is seen how the algorithm progressively improves the accuracy of the solution. For S_1 and M_1, the convergence is steady and direct, leading to highly accurate solutions quickly, highlighting Bi-CGSTAB's efficiency. L_1 shows slight oscillations, typical of

Bi-CGSTAB's method of convergence, possibly due to adjustments in the search direction or preconditioning techniques. XL_1 initially reduces consistently but then temporarily increases around the 6th iteration, likely due to how the preconditioner works or the system's matrix structure. XL_1 shows a slower but continuous reduction in the residual, indicating a gradual convergence that's effective for large-scale systems.

6.9. Comparison of various iterative methods

Comparing the Bi-CGSTAB and GMRES methods, they exhibit very similar behaviour. Both methods demonstrate robustness and efficacy across all of the network systems. Table 6.9 provides a comparison between them.

Solver Method	S_1 Mean Time	S_1 Std. Dev.	M_1 Mean Time	M_1 Std. Dev.
GMRES	$\mathbf{1.83 \times 10^{-2}}$	$\mathbf{2.9 \times 10^{-3}}$	$\mathbf{5.11 \times 10^{-1}}$	$\mathbf{5.7 \times 10^{-2}}$
Bi-CGSTAB	1.88×10^{-2}	5.0×10^{-3}	5.24×10^{-1}	6.3×10^{-2}
Solver Method	L_1 Mean Time	L_1 Std. Dev.	XL_1 Mean Time	XL_1 Std. Dev.
GMRES	$\mathbf{2.75 \times 10^{0}}$	$\mathbf{1.8 \times 10^{-1}}$	$\mathbf{3.92 \times 10^{1}}$	$\mathbf{2.4 \times 10^{0}}$
Bi-CGSTAB	2.76×10^{0}	1.8×10^{-1}	4.02×10^{1}	2.0×10^{0}

Table 6.9: Computational times GMRES versus Bi-CGSTAB, bold implies fastest method for that network

GMRES shows a trend of being slightly faster in mean computational time across the board. GMRES also generally exhibits a lower standard deviation, suggesting it has more consistent performance across iterations, except in the L_1 system where Bi-CGSTAB demonstrates slightly less variability. In practice, their performance is fairly similar. Hence, both GMRES and Bi-CGSTAB are effective iterative solvers.

6.10. Comparison

The analysis now extends to comparing all methods seen so far. For all networks up to the large size, it can be noted that GMRES is the most efficient method. Table 6.10 compares the values for the L_1 and XL_1 networks. Moving to extra large networks, the direct methods become more competitive. For the XL_1 network, as seen in table 6.10, SuperLU and SPSolve appear to be the optimal solvers. While GMRES also provides a fast solution and potentially uses less iterations, it performs slightly worse.

L_1 Network		XL_1 Network	
Solver Method	Mean Time	Solver Method	Mean Time
Inverse Jacobian	5.44×10^{0}	Inverse Jacobian	5.35×10^{1}
Linear Solve	3.83×10^{0}	Linear Solve	3.68×10^{1}
Scipy's SPSolve	3.28×10^{0}	Scipy's SPSolve	3.05×10^{1}
SuperLU	3.31×10^{0}	SuperLU	$\mathbf{3.05 \times 10^{1}}$
GMRES	$\mathbf{2.75 \times 10^{0}}$	GMRES	3.92×10^{1}
Bi-CGSTAB	2.76×10^{0}	Bi-CGSTAB	4.02×10^{1}

Table 6.10: Computational times for the L_1 and XL_1 networks across various solver methods, with the fastest method in bold

The log-log linear fit plot of figure 6.4 shows the scaling behaviours. The data is plotted on a log-log scale, indicating the power-law nature of the computational time scaling with network size. For dense solvers like those in `numpy.linalg`, the plot displays an improved scaling compared to the anticipated $\mathcal{O}(n^3)$. However, for sparse matrix solvers, the plot indicates deviations from the expected theoretical complexities, suggesting that factors such as matrix structure and sparsity patterns significantly influence the actual computational costs. The linear fit in a log-log scale indicates a power-law relationship between network size and computational time, expressed mathematically as:

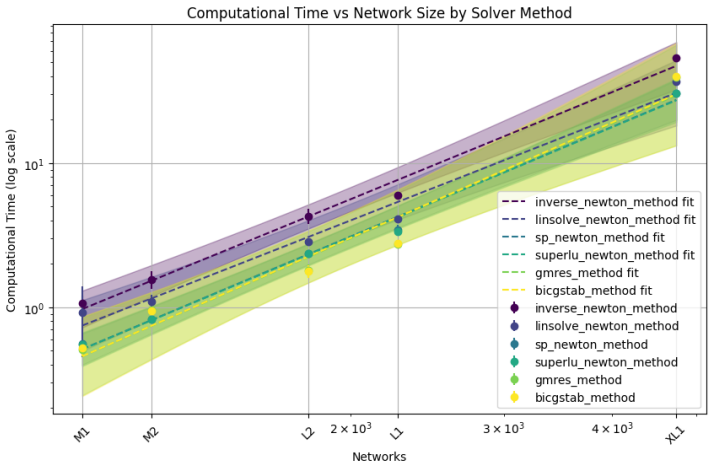


Figure 6.4: Relative difference of solver methods, linear fit

$$\text{Computational Time} \propto \text{Node Size}^{\text{Slope}}$$

Lower slopes indicate better efficiency since the computational time increases more slowly with network size. The overlapping confidence intervals suggest that the performance of the solver methods is quite similar across the range of network sizes tested. The error bars on each point represent the variability or standard deviation of the computational times for each method. Tighter error bars imply more consistency in the solver's performance, whereas wider bars suggest more variability.

The slopes, detailed in Table 6.11, suggest similar growth patterns across various methods. The Inverse Jacobian method displays a slope of 2.46, indicating moderate complexity growth with increasing problem size. Relative to the scaling behaviour typical in methods involving matrix inversion operations (typically with a $\mathcal{O}(n^3)$ scaling), they perform better. The Linear Solve method, with a slope of 2.36, shows slightly better scalability compared to the Inverse Jacobian method. Both SPSolve and SuperLU exhibit slopes above 2.5, implying a slightly steeper rise in computational demands with growing problem size. Conversely, the iterative GMRES and Bi-CGSTAB methods display slopes of 2.66 and 2.64, respectively. Although these values are slightly higher, the overall differences in slope among methods are not significantly pronounced, suggesting comparable sensitivity to increases in problem size across the evaluated techniques.

Solver Method	Slope of Log-Log Regression
Inverse Jacobian	2.464074
Linear Solve	2.357803
Scipy's SPSolve	2.525384
SuperLU	2.522780
GMRES	2.662067
Bi-CGSTAB	2.640319

Table 6.11: Slope of Log-Log Regression for Different Solver Methods

In conclusion, the log-log regression analysis emphasises the critical role of solver selection based on problem-specific characteristics and size. For significantly scaling matrices, methods with lower slopes might be preferable. Conversely, for smaller or matrix property-sensitive problems, such as sparsity or condition number, GMRES or Bi-CGSTAB could still be advantageous despite their scaling behaviour.

7

Further optimizing methods

The upcoming chapter explores the role of initial conditions in the Newton-Raphson method. It examines random and strategic initialization methods in section 7.1. Additionally, the chapter investigates whether evaluations of the Jacobian could be avoided, by using previous iterations in section 7.2. Afterwards, section 7.3 explores the potential errors of this report, and suggests how these could be omitted in the further research.

7.1. The initial conditions

The choice of initial conditions plays a crucial role in the convergence and efficiency of Newton-Raphson's method. Randomly chosen initial values are often used in practical implementations of the Newton-Raphson method. While this approach is convenient, it may not always yield satisfactory results. The convergence of the method heavily depends on the distance of the initial guess to the actual root. Random initialization can lead to scenarios where the initial guess is too far from the root, resulting in slow convergence or even divergence of the algorithm.

Rather than employing randomised vectors, an alternative approach involves generating initial vectors of constants. Three distinct constant vectors are examined: one where all values are set to one, another with values of ten, and a third with values of one hundred. This strategy helps mitigate the computational costs linked to random input vectors, which, though relatively minor, remain significant. Additionally, using constant values eliminates the possibility of encountering outlier values.

Figure 7.1 displays the mean computational times for various solver methods when initialised with different types of vectors. The mean time increases as the value of the constants increases, which suggests that larger constants might be further from the actual solution. The results demonstrate that both randomised and constant vectors, when scaled by a factor of one hundred, exhibit a notable increase in computational time. Conversely, the disparity in computational time between scaling by a factor of one and ten is less pronounced. Notably, the error bars associated with direct computational methods are consistently similar and appear narrower compared to those of iterative methods. Given the slight difference between randomised and constant input vectors, it is unlikely to yield substantial computational efficiency gains in further research on optimizing initial values. Based on this analysis, it is safe to conclude that choosing any vector with values below ten will suffice. Opting for a vector filled with ones is particularly advantageous due to its simplicity of implementation.

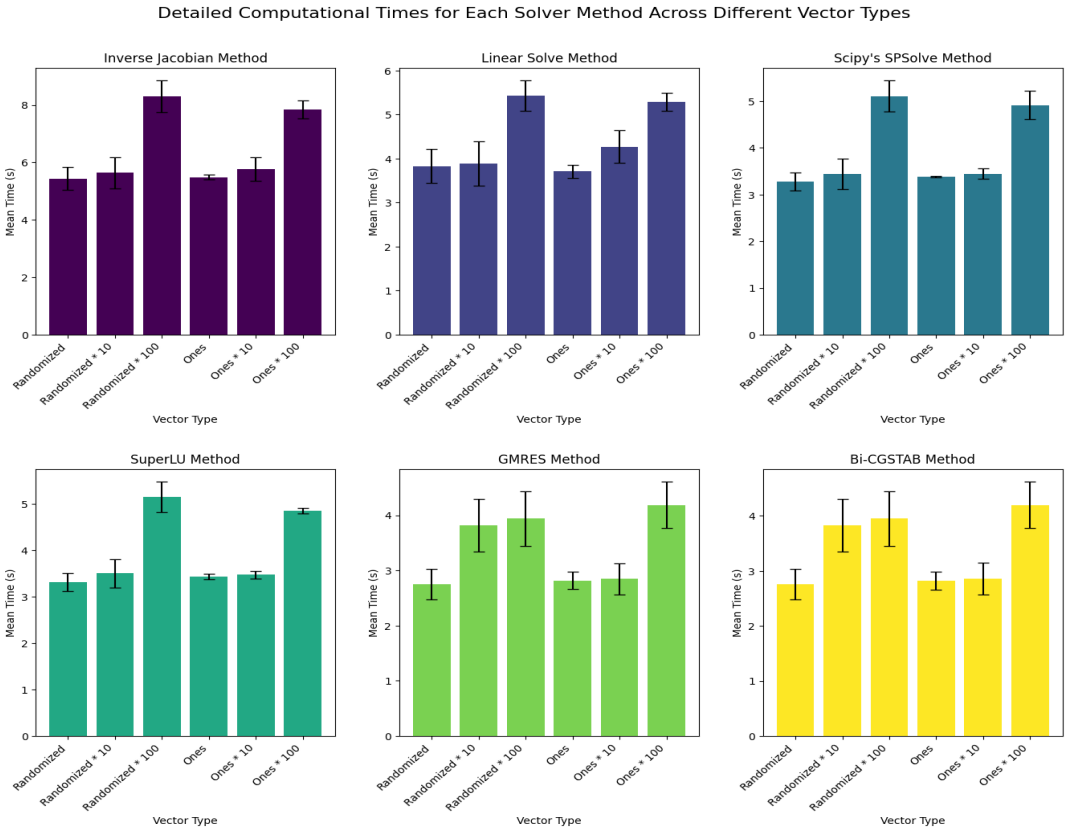


Figure 7.1: Different input vectors compared for the various methods

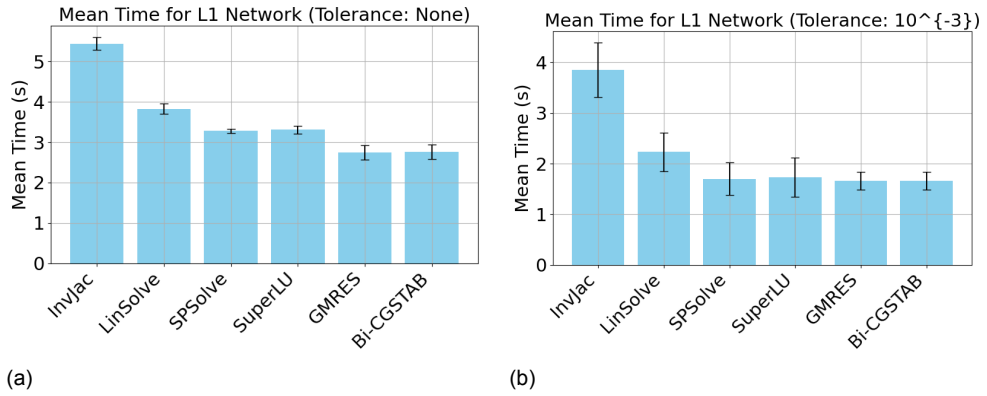
7.2. Freezing the Jacobian or evaluation

In the process of using Newton-Raphson, evaluating the Jacobian matrix at each iteration can be computationally expensive, especially for large-scale problems. However, if the sparsity pattern of the Jacobian matrix remains constant across iterations, it is unnecessary to recompute the entire Jacobian matrix at every step. Instead, reusing the Jacobian can significantly accelerate the convergence of the solver. This could make the optimization process more efficient and scalable for complex nonlinear systems.

