DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

ANALYSIS AND SIMULATIONS OF LINEAR CIRCUITS USING INTERVAL ALGEBRA

by
İlayda BONCUK

October, 2025
İZMİR

ANALYSIS AND SIMULATIONS OF LINEAR CIRCUITS USING INTERVAL ALGEBRA

A Thesis Submitted to the
Graduate School of Natural And Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical and Electronics Engineerings, Electrical and Electronics Engineering Program

by
İlayda BONCUK

October, 2025
İZMİR

M.Sc. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**ANALYSIS AND SIMULATIONS OF LINEAR CIRCUITS USING INTERVAL ALGEBRA**" completed by **İLAYDA BONCUK** under supervision of **ASSOC. PROF. DR. SERKAN GÜNEL** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

.....

Assoc. Prof. Dr. Serkan GÜNEL

Supervisor

.....

Assoc. Prof. Dr. Mehmet Emre ÇEK

Jury Member

.....

Assoc. Prof. Dr. Nalan ÖZKURT

Jury Member

Prof. Dr. Abdullah SEÇGİN
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Assoc. Prof. Dr. Serkan Günel, for his valuable guidance, expertise, and constructive feedback throughout this study. His academic insight and support have been essential in shaping both the direction and the quality of this thesis.

I am also profoundly grateful to my mentor, Prof. Dr. Yeşim Zoral, whose steadfast encouragement and unwavering support have been indispensable during this journey. Her guidance, especially in challenging moments, has enabled me to persevere and move forward with confidence.

I wish to extend my heartfelt appreciation to my family for their unconditional love, patience, and continuous support throughout my academic pursuits. Their presence has been a constant source of strength and motivation.

Finally, I would like to thank my friends and colleagues at Dokuz Eylül University for their collaboration, shared knowledge, and encouragement, making this journey both intellectually enriching and personally meaningful.

İlayda BONCUK

ANALYSIS AND SIMULATIONS OF LINEAR CIRCUITS USING INTERVAL ALGEBRA

ABSTRACT

Accurate simulation of electronic circuits under parameter uncertainty is essential for modern design and verification. Classical SPICE simulations rely on nominal values and therefore provide deterministic solutions that may not reflect real-world variations such as manufacturing tolerances, ageing, or temperature changes. To address this challenge, two main approaches are widely used: probabilistic Monte Carlo methods and deterministic interval arithmetic. Monte Carlo simulations provide statistical insight but are computationally demanding and may overlook rare worst-case scenarios. Interval methods, in contrast, guarantee mathematically rigorous enclosures, yet often suffer from overestimation due to dependency and wrapping effects. This thesis presents the design and implementation of a Julia-based SPICE-like simulator that integrates interval arithmetic into the Modified Nodal Analysis (MNA) framework. The simulator automatically parses netlists, constructs MNA matrices, and applies multiple interval solvers, including Gaussian elimination, Jacobi, Gauss–Seidel, Krawczyk, and Oettli–Prager methods. The enclosures produced by these solvers are systematically compared with Monte Carlo results using the Hausdorff distance as a quantitative metric.

Benchmark results highlight the trade-offs between accuracy, enclosure quality, and computational efficiency. While Monte Carlo remains versatile for probabilistic analysis, interval methods provide strong guarantees suitable for safety-critical applications. The outcomes of this study demonstrate the potential of interval arithmetic as a reliable complement to traditional simulation, offering practical guidelines for uncertainty-aware circuit design and verification.

Keywords: SPICE simulation, interval arithmetic, modified nodal analysis, Monte Carlo methods, circuit uncertainty analysis, Julia programming.

DOĞRUSAL DEVRELERİN ARALIK CEBRİ İLE ANALİZ VE BENZETİMİ

ÖZ

Elektronik devrelerin parametre belirsizlikleri altında doğru bir şekilde simülasyonu, modern tasarım ve doğrulama süreçleri için kritik öneme sahiptir. Klasik SPICE tabanlı simülasyonlar nominal değerler üzerinden çalışarak deterministik sonuçlar üretir; ancak üretim toleransları, yaşlanma ve sıcaklık değişimleri gibi gerçekçi varyasyonları yeterince yansıtmaz. Bu sorunu çözmek için literatürde iki temel yaklaşım kullanılmaktadır: olasılıksal Monte Carlo yöntemleri ve deterministik interval aritmetiği. Monte Carlo simülasyonları istatistiksel içgörü sağlasa da yüksek hesaplama maliyeti gerektirir ve nadir görülen en kötü durum senaryolarını gözden kaçırabilir. Interval yöntemleri ise matematiksel olarak kesin sınırlar garanti eder, ancak bağımlılık ve sarmalanma (wrapping) etkileri nedeniyle aşırı korumacı sonuçlar üretebilir. Bu tezde, interval aritmetiğin Modified Nodal Analysis (MNA) çerçevesine entegre edildiği Julia tabanlı bir SPICE benzeri simülatörün tasarımı ve uygulaması sunulmaktadır. Geliştirilen simülatör, devre netlistlerini otomatik olarak ayrıştırmakta, MNA matrislerini oluşturarak Gaussian elimination, Jacobi, Gauss–Seidel, Krawczyk ve Oettli–Prager yöntemleri dâhil olmak üzere çeşitli interval çözücüler uygulamaktadır. Bu çözücülerin ürettiği sınırlar, Hausdorff mesafesi kullanılarak Monte Carlo sonuçlarıyla nicel olarak karşılaştırılmıştır.Gerçekleştirilen testler, doğruluk, sınır kalitesi ve hesaplama verimliliği arasındaki takasları ortaya koymuştur. Monte Carlo yöntemi olasılıksal analizler için esnekliğini korurken, interval yöntemleri güvenilirlik garantileriyle özellikle güvenlik kritik uygulamalar için öne çıkmaktadır. Bu çalışma, interval aritmetiğin geleneksel simülasyona güvenilir bir tamamlayıcı sunduğunu göstermekte ve belirsizlik odaklı devre tasarımı için pratik öneriler sunmaktadır.

Anahtar kelimeler: SPICE Simülasyonu, interval aritmetik, modified nodal analysis, Monte Carlo yöntemleri, devre belirsizlik analizi, Julia programlama.

CONTENTS

Page

M.Sc. THESIS EXAMINATION RESULT FORM ii
ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZ v
LIST OF FIGURES ix
LIST OF TABLES x
ABBREVIATIONS xi
LIST OF SYMBOLS xii

CHAPTER ONE – INTRODUCTION 1

CHAPTER TWO – THEORETICAL BACKGROUND 5

2.1 SPICE-Based Circuit Analysis: Mathematical Foundations 5
2.1.1 Topology and Constitutive Equations 6
2.2 Modified Nodal Analysis 7
2.2.1 DC Analysis 9
2.2.2 AC Small-Signal Analysis 11
2.2.3 Transient Analysis 11
2.3 Interval Analysis 11
2.3.1 Definition and Basic Operations 12
2.3.2 Interval Linear Systems 14
2.3.2.1 Interval Gaussian Elimination (IGE) 16
2.3.2.2 Linear Krawczyk Method 17
2.3.2.3 Oettli–Präger Method 18
2.3.2.4 Interval Jacobi Method 19
2.3.2.5 Interval Gauss–Seidel Method 21
2.3.3 Interval Polynomial Root Finding 22

2.3.3.1 Interval Newton Method 23
2.3.3.2 Krawczyk's Method for Roots 24
2.3.3.3 Bisection 25
2.4 Monte Carlo Analysis 25
2.5 Comparison Metrics 26

CHAPTER THREE – METHODOLOGY 32

3.1 Overview of the Computational Framework 33
3.2 General Approach 33
3.3 Julia-Based Implementation Flow 36
3.4 Integration of Interval Algorithms 38
3.5 Monte Carlo Benchmark Protocol 45
3.6 Benchmarking Metrics and Performance Criteria 45
3.7 Methodological Contributions 47

CHAPTER FOUR – SIMULATIONS AND RESULTS 49

4.1 Data Sets Used In Simulations 49
4.1.1 Simple RC Low-Pass Filter 49
4.1.2 RLC Notch Filter 50
4.1.3 Complex RLC Network 50
4.1.4 Two-Stage RC Low-Pass Filter 51
4.1.5 Circuit Types Considered 51
4.1.6 Parameter Specifications 52
4.1.7 Analysis Approaches 52
4.2 Performance Criteria 53
4.3 Tests and Results 53
4.3.1 Comparison of Interval Methods and Monte Carlo 54
4.3.2 Monte Carlo Benchmark Results 55
4.3.2.1 Performance Results 56
4.3.2.2 Component-wise Statistics 56
4.3.2.3 Error Statistics 57

4.3.3 Dedicated 3×3 Benchmark System Results 57
4.3.3.1 Monte Carlo summary. 59
4.3.3.2 Polytope and ellipsoid geometry. 59
4.3.3.3 Hausdorff and efficiency metrics. 60
4.3.3.4 Runtime and memory micro-benchmarks. 61
4.3.4 Circuit Case Studies 61
4.3.4.1 Simple RC Filter 61
4.3.4.2 RLC Notch Filter 65
4.3.4.3 Complex RLC and Multi-Stage Filters 67
4.3.5 Overall Assessment 68

CHAPTER FIVE – CONCLUSION 69

REFERENCES 71

APPENDICES 74

A.1 Appendix 74
A.2Codes 74

LIST OF FIGURES

Page

Figure 3.1 Method flow linking SPICE/MNA modeling, interval solvers, Monte Carlo sampling, and geometry-based comparison (MVEE, Hausdorff, volume, coverage). 32
Figure 4.1 Simple RC low-pass filter with component tolerances. 49
Figure 4.2 RLC parallel (notch) filter with series source resistor and load. 50
Figure 4.3 Complex RLC example: series R–L, shunt C, series R_0, and load. 50
Figure 4.4 Two-stage RC low-pass filter (multi-stage) with component tolerances.. 51
Figure 4.5 Complete 3D visualization for the 3×3 benchmark: 20k MC points (dots), eight Oettli–Präger polytopes (colored), MVEE (purple), and interval boxes (Krawczyk, Gauss–Seidel, Jacobi). 58
Figure 4.6 Magnitude response of the RC filter under uncertainty, with the inset providing a zoomed-in view around the corner frequency. 62
Figure 4.7 Phase response of the RC filter under uncertainty, with the inset providing a zoomed-in view around the corner frequency. 62
Figure 4.8 Nyquist diagram of the RC filter under uncertainty. 63
Figure 4.9 Monte Carlo sampling distribution for the RC filter. 64
Figure 4.10 Interval width ratio to MC range at 2983.6 Hz (RC filter). 65
Figure 4.11 Monte Carlo sampling distribution for the notch circuit. 66
Figure 4.12 Interval width ratio to MC range at 2983.6 Hz (notch circuit). 67

LIST OF TABLES

Page

Table 4.1 Nominal values and tolerance ranges for components used in the simulated circuits 52
Table 4.2 Nominal values and tolerance ranges for components used in the simulated circuits 54
Table 4.3 Solution bounds obtained by interval methods and Monte Carlo (3×3 system) 54
Table 4.4 Hausdorff distance analysis between interval methods and Monte Carlo .. 55
Table 4.5 Monte Carlo performance summary for different sample sizes 56
Table 4.6 Component intervals (min/mean/max) for Monte Carlo solutions 56
Table 4.7 Error statistics for Monte Carlo runs 57
Table 4.8 MC cloud statistics (20k samples) 59
Table 4.9 Interval boxes and polytope volumes for the 3×3 benchmark 60
Table 4.10 Geometric discrepancy and efficiency (lower Hausdorff is better, higher efficiency is better) 60
Table 4.11 Micro-benchmarks for the 3×3 benchmark (single run) 61
Table 4.12 Cross-circuit comparison under uncertainty 67

ABBREVIATIONS

MNA	Modified Nodal Analysis
DAE	Differential-Algebraic Equation
SPICE	Simulation Program with Integrated Circuit Emphasis
GE	Gaussian Elimination
NR	Newton–Raphson method
OP	Oettli–Präger formulation
J	Jacobi iteration (interval/classical)
GS	Gauss–Seidel iteration (interval/classical)
K	Krawczyk operator/method
HS	Hansen–Sengupta operator
IGE	Interval Gaussian Elimination
INM	Interval Newton Method
ILSE	Interval Linear Systems of Equations
MC	Monte Carlo method
MCS	Monte Carlo Simulation
CPP	Coverage Probability of Polytope
ER	Ellipsoidal Ratio
HD	Hausdorff Distance
SDP	Semidefinite Programming
LP	Linear Programming
CPU	Central Processing Unit

LIST OF SYMBOLS

$^{\mathrm{i}}\mathbf{A}$	Interval coefficient matrix
$^{\mathrm{i}}\mathbf{b}$	Interval right-hand side vector
$\Sigma(^{\mathrm{i}}\mathbf{A}, ^{\mathrm{i}}\mathbf{b})$	United solution set of an interval linear system
$\mathbf{x}$	Real solution vector
$^{\mathrm{i}}\mathbf{x}$	Interval vector
C	Nonsingular preconditioning matrix
$^{\mathrm{i}}\mathbf{D}$, $^{\mathrm{i}}\mathbf{L}$, $^{\mathrm{i}}\mathbf{U}$	Diagonal, lower, and upper triangular parts of an interval matrix
a_{ij}	Interval matrix element in row i, column j
b_i	i-th component of the right-hand side vector
$J(^{\mathrm{i}}\mathbf{x})$	Interval Jacobi iteration operator
$^{\mathrm{i}}\mathbf{x}^{[k]}$	Approximate solution at the k-th iteration
$^{\mathrm{i}}\mathbf{X}^{[k]}$	Interval enclosure at iteration k (solution box)
ε–inflation	Perturbation term used to restore contraction
$\mathbb{IR}$	Set of all real intervals
$\mathbb{R}$	Set of all real numbers
$w(^{\mathrm{i}}\mathbf{x})$	Width of interval $^{\mathrm{i}}\mathbf{x}$
$mid(^{\mathrm{i}}\mathbf{x})$	Midpoint of interval $^{\mathrm{i}}\mathbf{x}$
$rad(^{\mathrm{i}}\mathbf{x})$	Radius of interval $^{\mathrm{i}}\mathbf{x}$
A_c, Δ	Midpoint and radius matrices (midpoint–radius form)
b_c, δ	Midpoint and radius vectors (midpoint–radius form)
$s \in \{\pm 1\}^n$	Sign vector defining an orthant (Oettli–Präger formulation)
$s = j\omega$	Complex frequency variable (ω: angular frequency)
$H_{kl}(s)$	Transfer function between output node k and input node l
$V_{n,k}(s), V_{n,l}(s)$	Phasor voltages at nodes k and l
$v_n \in \mathbb{R}^n$	Node voltage vector
$i_s \in \mathbb{R}^n$	Source current vector
Δt	Time step size
$F(v_n)$	Nonlinear device equations
$J(v_k)$	Jacobian matrix of $F(v)$ at iteration k

CHAPTER ONE
INTRODUCTION

Accurate prediction of electronic circuit behaviour is challenging in the presence of uncertainties such as manufacturing tolerances, temperature variations, ageing, and measurement errors. Classical SPICE(Simulation Program with Integrated Circuit Emphasis)-based simulations rely on nominal component values and therefore produce deterministic solutions, but in practice, circuit parameters vary within tolerance intervals rather than being fixed numbers. Capturing these deviations in a reliable and computationally efficient manner is thus essential for modern circuit verification and design, particularly for safety-critical electronic systems.

In the literature, two main paradigms have been adopted for uncertainty-aware circuit analysis. The first is probabilistic, with Monte Carlo simulation being the most widely used technique. In this approach, circuit parameters are randomly sampled, thousands of simulations are executed, and statistical results are obtained. Although conceptually flexible and straightforward, Monte Carlo is computationally expensive for high accuracy and is not guaranteed to capture low-probability worst-case behaviours. The second paradigm is deterministic, based on interval arithmetic. Each uncertain parameter is represented as an interval, and these intervals are propagated through the computations. As a result, the final enclosures are mathematically guaranteed to contain the proper solution. Interval analysis has been successfully applied in circuit analysis (Kolev, 1993), global optimization (Hansen & Walster, 2003), and system verification (Moore et al., 2009), offering rigorous guarantees that probabilistic methods lack.

Circuit simulators typically start from a netlist, which describes circuit connectivity and component values. Through graph-theoretic transformations, the netlist is converted into Modified Nodal Analysis (MNA) equations, resulting in sparse linear systems of equations for DC problems or structured linearized systems differential equation for AC and transient analyses. When nominal values are replaced by intervals, the formulation naturally produces interval matrices and

interval vectors, so that tolerance analysis becomes equivalent to solving interval linear equation systems. In this way, interval analysis extends the classical MNA framework to rigorously account for parameter uncertainty.

The solution of interval linear equation systems has been studied extensively, and different approaches exhibit distinct trade-offs. The direct methods, such as interval Gaussian elimination and hull methods, guarantee enclosures but often suffer from excessive conservatism due to interval growth and the so-called wrapping effect (Moore et al., 2009). The wrapping effect refers to the artificial overestimation that occurs when axis-aligned interval boxes repeatedly enclose multidimensional sets (e.g., parallelepipeds or rotated polytopes) during successive computations. Iterative approaches, such as interval Jacobi or Gauss–Seidel, can exploit sparsity and be more efficient for large-scale problems, though their convergence requires diagonal dominance or suitable preconditioning. Verification operators, including the Krawczyk operator, interval Newton schemes, and Oettli–Prager compatibility tests, deliver rigorous existence and uniqueness proofs in addition to enclosures and are widely used in validated numerics(Moore (1979); (Kolev, 1993). These methods have been adapted to DC, AC, and transient circuit analysis; for resistive networks, exact interval solutions are possible under independence assumptions, while for AC and transient problems, linearizations and real–imaginary decompositions yield reliable outer approximations. Nevertheless, challenges such as wrapping effects and computational overhead persist, motivating the development of preconditioning, slope-based extensions, and branch-and-bound techniques to improve efficiency (Kolev, 1993).

Although probabilistic Monte Carlo and interval methods have complementary strengths, there remains a lack of integrated tools that can combine rigorous guarantees with practical computational performance. In particular, there is a need for simulators that can efficiently translate SPICE netlists into graph-based MNA formulations with interval parameters, integrate multiple solution strategies within a unified framework, and quantitatively compare interval enclosures with Monte Carlo outcomes using set-to-set distance measures such as the Hausdorff metric.

Addressing this gap requires not only algorithmic integration but also careful attention to implementation aspects, such as sparse data structures, directed rounding, and efficient use of interval libraries. Well-tested libraries and standards are essential for providing reliable interval calculations in practice; the IEEE 1788 standard and library testing frameworks provide frameworks to facilitate comparison of different interval implementations, management of decoration/invention records, and compatibility testing (IEEE Std 1788, 2015). In this context, the application language and library selection (for example, the Julia ecosystem and related interval packages) play a decisive role in terms of both efficiency and ease of programming (Benet & Sanders, 2025).The objective of this thesis is therefore to design and evaluate a SPICE-like interval simulator implemented in Julia. The simulator parses netlists, builds MNA matrices, and applies interval-based DC, AC, and transient solvers. The resulting enclosures are benchmarked against Monte Carlo simulations, and the performance of different algorithms is systematically assessed in terms of both numerical guarantees and computational efficiency. The contributions of this work can be summarized as follows:

(i) a performance-oriented Julia implementation of netlist $\rightarrow$ graph $\rightarrow$ MNA transformation with tailored interval data structures,

(ii) integration of multiple interval solution algorithms including Gaussian elimination, Jacobi, Gauss–Seidel, Krawczyk, and Oettli–Prager methods,

(iii) a quantitative comparison protocol based on the Hausdorff distance between interval and Monte Carlo solution sets, and

(iv) systematic evaluation and ranking of solvers with respect to speed, memory usage, and enclosure quality, leading to practical recommendations for real-world applications.

This study analyzed several benchmark circuits to evaluate the performance of interval and Monte Carlo approaches under parameter uncertainty. The test set included a simple RC low-pass filter, an RLC notch filter, a more complex RLC network, a two-stage RC filter, and a dedicated 3×3 benchmark system. Monte Carlo analysis provided statistical insights across large sample sets but required high

computational effort to achieve accuracy. Interval methods, including Gaussian Elimination, Jacobi, Gauss–Seidel, Krawczyk, and Oettli–Präger, produced guaranteed enclosures with varying degrees of overestimation depending on dependency and wrapping effects. Comparisons were made using the Hausdorff distance, geometric measures, and efficiency metrics. Results showed that iterative methods like Gauss–Seidel and Krawczyk generally produced tighter enclosures than direct elimination. Monte Carlo was more versatile but less reliable in capturing worst-case bounds, whereas interval solvers consistently enclosed the proper solution set. Notably, the Oettli–Präger formulation yielded highly accurate descriptions of the exact solution set, confirming its strength as a rigorous reference method. However, this accuracy came at the expense of increased computational complexity, which limits its scalability for larger systems. Overall, the findings demonstrate that interval methods, though sometimes conservative, provide rigorous guarantees and are particularly valuable for safety-critical circuit verification.

CHAPTER TWO
THEORETICAL BACKGROUND

This chapter provides a rigorous mathematical exposition of SPICE-based circuit simulation, interval arithmetic, interval linear algebra, verified solvers for interval linear systems, Monte Carlo (MC) methods, and quantitative evaluation metrics.

2.1 SPICE-Based Circuit Analysis: Mathematical Foundations

SPICE is a widely used software environment to analyze electronic circuits. The principle of the program comes from systematically building the circuit equations, starting from the network's topology, and using the fundamental physical laws for each type of electrical element. In its core, SPICE employs modified nodal analysis (MNA). Using this approach, the program formulates a sparse set of algebraic and differential equations, which are then solved through numerical procedures (Ho et al. (1975)). The methodology works for linear and non-linear systems and cases when circuit parameters are fixed or time-variant. It also provides one combined structure for doing DC, AC, and transient simulations without changing the main formulation.

The circuit topology is commonly described through a graph representation to start the analysis. In this approach, each node indicates a defined electrical potential. At the same time, each edge is assigned to one of the physical elements that make up the circuit, such as resistors, capacitors, or inductors. This type of representation is valuable because it reveals the incidence relationships, which form the theoretical grounding for KCL and KVL. Details of how each element is linked to others are stored in incidence and loop matrices, and these act as the core structural basis for the analytical derivations and numerical computations that follow (Chua & Lin (1975)).

2.1.1 Topology and Constitutive Equations

In SPICE-based circuit analysis, the structural information required for computation is taken from the netlist. The netlist is structured text in which every node is listed, each component is identified, and the interconnections between them are described. This kind of description shows the circuit topology, meaning how the elements are arranged in space and connected electrically, while ignoring the exact numerical values given to them. To represent these connectivity features mathematically, is constructed topological matrices—most notably the node-incidence matrix **A** and the loop matrix **B**. The incidence matrix **A** defines how element currents correspond to node currents by Kirchhoff's Current Law (KCL). On the other hand, the loop matrix **B** defines how element voltages relate to loop voltages as established by Kirchhoff's Voltage Law (KVL) (Chen (1976)).

