JOURNAL OF COMPUTATIONAL INNOVATION AND ANALYTICS
https://e-journal.uum.edu.my/index.php/jcia

How to cite this article:
Bala, M. M., & Bahri, S. (2026). A compiler-based framework for automated numerical method selection via deterministic finite automata. *Journal of Computational Innovation and Analytics*, *5*(2), 1-24. https://doi.org/10.32890/jcia2026.5.2.1

A compiler-based framework for automated numerical method selection via deterministic finite automata

[1]Bala Muhammad Muhammad & [2]Susila Bahri
[1,2]Department of Mathematics and Data Science,
Universitas Andalas, Padang, 25175, Indonesia

[1]*Corresponding author:* susilabahri@sci.unand.ac.id

Received: 11/02/2026 Revised: 04/05/2026 Accepted: 07/05/2026 Published: 31/07/2026

ABSTRACT

This study introduces a compiler-integrated framework for the automated selection of numerical methods through deterministic finite automata (DFA)-based structural classification of mathematical expressions. Traditional numerical method selection approaches typically rely on expert knowledge or data-driven models, which often lack interpretability and fail to integrate seamlessly with compiler systems. In contrast, the proposed framework employs a domain-specific language, lexical analysis, canonicalization, and DFA-based pattern recognition to classify mathematical expressions into categories such as linear, polynomial, nonlinear, recurrence relations, and ordinary differential equations (ODEs). Subsequently, a rule-based selection engine maps each expression to the most appropriate numerical solver, including Newton's method, Bisection, Secant, Euler, RK2, and RK4 methods. Evaluation of the framework on a dataset of 150 expressions yielded 96.67% classification accuracy and 97% correctness in method selection. The solvers demonstrate convergence behavior consistent with theoretical expectations, while the computational overhead remains under 2 milliseconds. This approach offers an interpretable and efficient alternative to machine-learning-based methods and demonstrates the feasibility of integrating numerical reasoning directly into compiler systems and mathematical software.

Keywords: Automated solvers, compiler systems, DFA classification, mathematical expressions, numerical methods.

INTRODUCTION

Selecting appropriate numerical methods for solving mathematical expressions remains a significant challenge in scientific computing. While classical methods such as Newton's method, Bisection, and Runge-Kutta schemes are well-established for solving various problem types, selecting the most suitable method for a given expression depends on recognizing its structural properties, including linearity, differentiability, and recurrence behavior (Atkinson & Wiley, 1978; Press et al., 1988). Despite their widespread use, determining the most suitable numerical method often relies on manual expertise, making the process error-prone and time-consuming. Furthermore, the convergence rates, stability properties, and computational costs of different methods complicate the selection process (Dennis & Schnabel, 1996; Saad, 2003).

Compiler Systems and Domain-Specific Languages (DSLs)

In recent years, compiler systems have been explored as tools for automating the selection of numerical methods. Domain-specific languages (DSLs) have been particularly instrumental in this regard, allowing the automatic generation of code tailored to specific domains. DSL frameworks have been shown to encode domain knowledge and facilitate process automation, improving both efficiency and scalability (Khakpour et al., 2023; Shen et al., 2020).

For instance, DSL-based compiler systems can automatically generate computational workflows for scientific applications, incorporating complex domain logic directly into the compiler. This eliminates the need for human intervention in selecting appropriate numerical methods for specific problem classes, such as linear or nonlinear systems. However, existing DSL frameworks often fail to classify mathematical expressions by structure systematically and to link this classification directly to the selection of numerical methods. This gap in automated decision-making in compiler design is particularly significant because it limits the interpretability and integration of such systems into standard numerical computation workflows.

Machine Learning-Based Solver Selection

Recent advancements have also seen the application of machine learning (ML) techniques to automate the selection of numerical methods. ML frameworks have been increasingly employed to predict the most suitable solver based on the characteristics of the problem at hand. Zabegaev et al. (2024) proposed a machine learning framework for selecting linear solvers in multiphysics simulations, while Lee et al. (2023) introduced a meta-solver architecture that combines classical numerical methods with neural operators for solving nonlinear partial differential equations (PDEs). Other works, such as Solaiman et al. (2023), have explored hybrid optimization strategies that combine Newton's method with swarm intelligence techniques to improve the selection process. Extensions of DSL-based approaches demonstrate that compiler frameworks can encapsulate complex domain logic and automatically generate computational workflows in industrial and scientific applications (Sal et al., 2024). Furthermore, machine learning (ML), deep learning (DL), and reinforcement learning (RL) techniques have been increasingly applied to automate compiler optimizations, including parameter tuning, optimization selection, and performance prediction (Liu et al., 2022; Mithul et al., 2024; Wang et al., 2022). These approaches highlight a broader trend toward intelligent and adaptive compilation systems.