The function evaluation in every Newton iteration is also requires adequate computational efforts, but is needed to be precise. Even minor changes in the function's value can destabilise this approximation, causing divergence or slowing down convergence. Thus, reevaluating the function at each iteration is essential to uphold the accuracy of the linear model.

As seen in table 7.1, the efficiency of different solver methods varies with tolerance levels. The values are all given in seconds, with a significant value of $\times 10^0$. InvJac method starts as the least efficient but shows a consistent decrease in mean time with increasing tolerance. Linear Solve and SuperLU improve with higher tolerances, with SuperLU slightly better. SPSolve emerges as the most efficient solver, particularly with looser tolerances, possibly due to its effective exploitation of system sparsity. Iterative methods like GMRES and Bi-CGSTAB also exhibit a modest decrease in mean time with increased tolerance, indicating their robustness across different accuracy requirements.

Figures 7.2 showcases the mean computation times for various methods applied to the L_1 network with standard deviation as error bars, comparing no threshold and a threshold of10^{-3}. Similar plots for the other thresholds are to be found in appendix B. As the tolerance becomes less strict (from 10^{-7} to 10^{-3}), the mean time required by each solver decreases. SPSolve and SuperLU obtain smaller error bars across all tolerances, indicating stable performance. Conversely, InvJac shows larger error bars, particularly at tighter tolerances, suggesting greater variability in its performance. The plots in figure



(a) (b)

Figure 7.2: Reusing Jacobian evaluations, (a) without threshold, (b) with a threshold of 10^{-3}

Solver	Tolerance					
	None	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Inverse Jac	5.44 ± 0.16	5.14 ± 0.50	4.93 ± 0.36	4.69 ± 0.81	4.56 ± 0.73	3.85 ± 0.54
Linear Solve	3.83 ± 0.13	3.54 ± 0.27	3.35 ± 0.63	3.12 ± 0.29	3.00 ± 0.30	2.23 ± 0.38
SPSolve	3.28 ± 0.048	2.94 ± 0.23	2.71 ± 0.22	2.44 ± 0.32	2.29 ± 0.16	1.70 ± 0.32
SuperLU	3.31 ± 0.093	3.00 ± 0.23	2.78 ± 0.21	2.53 ± 0.32	2.39 ± 0.17	1.73 ± 0.39
GMRES	2.75 ± 0.18	2.66 ± 0.18	2.51 ± 0.18	2.12 ± 0.18	2.29 ± 0.18	1.66 ± 0.18
Bi-CGSTAB	2.76 ± 0.18	2.68 ± 0.18	2.52 ± 0.18	2.14 ± 0.18	2.29 ± 0.18	1.66 ± 0.18

Table 7.1: Solver Performance Comparison for Network L_1

7.2 indicate that for the L_1 network, sparse matrix solvers like SPSolve and SuperLU offer superior performance, particularly with some tolerance. Also the iterative methods demonstrate consistent strong performance across different tolerance levels.

7.3. Potential errors of this research

While aiming for a fair comparison among all methods, factors may still influence the results, underscoring the importance of acknowledging and attempting to mitigate them in future research.

7.3.1. More networks

This report analysed the networks S_1, S_2, M_1, M_2, L_1, L_2 and XL_1. This is a very limited number of networks, and therefore quite bold to make strong conclusions out of the data that is obtained. The patterns that are established are likely to also occur in the other networks, but the conclusions would be more grounded if more networks would be examined.

In order to get an even better view on how well the methods work, the networks could potentially be extrapolated to create more data points. That way, larger and smaller networks can be designed and evaluated. There are many ways to do this, but they fall outside of the scope of this research.

7.3.2. Randomness

In this study, new random vectors are generated each time the Newton-Raphson method is applied. This means that the methods are tested on similar yet different inputs. As discussed in section 7.1, the initialization is not expected to significantly alter the outcome but is worth noting. For future research, it is advisable to evaluate various methods using the same vectors, even if randomization is desired.

Furthermore, paying attention to the random seed is vital. In this study, random input arrays were initially constructed with values between zero and one for the values in chapter 6. Later, the random seed was set to 42, resulting in values that no longer fell within these boundaries, generating significantly different outcomes. Eventually, the random array was adjusted to produce entries based on random seed 42, maintaining values between 0 and 1. Understanding the initial values is crucial and

requires careful consideration.

7.3.3. Conversion to sparse matrix format

All functions, except for the Inverse Jacobian and the Linear Solve, operate on matrices provided in sparse formats. As the conversion process may also consume some time, the reported time appears higher than the actual time taken for linear solving. While this discrepancy is assumed to be minor, it remains notable for accuracy.

7.3.4. Fill factor

In the Bi-CGSSTAB method, the fill factor in the Python implementation is set to 25. This is because a singular matrix arised in the default settings. A larger fill factor in a computational context refers to the increased density of non-zero elements in a matrix. However, this enlarged fill factor may also imply longer computational times since more non-zero elements need to be stored and processed. Further research could be to change other library parameters, as discussed in chapter 5.

7.3.5. Internet connection

As Gradyent operates on virtual machines that rely on internet connectivity to function, it wasn't feasible to disconnect from the internet while running the code, as the software environment resides within the virtual machine. This limitation may have led to increased computational times. Therefore, it is recommended to restructure the software architecture, if feasible, to allow for offline operation during code execution.

8

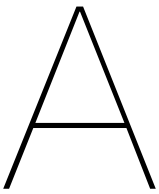
Conclusion

This report investigates ways to enhance Gradyent's implementation of the Newton-Raphson method for solving rootfinding functions. It begins by highlighting the significance of the linear solving process within the Newton iterations. By examining the structure of the networks' Jacobian matrices, opportunities for more efficient computations are identified, particularly by capitalising on their nonsymmetry and sparsity. The report delves into strategies for smart implementation of this information.

SuperLU and Scipy emerge as the fastest solvers for the networks up to sizes of roughly $(1,000 \times 1,000)$. For larger networks, iterative methods like GMRES and Bi-CGSTAB perform better than the direct linear solving methods. These methods are conveniently accessible through user-friendly Python packages. This report also highlights the impact of initial conditions on function behaviour, recommending input vectors closer to zero over those closer to one hundred. Additionally, implementing a threshold for reusing similar Jacobian evaluations can significantly boost method performance.

However, certain challenges arise due to various reasons. The Newton-Raphson method sometimes failed to converge in scenarios where the Jacobian matrix was nearly singular or the initial guess was distant from the true solution. Sharp function discontinuities also hindered convergence. Similarly, the PARDISO solver exhibited non-convergence in larger network systems due to insufficient solver parameters or preconditioning, alongside stability issues in LU factorization. The effectiveness of iterative methods like GMRES and Bi-CGSTAB was notably dependent on the application of well-tuned preconditioners, suggesting a critical need for robust preconditioning strategies.

Future research could concentrate on refining method parameters and possibly crafting a hybrid algorithm that merges a reliable iterative solver with an effective linear solving approach. This could involve devising clever methods to leverage the fact that each edge can only connect two nodes, leading to specific matrix structures as discussed in chapter 4. Therefore, preconditioning techniques for district heating systems could be explored in greater depth. Additionally, parallelization techniques are crucial, considering the observed disparity between wall and CPU times. Exploring these options has the potential to improve computational efficiency.



Python implementation

A.1. Packages

```
import time
import pickle as pkl
import joblib
import matplotlib.pyplot as plt
import numpy as np
import pypardiso
import scipy.sparse
np.random.seed(42)
```

A.2. Functions

```
def newton_raphson_solver(f, input_array, newton_method_func, max_iter=100, tol=1e-7):
    x_n = input_array
    residual_history = []
    start_time = time.time()

    for i in range(max_iter):
        eval = f.evaluate(x_n)
        residual = np.linalg.norm(eval, ord=np.inf)
        residual_history.append(residual)

        if residual < tol:
            print("Converged in {} iterations".format(i))
            break

        j = f.evaluate_jacobian(x_n)
        x_n = newton_method_func(x_n, j, eval)

    end_time = time.time()
    duration = end_time - start_time

    if residual >= tol:
        residual_history.append(residual)

    return x_n, duration, residual_history

def THRESHOLD_newton_raphson_solver(f, input_array, newton_method_func, max_iter=100,
tol=1e-7, eval_tol=1e-3):
```

```
    x_n = input_array
    residual_history = []
    start_time = time.time()
    prev_j = None
    prev_x_n = None

    for i in range(max_iter):
        eval = f.evaluate(x_n)  # Always evaluate the function
        if prev_x_n is not None and np.linalg.norm(x_n - prev_x_n, ord=np.inf)
        < eval_tol:
            j = prev_j
        else:
            j = f.evaluate_jacobian(x_n)
            prev_j = j
            prev_x_n = x_n

        residual = np.linalg.norm(eval, ord=np.inf)
        residual_history.append(residual)

        if residual < tol:
            break

        x_n = newton_method_func(x_n, j, eval)

    end_time = time.time()
    duration = end_time - start_time
    return x_n, duration, residual_history

def inverse_newton_method(x_n, jacobian, eval):
    return x_n - np.linalg.inv(jacobian) @ eval

def linsolve_newton_method(x_n, jacobian, evaluation):
    return x_n - np.linalg.solve(jacobian, evaluation)

def sp_newton_method(x_n, jacobian, evaluation):
    j_sparse = scipy.sparse.csc_matrix(jacobian)
    return x_n - spsolve(j_sparse, evaluation)

def sp_gmres_method(x_n, jacobian, evaluation, tol= 1e-9):
    Mx= scipy.sparse.linalg.spilu(jacobian, fill_factor= 10).solve
    M= scipy.sparse.linalg.LinearOperator(jacobian.shape, Mx)
    j_sparse = scipy.sparse.csc_matrix(jacobian)
    x, _ = gmres(j_sparse, evaluation, M=M, tol= tol, atol= tol)
    return x_n - x

def sp_bicgstab_method(x_n, jacobian, evaluation, tol=1e-9):
    Mx= scipy.sparse.linalg.spilu(jacobian, fill_factor=25).solve
    M= scipy.sparse.linalg.LinearOperator(jacobian.shape, Mx)
    j_sparse = scipy.sparse.csc_matrix(jacobian)
    x, _ = bicgstab(j_sparse, evaluation, M=M, rtol= tol, atol= tol)
    return x_n - x

def par_newton_method(x_n, jacobian, evaluation):
    eval_np = evaluation._value if hasattr(evaluation, '_value') else evaluation
    j_np = np.array(jacobian)
    j_csc = scipy.sparse.csc_matrix(j_np)
    return x_n - par_spsolve(j_csc, eval_np)
```

```
def UMFpar_newton_method(x_n, jacobian, evaluation):
    eval_np = evaluation._value if hasattr(evaluation, '_value') else evaluation
    j_np = np.array(jacobian)
    j_csc = scipy.sparse.csc_matrix(j_np)
    return x_n - par_spsolve(j_csc, eval_np, use_umfpack=False)

def superlu_newton_method(x_n, jacobian, evaluation):
    j_sparse = scipy.sparse.csc_matrix(jacobian)
    solution = scipy.sparse.linalg.spsolve(j_sparse, evaluation)
    return x_n - solution

def confidence_interval(x, y, x_fit, confidence=0.95):
    _, intercept, r_value, _, std_err = linregress(np.log(x), np.log(y))
    y_fit = np.exp(intercept + slope * np.log(x_fit))
    n = len(x)
    t = scipy.stats.t.ppf((1 + confidence) / 2., n - 2)
    ci = t * std_err * np.sqrt(1/n + (np.log(x_fit) - np.log(x).mean())**2 /
    ((np.log(x) - np.log(x).mean())**2).sum())
    return y_fit, np.exp(np.log(y_fit) - ci), np.exp(np.log(y_fit) + ci)
```