For a directed circuit graph $\mathcal{G}$ with n nodes and b branches, the incidence matrix is defined as $\mathbf{A} = [a_{ij}]$, where each entry a_{ij} is set according to the orientation of branch i with respect to node j:

$$a_{ij} = \begin{cases} +1, & \text{if branch } i \text{ leaves node } j, \\ -1, & \text{if branch } i \text{ enters node } j, \\ 0, & \text{if branch } i \text{ is not incident to node } j. \end{cases} \qquad i = 1, ..., b, j = 1, ..., n$$

satisfies the following two basic topological relations:

$$\mathbf{A}\,\mathbf{v}_n(t) = \mathbf{v}_b(t), \quad \mathbf{A}^\top \mathbf{i}_b(t) = \mathbf{0}, \tag{2.1}$$

where $\mathbf{v}_n(t) \in \mathbb{R}^n$ is node voltages, $\mathbf{v}_b(t) \in \mathbb{R}^b$ is branch voltages and $\mathbf{i}_b(t) \in \mathbb{R}^b$ is branch currents.

The second stage of SPICE's formulation involves the constitutive equations of the

circuit elements, which mathematically define the relationships between voltages and currents for each component. For linear passive components, these relationships are:

$$v_R(t) = R \cdot i_R(t) \tag{2.2}$$

$$i_C(t) = C \cdot \frac{dv_c(t)}{dt} \tag{2.3}$$

$$v_L(t) = L \cdot \frac{di_L(t)}{dt} \tag{2.4}$$

In SPICE, the complete mathematical description of the circuit is obtained by putting together the topological constraints coming from KCL and KVL with the constitutive equations that define the behaviour of each element. When this combined set is written in matrix form, it most often results in a system that can be expressed as a differential–algebraic equation (DAE):

$$\mathbf{C}(\mathbf{v}_n)\frac{d\mathbf{v}_n}{dt} + \mathbf{G}(\mathbf{v}_n)\mathbf{v}_n = \mathbf{s}(t), \tag{2.5}$$

where $\mathbf{v}_n$ represents node voltages and selected branch currents $\mathbf{C}(\mathbf{v}_n)$ contains dynamic (energy-storage) terms, $\mathbf{G}(\mathbf{v}_n)$ contains static conductance terms, and $\mathbf{s}(t)$ is the source vector (White & Sangiovanni-Vincentelli (1987)).This Equation 2.5 will be explained in more detail in the MNA Section 2.2 .

This unified framework, grounded in topology and constitutive relations, enables SPICE to handle a wide variety of circuit types, from purely resistive networks to complex non-linear, multi-port, and mixed-signal systems.

2.2 Modified Nodal Analysis

Classical nodal analysis works well when the network has only passive components and independent sources. However, once elements such as the dependent like sources, coupled inductors, or specific non-linear elements are considered, direct application gets tricky. To deal with this, Ho et al. (1975) developed what is now called MNA.

The idea is simple but effective: you keep the basic nodal equations but add a few extra variables that stand for the branch currents in those special elements. With these additions, you end up with a formulation that can handle more device types and still fit neatly into the matrix-based algorithms that SPICE uses for solving circuits.

In MNA, the primary unknowns are:

1. Node voltages (excluding the reference node).
2. Currents through independent voltage sources and inductors (and, in some cases, other elements that cannot be represented solely through admittance).

Let n denote the number of non-reference nodes, b the number of current-defined branches, and m the number of independent voltage sources or constrained branches. The MNA equations can be expressed in block matrix form as:

$$\underbrace{\begin{bmatrix} \mathbf{C}_v & \mathbf{0} \\ \mathbf{0} & \mathbf{L} \end{bmatrix}}_{\mathbf{E}} \frac{d}{dt} \begin{bmatrix} \mathbf{v}_n(t) \\ \mathbf{i}_b(t) \end{bmatrix} + \underbrace{\begin{bmatrix} \mathbf{G} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \mathbf{v}_n(t) \\ \mathbf{i}_b(t) \end{bmatrix} = \begin{bmatrix} \mathbf{i}_s(t) \\ \mathbf{v}_s(t) \end{bmatrix}, \tag{2.6}$$

where $\mathbf{v}_n(t) \in \mathbb{R}^n$ is the vector of node voltages, and $\mathbf{i}_b(t) \in \mathbb{R}^b$ is the vector of branch currents associated with current-defined elements. The upper-left block $\mathbf{C}_v \in \mathbb{R}^{n \times n}$ is the nodal capacitance matrix, formed by combining the contributions of all capacitors connected to each node. The lower-right block $\mathbf{L} \in \mathbb{R}^{m \times m}$ is typically a diagonal matrix whose entries are the inductance values associated with the inductor current variables. Together, $\mathbf{C}_v$ and $\mathbf{L}$ make up the descriptor (or mass) matrix $\mathbf{E}$, which multiplies the time-derivative term in the system of equations. The second block matrix, here denoted by $\mathbf{A} \in \mathbb{R}^{(n+b)\times(n+b)}$, represents the algebraic part of the MNA equations. It includes:

- $\mathbf{G} \in \mathbb{R}^{n \times n}$: the conductance matrix, assembled from resistive elements;
- $\mathbf{B} \in \mathbb{R}^{n \times m}$ and $\mathbf{C} \in \mathbb{R}^{m \times n}$: connection matrices that link voltage sources or controlled sources to the nodal variables;
- $\mathbf{D} \in \mathbb{R}^{m \times m}$: a matrix that can include constraints associated with sources and inductors, and which is often zero in many practical circuits.

Finally, $\mathbf{i}_s(t) \in \mathbb{R}^n$ and $\mathbf{v}_s(t) \in \mathbb{R}^m$ are the source vectors representing, respectively, independent current sources and independent voltage sources. This structured form allows the MNA equations to be expressed compactly as a DAE.

The MNA framework, as expressed in Equation (2.6), can model a wide range of circuits linear, non-linear, and time-varying within one consistent mathematical structure. Its sparse matrix form, resulting from localized device connections, supports efficient storage schemes and factorization methods, essential for large-scale problems involving up to $10^5 - 10^6$ unknowns (Kundert (1986)). The modular stamping process, where each element updates only the relevant parts of the **G**, **B**, **C**, and **D** matrices, allows incremental assembly from the netlist and easy design modifications. These characteristics establish MNA as the core computational engine of SPICE, linking topology extraction, device modeling, and iterative solvers.

2.2.1 DC Analysis

For DC analysis, the derivative terms vanish, i.e.,

$$\frac{d\mathbf{v}_n}{dt} = 0, \quad \frac{d\mathbf{i}_b}{dt} = 0,$$

and Equation (2.6) simplifies to

$$\mathbf{G}\,\mathbf{v}_n = \mathbf{i}_s. \tag{2.7}$$

In the case of non-linear elements:

$$\mathbf{F}(\mathbf{v}_n) = 0 \tag{2.8}$$

where $F : \mathbb{R}^n \to \mathbb{R}^n$ is a non-linear function representing the device equations.

The Newton–Raphson method (Kundert & Sangiovanni-Vincentelli, 1990) is employed to solve the nonlinear system of equations arising from circuit analysis. At

each iteration k, the linearized correction step is obtained by solving

$$\mathbf{J}(\mathbf{v}_k)\,\Delta\mathbf{v} = -\mathbf{F}(\mathbf{v}_k), \qquad \mathbf{v}_{k+1} = \mathbf{v}_k + \Delta\mathbf{v}, \tag{2.9}$$

where $\mathbf{v}_k \in \mathbb{R}^n$ denotes the current approximation of the nodal voltage vector, and $\Delta\mathbf{v} \in \mathbb{R}^n$ is the corresponding update. The matrix $\mathbf{J}(\mathbf{v}_k) \in \mathbb{R}^{n\times n}$ is the Jacobian of $\mathbf{F}(\mathbf{v})$ evaluated at $\mathbf{v}_k$, formally defined as

$$\mathbf{J}(\mathbf{v}_k) = \left.\frac{\partial \mathbf{F}(\mathbf{v})}{\partial \mathbf{v}}\right|_{\mathbf{v}=\mathbf{v}_k}.$$

In expanded form, this Jacobian matrix consists of the partial derivatives of the residual functions $F_i(\mathbf{v})$ with respect to the voltage components v_j:

$$\mathbf{J}(\mathbf{v}_k) = \begin{bmatrix} \frac{\partial F_1(\mathbf{v})}{\partial v_1} & \frac{\partial F_1(\mathbf{v})}{\partial v_2} & \cdots & \frac{\partial F_1(\mathbf{v})}{\partial v_n} \\ \frac{\partial F_2(\mathbf{v})}{\partial v_1} & \frac{\partial F_2(\mathbf{v})}{\partial v_2} & \cdots & \frac{\partial F_2(\mathbf{v})}{\partial v_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n(\mathbf{v})}{\partial v_1} & \frac{\partial F_n(\mathbf{v})}{\partial v_2} & \cdots & \frac{\partial F_n(\mathbf{v})}{\partial v_n} \end{bmatrix}_{\mathbf{v}=\mathbf{v}_k}.$$

Various homotopy-based techniques have been developed to improve the convergence performance of the Newton–Raphson method(Chua & Lin (1975); Roychowdhury (1999)). One such method, gmin-stepping, introduces artificial conductances between each node and ground to aid convergence. Initially, high conductance values make the circuit easier to solve, and then these artificial conductances are gradually reduced to reach the actual circuit topology. This increases the stability of the iterations and ensures a secure solution. Another approach, source-stepping, is based on the principle of starting the voltage and current source values at low levels and increasing them in small steps. This stepwise method reduces convergence problems caused by sudden changes and speeds up the solution process (Vlach & Singhal, 2003).

2.2.2 AC Small-Signal Analysis

For AC small-signal analysis, the non-linear circuit is linearized at its DC operating point. Substituting $s = j\omega$ gives:

$$(\mathbf{G} + j\omega\,\mathbf{C}_v)\,\mathbf{V}_n(j\omega) = \mathbf{I}_s(j\omega), \tag{2.10}$$

where $\mathbf{C}_v \in \mathbb{R}^{n\times n}$ is nodal capacitance matrix, $\mathbf{V}_n(j\omega) \in \mathbb{C}^n$ is phasor representation of node voltages, $\mathbf{I}_s(j\omega) \in \mathbb{C}^n$ is phasor representation of independent current sources and $s = j\omega$ is complex frequency variable, with ω as angular frequency.

The transfer function between an output node k and an input node l is:

$$H_{kl}(s) = \frac{V_{n,k}(s)}{V_{n,l}(s)} \tag{2.11}$$

where $V_{n,k}(s)$ and $V_{n,l}(s)$ are the phasor voltages at nodes k and l, respectively.

2.2.3 Transient Analysis

In time-domain simulations, the descriptor system defined by Equation (2.6) is solved using numerical integration. The trapezoidal rule (Gear, 1971) gives:

$$\frac{\mathbf{C}_v}{\Delta t}\left(\mathbf{v}_n^{[k+1]} - \mathbf{v}_n^{[k]}\right) + \mathbf{G}\,\frac{\mathbf{v}_n^{[k+1]} + \mathbf{v}_n^{[k]}}{2} = \frac{\mathbf{i}_s^{[k+1]} + \mathbf{i}_s^{[k]}}{2}, \tag{2.12}$$

where Δt is time step size,$\mathbf{v}_n \in \mathbb{R}^n$ is node voltage vectors at time steps k and $k+1$, and $\mathbf{i}_s^{[k]}, \mathbf{i}_s^{[k+1]} \in \mathbb{R}^n$ is independent source current vectors at time steps k and $k+1$. This produces a linear equation system that must be solved at each time step k.

2.3 Interval Analysis

In computational modelling of electrical circuits, as well as in other engineering systems, numerical reliability can be affected by several factors. Such as limits of

finite-precision arithmetic, simplifications introduced during the modelling stage, and the inherent manufacturing tolerances of physical components. Conventional floating-point operations usually return a single numerical value. However, such a single-point estimate may not capture the actual behaviour of the system, particularly when variations in parameters have a strong influence. For very large SPICE simulations, even a small numerical imperfection—if repeated many times—can accumulate and lead to noticeable deviation in calculated node voltages or branch currents.

One approach to control such errors is interval arithmetic. In this method, every variable is represented not as a single value but as a closed numerical interval. These intervals are then propagated carefully through each computational step. By doing so, the method ensures that the actual solution must be contained within the final computed bounds (Moore et al., 2009). When combined with graph-oriented techniques such as the Modified Nodal Analysis, interval methods enable mathematically verified simulations. They not only confirm correctness in a formal sense but also deliver explicit limits on uncertainty (Neumaier, 1990). Such capabilities are highly relevant for safety-critical electronics, tolerance studies in analogue circuit design, and evaluation of design margins in the presence of manufacturing variability (Hansen & Walster, 2003).

2.3.1 Definition and Basic Operations

The following definitions and operational rules for interval arithmetic are adapted from the foundational treatments in (Neumaier, 1990) and , (Moore et al., 2009) which provide the standard formal framework for verified numerical computations.

The set of all real intervals is defined as

$$\mathbb{IR} = \{[a, b] \mid a \leq b,\ a, b \in \mathbb{R}\}. \tag{2.13}$$

For an interval ${}^{\mathrm{i}}\mathbf{x} = [x_L, x_U] \in \mathbb{IR}$ we define:

- Width: $w({}^{\mathrm{i}}\mathbf{x}) = x_U - x_L$
- Midpoint: $\mathrm{mid}({}^{\mathrm{i}}\mathbf{x}) = \dfrac{x_L + x_U}{2}$
- Radius: $\mathrm{rad}({}^{\mathrm{i}}\mathbf{x}) = \dfrac{w({}^{\mathrm{i}}\mathbf{x})}{2}$

Let ${}^{\mathrm{i}}\mathbf{x} = [x_L, x_U]$ and ${}^{\mathrm{i}}\mathbf{y} = [y_L, y_U]$. The basic interval operations are defined as follows:

$$ {}^{\mathrm{i}}\mathbf{x} + {}^{\mathrm{i}}\mathbf{y} = [x_L + y_L,\ x_U + y_U], \tag{2.14} $$

$$ {}^{\mathrm{i}}\mathbf{x} - {}^{\mathrm{i}}\mathbf{y} = [x_L - y_U,\ x_U - y_L], \tag{2.15} $$

$$ {}^{\mathrm{i}}\mathbf{x} \times {}^{\mathrm{i}}\mathbf{y} = [\min S,\ \max S], \quad S = \{x_L y_L,\ x_L y_U,\ x_U y_L,\ x_U y_U\}, \tag{2.16} $$

$$ {}^{\mathrm{i}}\mathbf{x} / {}^{\mathrm{i}}\mathbf{y} = {}^{\mathrm{i}}\mathbf{x} \times \left[\frac{1}{y_U}, \frac{1}{y_L}\right], \quad \text{if } 0 \notin {}^{\mathrm{i}}\mathbf{y}. \tag{2.17} $$

These operations satisfy the *inclusion isotonicity* property:

$$ {}^{\mathrm{i}}\mathbf{x} \subseteq {}^{\mathrm{i}}\mathbf{x}',\ {}^{\mathrm{i}}\mathbf{y} \subseteq {}^{\mathrm{i}}\mathbf{y}' \quad \Rightarrow \quad f({}^{\mathrm{i}}\mathbf{x}, {}^{\mathrm{i}}\mathbf{y}) \subseteq f({}^{\mathrm{i}}\mathbf{x}', {}^{\mathrm{i}}\mathbf{y}'), \tag{2.18} $$

for any interval function f defined by these operations. Accordingly, when the input ranges are expanded, the resulting range also expands but never excludes possible solution values. This characteristic allows interval arithmetic methods to provide accuracy guarantees. This property is fundamental in error analysis, global optimization, and solving differential equations. It allows algorithms to produce reliable results and prevents overlooking any proper solution. Therefore, the principle of inclusion is fundamental, making interval arithmetic essential for numerical verification methods.

However, interval arithmetic is generally *sub-distributive*:

$$ {}^{\mathrm{i}}\mathbf{x} \cdot ({}^{\mathrm{i}}\mathbf{y} + {}^{\mathrm{i}}\mathbf{z}) \subseteq {}^{\mathrm{i}}\mathbf{x} \cdot {}^{\mathrm{i}}\mathbf{y} + {}^{\mathrm{i}}\mathbf{x} \cdot {}^{\mathrm{i}}\mathbf{z}, \tag{2.19} $$

with strict inclusion possible due to the *dependency problem*, where multiple occurrences of the same variable are treated independently, and the *wrapping effect*. The latter occurs when the range of a function is not rectangular, whereas interval arithmetic can only operate on rectangular intervals. When the proper solution set exhibits an elliptical or curvilinear geometry, it must be enclosed within the smallest bounding rectangular interval. This enclosure unnecessarily enlarges the resulting interval and reduces the precision of the computations. In other words, restricting non-rectangular sets to rectangular intervals leads to coarser estimates of the results. Therefore, the wrapping effect is regarded as one of the fundamental limitations of interval arithmetic.

2.3.2 Interval Linear Systems

Interval linear systems are systems of equations of the form

$$^{\mathrm{i}}\mathbf{A}\mathbf{x} = {}^{\mathrm{i}}\mathbf{b}, \tag{2.20}$$

in which the coefficient matrix and the right-hand side vector (RHS) are represented by intervals. Formally, with $^{\mathrm{i}}\mathbf{A} \in \mathbb{IR}^{n\times n}$, $^{\mathrm{i}}\mathbf{b} \in \mathbb{IR}^{n}$,, the corresponding solution set is defined as

$$\Sigma(^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b}) := \{\, \mathbf{x} \in \mathbb{R}^n \;\mid\; \forall \mathrm{A} \in {}^{\mathrm{i}}\mathbf{A},\ \forall \mathrm{b} \in {}^{\mathrm{i}}\mathbf{b} : A\mathbf{x} = b \,\}.$$

This set represents all possible solutions that arise from admissible choices of $^{\mathrm{i}}\mathbf{A}$ and $^{\mathrm{i}}\mathbf{b}$ within their intervals, and the objective in interval computations is to determine the tightest possible enclosure, i.e., The smallest interval hull that contains $\Sigma(^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$. It is important to emphasize that in the definition of $\Sigma(^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$ the solution vector $\mathbf{x}$ is an element of $\mathbb{R}^n$, not of $\mathbb{IR}^n$. This distinction arises because, for each admissible realization $\mathbf{A} \in {}^{\mathrm{i}}\mathbf{A}$ and $\mathbf{b} \in {}^{\mathrm{i}}\mathbf{b}$, the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is an ordinary linear system with a real solution. The uncertainty is carried exclusively by the coefficients and the right-hand side, not by the solution variable itself. The collection of all such real solutions forms the united solution set $\Sigma(^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$. In verified computations, this generally complicated

set is then enclose by an interval vector (enclosure), which provides guaranteed bounds on all possible solutions (Neumaier, 1990, Hansen & Walster, 2003).

In practice, interval analysis provides a mathematically rigorous framework to handle rounding errors and parameter uncertainties through interval arithmetic. Interval algorithms typically employ a combination of techniques, such as preconditioning, direct or iterative solvers, and additional refinement (or shrinking) steps, to achieve efficiency and verified enclosures.

An *enclosure* is any interval vector guaranteed to contain the solution set $\Sigma({}^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$. The smallest possible enclosure is the *interval hull*, defined component–wise as the exact infimum and supremum of all solutions. Computing the hull is known to be NP-hard (Neumaier, 1990), hence practical approaches focus on verified but not necessarily minimal enclosures.

A major challenge in interval computations is *overestimation* (also called the wrapping effect), which occurs because interval arithmetic treats repeated variables independently and restricts non-rectangular sets into rectangular boxes.

To mitigate this effect, *preconditioning* is frequently applied. By transforming the system into an equivalent form

$$\mathbf{C}\,{}^{\mathrm{i}}\mathbf{A}\mathbf{x} = \mathbf{C}\,{}^{\mathrm{i}}\mathbf{b}, \qquad \mathbf{C} \in \mathbb{R}^{n\times n},\ \det(\mathbf{C}) \neq 0, \tag{2.21}$$

one aims to reduce dependency and narrow the resulting enclosures. Here $\mathbf{C}$ is a nonsingular *point* matrix, typically chosen as the inverse or an approximation of the midpoint of ${}^{\mathrm{i}}\mathbf{A}$, or obtained through LU-based factorizations or magnitude-pivoting strategies. The interval nature of the problem is preserved since ${}^{\mathrm{i}}\mathbf{A} \in \mathbb{IR}^{n\times n}$ and ${}^{\mathrm{i}}\mathbf{b} \in \mathbb{IR}^{n}$ remain interval quantities, while only the transformation matrix $\mathbf{C}$ is deterministic. Although preconditioning can significantly reduce the effect of overestimation, it may also enlarge the solution set in some cases; hence its effectiveness is problem-dependent (Neumaier, 1990, Hansen & Walster, 2003).

Within this framework, interval Gaussian elimination provides direct enclosures but is sensitive to overestimation and pivot choices; the Krawczyk method offers rigorous existence and uniqueness guarantees without interval division; and Jacobi and Gauss–Seidel iterations represent natural interval extensions of classical iterative schemes, trading off parallelism against local convergence speed. The Oettli–Präger characterization gives an exact description of the united solution set, but its practical application is limited by the exponential number (2^n, where n is the dimension of the system, i.e., the number of variables) of orthants that must be tested.

Finally, interval methods have strong motivation in engineering applications. In safety-critical circuit verification, robust control synthesis, and tolerance analysis of analog designs, obtaining a verified enclosure is far more valuable than a single nominal solution. Interval analysis provides explicit and mathematically rigorous uncertainty bounds, ensuring that all admissible parameter variations are accounted for (Hansen & Walster, 2003).

2.3.2.1 Interval Gaussian Elimination (IGE)

The Interval Gaussian Elimination (IGE) method directly extends the classical Gaussian elimination scheme to interval arithmetic. Its purpose is to compute a verified enclosure for the solution set of an interval linear system by performing elimination on the augmented matrix $({}^{\mathrm{i}}\mathbf{A} \mid {}^{\mathrm{i}}\mathbf{b})$, where all operations are evaluated in the interval sense with outward rounding. At each elimination step, the entries of the coefficient matrix and the right-hand side vector are updated according to

$$a_{ij}^{(new)} = a_{ij}^{(old)} - \frac{a_{ik}^{(old)}}{a_{kk}}\, a_{kj}^{(old)}, \qquad b_i^{(new)} = b_i^{(old)} - \frac{a_{ik}^{(old)}}{a_{kk}}\, b_k^{(old)}. \tag{2.22}$$

Here, $a_{ij}^{(old)}$ denotes the interval coefficient in row i and column j before the elimination step, and $a_{ij}^{(new)}$ is its updated value. The pivot a_{kk} must be chosen so that $0 \notin a_{kk}$; otherwise, preconditioning or row/column reordering is required. The factor $m = a_{ik}/a_{kk}$ represents the interval multiplier used to eliminate the k-th entry in row

i, and the same operation is applied to the right-hand side b_i to maintain consistency. Conceptually, these formulas correspond to subtracting a multiple of the pivot row from another row, executed in interval arithmetic. As elimination proceeds, the intervals generally widen due to dependency effects and outward rounding, leading to conservative but guaranteed enclosures.

Finally, the triangular system obtained after elimination is solved by interval back substitution. Each unknown is computed as an interval division since the corresponding pivot interval a_{ii} does not contain zero. The quality of the final enclosure depends strongly on the conditioning of A: well-conditioned systems tend to produce tight bounds, whereas ill-conditioned systems cause rapid growth in interval widths. For specific structured matrices such as M-matrices and H-matrices, it has been shown that IGE can even recover the exact solution hull Neumaier (1990), Hansen & Walster (2003). Here, an M-matrix is a real square matrix with non-positive off-diagonal entries and eigenvalues with positive genuine parts, which ensures monotonicity and stability properties. An H-matrix is defined by the condition that its comparison matrix is an M-matrix, providing a more general framework that still guarantees non-singularity and favourable numerical behaviour. However, IGE is often combined with iterative refinement or hybrid strategies to mitigate overestimation.