A.3. Program

Listing A.1: Example Python code

```
tolerances = [10**-3, 10**-4, 10**-5, 10**-6, 10**-7]
networks = ["S0", "S1", "S2", "M1", "M2", "L1", "L2", "XL1"]
solvers = {
    "inverse_newton_method": inverse_newton_method,
    "linsolve_newton_method": linsolve_newton_method,
    "sp_newton_method": sp_newton_method,
    "superlu_newton_method":  superlu_newton_method
    "gmres_method": sp_gmres_method,
    "bicgstab_method": sp_bicgstab_method,
    "par_newton_method": par_newton_method,
    "UMFpar": UMFpar_newton_method,
}

results = []

for tol in tolerances:
    for network in networks:
        if network == "S0":
            residual_function_file_path = "test_network_residual_function_
␣␣␣␣␣␣␣␣␣␣␣␣untransformed.pkl"
        else:
            residual_function_file_path = f"networks/{network}/residual_function
␣␣␣␣␣␣␣␣␣␣␣␣_full.pkl.z"

        with open(residual_function_file_path, "rb") as residual_function_file:
            residual_function = joblib.load(residual_function_file)

        input_array_shape = residual_function.args[0].shape
        random=[]
        for i in range(11):
            r= np.random.random(low= 0.0, high= 1.0, size= input_array_shape) * 10
            residual_function.evaluate(r)
```

```
            residual_function.evaluate_jacobian(r)
            random.append(r)

        for i in range(1, 11):  # Start from 1 to avoid initial cold start
            random_input_array = random[i]
            for solver_name, solver_func in solvers.items():
                _, exec_time, _ = newton_raphson_solver(residual_function,
                random_input_array, solver_func, tol=tol)
                timesave[solver_name].append(exec_time)

        timesave = {solver_name: [] for solver_name in solvers}
        residualhistory = {solver_name: [] for solver_name in solvers}

        for solver_name, times in timesave.items():
            mean_time = np.mean(times)
            std_time = np.std(times)
            results.append({
                "Network": network,
                "Solver": solver_name,
                "Tolerance": tol,
                "Mean Time": mean_time,
                "Standard Deviation Time": std_time,
            })

for result in results:
    print(f"Network: {result['Network']}, Solver: {result['Solver']},
    Mean Time: {result['Mean Time']}, Standard Deviation Time:
    {result['Standard Deviation Time']}, Tolerance: {tolerances}")

 # Perform linear regression in log-log space and plot fit
slope, intercept, _, _, _ = linregress(np.log(node_sizes), np.log(mean_times))
slopes.append(slope)

# Calculate linear fit with confidence intervals
x_fit = np.linspace(min(node_sizes), max(node_sizes), 100)
y_fit, lower_bounds, upper_bounds = confidence_interval(node_sizes, mean_times,
x_fit)
```

B

Additional data

B.1. Jacobians

Figure B.1 shows that the structure of the matrix M_1 does not visualize great when plotted without clipping or setting a threshold for low values.

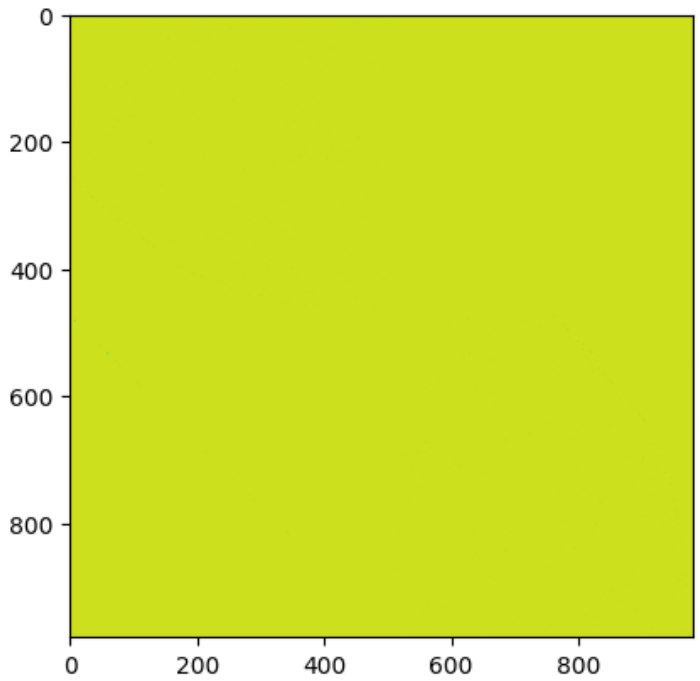


Figure B.1: Jacobian of medium sized network M_1

In figure B.2, the Jacobians plotted after the direct Newton methods are applied are seen. The figures include the Inverse Newton Method, the Linear Solve Newton Method, Scipy's SPSolve Method, PARDISO and the SuperLU method.

Figure B.3 plots the eigenvalues of the networks S_0, S_1, M_1 and L_1. They show similar behaviour as the eigenvalues of XL_1, provided in the main text.

To ensure that the characteristics of the matrices remain throughout the algorithms, a comparison is made with the eigenvalues at the last iteration. The eigenvalues presented in figure B.4 are based on Jacobians evaluated at the approximated root of the systems. This is done for various methods, to ensure the properties remain for them all.

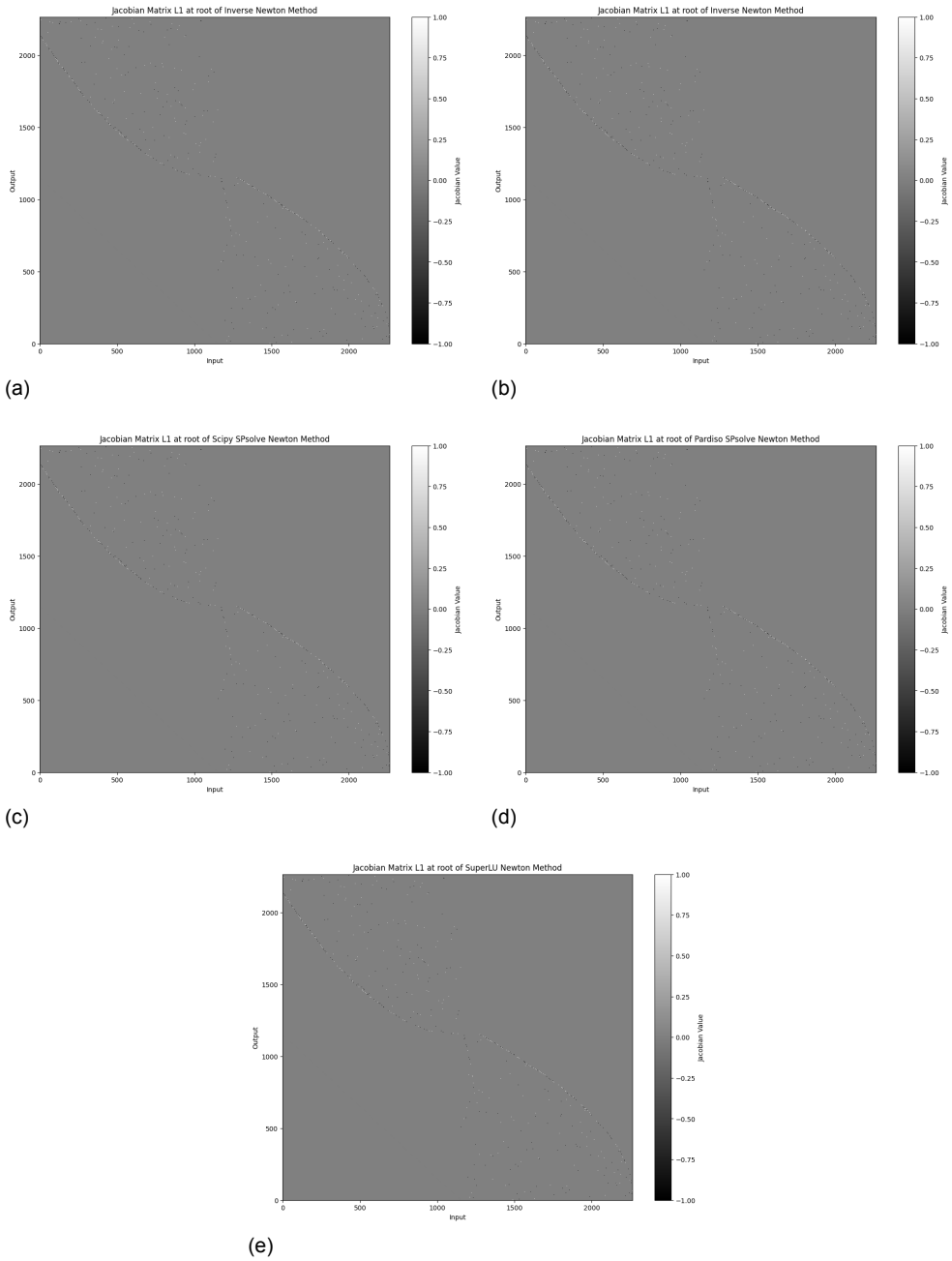


Figure B.2: Jacobians of L_1, evaluated at the final iteration of various Newton methods (Inverse Jacobian, Linear Solve, Scipy's SPSolve, PARDISO, SuperLU)

B.2. Residual versus iteration graphs

The following figures plot the residuals scaled against the iterations, for all sizes of networks. This is done per function, and all graphs that are not supplied in the main text are given here. Figure B.5 represents the plots for the Linear Solve, figure B.6 for SPSolve, figure B.7 for SuperLU, figure B.10 for GMRES and lastly, figure B.11 for Bi-CGSTAB.

	S_1	M_1	L_1	XL_1
#iter	5	12	10	13
CPU time (ms)	14.7×10^2	6.17×10^3	34.5×10^3	3.62×10^4
Wall clock time (ms)	2.80×10^2	1.22×10^3	5.80×10^3	5.86×10^4
#iter	5	12	10	13
CPU time (ms)	13.6×10^2	9.44×10^3	27.2×10^3	2.49×10^4
Wall clock time (ms)	2.87×10^2	1.70×10^3	4.62×10^3	3.88×10^4
#iter	5	12	10	13
CPU time (ms)	3.12×10^1	20.9×10^2	18.3×10^3	2.04×10^4
Wall clock time (ms)	2.56×10^1	5.57×10^2	3.42×10^3	3.29×10^4
#iter	5	12	10	-
CPU time (ms)	14.1×10^1	35.1×10^2	20.2×10^4	-
Wall clock time (ms)	3.72×10^1	6.68×10^2	3.59×10^4	-
#iter	6	14	12	13
CPU time (ms)	7.81×10^1	26.4×10^2	21.7×10^3	22.3×10^3
Wall clock time (ms)	3.01×10^1	6.73×10^2	4.12×10^3	3.30×10^3
#iter	4	11	10	13
CPU time (ms)	4.69×10^1	23.4×10^2	18.7×10^3	20.8×10^4
Wall clock time (ms)	3.01×10^1	6.52×10^2	4.07×10^3	3.58×10^4
#iter	4	11	10	13
CPU time (ms)	7.81×10^{-2}	20.2×10^{-1}	18.6×10^0	20.8×10^1
Wall clock time (ms)	3.16×10^{-2}	6.62×10^{-1}	3.97×10^0	3.56×10^1

Table B.1: Computational time Inverse Jacobian method

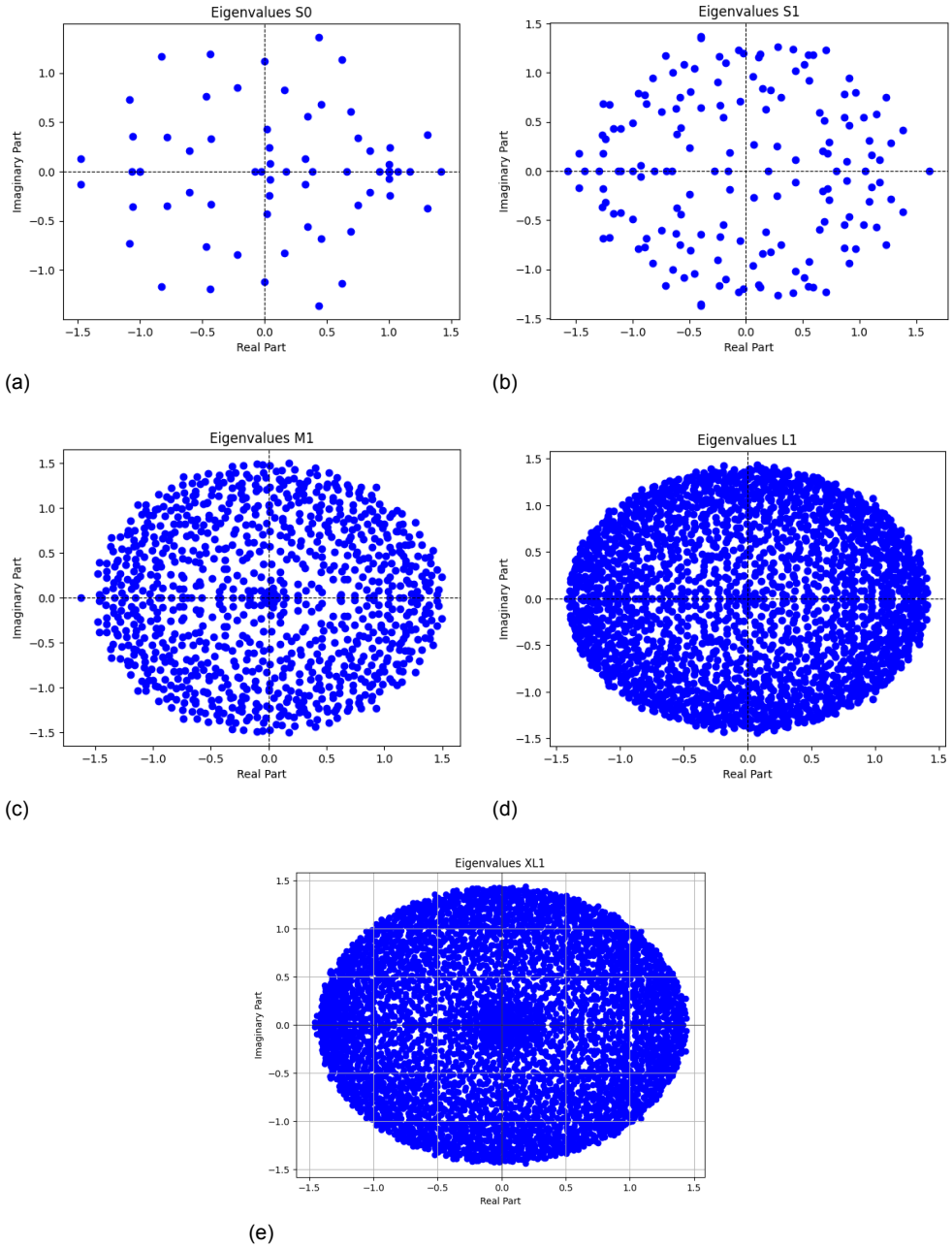
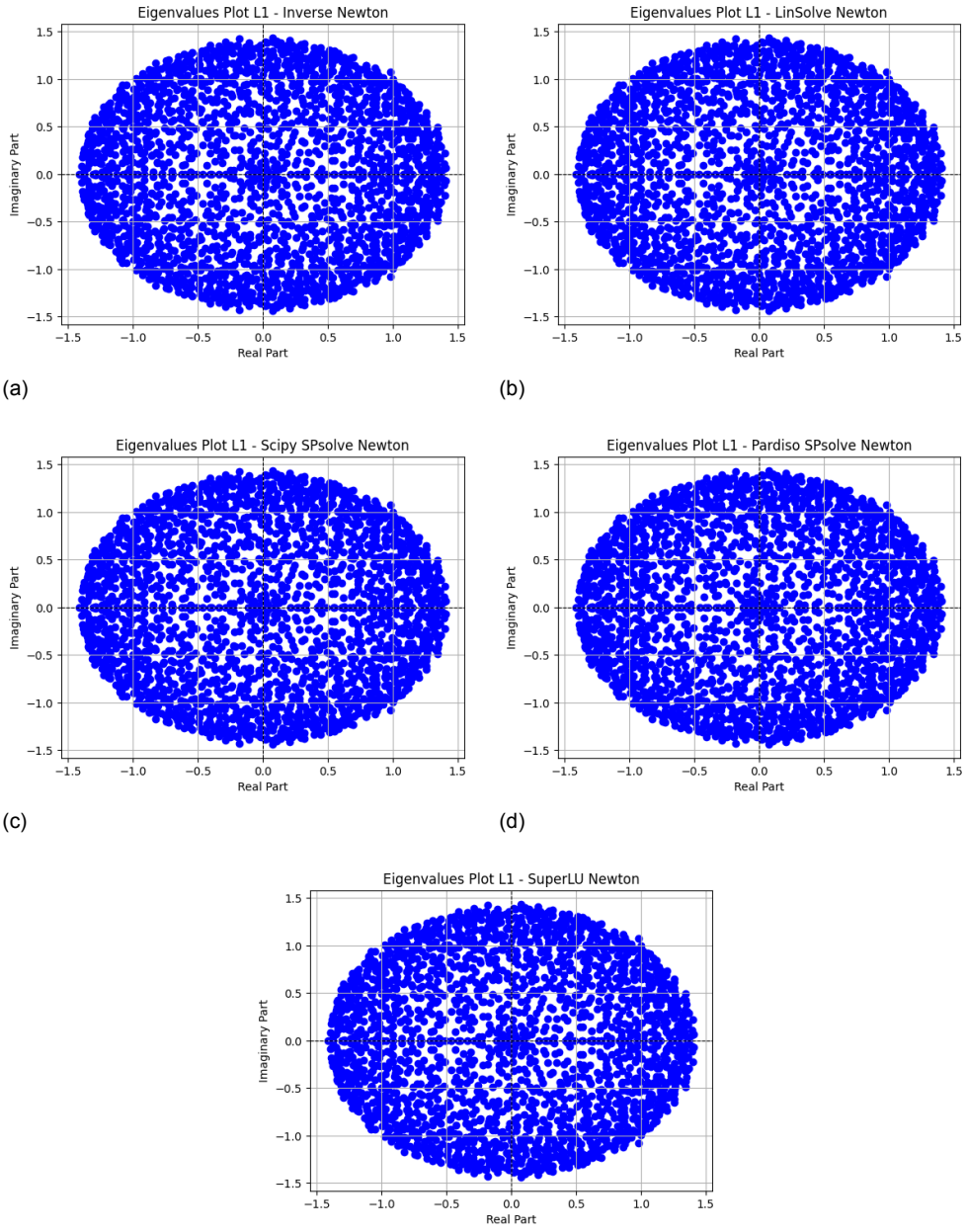


Figure B.3: Eigenvalues of different networks.



(a) (b)

(c) (d)

(e)

Figure B.4: Eigenvalues of L_1, after methods Inverse Jacobian, Linear Solve, Scipy's SPSolve, PARDISO, SuperLU

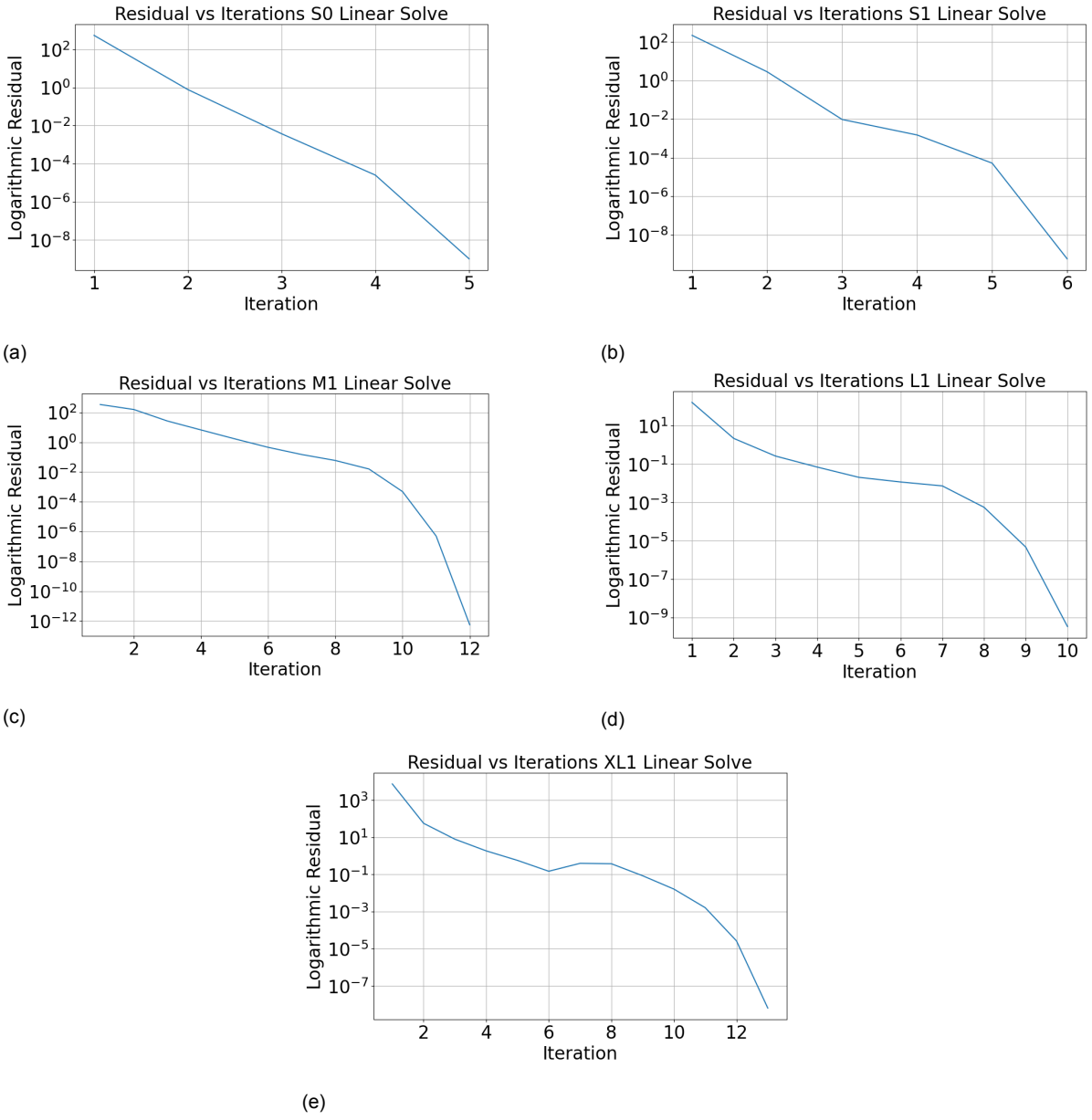
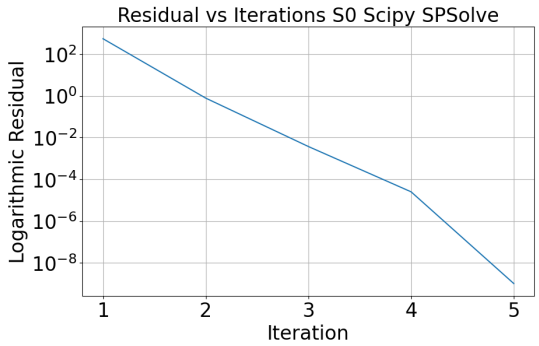
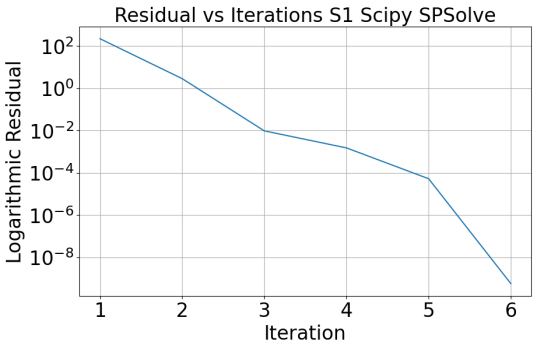


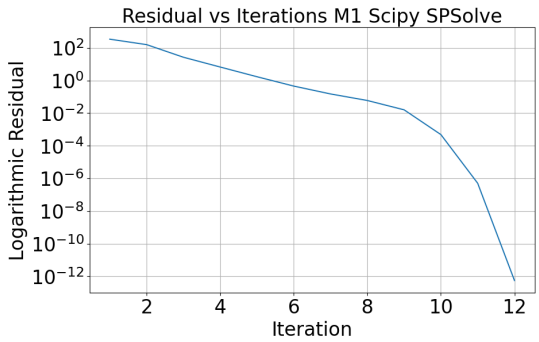
Figure B.5: Convergence of the LinSolve applied directly, on different networks.