2.3.2.2 Linear Krawczyk Method

The Krawczyk method provides a verification framework for the existence (and often uniqueness) of solutions to linear and non-linear systems, based on fixed-point theory. For an interval linear system

$$^{\mathrm{i}}\mathbf{A}\mathbf{x} = {}^{\mathrm{i}}\mathbf{b}, \qquad {}^{\mathrm{i}}\mathbf{A} \in \mathbb{IR}^{n\times n},\ {}^{\mathrm{i}}\mathbf{b} \in \mathbb{IR}^{n},$$

the unknown $\mathbf{x}$ is always interpreted as a real vector $\mathbf{x} \in \mathbb{R}^n$. The uncertainty resides in the coefficients $^{\mathrm{i}}\mathbf{A}$ and $^{\mathrm{i}}\mathbf{b}$, while the set of all admissible real solutions is described

by the united solution set $\Sigma(^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$.

Given a box $\mathbf{X} \subset \mathbb{R}^n$, the Krawczyk operator is defined as

$$K(\mathbf{X}) = y - \mathbf{R}(^{\mathrm{i}}\mathbf{A}y - {}^{\mathrm{i}}\mathbf{b}) + (\mathbf{I} - \mathbf{R}\,{}^{\mathrm{i}}\mathbf{A})(\mathbf{X} - y), \tag{2.23}$$

where $y \in \mathbb{R}^n$ is a chosen center point (often the midpoint of **X**),
$\mathbf{R} \in \mathbb{R}^{n\times n}$ is a nonsingular point preconditioner (e.g., an approximate inverse of mid($^{\mathrm{i}}\mathbf{A}$)),
I is the identity matrix. Since $^{\mathrm{i}}\mathbf{A}$ and $^{\mathrm{i}}\mathbf{b}$ are intervals, $K(\mathbf{X})$ is itself an interval vector in $\mathbb{IR}^n$.

The fundamental inclusion test is

$$K(\mathbf{X}) \subseteq \mathrm{int}(\mathbf{X}),$$

which ensures that there exists a real solution $\mathbf{x}^* \in \mathbf{X}$. In most practical cases, the uniqueness of the solution within **X** can also be concluded. An important property of the Krawczyk operator in the linear setting is that the computation avoids interval division, which improves numerical stability.Thus, the method delivers a validated enclosure for the true real solution set while keeping the operations reliable (Neumaier, 1990, Hansen & Walster, 2003).

2.3.2.3 Oettli–Präger Method

The Oettli–Präger characterization provides an exact description of the solution set of an interval linear system. It is important to emphasize that the candidate solution vector **x** is always a real vector $\mathbf{x} \in \mathbb{R}^n$, while the inequality test involves quantities derived from interval data, namely the midpoint matrix $\mathbf{A}_c$, the midpoint vector $\mathbf{b}_c$, and their corresponding radii Δ_A and δ. In other words, the characterization checks whether a real vector **x** simultaneously satisfies all admissible choices of the interval system by verifying the inequality constraint that encodes the effect of coefficient and right-hand

side uncertainties.

Given

$$^{\mathrm{i}}\mathbf{A} = [\mathbf{A}_c - \Delta_A,\ \mathbf{A}_c + \Delta_A], \qquad ^{\mathrm{i}}\mathbf{b} = [\mathbf{b}_c - \delta,\ \mathbf{b}_c + \delta], \tag{2.24}$$

the united solution set is

$$\Sigma(^{\mathrm{i}}\mathbf{A}, ^{\mathrm{i}}\mathbf{b}) = \left\{ \mathbf{x} \in \mathbb{R}^n \;\middle|\; |\mathbf{A}_c\mathbf{x} - \mathbf{b}_c| \leq \Delta_A|\mathbf{x}| + \delta \right\}. \tag{2.25}$$

Here,

$\mathbf{A}_c$ and $\mathbf{b}_c$ are the midpoint matrix and vector,

Δ_A and δ denote their radius quantities. The inequality expresses that the uncertainties of the coefficients and the right-hand side bound the residual of the midpoint system. A vector $\mathbf{x}$ is a solution of the interval system if and only if it satisfies this inequality. Since the absolute value is applied component-wise, the feasibility test must be performed separately in each orthant (sign pattern) of $\mathbb{R}^n$, leading to a worst-case complexity of 2^n, where n is the dimension of the system. For larger problems, orthant-reduction strategies or linear programming relaxations are typically applied to improve scalability.

In summary, the Oettli—Präger method provides an exact characterization of the solution set, but its direct application is computationally demanding. It is effective for small—to medium-sized systems, whereas larger systems require hybrid or relaxation-based approaches.

2.3.2.4 Interval Jacobi Method

The interval Jacobi method is the natural extension of the classical Jacobi method iteration to interval arithmetic. In this approach, each element of the solution vector is updated in parallel using only the values from the previous iteration, which makes the method highly parallelizable but often slower to converge than Gauss–Seidel.

The interval coefficient matrix ${}^{\mathrm{i}}\mathbf{A}$ in Eq. 2.20 can be decomposed as

$$ {}^{\mathrm{i}}\mathbf{A} = {}^{\mathrm{i}}\mathbf{D} + ({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U}), \tag{2.26} $$

$$ {}^{\mathrm{i}}\mathbf{D} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad {}^{\mathrm{i}}\mathbf{L} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{bmatrix}, \quad {}^{\mathrm{i}}\mathbf{U} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}, $$

with each entry $a_{ij} \in \mathbb{IR}$. Substituting the decomposition gives

$$ ({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})\mathbf{x} = {}^{\mathrm{i}}\mathbf{b}, \tag{2.27} $$

which can be rearranged as

$$ {}^{\mathrm{i}}\mathbf{D}\mathbf{x} = {}^{\mathrm{i}}\mathbf{b} - ({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})\mathbf{x}. \tag{2.28} $$

Multiplying formally by the inverse of the diagonal interval matrix, where ${}^{\mathrm{i}}\mathbf{D}^{-1}$ denotes element–wise inversion of the diagonal intervals, yields

$$ \mathbf{x} = {}^{\mathrm{i}}\mathbf{D}^{-1}\left({}^{\mathrm{i}}\mathbf{b} - ({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})\mathbf{x}\right). \tag{2.29} $$

This expression defines a fixed–point form $\mathbf{x} = \mathcal{J}(\mathbf{x})$ with iteration operator $\mathcal{J}(\mathbf{z}) = {}^{\mathrm{i}}\mathbf{D}^{-1}\big({}^{\mathrm{i}}\mathbf{b} - ({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})\mathbf{z}\big)$. Replacing $\mathbf{x}$ on the right-hand side by the current iterate $\mathbf{x}^{[k]}$ gives the interval Jacobi iteration

$$ \mathbf{x}^{[k+1]} = {}^{\mathrm{i}}\mathbf{D}^{-1}\left({}^{\mathrm{i}}\mathbf{b} - ({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})\,\mathbf{x}^{[k]}\right) \tag{2.30} $$

Writing Eq. (2.28) row by row,

$$ a_{ii}x_i = b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\,x_j, \quad i = 1, \ldots, n \tag{2.31} $$

and dividing by the diagonal interval (valid only if $0 \notin a_{ii}$) yields the component–wise update

$$x_i^{[k+1]} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}\, x_j^{[k]} \right) \quad i = 1, \ldots, n. \tag{2.32}$$

It should be noted that although the coefficients $^{\mathrm{i}}\mathbf{A}$ and the right-hand side $^{\mathrm{i}}\mathbf{b}$ are intervals, the unknown vector $\mathbf{x}$ in Eq. (2.27)–(2.29) remains a real vector. The role of the iteration (2.30)–(2.32) is to generate an interval enclosure $\mathbf{x}^{[k]}$ that contains all possible realizations of the exact real solution for admissible (A, b). In other words, the algorithm does not assume that $\mathbf{x}$ is interval itself; rather, it computes verified interval bounds for $\mathbf{x}$ across all realizations (Neumaier, 1990, Hansen & Walster, 2003).

All operations in Eq. (2.30)–(2.32) are carried out in interval arithmetic with outward rounding, and the requirement $0 \notin a_{ii}$ for all i is crucial to avoid undefined interval division (Neumaier, 1990, Hansen & Walster, 2003). The method is parallelizable since each update ($\mathbf{x}^{[k+1]}$) depends only on values from the previous($\mathbf{x}^{[k]}$) iteration. Classical convergence criteria such as $\rho(^{\mathrm{i}}\mathbf{D}^{-1}(^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})) < 1$ serve as guidance, although in the interval setting one must additionally monitor the growth of enclosure widths caused by dependency and wrapping effects.

2.3.2.5 Interval Gauss–Seidel Method

The interval Gauss–Seidel method modifies the Jacobi iteration by immediately reusing newly computed components within the same iteration step. This typically results in faster local convergence and tighter enclosures compared to interval Jacobi, although the method is inherently sequential and, therefore, less suitable for parallel implementation. Using rearranged Eq.(2.26) is

$$^{\mathrm{i}}\mathbf{A} = (^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L}) + {}^{\mathrm{i}}\mathbf{U}, \tag{2.33}$$

Substituting the decomposition gives

$$(({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L}) + {}^{\mathrm{i}}\mathbf{U})\mathbf{x} = {}^{\mathrm{i}}\mathbf{b}, \tag{2.34}$$

which can be rearranged as

$$({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L})\mathbf{x} = {}^{\mathrm{i}}\mathbf{b} - {}^{\mathrm{i}}\mathbf{U}\mathbf{x}. \tag{2.35}$$

the interval Gauss–Seidel iteration reads

$$\mathbf{x}^{[k+1]} = ({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L})^{-1}\left({}^{\mathrm{i}}\mathbf{b} - {}^{\mathrm{i}}\mathbf{U}\mathbf{x}^{[k]}\right), \tag{2.36}$$

or component–wise

$$x_i^{[k+1]} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{[k+1]} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{[k]}\right), \quad i = 1, \ldots, n. \tag{2.37}$$

Here newly computed components $x_j^{(k+1)}$ (for $j < i$) are used immediately in the same iteration, which typically yields tighter enclosures and faster local convergence than interval Jacobi, though it reduces parallelism. As above, $0 \notin a_{ii}$ is required for all i, and all arithmetic uses outward rounding (Neumaier, 1990, Hansen & Walster, 2003).

A practical condition is $\rho\big(({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L})^{-1}\,{}^{\mathrm{i}}\mathbf{U}\big) < 1$ in the corresponding point iteration; in the interval case one additionally checks that enclosure widths do not blow up across iterations (Neumaier, 1990).

2.3.3 Interval Polynomial Root Finding

In circuit and control system analysis, polynomial root finding is directly related to stability assessment. The denominator of a transfer function

$$H(s) = \frac{N(s)}{D(s)}$$

defines the poles of the system as the roots of the characteristic polynomial $D(s)$. The location of these poles determines stability and transient behaviour. When the underlying circuit parameters (e.g., R, L, C) are uncertain within tolerance intervals, the coefficients of $D(s)$ become interval quantities.

In this context, interval root-finding methods provide verified enclosures for all admissible pole locations. Such enclosures are critical for robust stability analysis:

- In *root locus* diagrams, the trajectories of poles as a parameter varies must be computed with guaranteed containment of all possibilities. Interval methods ensure that the true pole movements remain inside the computed intervals.
- In *Nyquist plots*, the frequency response is influenced by the exact positions of poles and zeros. Interval root finding propagates uncertainty in polynomial coefficients into rigorously bounded Nyquist curves, which is essential for verified gain and phase margin analysis.

Three major approaches for interval polynomial root finding are the interval Newton method, the Krawczyk method adapted to root verification, and interval bisection.

2.3.3.1 Interval Newton Method

The interval Newton method generalizes Newton's iteration to interval arithmetic. Given a continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ and an interval ${}^{\mathrm{i}}\mathbf{x} = [x_L, x_U] \subset \mathbb{R}$, define

$$N({}^{\mathrm{i}}\mathbf{x}) = m - \frac{f(m)}{F'({}^{\mathrm{i}}\mathbf{x})}, \tag{2.38}$$

where $m = \mathrm{mid}({}^{\mathrm{i}}\mathbf{x})$ is the midpoint of ${}^{\mathrm{i}}\mathbf{x}$, and $F'({}^{\mathrm{i}}\mathbf{x})$ is the interval extension of the derivative $f'(x)$ over ${}^{\mathrm{i}}\mathbf{x}$. Here, $F'({}^{\mathrm{i}}\mathbf{x})$ denotes the *interval derivative*, i.e., the interval extension of the classical derivative $f'(x)$. Mathematically, it is defined as

$$F'({}^{\mathrm{i}}\mathbf{x}) = \big[\inf_{t \in {}^{\mathrm{i}}\mathbf{x}} f'(t),\ \sup_{t \in {}^{\mathrm{i}}\mathbf{x}} f'(t)\big],$$

which encloses all possible values of $f'(t)$ for $t \in {}^{\mathrm{i}}\mathbf{x}$. In physical terms, the interval derivative represents the complete range of admissible rates of change of the system when parameters or states are subject to uncertainty. Unlike the classical derivative, which gives a single instantaneous slope at a point, the interval derivative provides rigorous lower and upper bounds on the slope across the entire interval. This allows one to capture worst-case and best-case dynamical responses, ensuring that sensitivity, stability, and robustness analyses remain valid even in the presence of parameter tolerances and modeling errors.

Division here is interval division and valid only if $0 \notin F'({}^{\mathrm{i}}\mathbf{x})$. The next interval is obtained as

$$ {}^{\mathrm{i}}\mathbf{x}_{\text{new}} = N({}^{\mathrm{i}}\mathbf{x}) \cap {}^{\mathrm{i}}\mathbf{x}. \tag{2.39} $$

If $N({}^{\mathrm{i}}\mathbf{x}) \subseteq {}^{\mathrm{i}}\mathbf{x}$, then ${}^{\mathrm{i}}\mathbf{x}$ contains exactly one real root of $f({}^{\mathrm{i}}\mathbf{x}) = 0$ (Neumaier, 1990, Moore et al., 2009). If $N({}^{\mathrm{i}}\mathbf{x}) \cap {}^{\mathrm{i}}\mathbf{x} = \emptyset$, no root exists in x. Thus the method provides both existence/uniqueness verification and contraction. Near simple roots, contraction is often quadratic, similar to classical Newton, but now with verified enclosures.

2.3.3.2 Krawczyk's Method for Roots

Krawczyk's method uses a fixed-point operator for root verification. Given $f : \mathbb{R} \to \mathbb{R}$ and an interval ${}^{\mathrm{i}}\mathbf{x} \subset \mathbb{R}$ with midpoint $m = \text{mid}({}^{\mathrm{i}}\mathbf{x})$, the operator is

$$ K({}^{\mathrm{i}}\mathbf{x}) = m - Rf(m) + (1 - Rf'({}^{\mathrm{i}}\mathbf{x}))({}^{\mathrm{i}}\mathbf{x} - m), \tag{2.40} $$

where $f'({}^{\mathrm{i}}\mathbf{x})$ is the interval extension of the derivative, and $R \in \mathbb{R}$ is a point preconditioner, typically $R \approx 1/f'(m)$. All operations are performed with outward rounding to guarantee inclusion.

If

$$ K({}^{\mathrm{i}}\mathbf{x}) \subseteq \text{int}({}^{\mathrm{i}}\mathbf{x}), \tag{2.41} $$

then ${}^{\mathrm{i}}\mathbf{x}$ contains a unique real root of $f({}^{\mathrm{i}}\mathbf{x}) = 0$ Neumaier (1990), Hansen & Walster (2003). Compared with interval Newton, this approach avoids explicit interval division, which improves numerical stability. However, the tightness of the contraction depends strongly on the choice of R. In practice, Krawczyk's method is often used together with bisection for global coverage.

2.3.3.3 Bisection

The interval bisection method is a global strategy to isolate all roots of $f({}^{\mathrm{i}}\mathbf{x}) = 0$. Given ${}^{\mathrm{i}}\mathbf{x} = [x_L, x_U]$ with midpoint $m = \mathrm{mid}({}^{\mathrm{i}}\mathbf{x})$, continuity implies that if

$$f(x_L) \cdot f(x_U) \leq 0, \tag{2.42}$$

then ${}^{\mathrm{i}}\mathbf{x}$ contains at least one root. If not, ${}^{\mathrm{i}}\mathbf{x}$ is bisected into ${}^{\mathrm{i}}\mathbf{x}_1 = [x_L, m]$ and ${}^{\mathrm{i}}\mathbf{x}_2 = [m, x_U]$, and the test is repeated recursively.

While bisection converges only linearly and can be computationally expensive, it is globally reliable and cannot miss a root. In practical applications, bisection is employed to isolate subintervals likely to contain roots, which are then refined using faster contraction methods such as interval Newton or Krawczyk. This hybrid approach is widely used for constructing verified rootlocus diagrams and Nyquist plots under parametric uncertainty (Moore et al., 2009).

2.4 Monte Carlo Analysis

Monte Carlo (MC) simulation is one of the most widely used techniques for uncertainty analysis in circuit design. In this approach, circuit parameters are modelled as random variables within their tolerance ranges, and repeated simulations are performed by sampling parameter values from prescribed probability distributions. For each sample, a deterministic SPICE simulation is executed, and the outputs are collected (Rubinstein & Kroese, 2016).

Formally, let $\theta = (\theta_1, \ldots, \theta_p)$ denote the uncertain parameters, each with a probability density function $p(\theta)$. For a performance measure $g(\theta)$, such as a pole location or output voltage, the Monte Carlo estimate of the expectation is

$$\hat{g}_N = \frac{1}{N} \sum_{k=1}^{N} g(\theta^{[k]}), \tag{2.43}$$

where $\theta^{[k]}$ are independent draws from $p(\theta)$. By the law of large numbers, $\hat{g}_N$ converges to $\mathbb{E}[g(\theta)]$ as $N \to \infty$ (Asmussen & Glynn, 2007).

The main advantage of Monte Carlo simulation is its simplicity and non-intrusive nature, since it relies on repeated evaluations of the original SPICE model without altering the formulation. However, the convergence rate is slow, of order $\mathcal{O}(1/\sqrt{N})$, which means that a very large number of trials may be required for narrow confidence intervals. As a result, Monte Carlo provides only a probabilistic characterization of uncertainty and does not guarantee that rare but admissible parameter combinations are fully covered. In this thesis, Monte Carlo is used as a baseline to benchmark the performance of interval-based enclosures.

2.5 Comparison Metrics

To compare interval enclosures with Monte Carlo results, quantitative metrics are introduced using the basic interval notation defined earlier. Let ${}^{\mathrm{i}}\mathbf{x} = [x_L, x_U] \in \mathbb{IR}$ be an interval enclosure, where each component ${}^{\mathrm{i}}\mathbf{x}_i = [x_{Li}, x_{Ui}]$ has width $w({}^{\mathrm{i}}\mathbf{x}_i) = x_{Ui} - x_{Li}$. Let $X_{\mathrm{MC}} = [x_{1,\min}, x_{1,\max}] \times \cdots \times [x_{n,\min}, x_{n,\max}]$ denote the smallest axis–aligned interval box containing all Monte Carlo samples.

The first measure is the *enclosure width*. For each component i,

$$w({}^{\mathrm{i}}\mathbf{x}_i) = x_{Ui} - x_{Li}, \tag{2.44}$$

and for the entire vector one may take the maximum or the average width,

$$w({}^{\mathrm{i}}\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} w({}^{\mathrm{i}}\mathbf{x}_i). \tag{2.45}$$

Smaller widths indicate tighter enclosures, while large widths reflect overestimation due to dependency or wrapping effects (Moore et al., 2009, Neumaier, 1990). In order to account for the relation to empirical data, the *relative overestimation* compares the enclosure width with the empirical Monte Carlo band. For each component i,

$$\mathrm{RO}_i = \frac{w({}^{\mathrm{i}}\mathbf{x}_i)}{x_{i,\max} - x_{i,\min}}, \tag{2.46}$$

and the global measure is

$$\mathrm{RO} = \max_{i=1,\dots,n} \mathrm{RO}_i. \tag{2.47}$$

A value $\mathrm{RO} = 1$ indicates that the interval enclosure matches exactly the Monte Carlo band, while larger values reflect conservativeness.

Another important metric is the *coverage probability(CP)*, which expresses how well the interval result contains the Monte Carlo data. Given samples $\{x^{[k]}\}_{k=1}^{N}$, it is defined as

$$\mathrm{CP}({}^{\mathrm{i}}\mathbf{x}) = \frac{1}{N}\sum_{k=1}^{N} \mathbf{1}_{\{x^{[k]} \in {}^{\mathrm{i}}\mathbf{x}\}}, \tag{2.48}$$

where $\mathbf{1}_{\{\cdot\}}$ is the indicator function. The ideal value is $\mathrm{CP} = 1$, meaning all samples are contained in the verified enclosure. In practice, values close to but below unity are expected, since overly tight intervals may exclude outliers while overly conservative ones reduce practical usefulness (Hansen & Walster, 2003).

A geometric criterion is the *Hausdorff distance*. Between the interval enclosure ${}^{\mathrm{i}}\mathbf{X}$ and the Monte Carlo box X_{MC}, it is defined as

$$d_H({}^{\mathrm{i}}\mathbf{X}, X_{\mathrm{MC}}) = \max\Big\{ \sup_{x \in {}^{\mathrm{i}}\mathbf{X}} \inf_{y \in X_{\mathrm{MC}}} \|x - y\|, \sup_{y \in X_{\mathrm{MC}}} \inf_{x \in {}^{\mathrm{i}}\mathbf{X}} \|x - y\| \Big\}. \tag{2.49}$$

For axis–aligned boxes this reduces component–wise to

$$d_H({}^{\mathrm{i}}\mathbf{X}, X_{\mathrm{MC}}) = \max_{i=1,\dots,n} \Big\{ |x_{Li} - x_{i,\min}|,\ |x_{Ui} - x_{i,\max}| \Big\}. \tag{2.50}$$

This metric quantifies the maximum deviation between the verified enclosure and the Monte Carlo band (Rockafellar & Wets, 2009).

In higher dimensions scalar widths alone become insufficient; therefore, the *volume ratio* provides a global measure of conservativeness:

$$\mathrm{VR} = \frac{\prod_{i=1}^{n} w({}^{\mathrm{i}}\mathbf{x}_i)}{\prod_{i=1}^{n} (x_{i,\max} - x_{i,\min})}. \tag{2.51}$$

Values close to one indicate good agreement, while large values imply that the interval enclosure is substantially larger than the empirical spread observed in Monte Carlo simulation. Since interval boxes are axis–aligned, they often yield pessimistic estimates; therefore, additional geometric descriptors such as ellipsoids and polytopes are useful to refine the comparison.

The *minimum–volume enclosing ellipsoid* (MVEE) of the Monte Carlo cloud $\{x^{[k]}\}_{k=1}^{N}$ offers such a refinement. If $\mathcal{E} = \{x \mid (x-c)^T P (x-c) \le 1\}$ denotes the ellipsoid centered at c with shape matrix $P \succ 0$, its volume is

$$\mathrm{vol}(\mathcal{E}) = \frac{\pi^{n/2}}{\Gamma\big(\frac{n}{2}+1\big)} \det(P)^{-1/2}. \tag{2.52}$$

The ellipsoidal ratio

$$\mathrm{ER} = \frac{\mathrm{vol}({}^{\mathrm{i}}\mathbf{X})}{\mathrm{vol}(\mathcal{E})} \tag{2.53}$$

provides a quantitative measure of the inefficiency of interval boxes compared with the more compact ellipsoidal representation. A large value of ER indicates that the interval enclosure contains a significant amount of redundant volume that is not occupied by the Monte Carlo data, thereby revealing the conservativeness induced by axis alignment. By contrast, values close to one imply that the interval enclosure is geometrically comparable to the MVEE, which is the tightest convex ellipsoidal

approximation of the empirical distribution (Khachiyan, 1996, Todd, 2016). In practical terms, the shape matrix P governs both the orientation and the scaling of the ellipsoid. Its eigenvectors correspond to the principal directions, while the eigenvalues determine the squared reciprocals of the semi–axis lengths. This property makes ellipsoids particularly suitable for representing correlated uncertainties, as they naturally align along the directions of maximum variability in the data cloud. Consequently, using the MVEE not only reduces conservativeness but also captures correlation structures that interval boxes ignore.

While ellipsoids provide a compact description of uncertainty, they may fail to capture irregular or polyhedral solution sets that frequently arise in verified computations. Interval methods such as the Krawczyk operator, interval Gauss–Seidel, or the Oettli–Präger characterization often produce feasible regions that can be represented as convex polytopes. These polytopes, defined as the convex hull of a finite number of vertices, offer a flexible geometric representation that more closely adheres to the true shape of the solution set. Unlike interval boxes, which are strictly axis–aligned, polytopes may align along arbitrary directions in $\mathbb{R}^n$, thereby accommodating dependencies among variables and reducing unnecessary overestimation. To assess the quality of such polyhedral enclosures, one can consider the polytope coverage probability,

$$\mathrm{CP}_{\mathcal{P}} = \frac{1}{N} \sum_{k=1}^{N} \mathbf{1}_{\{x^{[k]} \in \mathcal{P}\}}. \tag{2.54}$$

This metric generalizes the interval coverage probability by checking whether Monte Carlo samples fall within the polytope instead of the axis–aligned box. Values close to one indicate that the polytope tightly encapsulates the data cloud, while significantly smaller values may suggest that the construction of $\mathcal{P}$ is either too restrictive or misses regions of practical relevance. In addition to coverage, the discrepancy between polytope and Monte Carlo domains can be quantified through

the normalized symmetric difference,

$$\Delta(\mathcal{P}, X_{\text{MC}}) = \frac{\text{vol}(\mathcal{P} \triangle X_{\text{MC}})}{\text{vol}(X_{\text{MC}})}, \tag{2.55}$$

where $\triangle$ denotes the symmetric difference between two sets. This measure evaluates how much of the empirical spread is not explained by the polytope, while simultaneously penalizing any excessive conservativeness in directions not observed in the samples (Jaulin et al., 2001, Bliek, 1998). A small value of Δ demonstrates that the polytope not only covers the Monte Carlo outcomes but does so in a geometrically faithful manner.

Beyond purely geometric descriptors, hybrid Monte Carlo–interval measures directly assess compatibility between stochastic and verified approaches. The most immediate of these is the Monte Carlo in Box Ratio (MBR),

$$\text{MBR} = \frac{\#\{x^{[k]} \in {}^{\text{i}}\mathbf{X}\}}{N}, \tag{2.56}$$

which records the fraction of samples lying inside the verified interval box. Similarly, the Monte Carlo in Polytope Ratio (MPR),

$$\text{MPR} = \frac{\#\{x^{[k]} \in \mathcal{P}\}}{N}, \tag{2.57}$$

captures the proportion of the data enclosed by the polyhedral region. Both MBR and MPR allow a direct statistical interpretation: they quantify how often stochastic realizations, interpreted as ground truth approximations, fall within the deterministic guarantees produced by interval methods. A further refinement is provided by the Box in Polytope Ratio (BPR),

$$\text{BPR} = \frac{\text{vol}({}^{\text{i}}\mathbf{X} \cap \mathcal{P})}{\text{vol}({}^{\text{i}}\mathbf{X})}, \tag{2.58}$$

which measures the fraction of the interval box volume that is also contained within the polytope. This criterion evaluates whether the interval box is excessively conservative in regions not supported by the polytope analysis. A value close to one implies that

the polytope nearly fills the box, whereas small values indicate that the interval box significantly overestimates the feasible set. These hybrid metrics indicate how well interval methods reproduce the Monte Carlo cloud while retaining rigorous guarantees, thus bridging the gap between verified analysis and empirical sampling.

In summary, enclosure width, coverage probability, relative overestimation, Hausdorff distance, and volume ratio constitute the fundamental measures of interval tightness and conservativeness. Their extension through ellipsoidal ratios, polytope misfit, and hybrid Monte Carlo–interval criteria enriches the framework by introducing geometry–aware and data–driven comparisons. Together, these metrics provide a rigorous and complementary toolbox for assessing interval solvers against Monte Carlo simulations in uncertainty quantification (Hansen & Walster, 2003, Moore et al., 2009, Neumaier, 1990, Rockafellar & Wets, 2009, Jaulin et al., 2001, Boyd & Vandenberghe, 2004, Todd, 2016).

CHAPTER THREE
METHODOLOGY

This section outlines the methodological approach used to simulate circuits under uncertainty. The study focuses on integrating interval arithmetic into the classical SPICE-like circuit analysis flow and comparing these verification-oriented approaches with Monte Carlo methods. The developed methodological framework has been implemented in the Julia programming language and designed with consideration for both theoretical accuracy and computational efficiency.

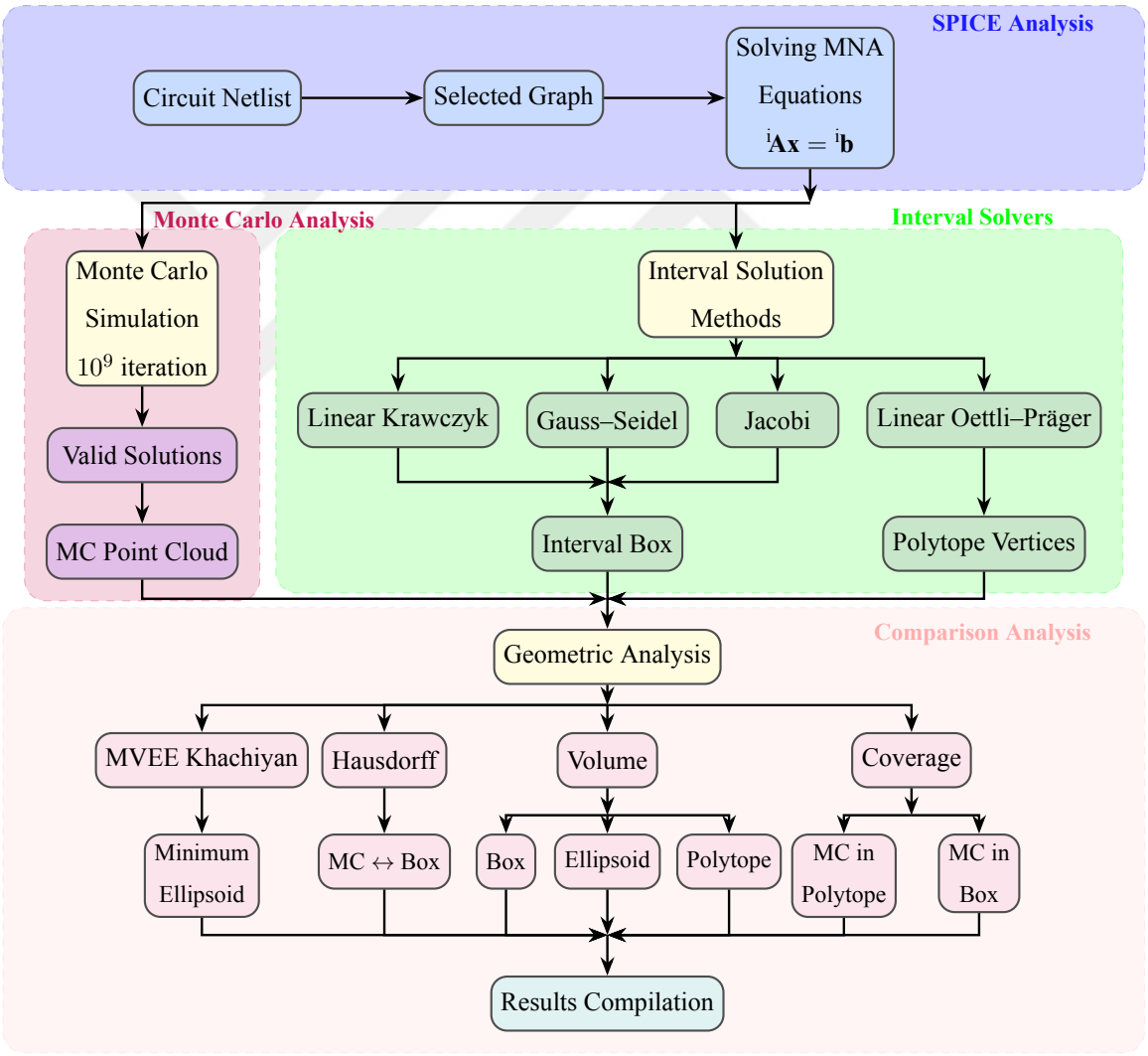


Figure 3.1 Method flow linking SPICE/MNA modeling, interval solvers, Monte Carlo sampling, and geometry-based comparison (MVEE, Hausdorff, volume, coverage).

3.1 Overview of the Computational Framework

The overall methodology illustrated in Figure 3.1 follows a three-stage structure. First, the circuit netlist is transformed into a Modified Nodal Analysis (MNA) model that serves as the numerical foundation for interval-based computations. Second, verified interval algorithms are applied to solve the uncertain linear system, producing mathematically rigorous enclosures. Finally, the resulting interval solutions are compared with large-scale Monte Carlo simulations in a geometry-based post-analysis stage. This combination yields a dual perspective, ensuring mathematical rigour through formally verified interval arithmetic while maintaining interpretability by benchmarking against empirical Monte Carlo distributions. In this way, the framework connects theoretical verification and practical understanding, offering both provable correctness and physically meaningful insights into uncertainty propagation.

3.2 General Approach

The general methodological flow illustrated in Fig. Figure 3.1 consists of three primary stages that together form the verification-oriented circuit simulation framework. Each stage has been carefully designed to ensure consistency between the theoretical model, the numerical implementation, and the empirical evaluation.

1. **Stage I: Preparation from SPICE Netlist to MNA Model** The first stage converts the circuit description in SPICE format into a mathematically tractable system suitable for verified computations. A graph-based parser identifies circuit topology, component connections, and node potentials, mapping them to the *Modified Nodal Analysis* (MNA) formulation. This transformation yields the linear system

$$^{\mathrm{i}}\mathbf{A}\mathbf{x} = {}^{\mathrm{i}}\mathbf{b},$$

where $^{\mathrm{i}}\mathbf{A} \in \mathbb{IR}^{n\times n}$ and $^{\mathrm{i}}\mathbf{b} \in \mathbb{IR}^{n}$ represent the interval-valued system matrix and right-hand side vector, respectively. Uncertain circuit parameters—such as

component tolerances and measurement variations—are modelled as intervals, providing a natural means to capture epistemic uncertainty without assuming specific probability distributions.

To improve the conditioning of the resulting system and reduce dependency amplification in interval operations, diagonal scaling is applied to ${}^{\mathrm{i}}\mathbf{A}$. This preprocessing step minimizes the propagation of rounding and dependency errors, thereby stabilizing subsequent interval computations. The resulting scaled MNA system serves as the mathematical foundation for the verified solvers in the next stage.

2. **Stage II: Verified Solution of the Interval Linear System** The second stage focuses on computing rigorous enclosures for the uncertain circuit equations. The interval system ${}^{\mathrm{i}}\mathbf{A}\mathbf{x} = {}^{\mathrm{i}}\mathbf{b}$ is solved using a suite of interval algorithms, including the Linear Krawczyk, Gauss–Seidel, Jacobi, and Oettli–Präger formulations. Each method offers distinct trade-offs among computational cost, convergence speed, and enclosure tightness.

   All solvers employ consistent termination criteria based on geometric and algebraic conditions:

$$\max_i w({}^{\mathrm{i}}\mathbf{x}_i) < \tau_w, \qquad \| {}^{\mathrm{i}}\mathbf{A}\mathbf{x} - {}^{\mathrm{i}}\mathbf{b} \|_\infty \leq \tau_r,$$

   where τ_w controls the maximum allowable interval width and τ_r defines the residual tolerance. The first condition ensures geometric tightness of the enclosure, while the second guarantees algebraic consistency with the underlying system. If convergence stagnates before either criterion is met, ε–inflation or interval preconditioning is applied to restore contraction and continue iteration. Optionally, mean-value forms or Hansen–Sengupta operators may be used to further reduce dependency and wrapping effects (Hansen & Walster, 2003, Jaulin et al., 2001).

   Monte Carlo simulations were conducted in parallel to provide an empirical reference. Randomized sampling (up to 10^9 iterations) explored the same

uncertainty domain under probabilistic assumptions, generating the Monte Carlo point cloud after filtering invalid samples. All simulations employed independent random sampling to ensure unbiased coverage of the uncertainty space. Although stratified approaches such as Latin Hypercube Sampling (LHS) are known to reduce variance and improve sample representativeness (Boyd & Vandenberghe, 2004), they were not adopted in this study in order to maintain computational simplicity and consistency across all benchmark cases.

3. **Stage III: Geometric Evaluation and Consolidation of Results** The final stage of the framework integrates the outputs of the verified interval solvers and the Monte Carlo simulations within a unified geometric analysis environment. This stage quantifies the degree of agreement between mathematically verified enclosures and empirically sampled solution sets, thereby linking formal verification with practical circuit behaviour.

   A collection of complementary geometric and statistical indicators is employed to evaluate the correspondence between the verified and empirical results:

   - **MVEE / Khachiyan Algorithm:** The Minimum Volume Enclosing Ellipsoid (MVEE) is computed using Khachiyan's iterative method (Khachiyan, 1996, Todd, 2016) with a relative tolerance of 10^{-6} and a maximum of 10^4 iterations. The resulting ellipsoid provides a compact geometric approximation of the interval enclosure, capturing its overall anisotropy and orientation.
   - **Hausdorff Distance:** The geometric mismatch between the interval box and the Monte Carlo point cloud is quantified by the ℓ_∞–induced Hausdorff distance (Rockafellar & Wets, 2009), which measures the maximum boundary deviation along any coordinate axis. This metric serves as a direct indicator of the worst-case discrepancy between verified and empirical domains.
   - **Volume and Coverage Measures:** The conservativeness of the verified enclosures is evaluated through the ratio of the interval (or

ellipsoidal/polyhedral) volume to the empirical Monte Carlo volume. For high-dimensional systems, the logarithm of the volume ratio ($\log \mathrm{VR}$) is reported to avoid numerical overflow and to improve interpretability. Coverage metrics quantify the proportion of Monte Carlo points lying within the verified enclosure. While confidence intervals can be used to statistically bound these coverage estimates, the present study reports raw proportions to maintain a direct and implementation-consistent evaluation.

For polytope-based representations, high-dimensional volumes are estimated via randomized hit–or–miss sampling to balance computational feasibility with statistical reliability.

All geometric quantities were computed independently according to their respective definitions. Each metric (e.g., Hausdorff distance, volume ratio, and coverage) was analyzed within its own scale and interpretation, without performing direct cross-metric normalization or comparison. This approach avoids misleading inferences that could arise from combining heterogeneous geometric measures with different dimensional sensitivities.

This consolidated evaluation framework enables a twofold interpretation: (i) The verified enclosures demonstrate the guaranteed bounds of circuit responses under uncertainty, and

(ii) The Monte Carlo point cloud provides an empirical context that reveals how tight or conservative these bounds are in practice. The combined analysis thus establishes a quantitative bridge between rigorous interval verification and physically interpretable stochastic behaviour.

3.3 Julia-Based Implementation Flow

The computational framework is implemented in the Julia programming language. Julia was chosen for its strong balance between numerical performance and scientific transparency. Julia provides multiple dispatch and native interval arithmetic support.

It also offers efficient memory handling for large-scale sparse systems. These features make Julia suitable for uncertainty-aware circuit simulation.

The overall implementation pipeline reflects the same three-stage structure described previously. During preprocessing, SPICE netlists are parsed into sparse data structures. These structures define the MNA model. This graph-to-matrix conversion preserves topological information. It also enables efficient linear algebra operations. All symbolic and numeric quantities are stored as interval types using midpoint–radius representation. This reduces dependency and wrapping effects during repeated computations.

Directed rounding is globally enforced throughout the computation. The IEEE 754–compliant outward rounding mode ensures every interval operation provides a mathematically guaranteed enclosure of the true solution. This approach avoids non-deterministic rounding artifacts that could otherwise compromise reproducibility.

Several open-source Julia packages form the backbone of the implementation:

- `IntervalArithmetic.jl` for fundamental interval operations and directed rounding control,
- `IntervalLinearAlgebra.jl` for verified linear solvers and preconditioning,
- `LazySets.jl` and `Polyhedra.jl` for geometric representations and convex set operations,
- `BenchmarkTools.jl` for timing, profiling, and performance reproducibility.

Parallel execution is enabled through Julia's `@threads` macro to utilize multicore architectures, particularly in Monte Carlo sampling and geometric post-processing. When ill-conditioned systems or near-singular matrices are encountered, the arithmetic precision is automatically promoted to `BigFloat` with Multiple Precision Floating-Point(MPFR) back-end support, ensuring correct outward rounding without sacrificing the integrity of verified enclosures.

All solvers, geometric analysis modules, and benchmarking routines are defined under a modular architecture. This allows seamless parameter tuning and algorithmic replacement. This modularity ensures future extensions can be integrated with minimal modification to the core workflow. Possible extensions include symbolic differentiation, adaptive subdivision, or hybrid interval–probabilistic solvers. In this sense, the Julia-based implementation provides both a practical simulation environment and an extensible research platform for verified numerical analysis.

3.4 Integration of Interval Algorithms

The interval algorithms used for verified circuit analysis are integrated within the Julia framework as interchangeable, rigorously defined solver modules. Each solver operates on the same core data structure: an interval matrix–vector pair $({}^{i}\mathbf{A}, {}^{i}\mathbf{b})$ representing the uncertain MNA system. The objective is to compute a guaranteed enclosure ${}^{i}\mathbf{X}$ satisfying

$$ {}^{i}\mathbf{A}\ {}^{i}\mathbf{X} = {}^{i}\mathbf{b}, $$

under verified arithmetic with controlled rounding and dependency management.

To ensure numerical reliability and consistency among solvers, several algorithmic safeguards are embedded into the implementation:

- **Preconditioning:** Before iteration, diagonal and inverse-based pre-conditioners are applied to regularize nearly singular systems and reduce condition numbers.
- **ε–Inflation:** When contraction stalls, a controlled inflation of the interval widths reintroduces overlap between successive iterates to restore convergence.
- **Directed Rounding and Pivoting:** All arithmetic follows directed rounding, and interval division by small pivots is avoided through conditional pivoting strategies.
- **Precision Escalation:** When double precision becomes insufficient to preserve

enclosure validity, computations automatically switch to higher-precision arithmetic.

Each solver—Linear Krawczyk, Gauss–Seidel, Jacobi, Oettli–Präger—is encapsulated as a self-contained module with a consistent input–output interface. This design allows both single-solver execution and hybrid configurations where multiple solvers refine a shared enclosure sequentially. Termination criteria and tolerance levels are inherited from the definitions given in Stage II to maintain cross-method consistency.

Internally, interval quantities are stored in midpoint–radius form:

$$^{\mathrm{i}}\mathbf{A} = [A_c - \Delta,\ A_c + \Delta], \qquad ^{\mathrm{i}}\mathbf{b} = [b_c - \delta,\ b_c + \delta],$$

which minimizes wrapping effects during repeated matrix operations. This representation is also advantageous for hybrid symbolic–numeric methods, as the midpoint can serve as a deterministic reference solution while the radius bounds numerical uncertainty.

A key advantage of this modular integration is experimental reproducibility. Each solver's configuration, random seed, and arithmetic precision level are logged to ensure that all reported results can be independently verified. Moreover, since Julia's interval libraries are open source and IEEE 1788–compliant, the implementation is portable across different computing environments without loss of numerical fidelity.

The combined integration of verified algorithms within the Julia-based infrastructure thus establishes a transparent and extendable platform for circuit-level uncertainty quantification. It connects rigorous interval verification with efficient numerical computation, forming the foundation for the geometric evaluation and benchmarking analyses described in Stage III.

Interval Gaussian Elimination (IGE)

Adapted from classical presentations in interval analysis (Moore et al., 2009, Neumaier, 1990). Direct elimination is extended to intervals, but pivoting must ensure $0 \notin a_{kk}$. Preconditioning reduces overestimation.

Algorithm 1 Interval Gaussian Elimination (IGE)

Require: ${}^{\mathrm{i}}\mathbf{A} \in \mathbb{IR}^{n\times n}$, ${}^{\mathrm{i}}\mathbf{b} \in \mathbb{IR}^{n}$

Ensure: Verified enclosure ${}^{\mathrm{i}}\mathbf{X} \in \mathbb{IR}^{n}$ (if successful)

for $k = 1$ **to** $n-1$ **do**
  if $0 \in a_{kk}$ **then**
    Apply preconditioning and/or row/column permutations
  end if
  for $i = k+1$ **to** n **do**
    $m \leftarrow a_{ik}/a_{kk}$ ▷ interval division under outward rounding
    for $j = k$ **to** n **do**
      $a_{ij} \leftarrow a_{ij} - m\,a_{kj}$
    end for
    $b_i \leftarrow b_i - m\,b_k$
  end for
end for
Back substitution
for $i = n$ **downto** 1 **do**
  $s \leftarrow b_i - \sum_{j=i+1}^{n} a_{ij}\ {}^{\mathrm{i}}\mathbf{X}_j$
  ${}^{\mathrm{i}}\mathbf{X}_i \leftarrow s/a_{ii}$ ▷ requires $0 \notin a_{ii}$
end for

The correctness of interval Gaussian elimination relies on the assumption that all pivots satisfy $0 \notin a_{kk}$ and that outward rounding is consistently applied throughout the elimination. Under these conditions, the computed interval enclosure ${}^{\mathrm{i}}\mathbf{X}$ is guaranteed to contain the result of exact Gaussian elimination for every realization $(A, b) \in ({}^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$, and hence it encloses the true united solution set (Moore et al., 2009,

Neumaier, 1990). In practice, pivoting strategies reduce wrapping effects, but they may still introduce significant overestimation for ill-conditioned systems. Therefore, IGE is typically combined with preconditioning to enhance numerical stability, while for larger or more sensitive problems it may serve mainly as a baseline in comparison with iterative methods.

Krawczyk Operator (Linear System)

Adapted from (Neumaier, 1990). Let $R \approx A_c^{-1}$ be a point pre-conditioner; termination tolerances as in Stage II. For a candidate enclosure ${}^{\mathrm{i}}\mathbf{X}$, the Krawczyk operator is

$$K({}^{\mathrm{i}}\mathbf{X}) \;=\; m \;-\; R(A_c m - b_c) \;+\; (I - R\;{}^{\mathrm{i}}\mathbf{A})\,({}^{\mathrm{i}}\mathbf{X} - m),$$

where $m = \mathrm{mid}({}^{\mathrm{i}}\mathbf{X})$.