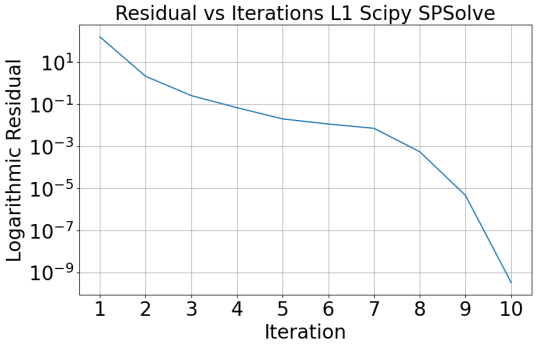
(a)



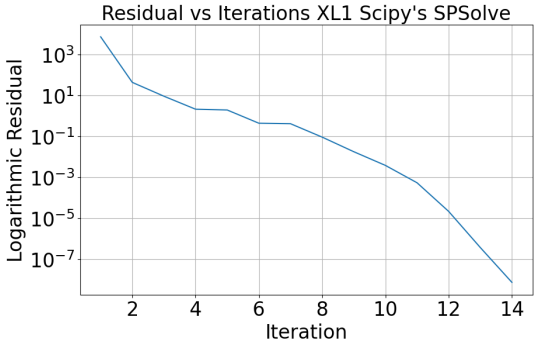
(b)



(c)



(d)



(e)

Figure B.6: Convergence of the Scipy SPSolve applied directly, on different networks.

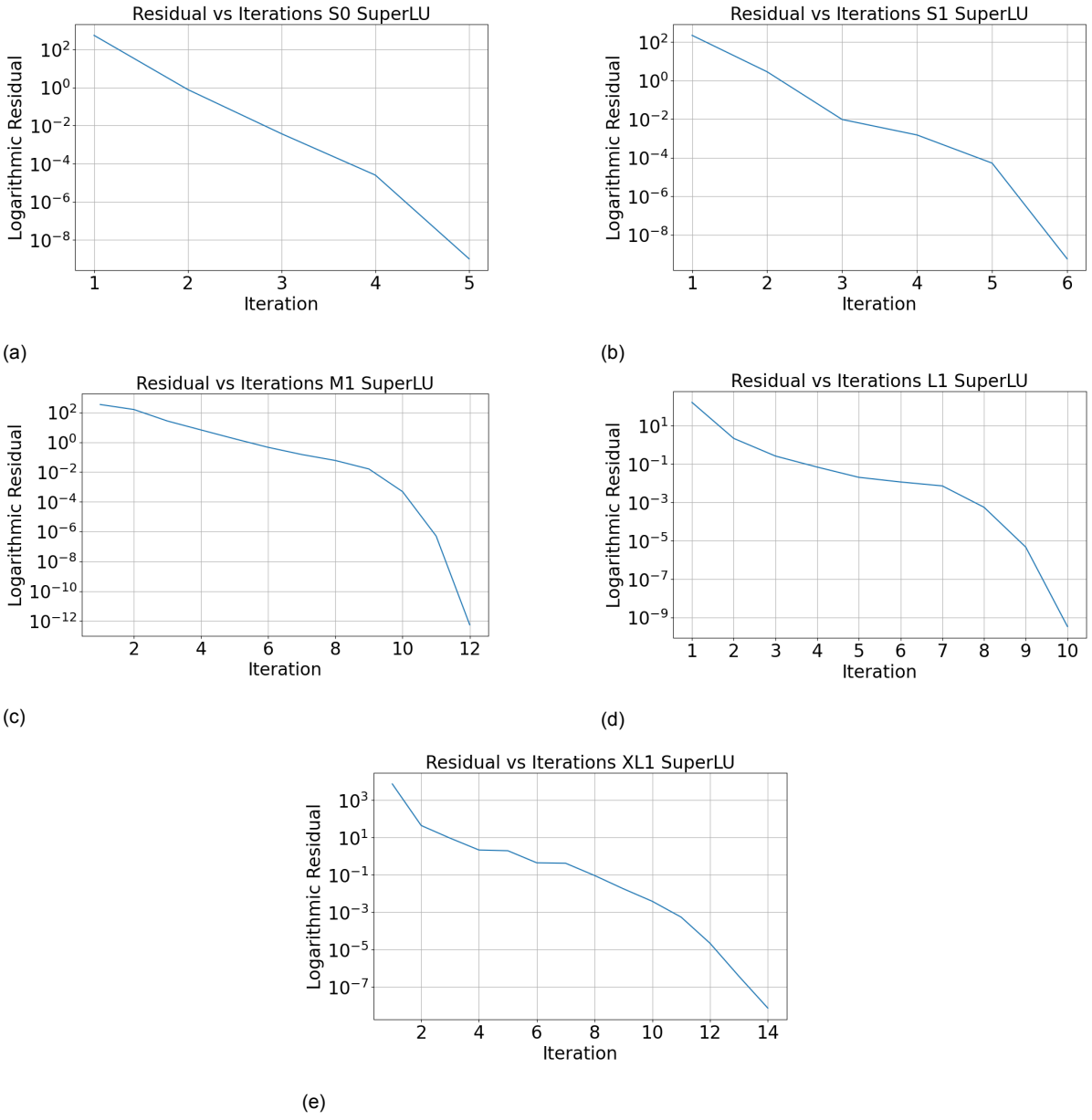
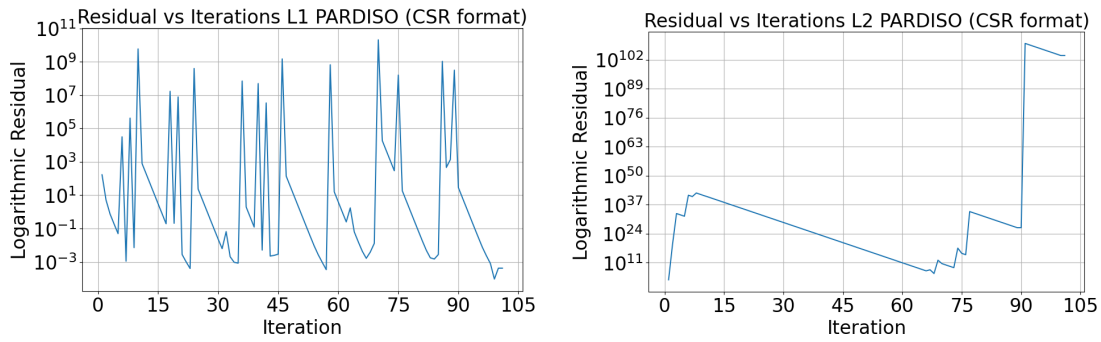
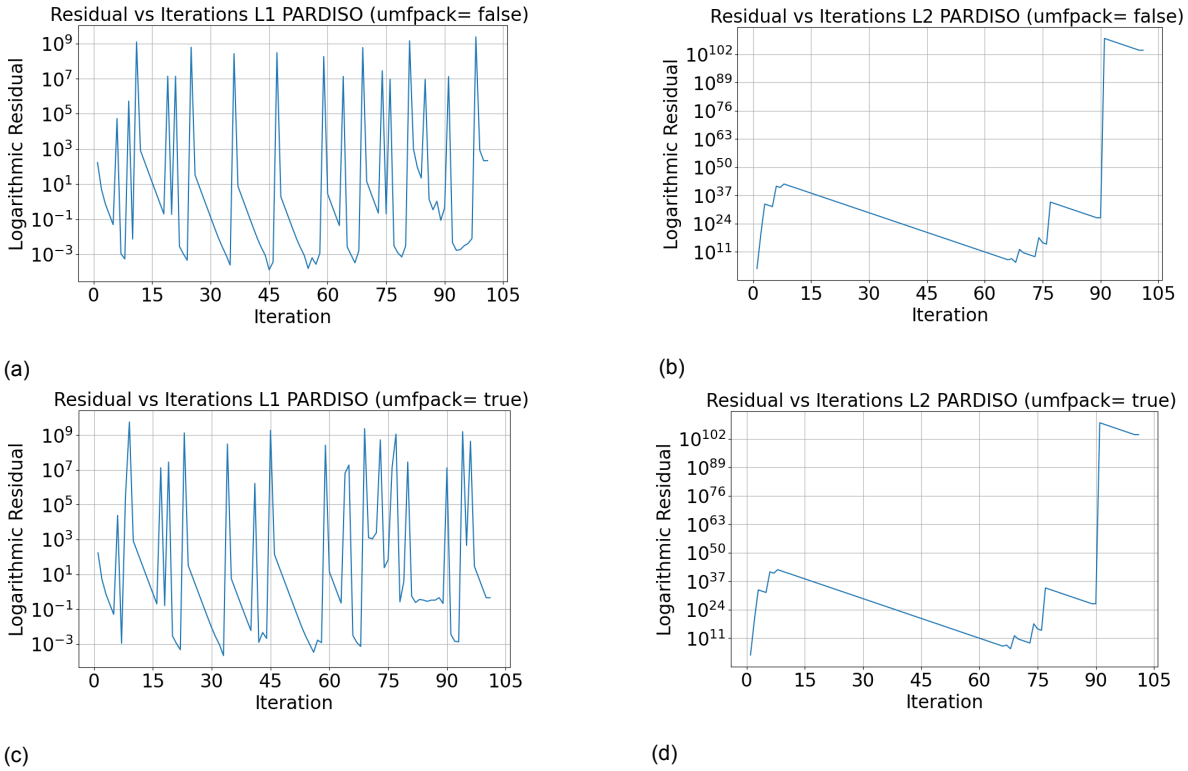


Figure B.7: Convergence of the SuperLU applied directly, on different networks.



(a) (b)

Figure B.8: PARDISO applied to L_1 and L_2, but then in CSR format (CSC is default)



(a) (b)

(c) (d)

Figure B.9: PARDISO applied to L_1 and L_2, differentiating between UMFPACK= True and False

B.3. Linear Regression

As discussed in Chapter 6, the linear fit of the different methods appears to be consistent. To explore the behavior of the fit when considering the smaller networks S_1 and S_2, Figure B.13 depicts the residuals with these networks included. Noticeably, the error bars are less pronounced with the inclusion of more data points. While the increased data aids accuracy, this observation is still noteworthy.

B.4. Initial values

Table B.2: Solver Performance Comparison for Networks S2, M2, and L2

Solver Method	S2 Mean Time	S2 Std. Dev.	M2 Mean Time	M2 Std. Dev.
inverse_newton_method	4.66×10^{-2}	96.7×10^{-4}	14.6×10^{-1}	20.8×10^{-3}
linsolve_newton_method	4.19×10^{-2}	53.5×10^{-4}	10.3×10^{-1}	29.6×10^{-3}
sp_newton_method	2.43×10^{-2}	2.63×10^{-4}	7.96×10^{-1}	13.4×10^{-3}
superlu_newton_method	$\mathbf{2.40 \times 10^{-2}}$	$\mathbf{6.10 \times 10^{-4}}$	$\mathbf{7.95 \times 10^{-1}}$	$\mathbf{9.53 \times 10^{-3}}$
Solver Method	**L2 Mean Time**	**L2 Std. Dev.**		
inverse_newton_method	4.24×10^{0}	3.18×10^{-2}		
linsolve_newton_method	3.62×10^{0}	1.17×10^{-2}		
sp_newton_method	$\mathbf{2.68 \times 10^{0}}$	$\mathbf{1.21 \times 10^{-2}}$		
superlu_newton_method	2.76×10^{0}	1.23×10^{-2}		

The set of figures 7.2 showcases the mean computation times for various Newton-Raphson methods applied to the L_1 network with standard deviation as error bars, across a range of tolerance levels from none to 10^{-7}.

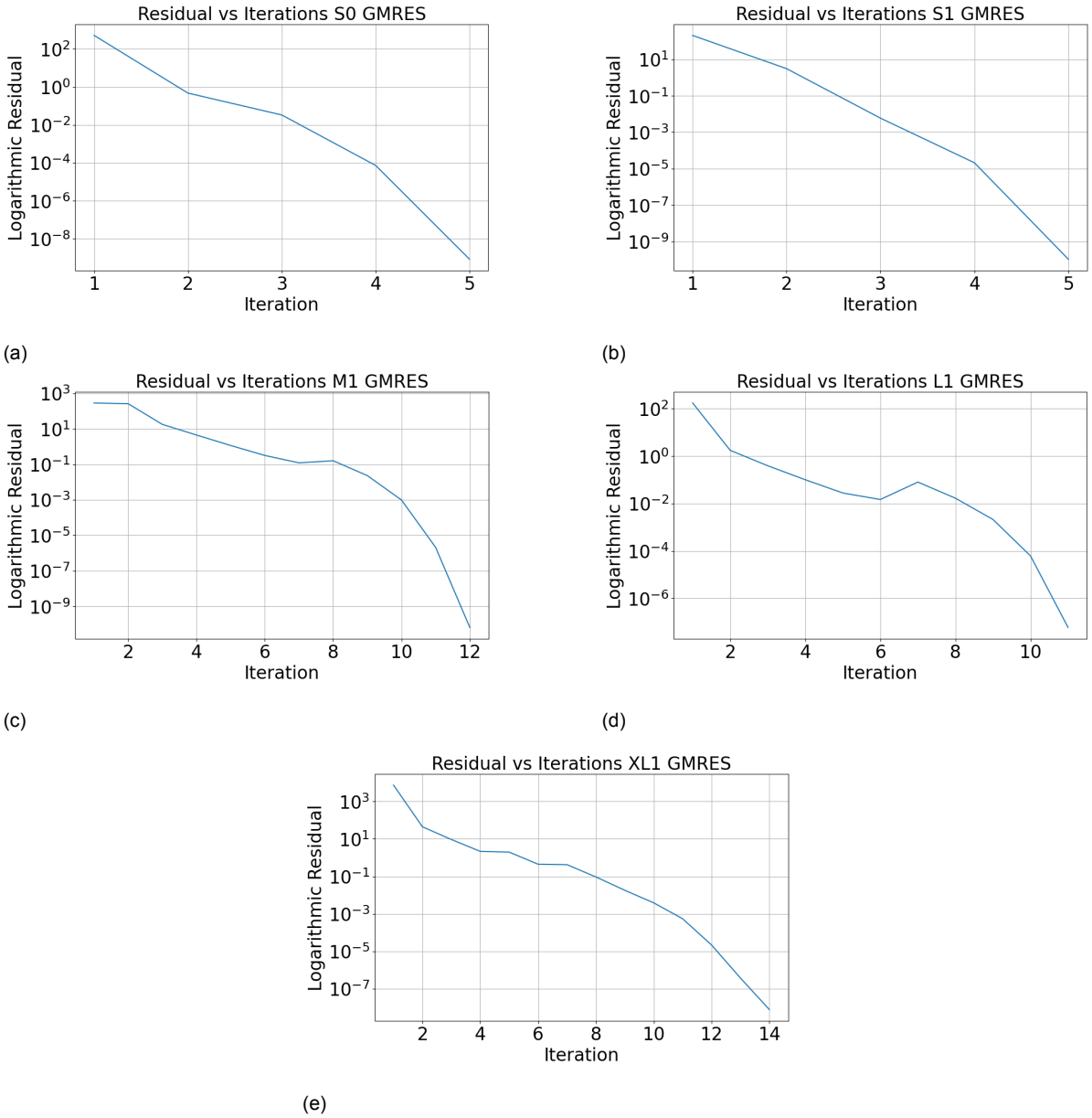


Figure B.10: Convergence of the GMRES applied directly, on different networks.

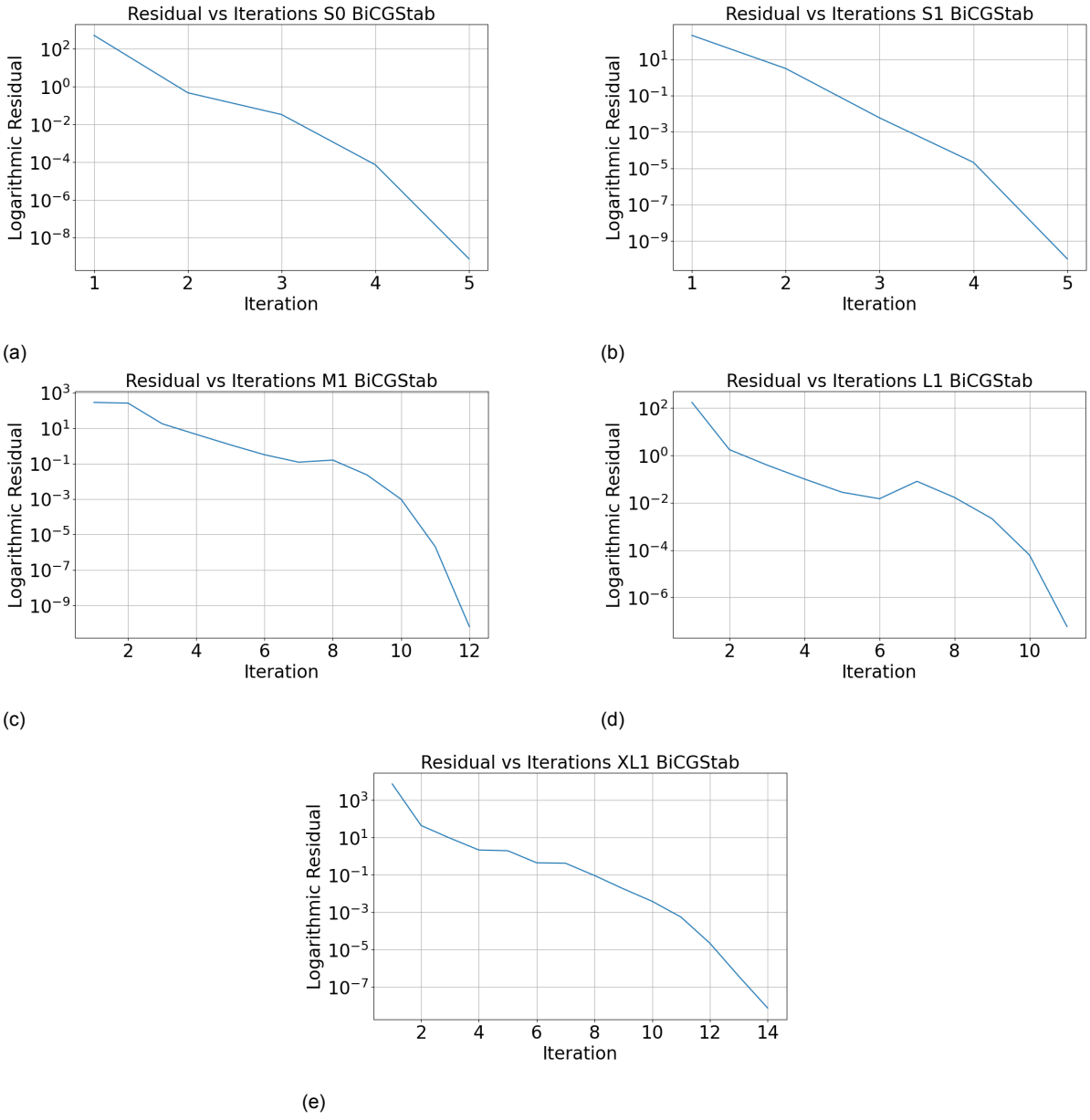


Figure B.11: Convergence of the Bi-CGSTAB applied directly, on different networks.

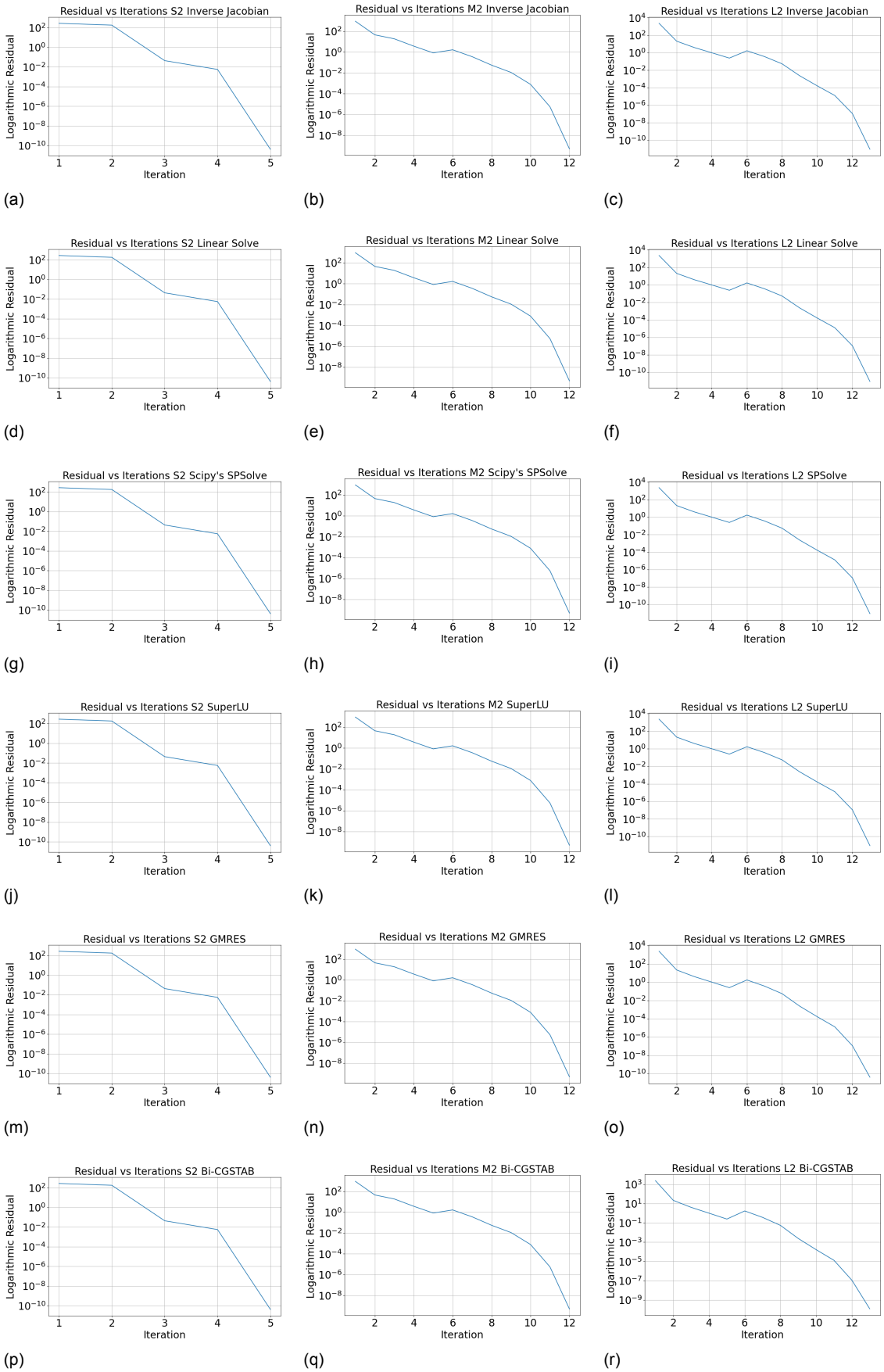


Figure B.12: Residual versus iterations for network solving of S_2, M_2, L_2, for the methods Inverse Jacobian, Linear Solve, Scipy's SPSolve, SuperLU, GMRES and Bi-CGSTAB