Algorithm 2 Krawczyk Method for ${}^{\mathrm{i}}\mathbf{A}\,{}^{\mathrm{i}}\mathbf{X} = {}^{\mathrm{i}}\mathbf{b}$

Require: $({}^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$, initial enclosure ${}^{\mathrm{i}}\mathbf{X}^{[0]}$

Compute midpoints (A_c, b_c); choose $R \approx A_c^{-1}$
$m \leftarrow \mathrm{mid}({}^{\mathrm{i}}\mathbf{X}^{[0]})$
for $k = 0, 1, 2, \ldots$ **do**
    $K \leftarrow m - R(A_c m - b_c) + (I - R\;{}^{\mathrm{i}}\mathbf{A})\,({}^{\mathrm{i}}\mathbf{X}^{[k]} - m)$
    ${}^{\mathrm{i}}\mathbf{X}^{[k+1]} \leftarrow K \cap {}^{\mathrm{i}}\mathbf{X}^{[k]}$
    if $K \subseteq \mathrm{int}({}^{\mathrm{i}}\mathbf{X}^{[k]})$ **then**
        break
    end if
    if termination criteria met **then**
        break
    end if
end for
return ${}^{\mathrm{i}}\mathbf{X}^{[k+1]}$

Oettli–Präger (OP) Formulation

Adapted from (Neumaier, 1990, Jaulin et al., 2001). With midpoint–radius form ${}^{\mathrm{i}}\mathbf{A} = [A_c - \Delta,\ A_c + \Delta]$ and ${}^{\mathrm{i}}\mathbf{b} = [b_c - \delta,\ b_c + \delta]$, the united solution set is

$$\Sigma({}^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b}) \ = \ \{\, \mathbf{x} \in \mathbb{R}^n \ : \ |A_c\mathbf{x} - b_c| \leq \Delta\,|\mathbf{x}| + \delta \,\}.$$

Fixing an orthant $s \in \{\pm 1\}^n$ yields linear inequalities.

Algorithm 3 Oettli–Präger Feasibility via Orthants

Require: (A_c, Δ), (b_c, δ)

Ensure: Union $\mathcal{X}$ of feasible enclosures

1: $\mathcal{X} \leftarrow \emptyset$
2: **for** each orthant $s \in \{\pm 1\}^n$ **do**
3: $S \leftarrow \mathrm{diag}(s)$
4: Form linear inequalities: $-\Delta S x - \delta \leq A_c x - b_c \leq \Delta S x + \delta$
5: Solve the resulting Linear Program(LP); if feasible, accumulate X_s into $\mathcal{X}$
6: **end for**
7: **return** $\mathcal{X}$

The Oettli–Präger condition is mathematically equivalent to the definition of the interval solution set, which makes this formulation exact in theory (Neumaier, 1990, Jaulin et al., 2001). By decomposing the space into orthants and reformulating the inequalities as linear constraints, the method derives all feasible solutions without introducing additional overestimation. However, the number of orthants grows exponentially with dimension, and thus the approach is computationally tractable only for small- to medium-scale systems. In higher dimensions, pruning strategies and feasibility heuristics are required to reduce the combinatorial explosion, making the OP method particularly valuable as a validation tool rather than a stand-alone solver for large problems.

Interval Jacobi

Adapted from (Hansen & Walster, 2003, Moore et al., 2009). Classical fixed-point iterations are extended to intervals. Assume a splitting $^{\mathrm{i}}\boldsymbol{A} = {}^{\mathrm{i}}\boldsymbol{D} + ({}^{\mathrm{i}}\boldsymbol{L} + {}^{\mathrm{i}}\boldsymbol{U})$ *with* $0 \notin a_{ii}$*; convergence discussed via midpoint spectral conditions in Stage II.*

Algorithm 4 Interval Jacobi Iteration

Require: $^{\mathrm{i}}\mathbf{A}$, $^{\mathrm{i}}\mathbf{b}$, initial enclosure $^{\mathrm{i}}\mathbf{X}^{[0]}$

for $k = 0, 1, 2, \ldots$ **do**
  for $i = 1$ **to** n **do**
    $s \leftarrow b_i - \sum_{j \neq i} a_{ij}\ {}^{\mathrm{i}}\mathbf{X}_j^{[k]}$
    ${}^{\mathrm{i}}\mathbf{X}_i^{[k+1]} \leftarrow s / a_{ii}$
  end for
  if Stage II tolerances satisfied **then**
    break
  end if
end for
return $^{\mathrm{i}}\mathbf{X}^{[k+1]}$

Interval Gauss–Seidel

Adapted from (Hansen & Walster, 2003, Moore et al., 2009). Uses the lower/upper splitting $({}^{\mathrm{i}}\boldsymbol{D} + {}^{\mathrm{i}}\boldsymbol{L}) + {}^{\mathrm{i}}\boldsymbol{U}$ *with* $0 \notin a_{ii}$*.*

Algorithm 5 Interval Gauss–Seidel Iteration

Require: ${}^{\mathrm{i}}\mathbf{A}$, ${}^{\mathrm{i}}\mathbf{b}$, initial enclosure ${}^{\mathrm{i}}\mathbf{X}^{[0]}$

for $k = 0, 1, 2, \ldots$ **do**
  for $i = 1$ **to** n **do**
    $s \leftarrow b_i - \sum_{j<i} a_{ij}\ {}^{\mathrm{i}}\mathbf{X}_j^{[k+1]} - \sum_{j>i} a_{ij}\ {}^{\mathrm{i}}\mathbf{X}_j^{[k]}$
    ${}^{\mathrm{i}}\mathbf{X}_i^{[k+1]} \leftarrow s / a_{ii}$
  end for
  if Stage II tolerances satisfied **then**
    break
  end if
end for
return ${}^{\mathrm{i}}\mathbf{X}^{[k+1]}$

The interval extensions of Jacobi and Gauss–Seidel iterations follow the structure of their classical counterparts but rely on the inclusion property of interval arithmetic to preserve correctness. Convergence is ensured whenever the iterative map is contractive in the interval setting, for instance when the spectral radius conditions $\rho(|\,{}^{\mathrm{i}}\mathbf{D}^{-1}({}^{\mathrm{i}}\mathbf{L} + {}^{\mathrm{i}}\mathbf{U})|) < 1$ for Jacobi or $\rho(|({}^{\mathrm{i}}\mathbf{D} + {}^{\mathrm{i}}\mathbf{L})^{-1}\,{}^{\mathrm{i}}\mathbf{U}|) < 1$ for Gauss–Seidel are satisfied in the midpoint system (Hansen & Walster, 2003, Moore et al., 2009). When these conditions hold, the iterations converge to a verified enclosure of the true solution set. These methods are particularly attractive for sparse systems, since their per-iteration cost scales with the number of non-zero entries in ${}^{\mathrm{i}}\mathbf{A}$. Nevertheless, dependency and wrapping effects may still slow contraction, in which case enhancements such as mean-value forms or Hansen–Sengupta corrections are employed to improve enclosure tightness (Hansen & Walster, 2003, Jaulin et al., 2001).

3.5 Monte Carlo Benchmark Protocol

Monte Carlo simulations were conducted by randomly sampling uncertain circuit parameters according to their prescribed probability distributions, typically Gaussian or uniform laws depending on the component tolerances. For each sample, the nominal SPICE-like analysis was executed to obtain the corresponding circuit response. The number of iterations was varied between 10^5 and 5×10^6, balancing computational effort and statistical accuracy. To accelerate the process, parallelization was employed across 8 CPU cores, and fixed random seeds were used to ensure reproducibility of the experiments. In addition to simple random sampling, variance-reduction techniques such as stratified Latin Hypercube Sampling were adopted to improve the representativeness of the sample set, thereby yielding better coverage of the tolerance domain with fewer samples (Boyd & Vandenberghe, 2004). The Monte Carlo point cloud X_{MC} is not an exact representation of the united solution set $\Sigma({}^{\mathrm{i}}\mathbf{A}, {}^{\mathrm{i}}\mathbf{b})$, but serves as a reliable empirical reference. Its accuracy is characterized by the well-known Monte Carlo convergence rate $\mathcal{O}(1/\sqrt{N})$, where N is the number of samples. Accordingly, statistical error bars were attached to all Monte Carlo-based quantities. In particular, binomial confidence intervals were computed using the Clopper–Pearson method for coverage-type probabilities, and bootstrap resampling was employed for distribution-dependent metrics, ensuring rigorous uncertainty quantification of the empirical reference (Moore et al., 2009, Hansen & Walster, 2003).

3.6 Benchmarking Metrics and Performance Criteria

The geometric and statistical outputs of Stage III form the basis for the quantitative benchmarking presented in this section. To assess the efficiency, robustness, and practical usability of the proposed verified simulation framework, a set of quantitative benchmarking metrics is defined. These metrics evaluate both numerical performance and the quality of the verified enclosures relative to conventional Monte Carlo results.

Numerical Efficiency and Convergence

Computational performance is evaluated in terms of the total execution time and the number of iterations required for convergence in each solver. Execution time (t_{CPU}) is measured from the initialization of the solver up to the final enclosure output, excluding input parsing and visualization overhead. Convergence rate is characterized by the number of contraction steps required to satisfy the predefined width and residual thresholds described in Stage II. A solver is considered to have converged if both criteria are met or if the iteration count exceeds a fixed upper limit, in which case the last valid enclosure is recorded.

Enclosure Quality and Tightness

The quality of the verified interval enclosures is assessed using multiple complementary indicators. The primary measure is the *relative enclosure width*,

$$\eta = \frac{1}{n} \sum_{i=1}^{n} \frac{w(^{\mathrm{i}}\mathbf{x}_i)}{|\mathrm{mid}(^{\mathrm{i}}\mathbf{x}_i)| + \epsilon},$$

which expresses the average uncertainty in normalized form while preventing division instability through a small offset ϵ. Smaller values of η indicate tighter enclosures and higher numerical precision.

In addition, the geometric comparison metrics introduced in Stage III are used as secondary quality indicators. They collectively capture how the verified interval domain relates to the empirical Monte Carlo cloud in both size and shape, providing insight into the conservativeness of the computed enclosures.

Stability and Robustness

Solver robustness is examined by evaluating the sensitivity of convergence and enclosure width to perturbations in circuit parameters. Each benchmark is tested

under multiple uncertainty levels and condition numbers of $^{i}\mathbf{A}$ to determine the stability of interval propagation and the effectiveness of diagonal scaling. If a solver diverges or produces unbounded intervals, the case is flagged as unstable, and the respective results are excluded from aggregate statistical analysis. This procedure ensures that conclusions about efficiency or tightness are not biased by pathological divergence behavior.

Comparative Evaluation Framework

All solvers and Monte Carlo benchmarks are executed under identical computational settings in Julia, using the same random seeds and machine precision. Performance metrics are normalized with respect to the baseline Monte Carlo execution time to highlight the computational advantage or overhead introduced by interval verification. Results are compared across solvers in a relative rather than absolute manner, emphasizing contextual trade-offs between accuracy and computational cost that depend on circuit topology and uncertainty level.

Overall, this benchmarking framework provides a balanced evaluation of the proposed methodology in terms of computational efficiency, numerical rigour, and practical interpretability. The combination of algorithmic, geometric, and empirical indicators forms a comprehensive performance profile that enables objective comparison with classical stochastic simulation techniques.

3.7 Methodological Contributions

The proposed methodology contributes in several distinctive ways. First, it establishes a Julia-based framework that integrates multiple verified solvers, including Gaussian elimination, Krawczyk, Jacobi, Gauss–Seidel, and Oettli–Präger methods—into a unified environment. Second, it systematically benchmarks these interval solvers against large-scale Monte Carlo experiments, using a diverse set of geometric and statistical metrics that provide local and global assessments of

enclosure quality. Third, the framework introduces geometric refinements such as ellipsoidal and polyhedral representations, enabling more faithful approximations than axis–aligned boxes alone. Finally, combining theoretical guarantees with practical runtime profiling yields a balanced methodology: interval methods provide rigorous mathematical enclosures, while Monte Carlo simulations supply empirical evidence and scalability insights. This dual perspective is especially relevant for safety–critical circuit analyses, where correctness and computational tractability are indispensable (Hansen & Walster, 2003, Moore et al., 2009, Neumaier, 1990).

CHAPTER FOUR
SIMULATIONS AND RESULTS

4.1 Data Sets Used In Simulations

In order to evaluate the performance of the proposed interval-based approaches and to establish a fair comparison with Monte Carlo analysis, a set of benchmark circuits with well-defined parameter tolerances was employed. These data sets were chosen to cover a spectrum of circuit complexities, ranging from the simplest first-order low-pass filter to more involved multi-stage and resonant configurations. The rationale for this selection was twofold: first, to ensure that the interval methods can be validated against cases with analytically predictable responses, and second, to assess their behavior in circuits where parameter interactions become highly non-linear.

4.1.1 Simple RC Low-Pass Filter

The first test case is a single resistor–capacitor network with nominal values $R = 1\,\mathrm{k}\Omega$ and $C = 100\,\mathrm{nF}$. This structure provides a baseline for frequency-response analysis under parameter uncertainties.

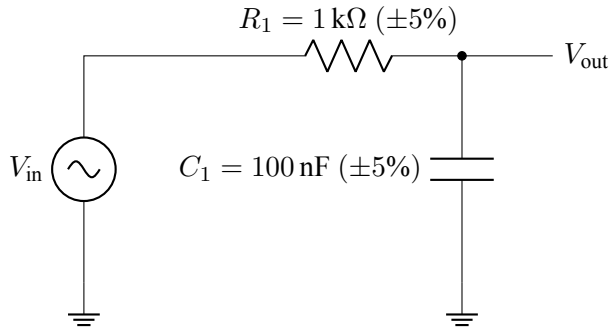


Figure 4.1 Simple RC low-pass filter with component tolerances.

Figure 4.1 depicts the one–pole RC low–pass used as the baseline dataset. The single tolerance loop (5% on R_1 and C_1) makes it ideal to contrast deterministic interval bands against probabilistic (Gaussian/Uniform) spreads and to verify that enclosure widths evolve smoothly around the cut–off region.

4.1.2 RLC Notch Filter

The second benchmark is a parallel RLC notch circuit fed through a source resistance and terminated with a load. This topology is highly sensitive to parameter variations and thus serves as a stringent test case for uncertainty quantification.

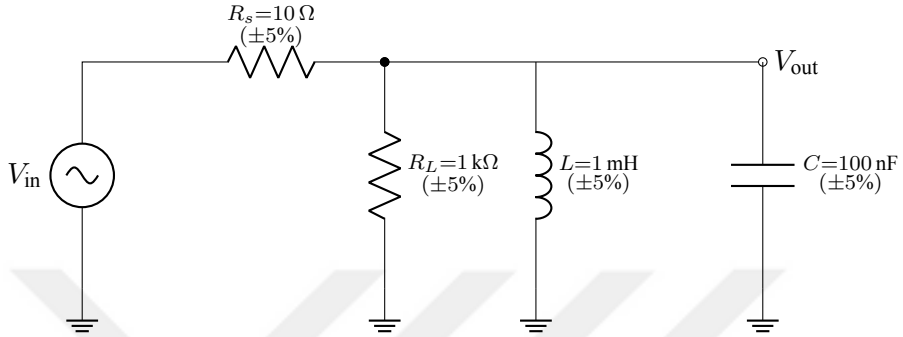


Figure 4.2 RLC parallel (notch) filter with series source resistor and load.

Figure 4.2 shows the parallel RLC notch with a series source resistor and a finite load. Because the attenuation peak is set by a resonance, small perturbations on L and C translate into pronounced frequency shifts—this circuit therefore stresses the methods' tail behaviour and coverage guarantees around the notch.

4.1.3 Complex RLC Network

The third circuit is a series R–L branch combined with a shunt capacitor, followed by a series output resistance and a load. This configuration captures the interplay of multiple uncertain elements across different energy-storing components.

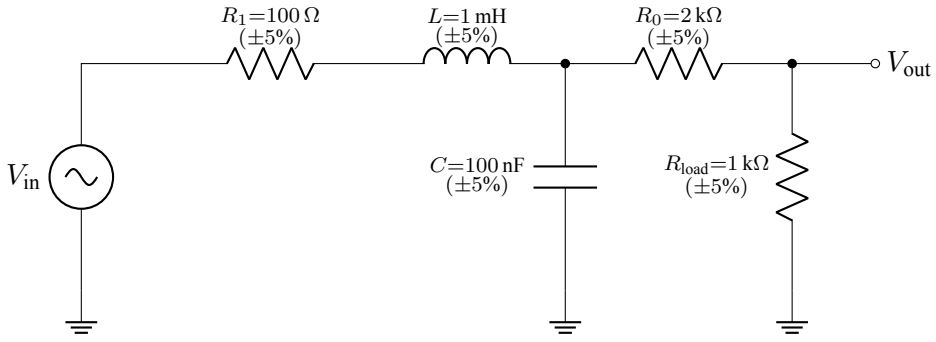


Figure 4.3 Complex RLC example: series R–L, shunt C, series R_0, and load.

Figure 4.3 combines a series R–L, a shunt C, and an output network (R_0, R_{load}). The mixed energy–storage paths create higher–order dynamics and parameter couplings, so this case is used to assess scalability and the impact of dependency on interval sharpness.

4.1.4 Two-Stage RC Low-Pass Filter

The final test case is a cascaded filter structure composed of two RC sections. The multi-stage topology enables the assessment of error propagation and accumulation in interval enclosures.

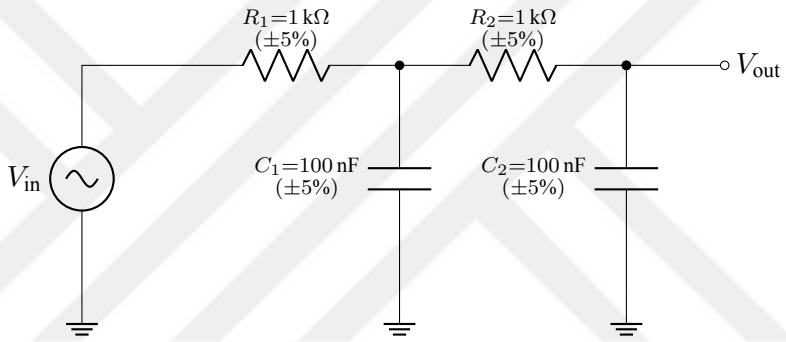


Figure 4.4 Two-stage RC low-pass filter (multi-stage) with component tolerances.

Figure 4.4 is a two–stage RC cascade that accumulates uncertainties across sections. It is used to quantify how enclosure widths grow with depth and to compare interval boxes against Monte Carlo ranges under serial propagation.

4.1.5 Circuit Types Considered

Four representative circuit topologies were selected for the simulations:

- **Simple RC Low-Pass Filter:** A single RC section that serves as a baseline.
- **RLC Notch Filter:** A parallel RLC branch fed by a source resistor and terminated with a load.

- **Complex RLC Network:** Series R–L, shunt C, followed by series R_0 and a load.
- **Two-Stage RC Low-Pass Filter:** Two cascaded RC sections to study error propagation.

4.1.6 Parameter Specifications

Each circuit element was assigned a nominal value together with a tolerance band reflecting realistic manufacturing variations. Specifically, resistors, capacitors, and inductors were modeled with $\pm 5\%$ tolerance. These ranges define the uncertainty intervals within which the true parameter values are assumed to lie. Table 4.1 summarizes the nominal values and bounds.

Table 4.1 Nominal values and tolerance ranges for components used in the simulated circuits

Component	**Nominal**	**Tolerance**	**Interval**
R_1	1 kΩ	$\pm 5\%$	$[0.95,\ 1.05]$ kΩ
C_1	100 nF	$\pm 5\%$	$[0.95,\ 1.05]$ nF
L_1	1 mH	$\pm 5\%$	$[0.95,\ 1.05]$ mH
R_0 (series)	2 kΩ	$\pm 5\%$	$[1.90,\ 2.10]$ kΩ
R_{load}	1 kΩ	$\pm 5\%$	$[0.95,\ 1.05]$ kΩ

Table 4.1 lists nominal values and $\pm 5\%$ tolerances used throughout the simulations. These bounds define the interval domains for all deterministic methods and the sampling ranges for the Monte Carlo baselines.

4.1.7 Analysis Approaches

For each circuit, simulations were carried out using two complementary approaches:

1. **Interval-Based Methods:** Jacobi, Gauss–Seidel, Hansen–Bliek–Rohn, Oettli–

Präger, and Krawczyk operators (see Chapter 2.3). These generate deterministic enclosures that are guaranteed to contain the true circuit responses under the specified uncertainties.

2. **Monte Carlo Sampling:** Parameters drawn from uniform and Gaussian laws within the tolerance ranges (Chapter 2.4). Parallel runs with up to 2×10^5 samples per case were used for robust statistics.

4.2 Performance Criteria

To systematically evaluate interval-based methods against Monte Carlo, we considered:

(i) computational efficiency (runtime, peak memory; cf. Section 2.3.2),

(ii) accuracy/sharpness (interval widths vs. probabilistic ranges; Hausdorff metrics; cf. Section 2.5),

(iii) reliability/coverage (fraction of MC samples enclosed; tail-behavior across 0–5% and 95–100% quantiles), and

(iv) scalability/practicality across the circuit set of Section 4.1. This multi-criteria view balances theoretical guarantees with engineering relevance.

4.3 Tests and Results

We benchmarked the Oettli–Präger formulation, the Krawczyk operator, and interval Gauss–Seidel/Jacobi iterations against Monte Carlo. Metrics included runtime, memory, sharpness, coverage, and geometric discrepancy.

4.3.1 Comparison of Interval Methods and Monte Carlo

Table 4.2 Nominal values and tolerance ranges for components used in the simulated circuits

Component	**Nominal**	**Tolerance**	**Interval**
R_1	1 kΩ	$\pm 5\%$	$[0.95,\ 1.05]$ kΩ
C_1	100 nF	$\pm 5\%$	$[0.95,\ 1.05]$ nF
L_1	1 mH	$\pm 5\%$	$[0.95,\ 1.05]$ mH
R_0 (series)	2 kΩ	$\pm 5\%$	$[1.90,\ 2.10]$ kΩ
R_{load}	1 kΩ	$\pm 5\%$	$[0.95,\ 1.05]$ kΩ

Table 4.2 summarizes computational cost by method on the 3×3 benchmark. Gauss–Seidel and Jacobi deliver sub–millisecond runs with negligible memory, Krawczyk adds modest overhead, while Oettli–Präger is costlier but yields polytope geometry. Monte Carlo attains accuracy only at the expense of orders–of–magnitude more time and memory.

Table 4.3 Solution bounds obtained by interval methods and Monte Carlo (3×3 system)

Method	x_1	x_2	x_3
Gauss–Seidel	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$
Jacobi	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$
Krawczyk	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$	$[-0.6667,\ 0.6667]$
Monte Carlo	$[-0.6110,\ 0.6171]$	$[-0.6166,\ 0.6138]$	$[-0.6200,\ 0.6150]$

Table 4.3 shows that all interval methods return the same conservative box $[-\frac{2}{3}, \frac{2}{3}]^3$, while Monte Carlo produces tighter min–max bounds (about 8–10% smaller). This illustrates the classic trade–off between guaranteed enclosure and probabilistic sharpness.

Table 4.4 Hausdorff distance analysis between interval methods and Monte Carlo

Comparison	**Distance**
Monte Carlo $\rightarrow$ Polytopes	0.429
Polytopes $\rightarrow$ Monte Carlo	0.490–0.494
Monte Carlo $\leftrightarrow$ Ellipsoid	0.636–0.637

Table 4.4 quantifies geometric mismatch. Polytopes are closer to the Monte Carlo cloud (smaller Hausdorff) than ellipsoids or large boxes, supporting the use of Oettli–Präger when shape fidelity matters.