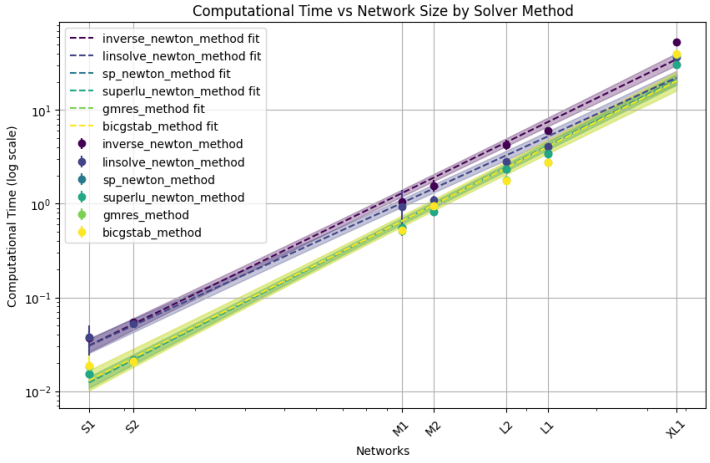


Figure B.13: Linear Regression of all methods combined

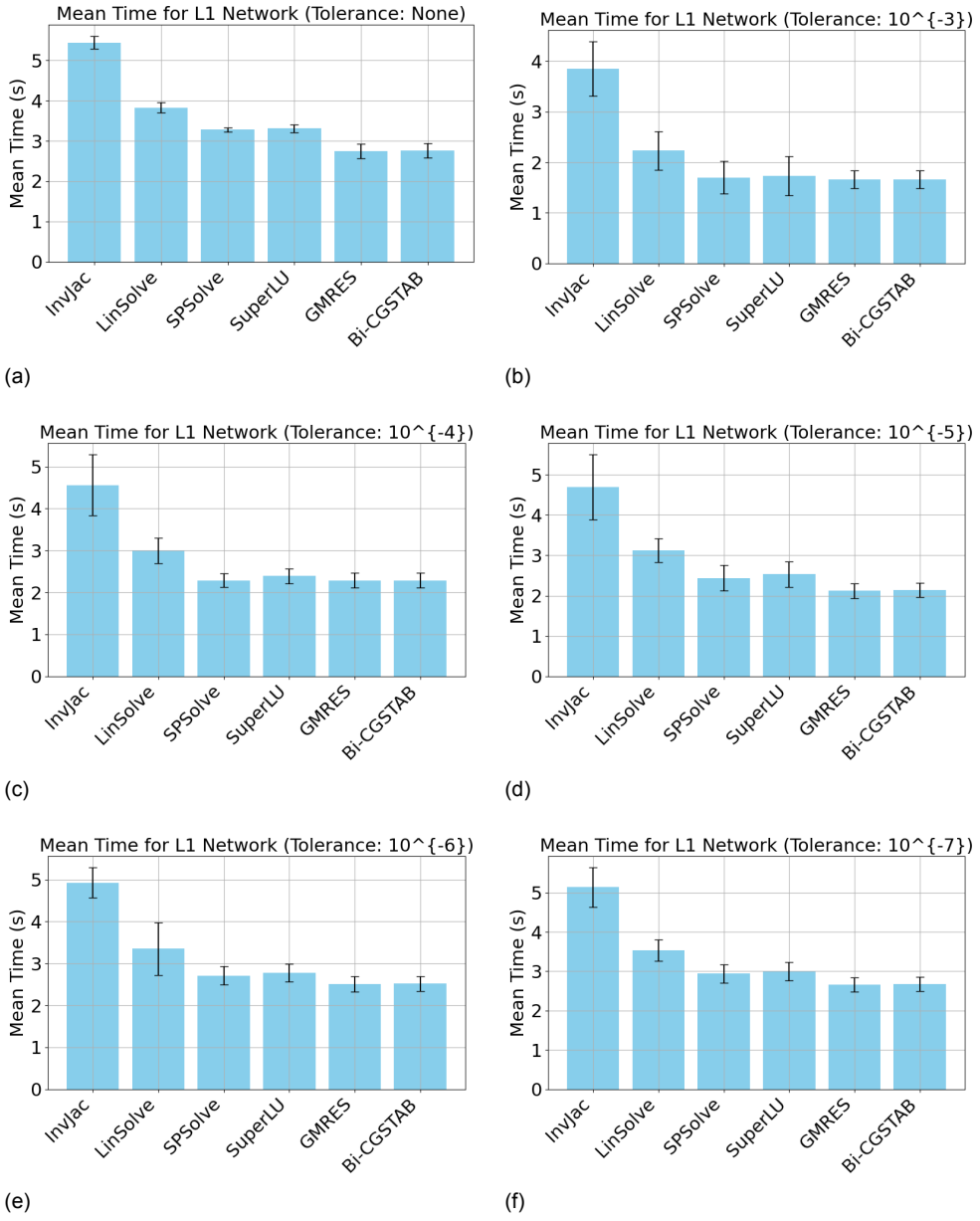


Figure B.14: Reusing Jacobian evaluations for various tolerances

C

Improving the iteration method of Newton-Raphson

Now knowing the numerical mathematical background, and understanding the algorithm, we can start to check ways in which we can improve this. The most obvious way to improve is to see whether Newton-Raphson can be adjusted for a typical heating network. This chapter delves in to the possibilities of adjusting this algorithm, and finding a more efficient way to do so. What methods can be employed to improve its performance, and how do they function? The mathematical basis behind these improvements is explored, along with the scenarios in which they are applicable. Following this, Chapter 3 moves on to discussing techniques for solving linear systems.

Damping and regularization are strategies employed to enhance the stability and performance of the Newton-Raphson algorithm, particularly in scenarios where issues like numerical instability or ill-conditioning may arise (**damping**). After examining these in sections C.1 and C.2, various line searching methods are exploited in section C.3, followed up by examples of accelerators in section C.3.

C.1. Damping

Damping involves introducing a damping factor to control the step size in each iteration of the Newton-Raphson algorithm. This can be especially useful when the algorithm encounters regions with steep slopes or sharp changes, preventing overshooting and ensuring more stable convergence. One common damping strategy is the introduction of a damping parameter, denoted as λ, where the updated iterate is calculated as$x_{n+1} = x_n - \lambda f'(x_n)^{-1} f(x_n)$. Adjusting λ allows for a balance between the Newton step and damping, promoting smoother convergence.

Damping involves multiplying the Jacobian matrix by a damping factor to reduce the step size and control the convergence rate. Damping is implemented as follows;

1. Introduce a damping factor, often denoted as α, into the Newton-Raphson update equation: $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} - \alpha \mathbf{J}^{-1} \mathbf{f}(\mathbf{x}_{\text{old}})$, where J is the Jacobian matrix.
2. The damping factor α is typically between 0 and 1. Smaller values of α lead to smaller steps, stabilizing the iterations.

C.2. Regularization

Regularization is employed to handle ill-conditioned problems or situations where the Hessian matrix is close to singular (**regularization**). By adding a regularization term to the Hessian matrix, the algorithm becomes more robust, and convergence is improved. For instance, Tikhonov regularization adds a multiple of the identity matrix to the Hessian, preventing singularities and stabilizing the Newton-Raphson method in ill-conditioned cases. The regularized Newton step is given by $\mathbf{x_{n+1}} = \mathbf{x_n} - [\mathbf{f''}(\mathbf{x_n}) + \alpha \mathbf{I}]^{-1} \mathbf{f'}(\mathbf{x_n})$, where α is the regularization parameter.

Regularization involves adding a stabilizing term to the system of equations, typically in the form of a penalty term. This term helps prevent divergence or oscillations in the iterations. This is how it works;

1. Modify the original equation to include a regularization term, often proportional to the norm of the solution or its derivatives.
2. The modified equation becomes: $f(x) + \lambda g(x) = 0$, where $f(x)$ is the original function, $g(x)$ is the regularization term, and λ is a regularization parameter.
3. The regularization parameter λ is chosen carefully to balance the trade-off between fitting the data and controlling instability

Both damping and regularization techniques are valuable tools in the Newton-Raphson algorithm's toolbox, providing mechanisms to address challenges associated with steep or ill-conditioned regions. These strategies ensure more robust convergence and extend the algorithm's applicability to a wider range of optimization problems.

These techniques aim to prevent overshooting and oscillations in the iterative process. It is common to use a combination of regularization and damping for improved stability. One can do that as follows. First modify the Newton-Raphson update equation with both a regularization term and a damping factor: $x_{\text{new}} = x_{\text{old}} - \alpha J^{-1}(f(x_{\text{old}}) + \lambda g(x_{\text{old}}))$. Then adjust the damping factor and regularization parameter to achieve the desired balance between stability and convergence.The choice of damping factor and regularization parameter depends on the characteristics of the specific problem. Too much damping or regularization may slow down convergence, while too little may lead to instability. Careful tuning and experimentation are often necessary to find an optimal combination for a given application.

C.3. Line searching methods

Line-searching methods play a crucial role in enhancing the performance of optimization algorithms. These methods focus on efficiently determining the step size along the search direction, ensuring that each iteration moves the algorithm toward the optimal solution. They are further examined in **num-nonlin**. Incorporating line search techniques in the Newton-Raphson algorithm can significantly improve its convergence properties, particularly in cases where the step size influences the algorithm's stability and speed.
For instance, the Armijo-Goldstein rule is a popular line-search method that adjusts the step size based on a sufficient decrease condition, preventing overstepping and facilitating convergence (**armijo**). Another example is the Wolfe conditions, which offer a more sophisticated set of criteria for step size adjustment, balancing the trade-off between speed and stability. By integrating such line-search strategies, the Newton-Raphson algorithm becomes more adaptive and robust, ensuring efficient convergence to the desired solution (**num-nonlin**).

C.3.1. Interpolated Line Search

The interpolated line search integrates the Newton-Raphson update direction (ΔU) with an adaptive step size (η). Initially, one computes (ΔU) using the regular Newton-Raphson method. However, instead of directly updating ($U_{n+1} = U_n + \Delta U$), the optimal step size (η) is determined by evaluating the residual function at various points along the direction (ΔU) to minimize the residual.

This approach can enhance accuracy by adjusting the step size based on the residual landscape, adapting to local curvature and potentially accelerating convergence. While it improves accuracy, the interpolated line search necessitates additional function evaluations to find the optimal step size, which may increase computational cost. Implementing the interpolated line search involves interpolating the residual function, introducing complexity compared to using a fixed step size.

C.3.2. Regula Falsi Line Search

The regula falsi (false position) line search employs a root-finding technique. One computes the function $s(\eta) = \Delta U R(U_n + \eta \Delta U)$, where R represents the residual. The step size (η) is updated based on the sign of ($s(\eta)$). If ($s(\eta)$) changes sign between two points, the interval is bisected to find the root.

Regula falsi balances accuracy and stability, providing reasonable convergence. While it may not be as precise as the interpolated line search, it offers a good compromise. It requires fewer function evaluations compared to the interpolated line search, making it computationally efficient. Regula falsi is straightforward to implement, involving root-finding techniques without additional complexities.

C.3.3. Bisection Line Search

The bisection line search maintains stability by employing fixed step sizes. The function is evaluated at two points along (ΔU), and the midpoint is chosen as the new estimate. The interval is halved in each iteration.

The bisection method maintains stability but sacrifices accuracy, leading to slower convergence due to fixed step sizes. It is computationally inexpensive, requiring only two function evaluations per iteration. Bisection is the simplest to implement, involving minimal additional logic beyond the basic iteration scheme.

C.4. Accelerators

Acceleration methods efficiently update iterates or adjust step sizes to overcome convergence limitations. Aitken's method employs linear extrapolation of iterates for faster convergence, while Anderson's combines previous iterates with a high-order correction term for iterative refinement (**accelerators2**). Steffensen's method is another approach, avoiding derivatives. These techniques are particularly effective for slowly converging sequences or challenging optimization problems, making Newton-Raphson more versatile and improving convergence across various scenarios (**accelerators1**).

C.4.1. Aitken's Delta-squared process (Shank's transformation)

The Shanks Transformation, also known as Aitken's delta-squared process, is a method used to accelerate the convergence of a sequence. Although not directly applicable to the Newton-Raphson algorithm itself, it can be employed on the sequence of iterates generated by the Newton-Raphson method to potentially improve its convergence speed (**aitken**).

Here's how you can apply the Shanks Transformation to the Newton-Raphson iterates:

1. Begin with an iterative process for solving a system of equations. For simplicity, consider a system with three equations and three unknowns, although the method can be generalized to larger systems.
2. Generate the iterative sequence $\{x_n\}$ where each x_n is an approximation to the solution. This sequence may exhibit slow convergence or divergence.
3. Apply Aitken's Delta-squared transformation to the iterative sequence. The transformed sequence is given by:
$$y_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \tag{C.1}$$
4. Use the transformed sequence $\{y_n\}$ as an updated sequence for further iterations. The idea is that this updated sequence converges faster than the original one.
5. Repeat

Repeat the iterative process using the updated sequence until convergence is achieved or until the desired level of accuracy is reached. Applying this transformation may help accelerate the convergence of the Newton-Raphson method. However, keep in mind that the effectiveness of the Shanks Transformation depends on the characteristics of the sequence and whether the underlying assumptions are met. It is important to test and verify the impact of such transformations on the specific problem.