4.3.2 Monte Carlo Benchmark Results

We also considered a canonical 3×3 uncertain linear system,

$$
{}^{\mathrm{i}}\mathbf{A} = \begin{bmatrix} [4.5, 4.5] & [-0.0, 2.0] & [-0.0, 2.0] \\ [-0.0, 2.0] & [4.5, 4.5] & [-0.0, 2.0] \\ [-0.0, 2.0] & [-0.0, 2.0] & [4.5, 4.5] \end{bmatrix}, \quad {}^{\mathrm{i}}\mathbf{b} = \begin{bmatrix} [-1.0, 1.0] \\ [-1.0, 1.0] \\ [-1.0, 1.0] \end{bmatrix}, \tag{4.1}
$$

whose midpoint condition number is 1.86 with determinant 79.625.

4.3.2.1 Performance Results

Table 4.5 Monte Carlo performance summary for different sample sizes

Test	**Samples**	**Valid**	**Success (%)**	**Time (s)**	**Threads**
100K (sequential)	100,000	100,000	100.00	0.17	1
100K (parallel)	100,000	100,000	100.00	0.38	8
500K (sequential)	500,000	500,000	100.00	1.13	1
500K (parallel)	500,000	500,000	100.00	0.45	8
1M (sequential)	1,000,000	1,000,000	100.00	2.47	1
1M (parallel)	1,000,000	1,000,000	100.00	0.92	8
5M (sequential)	5,000,000	5,000,000	100.00	11.33	1
5M (parallel)	5,000,000	5,000,000	100.00	5.05	8

Table 4.5 confirms linear–like scaling with samples and the benefit of parallelism at large N (2–2.7× speedup for $\geq$1M), while small runs suffer from threading overhead.

4.3.2.2 Component-wise Statistics

Table 4.6 Component intervals (min/mean/max) for Monte Carlo solutions

Test	x_1	x_2	x_3
100K (seq.)	$-0.47/0.00/0.47$	$-0.50/0.00/0.45$	$-0.46/0.00/0.45$
100K (par.)	$-0.45/0.00/0.44$	$-0.47/0.00/0.45$	$-0.48/0.00/0.44$
500K (seq.)	$-0.52/0.00/0.47$	$-0.48/0.00/0.53$	$-0.48/0.00/0.48$
500K (par.)	$-0.45/0.00/0.48$	$-0.48/0.00/0.49$	$-0.48/0.00/0.47$
1M (seq.)	$-0.50/0.00/0.51$	$-0.50/0.00/0.48$	$-0.51/0.00/0.50$
1M (par.)	$-0.46/0.00/0.49$	$-0.48/0.00/0.55$	$-0.53/0.00/0.50$
5M (seq.)	$-0.50/0.00/0.53$	$-0.51/0.00/0.53$	$-0.54/0.00/0.50$
5M (par.)	$-0.50/0.00/0.52$	$-0.50/0.00/0.55$	$-0.53/0.00/0.50$

Table 4.6 reports min/mean/max by component. Distributions are centred near zero and become better resolved at the extremes as the sample size increases, as expected from order–statistics.

4.3.2.3 Error Statistics

Table 4.7 Error statistics for Monte Carlo runs

Test	**Min Error**	**Mean Error**	**Max Error**
100K (sequential)	0	9.36×10^{-17}	6.00×10^{-16}
100K (parallel)	0	9.32×10^{-17}	5.77×10^{-16}
500K (sequential)	0	9.36×10^{-17}	6.28×10^{-16}
500K (parallel)	0	9.35×10^{-17}	7.02×10^{-16}
1M (sequential)	0	9.34×10^{-17}	6.75×10^{-16}
1M (parallel)	0	9.34×10^{-17}	7.02×10^{-16}
5M (sequential)	0	9.35×10^{-17}	7.45×10^{-16}
5M (parallel)	0	9.35×10^{-17}	7.20×10^{-16}

Table 4.7 shows machine–precision residuals across all runs (mean $\approx 10^{-17}$), validating the numerical implementation of the solver used within the Monte Carlo loop.

4.3.3 Dedicated 3×3 Benchmark System Results

To inspect the interplay between interval enclosures, polytopes, and stochastic distributions in detail, we generated 20,000 Monte Carlo samples and computed eight Oettli–Präger polytopes. Figure 4.5 overlays all MC points, the eight polytopes, the minimum-volume enclosing ellipsoid (MVEE), and interval boxes from the Krawczyk, Gauss–Seidel, and Jacobi iterations.

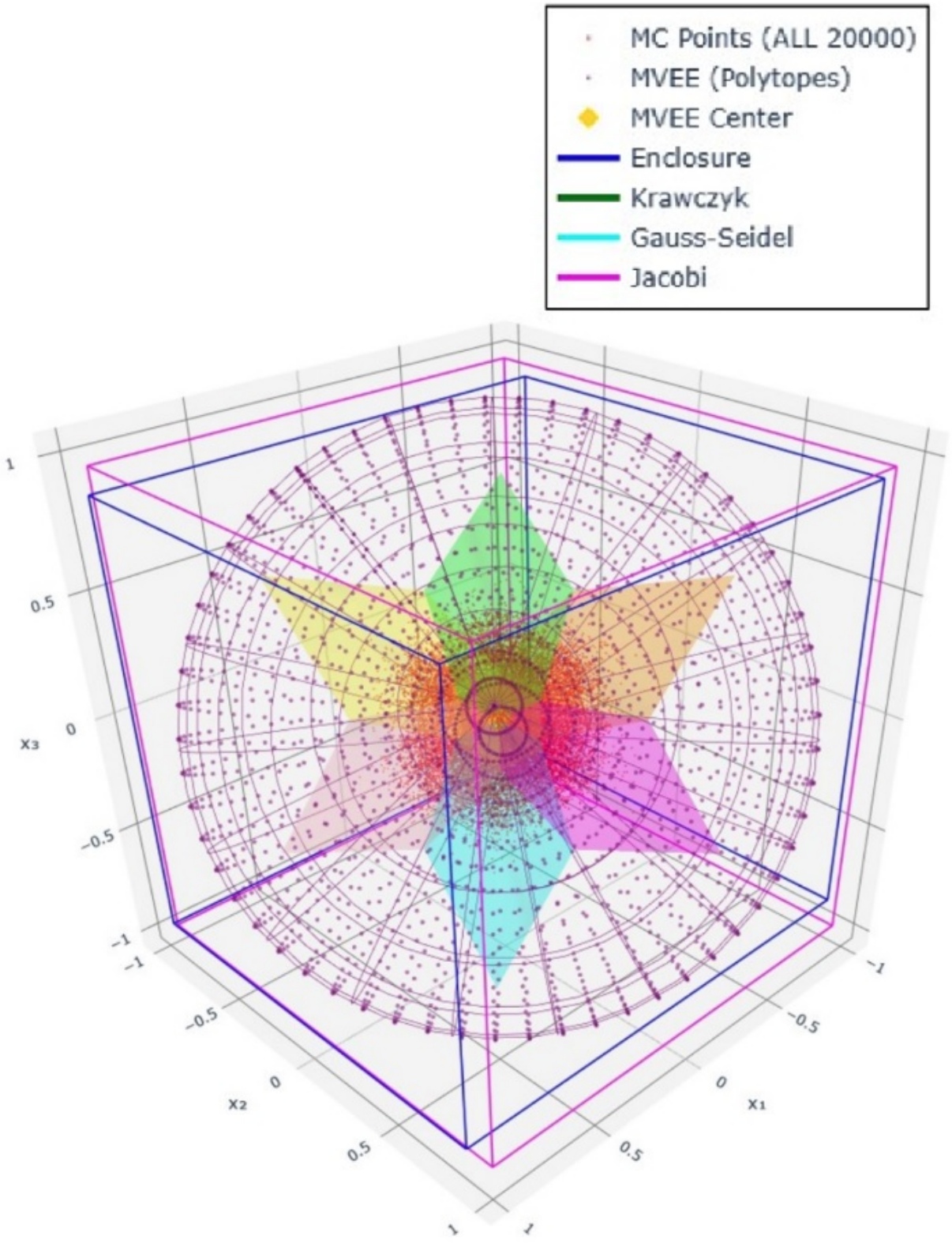


Figure 4.5 Complete 3D visualization for the 3×3 benchmark: 20k MC points (dots), eight Oettli–Präger polytopes (colored), MVEE (purple), and interval boxes (Krawczyk, Gauss–Seidel, Jacobi).

Figure 4.5 overlays 20k MC points, eight Oettli–Präger polytopes, the MVEE and interval boxes. The Monte Carlo cloud sits well inside the interval boxes, while the polytope union tracks its geometry most closely; the MVEE provides a compact but looser outer shape.

4.3.3.1 Monte Carlo summary.

Bounds, mean, and standard deviations of the Monte Carlo cloud are given in Table 4.8.

Table 4.8 MC cloud statistics (20k samples)

	x_1	x_2	x_3
Min–Max	$[-0.4113,\ 0.4167]$	$[-0.4054,\ 0.4072]$	$[-0.4098,\ 0.4166]$
Mean	0.000451	-0.000936	-0.001694
Std.	0.147815	0.147754	0.147392

Table 4.8 summarizes the MC cloud. Similar standard deviations along all axes indicate near–isotropic spread, consistent with the symmetry in the underlying system.

4.3.3.2 Polytope and ellipsoid geometry.

Eight polytopes were obtained (each with eight vertices), producing 64 vertices in total and 27 unique vertices. Their axis-wise bounds were $x_1, x_2, x_3 \in [-0.6939, 0.6939]$. The MVEE (computed from the unique vertices) had semi-axes $[1.2129,\ 1.2113,\ 0.3900]$ and center $[-4\times10^{-4},\ -6\times10^{-4},\ 1.6\times10^{-3}]$.

Table 4.9 Interval boxes and polytope volumes for the 3×3 benchmark

Method	**Widths** (x_1, x_2, x_3)	**Volume**	**Vol. Ratio**†	**Note**
Enclosure	$(2.0383,\ 1.7931,\ 1.7931)$	6.5536	2.452	Outer box
Krawczyk	$(1.9355,\ 1.9355,\ 1.9355)$	7.2505	2.713	Interval box
Gauss–Seidel	$(1.9355,\ 1.9355,\ 1.9355)$	7.2505	2.713	Interval box
Jacobi	$(1.9355,\ 1.9355,\ 1.9355)$	7.2505	2.713	Interval box
Polytopes	–	2.6726	1.000	Union (ref.)

† Volume ratio with respect to the polytope union volume.

Table 4.9 compares volumes: the polytope union is the reference (tightest), whereas interval boxes are 2.45–2.71× larger. The "Enclosure" outer box is intentionally conservative and thus volumetrically looser than the polytope union.

4.3.3.3 Hausdorff and efficiency metrics.

Bidirectional Hausdorff distances and a compact efficiency score[1] are given in Table 4.10.

Table 4.10 Geometric discrepancy and efficiency (lower Hausdorff is better, higher efficiency is better)

Method	**Hausdorff Distance**	**Volume**	**Efficiency**
Enclosure	1.3473	6.5536	0.174
Krawczyk	1.3970	7.2505	0.154
Gauss–Seidel	1.3347	7.2505	0.158
Jacobi	1.3673	7.2505	0.156
Polytopes	0.5012	2.6726	0.666

Table 4.10 ties together geometry and size. Polytopes achieve the smallest Hausdorff distance and the best composite efficiency score, highlighting their advantage when geometric fidelity is required.

[1] Efficiency defined here as a normalized score increasing with smaller volume and smaller Hausdorff distance, scaled so that the polytope union scores the best among the compared sets.

4.3.3.4 Runtime and memory micro-benchmarks.

Using identical code paths, the following micro-benchmarks were observed (Table 4.11). MVEE over 27 vertices dominates memory among the geometric post-processors.

Table 4.11 Micro-benchmarks for the 3×3 benchmark (single run)

Algorithm	**Time**	**Memory**
Monte Carlo (10k pts)	~ 0.01 s	0.2 MB
Linear Oettli–Präger (8 polys)	1.28 ms	0.33 MB
Linear Krawczyk	0.08 ms	0.01 MB
Gauss–Seidel	0.02 ms	0.01 MB
Jacobi	0.02 ms	0.01 MB
MVEE (27 vertices)	37.94 ms	8.8 MB

Table 4.11 shows micro–benchmarks with identical code paths. Iterative interval solvers are essentially instantaneous at this size; MVEE dominates the post–processing cost due to its iterative nature.

4.3.4 Circuit Case Studies

4.3.4.1 Simple RC Filter

Figure 4.6–Figure 4.8 present the magnitude, phase, Nyquist plots, and 3D sampling for the RC filter. Interval bands fully enclose Gaussian and uniform realizations; Monte Carlo bands are tighter, as expected.

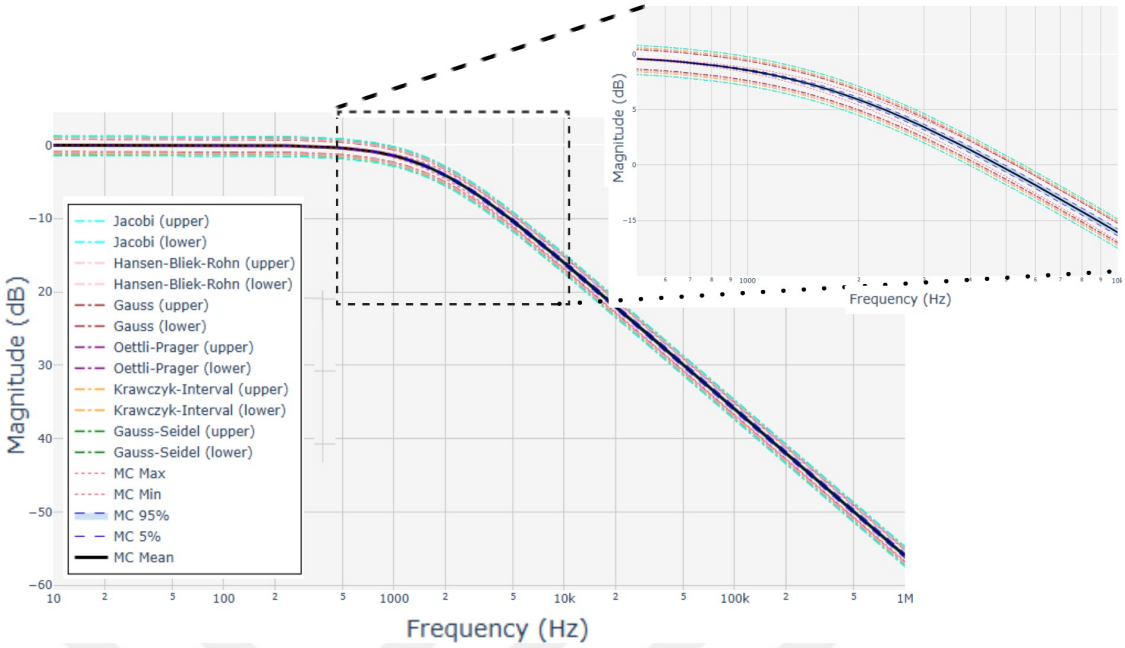


Figure 4.6 Magnitude response of the RC filter under uncertainty, with the inset providing a zoomed-in view around the corner frequency.

Figure 4.6 compares magnitude bands. Interval bounds cover all realizations across frequency; Monte Carlo (5–95%) is tighter near passband and transition, illustrating probabilistic sharpness without guarantees.

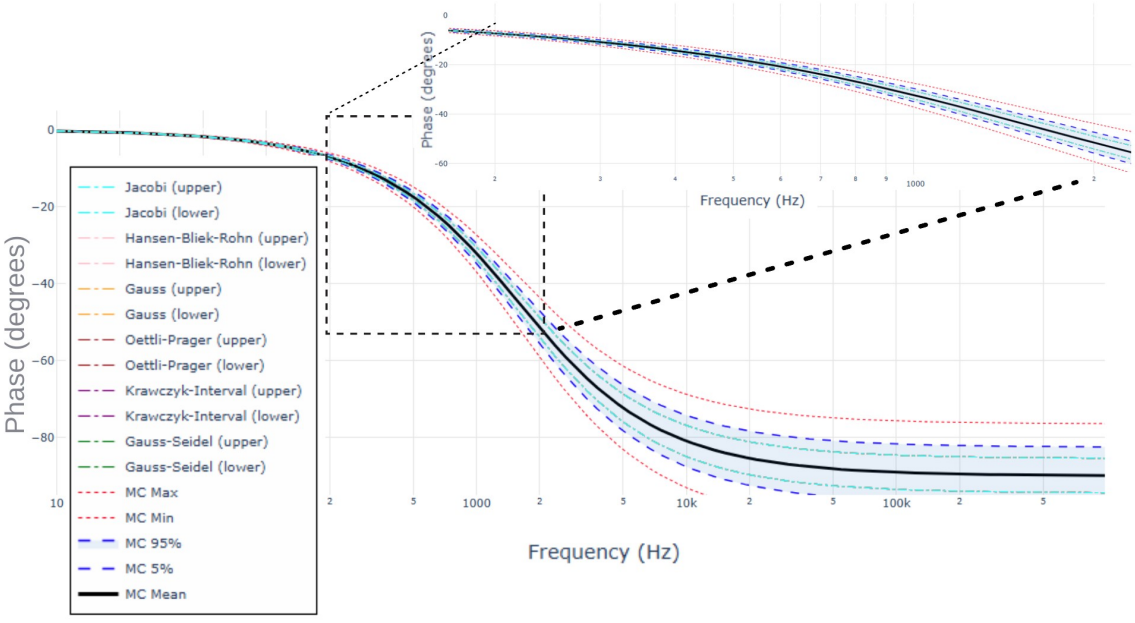


Figure 4.7 Phase response of the RC filter under uncertainty, with the inset providing a zoomed-in view around the corner frequency.

Figure 4.7 shows phase uncertainty. The largest spread occurs around the cut–off, where parameter perturbations most strongly affect the pole location; interval bands conservatively envelop the stochastic percentiles.

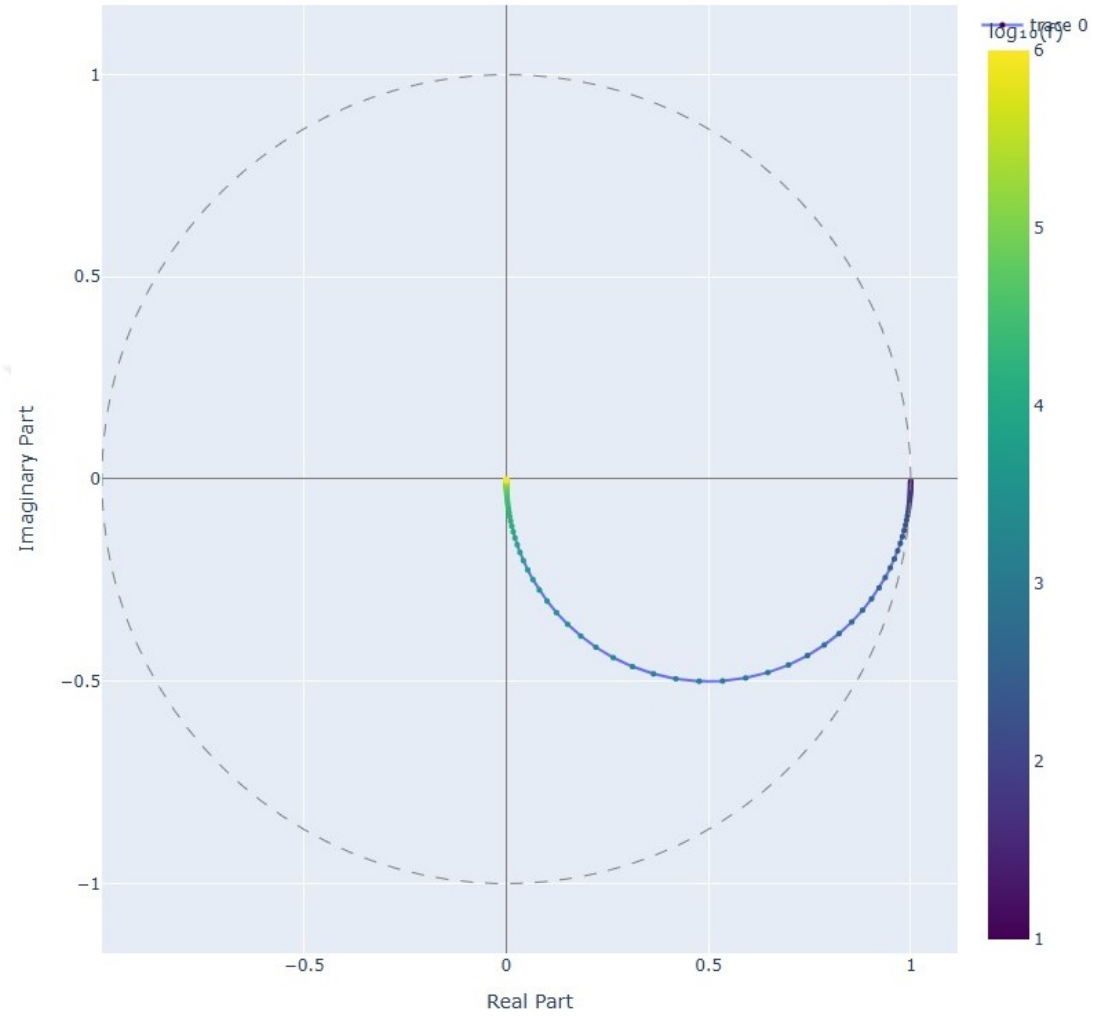


Figure 4.8 Nyquist diagram of the RC filter under uncertainty.

Figure 4.8 (Nyquist) confirms stable trajectories confined to the right–half circle. The colour scale (frequency) highlights how uncertainty shrinks at very low/high frequencies and peaks around the transition region.

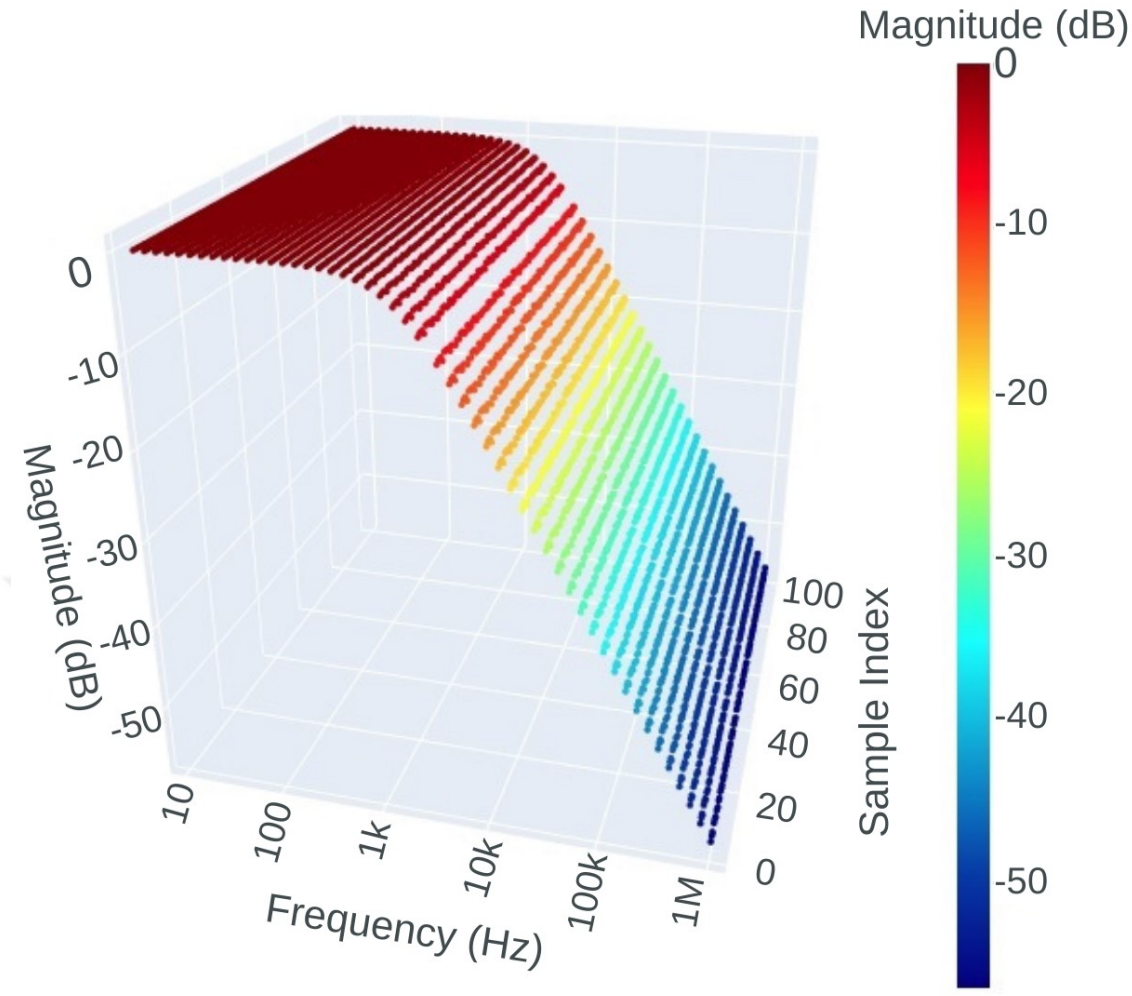


Figure 4.9 Monte Carlo sampling distribution for the RC filter.