Comparing Aitken's delta-squared method with the Newton-Raphson method reveals trade-offs. While Newton-Raphson offers quadratic convergence, it depends on derivatives, posing challenges

when derivatives are hard to obtain. Aitken's method, however, provides a derivative-free approach but lacks a guaranteed convergence. Thus, while Aitken's method accelerates convergence, especially with slowly converging sequences, its effectiveness varies, and Newton-Raphson remains competitive, particularly in cases of rapid convergence.

C.4.2. Anderson's transformation

Anderson's Acceleration is an iterative algorithm designed to accelerate the convergence of fixed-point iterations or other iterative methods for solving nonlinear equations or optimization problems (**accelerators1**). The basic iterative step in Anderson's Acceleration is given by:

$$x_{n+1} = \alpha_0 + \sum_{k=1}^{m} \alpha_k (F(x_n) - x_n) + H(x_n) \tag{C.2}$$

where x_n is the current approximation, $F(x_n)$ is the function value at x_n, α_0 is a damping parameter, α_k are coefficients, and $H(x_n)$ is a high-order correction term. To use Anderson's Acceleration, start with an initial approximation x_0, apply the iterative step until convergence is reached, and update the coefficients and damping parameter dynamically during the iterations.

When examining Anderson's Acceleration alongside the Newton-Raphson method, both aim to improve iterative processes for solving nonlinear equations, though they take different approaches. Anderson's Acceleration has advantages over Newton-Raphson, particularly in significantly speeding up convergence, especially for specific types of problems. It's widely used in electronic structure calculations and various scientific fields. However, Anderson's Acceleration isn't without limitations compared to Newton-Raphson. Its effectiveness can vary depending on the problem, and it can be computationally demanding due to the calculations needed for coefficients and correction terms.

While Anderson's Acceleration shows promise for improving convergence in iterative processes, its performance and computational requirements need careful consideration, especially when compared to the more traditional Newton-Raphson method.

C.4.3. Richardson Extrapolation

Richardson Extrapolation is a numerical technique utilized to enhance the accuracy of approximations obtained from iterative methods by extrapolating solutions from computations performed at multiple step sizes or levels of refinement. The method is particularly valuable for improving convergence rates and reducing numerical errors in iterative schemes (**richardson**).

The fundamental concept behind Richardson Extrapolation involves computing approximations of a desired quantity at different levels of refinement, typically achieved by varying parameters such as step sizes or grid resolutions. These approximations are then combined using a weighted sum or polynomial interpolation to generate a more accurate estimate of the desired quantity. By leveraging information from computations at different resolutions, Richardson Extrapolation can significantly enhance the accuracy of numerical solutions.

When compared to standard iterative methods, Richardson Extrapolation offers notable advantages in terms of accuracy improvement and convergence acceleration. However, its effectiveness may depend on factors such as the choice of refinement levels and the behavior of the underlying iterative scheme. Additionally, Richardson Extrapolation may introduce additional computational overhead, particularly when performing computations at multiple resolutions.

In conclusion, Richardson Extrapolation presents a powerful technique for enhancing the accuracy and convergence of iterative methods by leveraging computations performed at different levels of refinement. While offering significant benefits, its application requires careful consideration of factors such as computational cost and convergence behavior to ensure optimal performance.

C.5. Conclusion on methods for non-linear system

This chapter has examined various iterative methods and techniques aimed at optimizing the performance of numerical solvers in district heating network simulations.

C.5.1. Iterative methods compared

This chapter has explored a range of iterative methods designed to optimize the performance of numerical solvers in district heating network simulations, focusing on achieving a delicate balance between stability and accuracy. Damping and regularization techniques are fundamental for enhancing solver stability, controlling update steps through damping to prevent overshoot, and modifying the problem via regularization to introduce stability, although this may introduce a slight bias. Similarly, line search methods dynamically optimize step sizes to ensure meaningful contributions toward convergence with each iteration, crucial for achieving precise solutions in complex simulations. Accelerators also enhance convergence rates, but their success is dependent on accurate prior knowledge of system characteristics, with inaccurate estimations potentially leading to poor convergence or increased computational demands.

Integrating these methods—damping, regularization, line search, and accelerators—typically yields the best results. This integrated approach not only addresses the varied challenges posed by different network configurations but also maximizes the strengths of each method to enhance overall solver performance. Effective selection and implementation of these iterative methods necessitate a nuanced understanding of the mathematical properties of the problem and the practical constraints of the available computational resources.

C.5.2. Moving forward

While refining the overall iterative steps of the method can yield incremental benefits, substantial gains are realized through optimizing the linear solve step, which is typically the most computationally demanding part of the iteration.

Optimizing the linear solve step is crucial because it consumes considerable computational resources, especially in large-scale systems. By enhancing the efficiency of this step, there is a reduction in both time and computational power needed to achieve a solution. The performance of the linear solver is often the limiting factor in the speed of convergence; efficient linear solvers can drastically decrease the number of iterations required by enabling better step changes with each iteration. Such techniques can significantly improve the method's overall efficiency and speed.

Furthermore, the accuracy and stability of the solutions from the Newton-Raphson method heavily depend on the precision of the linear solve step. Inadequate solving can introduce errors that propagate through iterations, potentially causing divergence. As network size and complexity grow, focusing on this step is essential to develop scalable solutions that can manage increased matrix sizes and complexity without performance loss, particularly using scalable linear solvers that efficiently handle sparse matrices typical in such applications.

In conclusion, while iterative improvements are valuable, the optimization of the linear solve step is critical. This approach not only boosts the efficiency and speed of the numerical method but also ensures the scalability and robustness necessary for modeling complex district heating networks effectively. This research advances the state of the art in the simulation and optimization of thermal networks by focusing efforts on refining the linear solve step.

C.5.3. Future research iterative methods

Based on the observed limitations, the following future research directions are proposed to further advance the field of numerical optimization in district heating systems. Research could be done into adaptive methods that automatically adjust damping and regularization parameters in response to changing conditions within the solver could provide a more dynamic and effective approach to managing stability and convergence issues. Developing more efficient line search algorithms that minimize the number of required function evaluations could reduce the computational costs while retaining the benefits of

dynamic step sizing. These suggested directions not only aim to tackle the inherent limitations of the current iterative methods but also open new avenues for robust, efficient, and adaptable solutions in the optimization of district heating networks.

Bibliography

[1] Ahookhosh, M., & Ghaderi, S. (2017). On efficiency of nonmonotone Armijo-type line searches. *Applied Mathematical Modelling*, 43, 170-190. https://doi.org/10.1016/j.apm.2016.10.055

[2] Axelsson, O. (1996). *Iterative solution methods*. Cambridge University Press. (ISBN 978-0-521-55569-2)

[3] Bertaccini, D., & Durastante, F. (2018). Iterative methods and preconditioning for large and sparse linear systems with applications. Chapman and Hall/CRC.

[4] Chill, R. (2008). Three variations on Newton's method. *The Mathematics Student*, 77.

[5] Colebrook, C. F., & White, C. M. (1937). Experiments with fluid friction in roughened pipes. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 161, 367-381. https://doi.org/10.1098/rspa.1937.0150

[6] Chen, K. (2005). *Matrix preconditioning techniques and applications*. Cambridge University Press. (ISBN 978-0521838283)

[7] Crisfield, M. A. (1984). Accelerating and damping the modified Newton-Raphson method. *Computers & Structures*, 18(3), 395-407. https://doi.org/10.1016/0045-7949(84)90059-2

[8] Dahlquist, G., & Björck, Å. (1974). *Numerical Methods*. Translated by N. Anderson. Prentice Hall.

[9] Dancker, J., & Wolter, M. (2021). Improved quasi-steady-state power flow calculation for district heating systems: A coupled Newton-Raphson approach. *Applied Energy*, 295, 116930. https://doi.org/10.1016/j.apenergy.2021.116930

[10] del Hoyo Arce, I., Herrero López, S., López Perez, S., Rämä, M., Klobut, K., & Febres, J. A. (2018). Models for fast modelling of district heating and cooling networks. *Renewable and Sustainable Energy Reviews*, 82, 1863-1873. https://doi.org/10.1016/j.rser.2017.06.109

[11] Dennis, J. E. Jr., & Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall. (ISBN 0-13-627216-9)

[12] Fekete, I., & Lóczi, L. (2022). Linear multistep methods and global Richardson extrapolation. *Applied Mathematics Letters*, 133, 108267. https://doi.org/10.1016/j.aml.2022.108267

[13] Flores, P., & Lankarani, H. (2016). Contact force models for multibody dynamics.

[14] Frederiksen, S., & Werner, S. (2013). District Heating and Cooling. Studentlitteratur. https://books.google.com.cy/books?id=vH5zngEACAAJ

[15] Gradyent. (2024). Gradyent Digital Twin. Retrieved February 2, 2024, from https://www.gradyent.ai/

[16] Google. (2024). JAX: Autograd and XLA. Retrieved April 15, 2024, from https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

[17] Johansson, R. (2015). Sparse matrices and graphs. In Numerical Python: A Practical Techniques Approach for Industry (pp. 235–254). Apress. https://doi.org/10.1007/978-1-4842-0553-2_10

[18] Li, X. S. (2004). SuperLU: Sparse direct solver and preconditioner. In 13th DOE ACTS Collection Workshop.

[19] Menapace, A., Boscheri, W., Baratieri, M., & Righetti, M. (2020). An efficient numerical scheme for the thermo-hydraulic simulations of thermal grids. International Journal of Heat and Mass Transfer, 161, 120304. https://doi.org/10.1016/j.ijheatmasstransfer.2020.120304

[20] Neumaier, A. (1998). Solving Ill-Conditioned and Singular Linear Systems: A Tutorial on Regularization. SIAM Review, 40(3), 636-666. https://doi.org/10.1137/S0036144597321909

[21] NumPy Developers. (2020). NumPy: Fundamental Package for Scientific Computing with Python. Retrieved from https://numpy.org/

[22] OpenAI. (n.d.). ChatGPT: A Large-Scale Generative Pretrained Transformer. Retrieved April 20, 2024, from https://openai.com/gpt

[23] Ross, K. A. (2013). Elementary Analysis: The Theory of Calculus (2nd ed.). Springer. https://doi.org/10.1007/978-1-4614-6271-2

[24] Schenk, O., & Gärtner, K. (2002). Solving unsymmetric sparse systems of linear equations with PARDISO. In P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, & J. J. Dongarra (Eds.), Computational Science — ICCS 2002 (pp. 355–363). Springer Berlin Heidelberg.

[25] Schenk, O. (2024). Parallel Sparse Direct and Multi-Recursive Iterative Linear Solvers: PARDISO User Guide Version 8.2. Panua Technologies.

[26] Schrijfhulp. (2024). AWA. Retrieved from https://awa.schrijfhulp.be/index.php#

[27] SciPy Developers. (2020). GMRES documentation. Retrieved from https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.gmres.html

[28] SciPy Developers. (2022). BiCGSTAB documentation. Retrieved from https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.bicgstab.html

[29] SciPy. (n.d.). SciPy: Scientific Library for Python. Retrieved from https://www.scipy.org/

[30] Shahnaz, R., Usman, A., & Chughtai, I. R. (2005). Review of storage techniques for sparse matrices. In 2005 Pakistan Section Multitopic Conference (pp. 1–7). IEEE.

[31] SuperLU Developers. (n.d.). SuperLU: Direct Solver for Large Sparse Linear Systems. Retrieved from https://portal.nersc.gov/software/superlu/

[32] Trefethen, L. N., & Bau, D. (2022). Numerical Linear Algebra. SIAM.

[33] Vuik, K., Vermolen, F., van Gijzen, M., & Vuik, T. (2023). Numerical Methods for Ordinary Differential Equations. TU Delft OPEN Textbooks. Retrieved March 16, 2024, from https://textbooks.open.tudelft.nl/textbooks/catalog/view/57/152/394

[34] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... & SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods, 17, 261–272. https://doi.org/10.1038/s41592-019-0686-2

[35] White, F. M. (2011). Fluid Mechanics (7th ed.). McGraw-Hill Education. (ISBN 978-0073529349)