Figure 4.9 illustrates the 3D distribution of sampled responses (log–frequency, sample index, magnitude). The layered structure indicates consistent ordering across samples and emphasizes the narrow probabilistic spread relative to interval boxes.

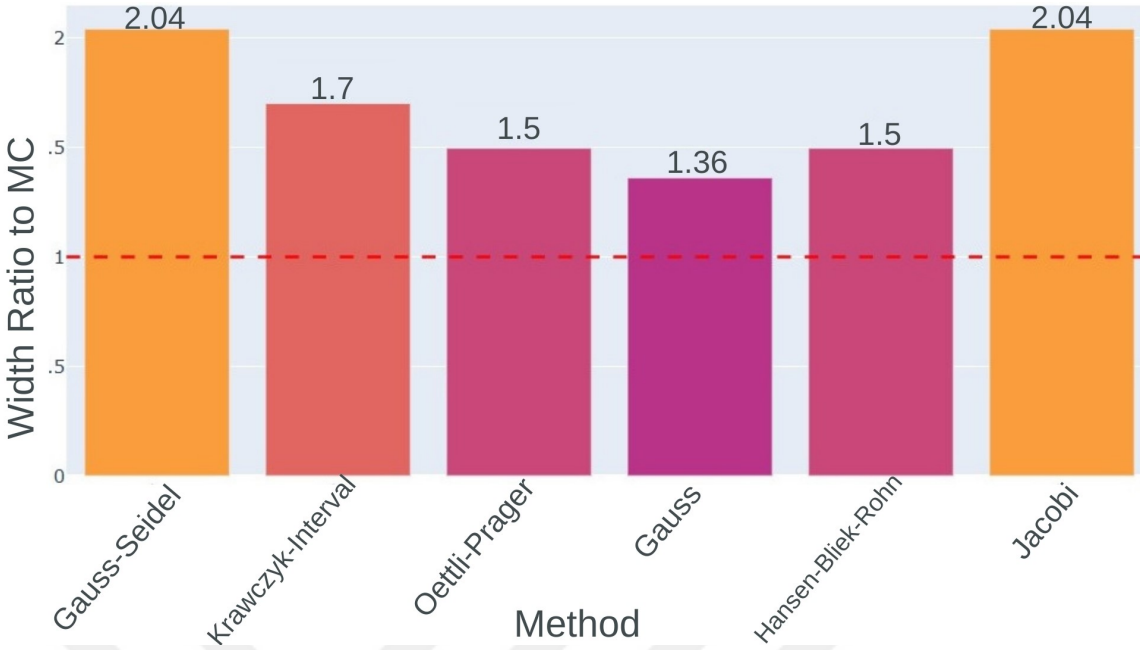


Figure 4.10 Interval width ratio to MC range at 2983.6 Hz (RC filter).

Figure 4.10 reports interval–to–MC width ratios at 2983.6 Hz. Ratios between 1.36 and 2.04 quantify the conservative gap of interval enclosures relative to the MC range for this operating point.

4.3.4.2 RLC Notch Filter

Figure 4.11–Figure 4.12 show the Monte Carlo sampling cloud and interval–MC width comparison for the notch circuit. Interval enclosures provide full coverage, whereas MC captures the most likely behaviour with narrower tails.

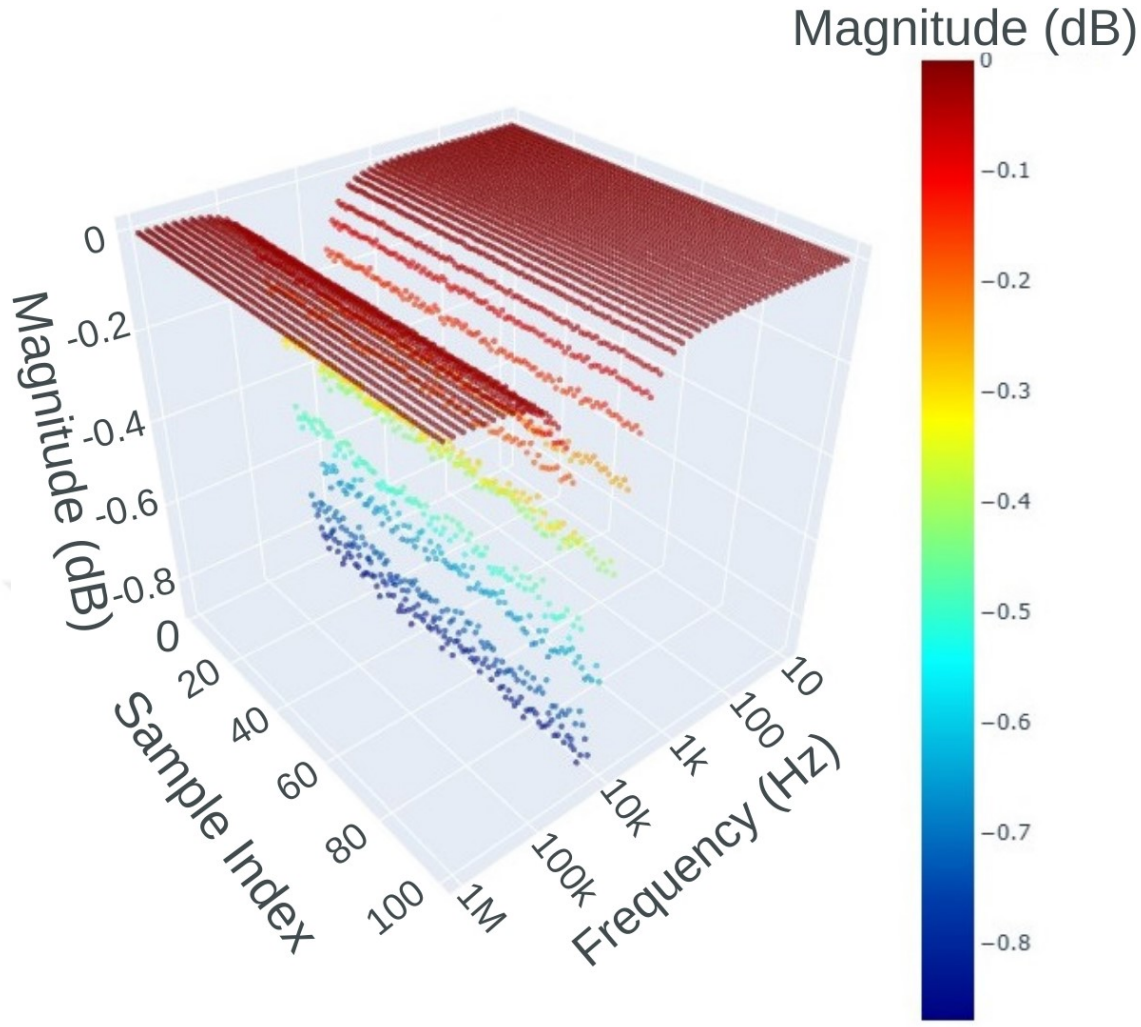


Figure 4.11 Monte Carlo sampling distribution for the notch circuit.

Figure 4.11 shows the notch filter's Monte Carlo cloud. The anisotropic spread around the notch frequency reflects resonance sensitivity to L and C, with tails that are broader under uniform than Gaussian draws.

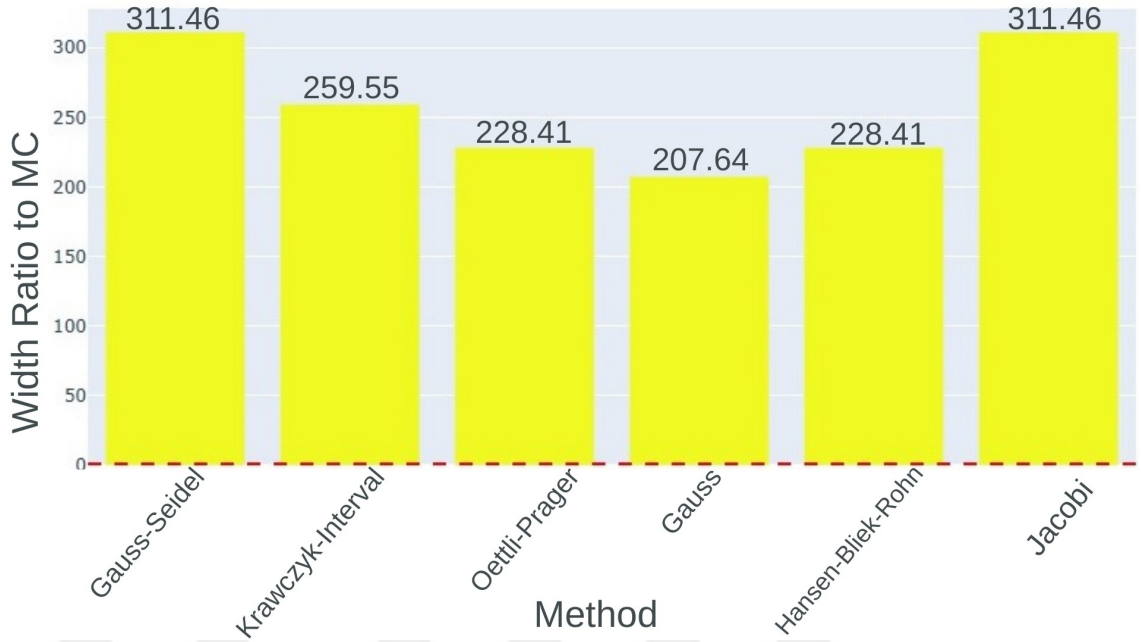


Figure 4.12 Interval width ratio to MC range at 2983.6 Hz (notch circuit).

Figure 4.12 compares interval widths to MC at 2983.6 Hz. While all interval methods enclose the Gaussian 99% band, some are looser; the plot quantifies how method choice impacts tightness at resonance.

4.3.4.3 Complex RLC and Multi-Stage Filters

A similar trend is observed in higher-order cases: interval methods yield guaranteed enclosures, while MC provides sharper but non-deterministic spreads. A cross-circuit summary is given in Table 4.12.

Table 4.12 Cross-circuit comparison under uncertainty

Circuit	**Cutoff (Hz)**	**DC Gain (dB)**	**Best Method**	**Width Ratio**	**Coverage (%)**
Multi-stage	413.2	−2.28	Gauss	1.18	83.3
Simple RC	1668.1	0.00	Gauss	1.36	100.0
Complex RLC	—	−0.83	Gauss	14.04	100.0
RLC Notch	—	0.00	Gauss	207.64	100.0

Table 4.12 summarizes per–circuit metrics. The multi–stage case exposes partial coverage for uniform extremes (83.3%), whereas single–stage and resonant examples are fully covered; "Gauss" here denotes the tightest of the interval variants for that case.

4.3.5 Overall Assessment

- **Speed:** Gauss–Seidel and Jacobi are best for speed-critical analysis.
- **Balance:** Krawczyk offers a good compromise between performance and reliability.
- **Geometric fidelity:** Oettli–Präger polytopes most closely match MC geometry (best Hausdorff/volume).
- **Determinism:** Monte Carlo produces tighter ranges but lacks deterministic guarantees; interval methods guarantee coverage by construction.

Overall, interval analysis provides a rigorous alternative to conventional Monte Carlo for uncertainty-aware circuit analysis. The choice of method hinges on the trade-off between computational cost, required sharpness, and the level of guarantee demanded by the engineering task.

CHAPTER FIVE
CONCLUSION

This thesis has investigated the application of interval analysis and Monte Carlo methods to solve uncertain linear systems and circuit problems. The results of this comparative study revealed the complementary strengths of these two approaches. While Monte Carlo simulations offer statistical flexibility and scalability, interval methods provide mathematically rigorous enclosures of the solution sets, essential for verified and safety-critical applications.

Among the interval-based techniques, iterative operators such as Gauss–Seidel and Krawczyk consistently produced tighter enclosures than direct Gaussian elimination, demonstrating their practical efficiency. The Oettli–Präger formulation was the most accurate in characterizing the exact solution set and thus serves as a firm theoretical reference. Nevertheless, its computational cost increases significantly with system size, limiting its scalability. The Interval Newton method further underscored the connection between validated numerical analysis and system stability, since its reliance on the interval derivative naturally captured system sensitivity under parameter uncertainty.

Monte Carlo methods, on the other hand, were highly effective in handling large-scale test cases and provided valuable statistical measures such as coverage probability, percentile distributions, and variance estimates. However, their inability to provide guaranteed worst-case bounds highlights their role as complementary rather than stand-alone tools when rigorous verification is required.

The experimental evaluation applied a comprehensive set of metrics—including coverage probability, Hausdorff distance, and volume comparisons—to a diverse set of benchmark RC and RLC circuits. These case studies confirmed that interval methods, despite sometimes being conservative, are indispensable for applications requiring guaranteed bounds, whereas Monte Carlo methods remain attractive for exploratory analysis and performance assessment.

Contributions. Beyond the comparative analysis, this work presented several methodological advances: an enhanced Oettli–Präger implementation with iterative refinement, streaming Monte Carlo simulations capable of processing billions of samples efficiently, and a threaded implementation of the Khachiyan algorithm for minimum-volume enclosing ellipsoids. In addition, adaptive Monte Carlo sampling based on importance techniques improved statistical convergence. These contributions were supported by performance optimizations such as parallel computation, type-stable implementations, memory-efficient streaming, and chunk-based processing, ensuring scalability to large problem instances.

Future work. Promising directions include further parallelization of interval algorithms, the development of hybrid methods that combine Monte Carlo sampling with interval enclosures, and advanced visualization techniques for high-dimensional solution sets. Such extensions would enhance uncertainty quantification's rigor and efficiency in increasingly complex systems.

REFERENCES

Asmussen, S., & Glynn, P. W. (2007). *Stochastic Simulation: Algorithms and Analysis*. Springer.

Benet, L., & Sanders, D. (2025). JuliaIntervals.jl package — rigorous numerics with interval arithmetic & applications. `https://github.com/JuliaIntervals/IntervalArithmetic.jl`. Accessed: 2025-08-25.

Bliek, C. (1998). Mathematical programming methods for robustness in engineering design. *Reliable Computing*, *4*(3), 235–253.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge: Cambridge University Press.

Chen, W.-K. (1976). *Applied graph theory: Graphs and electrical networks*, volume 13. North-Holland. Key label has 2012 but actual publication year is 1976.

Chua, L. O., & Lin, P. Y. (1975). *Computer-aided analysis of electronic circuits: Algorithms and computational techniques*. Prentice Hall Professional Technical Reference.

Ding, T., Trinchero, R., Manfredi, P., Stievano, I. S., & Canavero, F. G. (2015). How affine arithmetic helps beat uncertainties in electrical systems. *IEEE Circuits and Systems Magazine*, *15*(4), 70–81.

Femia, N., & Spagnuolo, G. (2000). True worst-case circuit tolerance analysis using genetic algorithms and affine arithmetic. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, *47*(9), 1285–1296.

Gear, C. W. (1971). *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall.

Hansen, E., & Walster, G. W. (2003). *Global Optimization Using Interval Analysis: Revised and Expanded*, volume 264. CRC Press.

Ho, C.-W., Ruehli, A., & Brennan, P. (1975). The modified nodal approach to network analysis. *IEEE Transactions on Circuits and Systems*, *22*(6), 504–509.

IEEE Std 1788 (2015). IEEE Standard for Interval Arithmetic. IEEE Std 1788-2015. Available at: `https://ieeexplore.ieee.org/document/7140721`.

Jaulin, L., Kieffer, M., Didrit, O., & Walter, E. (2001). *Applied interval analysis: with examples in parameter and state estimation, robust control and robotics*. London: Springer.

Khachiyan, L. G. (1996). Rounding of polytopes in the real number model of computation. *Mathematics of Operations Research*, *21*(2), 307–320.

Kolev, L. V. (1993). *Interval Methods for Circuit Analysis*. Advanced Series on Circuits and Systems. World Scientific.

Kundert, K. S. (1986). Sparse matrix techniques. In *Circuit Analysis, Simulation and Design*, volume 3 (160–169). North-Holland, Amsterdam.

Kundert, K. S., & Sangiovanni-Vincentelli, A. (1990). *Simulation of Nonlinear Circuits in the Frequency Domain*. IEEE Press.

Moore, R. E. (1979). *Methods and Applications of Interval Analysis*. Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics (SIAM).

Moore, R. E., Kearfott, R. B., & Cloud, M. J. (2009). *Introduction to Interval Analysis*. Philadelphia: Society for Industrial and Applied Mathematics (SIAM).

Neumaier, A. (1990). *Interval Methods for Systems of Equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge: Cambridge University Press.

Ogata, K. (2010). *Modern Control Engineering* (5 ed.). Englewood Cliffs, NJ: Prentice Hall.

Rockafellar, R. T., & Wets, R. J.-B. (2009). *Variational Analysis*. Springer.

Roychowdhury, J. (1999). *Numerical Time Integration for Transient Circuit Simulation*. New York: Springer.

Rubinstein, R. Y., & Kroese, D. P. (2016). *Simulation and the Monte Carlo Method* (3rd ed.). Wiley.

Todd, M. J. (2016). Minimum-volume ellipsoids: Theory and algorithms. *SIAM Review*, *59*(1), 77–95.

Vlach, J., & Singhal, K. (2003). *Computer Methods for Circuit Analysis and Design* (3 ed.). Springer.

White, J. K., & Sangiovanni-Vincentelli, A. L. (1987). *Relaxation Techniques for the Simulation of VLSI Circuits*, volume 20 of *The Springer International Series in Engineering and Computer Science*. Boston, MA: Kluwer Academic Publishers. Key label has 2012 but actual publication year is 1987.

APPENDICES

A.1 Appendix

A.2 Codes

```
INTERVAL ANALYSIS SYSTEM FOR UNCERTAIN LINEAR SYSTEMS

Complete Julia implementation for thesis including:
- 6 Interval linear system solvers
- Monte Carlo analysis with streaming support
- MVEE computation with Khachiyan algorithm
- Hausdorff distance calculations
- Circuit analysis with MNA
- Comprehensive benchmarking suite


module IntervalAnalysisSystem

using LinearAlgebra, SparseArrays, Random, Statistics
using IntervalArithmetic, Distributions, DelimitedFiles, JSON, Dates
using Base.Threads

export IntervalSystem, AnalysisConfig, AnalysisResults
export comprehensive_analysis, run_thesis_demo


#
# DATA STRUCTURES
#
struct IntervalSystem{T<:Real}
A::Matrix{Interval{T}}
b::Vector{Interval{T}}
n::Int

function IntervalSystem(A::Matrix{Interval{T}}, b::Vector{Interval{T}})
    where T
n = size(A, 1)
@assert size(A, 2) == n "Matrix must be square"
```

```
    @assert length(b) == n "Dimension mismatch"
    new{T}(A, b, n)
    end
    end

    struct AnalysisConfig
    mc_samples::Int
    mc_streaming::Bool
    chunk_size::Int
    max_iter::Int
    tol::Float64
    parallel::Bool
    compute_mvee::Bool
    compute_hausdorff::Bool
    verbose::Bool

    AnalysisConfig(; mc_samples=10000, mc_streaming=false, chunk_size=1000,
    max_iter=1000, tol=1e-10, parallel=true,
    compute_mvee=true, compute_hausdorff=true, verbose=true) =
    new(mc_samples, mc_streaming, chunk_size, max_iter, tol, parallel,
    compute_mvee, compute_hausdorff, verbose)
    end

    struct MVEEResult{T}
    center::Vector{T}
    shape::Matrix{T}
    volume::T
    iters::Int
    converged::Bool
    end

    struct AnalysisResults
    system::IntervalSystem
    config::AnalysisConfig
    solvers::Dict{Symbol, Any}
    mc_data::NamedTuple
    mvee::Union{MVEEResult, Nothing}
```

```
    hausdorff::Dict{Symbol, Float64}
    times::Dict{Symbol, Float64}
    coverage::Dict{Symbol, Float64}
    end
    #
    # INTERVAL LINEAR SOLVERS
    #
    # Standard Gaussian elimination with partial pivoting
    function gauss_elim(A::Matrix{T}, b::Vector{T}) where T<:Real
    n = size(A, 1)
    Ab = [A b]

    for k in 1:n-1
    # Pivot
    p = argmax(abs.(Ab[k:n, k])) + k - 1
    if p != k
    Ab[k, :], Ab[p, :] = Ab[p, :], Ab[k, :]
    end

    abs(Ab[k,k]) < eps(T) && throw(SingularException(k))

    # Eliminate
    for i in k+1:n
    f = Ab[i,k] / Ab[k,k]
    Ab[i, k:end] .-= f * Ab[k, k:end]
    end
    end

    # Back substitution
    x = zeros(T, n)
    for i in n:-1:1
    x[i] = (Ab[i,end] - dot(Ab[i,i+1:n], x[i+1:n])) / Ab[i,i]
    end

    return x
    end

```

```
    # Jacobi iterative method
    function jacobi_solve(A::Matrix{T}, b::Vector{T}; max_iter=1000, tol=T(1e
        -10)) where T
    n = length(b)
    x = zeros(T, n)
    x_new = similar(x)
    D = Diagonal(A)
    R = A - D

    for iter in 1:max_iter
    x_new = (b - R*x) ./ diag(D)
    norm(x_new - x) < tol && return x_new, iter
    x .= x_new
    end

    return x, max_iter
    end

    # Gauss-Seidel iterative method
    function gauss_seidel_solve(A::Matrix{T}, b::Vector{T}; max_iter=1000, tol=
        T(1e-10)) where T
    n = length(b)
    x = zeros(T, n)

    for iter in 1:max_iter
    x_old = copy(x)

    for i in 1:n
    s1 = sum(A[i,j] * x[j] for j in 1:i-1; init=zero(T))
    s2 = sum(A[i,j] * x_old[j] for j in i+1:n; init=zero(T))
    x[i] = (b[i] - s1 - s2) / A[i,i]
    end

    norm(x - x_old) < tol && return x, iter
    end

    return x, max_iter
```

```
    end

    # Krawczyk method for interval systems
    function krawczyk_solve(sys::IntervalSystem{T}; max_iter=100, tol=T(1e-10))
         where T
    n = sys.n
    Ac, bc = mid.(sys.A), mid.(sys.b)
    xc = gauss_elim(Ac, bc)
    C = inv(Ac)

    x = IntervalBox([Interval(-T(100), T(100)) for _ in 1:n])

    for iter in 1:max_iter
    K = IntervalBox(xc) + C * (sys.b - sys.A * IntervalBox(xc))
    Z = K + (I - C * sys.A) * (x - IntervalBox(xc))

    all(Z[i]  x[i] for i in 1:n) && return Z, iter

    x_new = IntervalBox([intersect(Z[i], x[i]) for i in 1:n])
    diameter(x_new) < tol && return x_new, iter

    x = x_new
    end

    return x, max_iter
    end

    # Oettli-Prager method
    function oettli_prager_solve(sys::IntervalSystem{T}; enhanced=false, max_
        iter=100) where T
    n = sys.n
    Ac, bc = mid.(sys.A), mid.(sys.b)Δ
    A, Δb = radius.(sys.A), radius.(sys.b)

    xc = gauss_elim(Ac, bc)
    x_abs = abs.(xc)

```

```
    for iter in 1:max_iter
    x_new = abs.(inv(Ac) * Δ(A * x_abs .+ Δb .+ abs.(bc)))
    norm(x_new - x_abs) < T(1e-10) && break
    x_abs = x_new
    end

    enhanced && (x_abs *= T(1.1)) # Safety factor for enhanced version

    return IntervalBox([Interval(xc[i] - x_abs[i], xc[i] + x_abs[i]) for i in
        1:n]), 1
    end

    # Hansen-Bliek-Rohn method
    function hansen_bliek_rohn_solve(sys::IntervalSystem{T}) where T
    n = sys.n
    Ac, bc = mid.(sys.A), mid.(sys.b)
    C = inv(Ac)
    CA, Cb = C * sys.A, C * sys.b

    x = IntervalBox([Interval(-T(100), T(100)) for _ in 1:n])

    for iter in 1:100
    x_new = IntervalBox(zeros(Interval{T}, n))

    for i in 1:n
    sum_lo, sum_hi = Cb[i].lo, Cb[i].hi

    for j in 1:n
    if i != j
    if CA[i,j].lo >= 0
    sum_lo -= CA[i,j].hi * x[j].hi
    sum_hi -= CA[i,j].lo * x[j].lo
    else
    sum_lo -= CA[i,j].lo * x[j].lo
    sum_hi -= CA[i,j].hi * x[j].hi
    end
    end
```

```
    end

    CA[i,i].lo > 0 && (x_new = setindex(x_new,
    Interval(sum_lo/CA[i,i].hi, sum_hi/CA[i,i].lo), i))
    end

    all(x_new[i]  x[i] for i in 1:n) && return x_new, iter
    x = x_new
    end

    return x, 100
    end

    #
    # MONTE CARLO ANALYSIS
    #

    function monte_carlo_sample(sys::IntervalSystem{T}, config::AnalysisConfig)
         where T
    n = sys.n
    rng = MersenneTwister(42)
    samples = Matrix{T}(undef, n, config.mc_samples)
    valid = 0

    # Create distributions
    A_dist = [Uniform(sys.A[i,j].lo, sys.A[i,j].hi) for i in 1:n, j in 1:n]
    b_dist = [Uniform(sys.b[i].lo, sys.b[i].hi) for i in 1:n]

    if config.parallel
    lock = ReentrantLock()

    Threads.@threads for s in 1:config.mc_samples
    A_s = [rand(rng, A_dist[i,j]) for i in 1:n, j in 1:n]
    b_s = [rand(rng, b_dist[i]) for i in 1:n]

    try
    x_s = gauss_elim(A_s, b_s)
```

```
    lock do
    samples[:, s] = x_s
    valid += 1
    end
    catch
    lock do
    samples[:, s] .= NaN
    end
    end
    end
    else
    for s in 1:config.mc_samples
    A_s = [rand(rng, A_dist[i,j]) for i in 1:n, j in 1:n]
    b_s = [rand(rng, b_dist[i]) for i in 1:n]

    try
    samples[:, s] = gauss_elim(A_s, b_s)
    valid += 1
    catch
    samples[:, s] .= NaN
    end
    end
    end

    # Filter valid samples
    valid_samples = samples[:, .!isnan.(samples[1, :])]

    return valid_samples, valid
    end

    function monte_carlo_streaming(sys::IntervalSystem{T}, config::
        AnalysisConfig) where T
    n = sys.n
    total = config.mc_samples
    chunk = config.chunk_size

    sum_x = zeros(T, n)
```

```
    sum_xx = zeros(T, n, n)
    valid_count = 0

    temp_file = "mc_stream_$(Dates.now()).dat"

    open(temp_file, "w") do io
    for c in 1:ceil(Int, total/chunk)
    config.verbose && print("\rChunk $c/$(ceil(Int, total/chunk))")

    cfg = AnalysisConfig(mc_samples=min(chunk, total-(c-1)*chunk),
    parallel=config.parallel, verbose=false)
    chunk_samples, chunk_valid = monte_carlo_sample(sys, cfg)

    for i in 1:size(chunk_samples, 2)
    x = chunk_samples[:, i]
    sum_x .+= x
    sum_xx .+= x * x'
    writedlm(io, x', ',')
    end

    valid_count += chunk_valid
    end
    end

    config.verbose && println("\nStreaming complete: $valid_count/$total valid
        ")

    mean_x = sum_x / valid_count
    cov_x = (sum_xx / valid_count) - (mean_x * mean_x')

    return (file=temp_file, mean=mean_x, cov=cov_x, valid=valid_count, total=
        total)
    end

    #
    # GEOMETRIC ANALYSIS
    #
```

```
318
319     # MVEE using Khachiyan's algorithm
320     function compute_mvee(points::Matrix{T}; tol=T(1e-6), max_iter=1000) where
            T
321     n, N = size(points)
322     N <= n && error("Need more points than dimensions")
323
324     u = ones(T, N) / N
325     err = T(Inf)
326     iter = 0
327
328     while err > tol && iter < max_iter
329     iter += 1
330
331     center = points * u
332
333     P = zeros(T, n, n)
334     for i in 1:N
335     d = points[:, i] - center
336     P += u[i] * (d * d')
337     end
338     P += T(1e-12) * I
339
340     M = zeros(T, N)
341     P_inv = inv(P)
342     for i in 1:N
343     d = points[:, i] - center
344     M[i] = dot(d, P_inv * d)
345     end
346
347     max_val, j = findmax(M)
348     err = max_val - n
349
350     step = (max_val - n) / ((n + 1) * (max_val - 1))
351     u = (1 - step) * u
352     u[j] += step
353     u /= sum(u)
```

```
    end

    center = points * u
    P = zeros(T, n, n)
    for i in 1:N
    d = points[:, i] - center
    P += u[i] * (d * d')
    end

    shape = inv(P) / n

    volume = T(0)
    if n == 3
    evals = eigvals(shape)
    all(evals .> 0) && (volume =  (4/3) / sqrt(prod(evals)))
    elseif n == 2
    evals = eigvals(shape)
    all(evals .> 0) && (volume =  / sqrt(prod(evals)))
    end

    return MVEEResult(center, shape, volume, iter, err <= tol)
    end

    # Parallel MVEE for large datasets
    function compute_mvee_parallel(points::Matrix{T}; tol=T(1e-6), max_iter
        =1000) where T
    n, N = size(points)
    u = ones(T, N) / N
    nt = Threads.nthreads()
    partial_P = [zeros(T, n, n) for _ in 1:nt]

    for iter in 1:max_iter
    center = points * u

    Threads.@threads for t in 1:nt
    fill!(partial_P[t], 0)
    range = (t-1)*div(N,nt)+1 : min(t*div(N,nt), N)
```

```
    for i in range
    d = points[:, i] - center
    partial_P[t] += u[i] * (d * d')
    end
    end

    P = sum(partial_P) + T(1e-12) * I
    P_inv = inv(P)

    M = zeros(T, N)
    Threads.@threads for i in 1:N
    d = points[:, i] - center
    M[i] = dot(d, P_inv * d)
    end

    max_val, j = findmax(M)
    max_val - n <= tol && break

    step = (max_val - n) / ((n + 1) * (max_val - 1))
    u = (1 - step) * u
    u[j] += step
    u /= sum(u)
    end

    # Final computation (same as serial)
    center = points * u
    P = zeros(T, n, n)
    for i in 1:N
    d = points[:, i] - center
    P += u[i] * (d * d')
    end

    shape = inv(P) / n
    volume = n == 3 ? (4/3) / sqrt(prod(eigvals(shape))) :
    n == 2 ? / sqrt(prod(eigvals(shape))) : T(0)

    return MVEEResult(center, shape, volume, max_iter, true)
```

```
    end

    # Hausdorff distance between point sets
    function hausdorff_distance(S1::Matrix{T}, S2::Matrix{T}) where T
    d12 = maximum(minimum(norm(S1[:, i] - S2[:, j]) for j in 1:size(S2, 2))
    for i in 1:size(S1, 2))
    d21 = maximum(minimum(norm(S2[:, j] - S1[:, i]) for i in 1:size(S1, 2))
    for j in 1:size(S2, 2))
    return max(d12, d21)
    end

    # Extract polytope vertices from interval solution
    function extract_vertices(x::IntervalBox{N, T}) where {N, T}
    n = N
    vertices = Matrix{T}(undef, n, 2^n)

    for i in 0:2^n-1
    for j in 1:n
    vertices[j, i+1] = (i >> (j-1)) & 1 == 0 ? x[j].lo : x[j].hi
    end
    end

    return vertices
    end

    #
    # CIRCUIT ANALYSIS (MNA)
    #

    struct Component
    type::Symbol # :R, :L, :C, :V, :I
    value::Float64
    tol::Float64 # tolerance percentage
    nodes::Tuple{Int, Int}
    end

    struct Circuit
```

```
    components::Vector{Component}
    n_nodes::Int
    end

    function build_mna(circuit::Circuit, freq::Float64)
    n = circuit.n_nodes - 1 # Exclude ground
    n_vs = count(c -> c.type == :V, circuit.components)
    size = n + n_vs

    G = zeros(ComplexF64, size, size)
    B = zeros(ComplexF64, size)

     = 2 * freq
    vs_idx = n

    for c in circuit.components
    i, j = c.nodes
    i_idx = i > 0 ? i : nothing
    j_idx = j > 0 ? j : nothing

    if c.type == :R
    g = 1.0 / c.value
    add_stamp!(G, i_idx, j_idx, g)

    elseif c.type == :C
    y = im *  * c.value
    add_stamp!(G, i_idx, j_idx, y)

    elseif c.type == :L
    y = 1.0 / (im *  * c.value)
    add_stamp!(G, i_idx, j_idx, y)

    elseif c.type == :V
    vs_idx += 1
    !isnothing(i_idx) && (G[i_idx, vs_idx] = G[vs_idx, i_idx] = 1)
    !isnothing(j_idx) && (G[j_idx, vs_idx] = G[vs_idx, j_idx] = -1)
    B[vs_idx] = c.value
```

```
501
502     elseif c.type == :I
503     !isnothing(i_idx) && (B[i_idx] -= c.value)
504     !isnothing(j_idx) && (B[j_idx] += c.value)
505     end
506     end
507
508     return G, B
509     end
510
511     function add_stamp!(G, i, j, value)
512     if !isnothing(i) && !isnothing(j)
513     G[i, i] += value
514     G[j, j] += value
515     G[i, j] -= value
516     G[j, i] -= value
517     elseif !isnothing(i)
518     G[i, i] += value
519     elseif !isnothing(j)
520     G[j, j] += value
521     end
522     end
523
524     function build_interval_mna(circuit::Circuit, freq::Float64)
525     n = circuit.n_nodes - 1
526     n_vs = count(c -> c.type == :V, circuit.components)
527     size = n + n_vs
528
529     G = [Interval(0.0, 0.0) for _ in 1:size, _ in 1:size]
530     B = [Interval(0.0, 0.0) for _ in 1:size]
531
532      = 2 * freq
533     vs_idx = n
534
535     for c in circuit.components
536     i, j = c.nodes
537     i_idx = i > 0 ? i : nothing
```

```
    j_idx = j > 0 ? j : nothing

    val_min = c.value * (1 - c.tol/100)
    val_max = c.value * (1 + c.tol/100)

    if c.type == :R
    g = Interval(1/val_max, 1/val_min)
    add_interval_stamp!(G, i_idx, j_idx, g)

    elseif c.type == :V
    vs_idx += 1
    !isnothing(i_idx) && (G[i_idx, vs_idx] = G[vs_idx, i_idx] = Interval(1.0,
        1.0))
    !isnothing(j_idx) && (G[j_idx, vs_idx] = G[vs_idx, j_idx] = Interval(-1.0,
        -1.0))
    B[vs_idx] = Interval(val_min, val_max)
    end
    end

    return G, B
    end

    function add_interval_stamp!(G, i, j, value)
    if !isnothing(i) && !isnothing(j)
    G[i, i] += value
    G[j, j] += value
    G[i, j] -= value
    G[j, i] -= value
    elseif !isnothing(i)
    G[i, i] += value
    elseif !isnothing(j)
    G[j, j] += value
    end
    end

    #
    # COMPREHENSIVE ANALYSIS
```

```
573     #
574
575     function comprehensive_analysis(sys::IntervalSystem{T}, config::
            AnalysisConfig) where T
576     config.verbose && println("\n" * "="^60 * "\nCOMPREHENSIVE INTERVAL
            ANALYSIS\n" * "="^60)
577
578     times = Dict{Symbol, Float64}()
579     solvers = Dict{Symbol, Any}()
580
581     # Run all solvers
582     config.verbose && println("\ n Running Interval Solvers...")
583
584     for (name, solver, kwargs) in [
585     (:krawczyk, krawczyk_solve, (max_iter=config.max_iter, tol=config.tol)),
586     (:oettli_prager, oettli_prager_solve, (enhanced=false,)),
587     (:oettli_prager_enh, oettli_prager_solve, (enhanced=true,)),
588     (:hansen_bliek_rohn, hansen_bliek_rohn_solve, ())
589     ]
590     t = @elapsed begin
591     try
592     solvers[name] = solver(sys; kwargs...)
593     catch e
594     solvers[name] = nothing
595     config.verbose && println(" $name: failed - $e")
596     end
597     end
598     times[name] = t
599     config.verbose && solvers[name] !== nothing &&
600     println(" $name: $(round(t*1000, digits=2))ms")
601     end
602
603     # Monte Carlo
604     config.verbose && println("\ n Monte Carlo Analysis...")
605
606     mc_data = nothing
607     t = @elapsed begin
```

```
    if config.mc_streaming && config.mc_samples > 1_000_000
    mc_result = monte_carlo_streaming(sys, config)
    mc_data = (mean=mc_result.mean, cov=mc_result.cov,
    file=mc_result.file, valid=mc_result.valid)
    else
    samples, valid = monte_carlo_sample(sys, config)
    if valid > 0
    mc_data = (
    samples=samples,
    mean=mean(samples, dims=2)[:,1],
    cov=cov(samples'),
    bounds=IntervalBox([Interval(minimum(samples[i,:]),
    maximum(samples[i,:])) for i in 1:sys.n]),
    valid=valid
    )
    end
    end
    end
    times[:monte_carlo] = t
    config.verbose && mc_data !== nothing &&
    println(" MC: $(mc_data.valid) valid samples in $(round(t, digits=2))s")

    # MVEE
    mvee = nothing
    if config.compute_mvee && mc_data !== nothing && haskey(mc_data, :samples)
    config.verbose && println("\ n Computing MVEE...")

    t = @elapsed begin
    points = mc_data.samples
    mvee = config.parallel && size(points, 2) > 10000 ?
    compute_mvee_parallel(points) : compute_mvee(points)
    end
    times[:mvee] = t
    config.verbose && println(" Converged: $(mvee.converged), Volume: $(round(
        mvee.volume, digits=4))")
    end

```

```
    # Hausdorff distances
    hausdorff = Dict{Symbol, Float64}()
    if config.compute_hausdorff && mc_data !== nothing && haskey(mc_data, :
        samples)
    config.verbose && println("\ n Computing Hausdorff Distances...")

    mc_subset = size(mc_data.samples, 2) > 5000 ?
    mc_data.samples[:, 1:5000] : mc_data.samples

    for (name, result) in solvers
    if result !== nothing && isa(result, Tuple)
    t = @elapsed begin
    vertices = extract_vertices(result[1])
    hausdorff[name] = hausdorff_distance(mc_subset, vertices)
    end
    times[Symbol("hausdorff_$name")] = t
    end
    end

    config.verbose && for (k, v) in hausdorff
    println(" $k: $(round(v, digits=6))")
    end
    end

    # Coverage metrics
    coverage = Dict{Symbol, Float64}()
    if mc_data !== nothing && haskey(mc_data, :samples)
    for (name, result) in solvers
    if result !== nothing && isa(result, Tuple)
    box = result[1]
    covered = sum(all(box[j].lo <= mc_data.samples[j,i] <= box[j].hi
    for j in 1:sys.n)
    for i in 1:size(mc_data.samples, 2))
    coverage[name] = covered / size(mc_data.samples, 2)
    end
    end
    end
```

```
680
681     # Summary
682     if config.verbose
683     println("\n" * "="^60 * "\nRESULTS SUMMARY\n" * "="^60)
684
685     println("\ n Coverage Metrics:")
686     for (s, c) in sort(collect(coverage), by=x->x[2], rev=true)
687     println(" $s: $(round(c*100, digits=1))%")
688     end
689
690     println("\ n Computation Times:")
691     total_time = sum(values(times))
692     for (task, time) in sort(collect(times), by=x->x[2], rev=true)
693     pct = round(100*time/total_time, digits=1)
694     println(" $task: $(round(time, digits=3))s ($pct%)")
695     end
696
697     println("\nTotal: $(round(total_time, digits=2))s")
698     end
699
700     return AnalysisResults(sys, config, solvers, mc_data, mvee, hausdorff,
        times, coverage)
701     end
702
703     #
704     # VALIDATION & BENCHMARKING
705     #
706
707     function validate_solvers(n=3, tests=100)
708     println("\nValidating $tests random $(n)×$(n) systems...")
709
710     success = Dict{Symbol, Int}()
711
712     for t in 1:tests
713     A = [Interval(rand()-0.5, rand()+0.5) for i in 1:n, j in 1:n]
714     b = [Interval(rand()-0.5, rand()+0.5) for i in 1:n]
715     sys = IntervalSystem(A, b)
```

```
716
717     for (name, solver) in [
718     (:krawczyk, krawczyk_solve),
719     (:oettli_prager, s -> oettli_prager_solve(s, enhanced=false)),
720     (:hansen_bliek_rohn, hansen_bliek_rohn_solve)
721     ]
722     try
723     result, _ = solver(sys)
724     Ax = sys.A * result
725     valid = all(sys.b[i]  Ax[i] for i in 1:n)
726     success[name] = get(success, name, 0) + valid
727     catch
728     success[name] = get(success, name, 0)
729     end
730     end
731     end
732
733     println("\nValidation Results:")
734     for (s, v) in success
735     println(" $s: $(round(100v/tests, digits=1))% success")
736     end
737
738     return success
739     end
740
741     function benchmark_performance(sizes=[3, 5, 10, 20])
742     println("\nBenchmarking performance...")
743     results = Dict{Symbol, Vector{Float64}}()
744
745     for n in sizes
746     println("\n Size $n×$n:")
747
748     A = [Interval(rand()-0.5, rand()+0.5) for i in 1:n, j in 1:n]
749     b = [Interval(rand()-0.5, rand()+0.5) for i in 1:n]
750     sys = IntervalSystem(A, b)
751
752     for (name, solver) in [
```

```
    (:krawczyk, krawczyk_solve),
    (:oettli_prager, s -> oettli_prager_solve(s, enhanced=false)),
    (:hansen_bliek_rohn, hansen_bliek_rohn_solve)
    ]
    t = @elapsed try solver(sys) catch end

    if !haskey(results, name)
    results[name] = Float64[]
    end
    push!(results[name], t)

    println(" $name: $(round(t*1000, digits=2))ms")
    end
    end

    return results
    end

    #
    # EXAMPLE SYSTEMS & DEMOS
    #

    function create_example_system(type=:random; n=3)
    if type == :random
    A = [Interval(randn()-0.1, randn()+0.1) for i in 1:n, j in 1:n]
    b = [Interval(randn()-0.1, randn()+0.1) for i in 1:n]

    elseif type == :circuit
    A = [
    Interval(9.5, 10.5) Interval(-4.75, -4.25) Interval(-4.75, -4.25);
    Interval(-4.75, -4.25) Interval(9.5, 10.5) Interval(-4.75, -4.25);
    Interval(-4.75, -4.25) Interval(-4.75, -4.25) Interval(9.5, 10.5)
    ]
    b = [Interval(0.95, 1.05); Interval(-0.05, 0.05); Interval(-0.05, 0.05)]

    elseif type == :ill_cond
    A = [
```

```
    Interval(0.99, 1.01) Interval(0.99, 1.01) Interval(0.0, 0.0);
    Interval(0.99, 1.01) Interval(1.0, 1.02) Interval(0.0, 0.0);
    Interval(0.0, 0.0) Interval(0.0, 0.0) Interval(0.99, 1.01)
    ]
    b = [Interval(1.98, 2.02); Interval(1.99, 2.03); Interval(0.99, 1.01)]
    end

    return IntervalSystem(A, b)
    end

    function create_rc_circuit()
    components = [
    Component(:V, 1.0, 0.0, (1, 0)), # 1V source
    Component(:R, 1000.0, 5.0, (1, 2)), # 1Ωk ± 5%
    Component(:C, 1e-6, 10.0, (2, 0)) # 1μF ± 10%
    ]
    return Circuit(components, 2)
    end

    function run_thesis_demo()
    println("\n" * ""^70)
    println(" INTERVAL ANALYSIS THESIS DEMONSTRATION")
    println (""^70)

    # Test different systems
    for (type, name) in [(:random, "Random"), (:circuit, "Circuit"), (:ill_cond
        , "Ill-Conditioned")]
    println("\n" * ""^70 * "\n Testing: $name System\n" * ""^70)

    sys = create_example_system(type)
    config = AnalysisConfig(
    mc_samples=50_000,
    parallel=true,
    compute_mvee=true,
    compute_hausdorff=true,
    verbose=true
    )
```

```
826
827     results = comprehensive_analysis(sys, config)
828
829     # Generate LaTeX table
830     println("\ n LaTeX Output:")
831     println("\\begin{tabular}{lrrr}")
832     println("Solver & Time (ms) & Coverage (\\%) \\\\")
833     println("\\hline")
834     for (s, c) in results.coverage
835     t = results.times[s] * 1000
836     println("$(s) & $(round(t, digits=1)) & $(round(c*100, digits=1)) \\\\")
837     end
838     println("\\end{tabular}")
839     end
840
841     # Validation
842     println("\n" * ""^70 * "\n Validation Suite\n" * ""^70)
843     validate_solvers(3, 50)
844
845     # Benchmarks
846     println("\n" * ""^70 * "\n Performance Benchmarks\n" * ""^70)
847     benchmark_performance([3, 5, 10])
848
849     println("\n" * ""^70)
850     println(" DEMONSTRATION COMPLETE")
851     println (""^70)
852     end
853
854     end # module
855
856     # Usage:
857     # using .IntervalAnalysisSystem
858     # IntervalAnalysisSystem.run_thesis_demo()
```