While these data-driven techniques have shown promise, they suffer from several limitations. Machine learning models often lack interpretability and require large amounts of training data, which may not always be available for specialized numerical methods. Additionally, many ML-based solvers operate

in black-box environments, where understanding the underlying decision process is difficult. This limits their applicability in scenarios where explainable and interpretable results are necessary, such as in educational tools or in scientific computing environments where the reasoning behind solver selection must be transparent.

AUTOMATED NUMERICAL SOLVER SELECTION SYSTEMS

Several automated numerical solver selection systems have been developed, such as the Lighthouse framework, which provides performance-based recommendations for linear algebra solvers (Jessup et al., 2016; Motter et al., 2015; Norris et al., 2014). While such systems are useful, they generally focus on performance metrics rather than the structural classification of expressions, and they often fail to integrate well with compiler systems. The Lighthouse framework, for instance, limits itself to specific solvers for linear algebra problems and does not address the broader problem of classification and solver selection across various mathematical expression types, such as ODEs, polynomials, or nonlinear equations.

Despite these advances, most existing approaches rely heavily on data-driven models, which often lack interpretability and require substantial training data. In addition, many solver-selection systems operate as external tools or specialized environments rather than being integrated directly into compiler pipelines. Mathematical software environments, such as automated ODE-solving systems and numerical computation platforms, provide high-level abstractions for solving equations (Driscoll et al., 2008; Petcu & Drăgan, 2000), but they do not explicitly address the structural classification of expressions to support the selection of systematic numerical methods. Similarly, symbolic computation systems focus on algebraic manipulation rather than on selecting automated numerical solvers based on structural properties (Winkler, 2024).

Another limitation of existing solver selection systems is their lack of integration into compilers. Most systems operate as external tools or specialized environments, rather than as part of a compiler pipeline that can automatically select solvers based on structural properties of expressions. This gap in integration is particularly significant, as it limits the automation and accessibility of numerical computing systems.

Application of Deterministic Finite Automata (DFAs)

In contrast to data-driven models, deterministic finite automata (DFAs) offer an interpretative framework for recognizing structured patterns in symbolic inputs. DFAs are widely used in compiler design for lexical analysis, token recognition, and regular language processing (Aho et al., 2007). Modern compiler systems further build upon these foundations to enable structured program analysis and transformation (Cooper & Torczon, 2012; Grune et al., 2012). Due to their deterministic behaviour and low computational complexity, DFAs are particularly suitable for recognizing structured patterns in symbolic inputs. However, despite their extensive use in language processing, DFAs have rarely been applied to the structural classification of mathematical expressions for selecting numerical methods.

However, despite their success in language processing, DFAs have rarely been applied to the structural classification of mathematical expressions for selecting numerical methods. The current literature often overlooks the potential of DFAs to provide efficient, deterministic, and low-complexity solutions for identifying mathematical structures and selecting corresponding solvers.

Table 1 presents a comparison of various existing approaches for numerical method selection, focusing on approach type, interpretability, compiler integration, and limitations. As shown in the table, most existing systems either rely on data-driven models or are limited to specific environments, while the proposed framework uniquely integrates DFA-based classification within a compiler system, ensuring interpretability and efficient integration.

Table 1

Comparison of Existing Approaches for Numerical Method Selection

Study	Approach Type	Technique Used	Scope	Interpret-ability	Compiler Integration	Limitations
(Motter et al., 2015)	Solver Selection System	Performance-based rules	Linear algebra	Medium	No	Limited to specific libraries
(Jessup et al., 2016)	Solver Recommen-dation	Empirical performance models	Linear systems	Medium	No	Domain-specific
(Zabegaev et al., 2024)	ML-based	Machine learning	Multiphysics solvers	Low	No	Requires training data
(Lee et al., 2025)	Meta-solver	Neural operators	PDEs	Low	No	Complex, black box
(Liu et al., 2022)	Compiler ML	Ensemble learning	Optimiza-tion parameters	Low	Yes	Not numerical methods
(Wang et al., 2022b)	RL-based compiler	Reinforcement learning	Code optimiza-tion	Low	Yes	No expression-level reasoning
(Petcu & Drăgan, 2000)	Mathematic-al system	ODE environment	ODE solving	Medium	No	Limited automation
(Driscoll et al., 2008)	Numerical software	Chebop system	Differential equations	Medium	No	Not classification-based
(Winkler, 2024)	Symbolic systems	Algebraic manipulation	General math	High	No	Not numerical selection
Proposed	DFA + Compiler	Rule-based + automata	General expressions	High	Yes	

RESEARCH GAP

This review of the literature highlights a clear gap in existing methods. While significant progress has been made with machine learning-based methods and DSL frameworks, none of these approaches adequately address the challenge of integrating structural classification with automated numerical reasoning within a compiler environment. Existing systems often lack interpretability, are not fully integrated into compilers, and rely on large amounts of training data. Additionally, many systems are either limited to specific problem types or fail to provide a systematic, rule-based framework for solver selection.

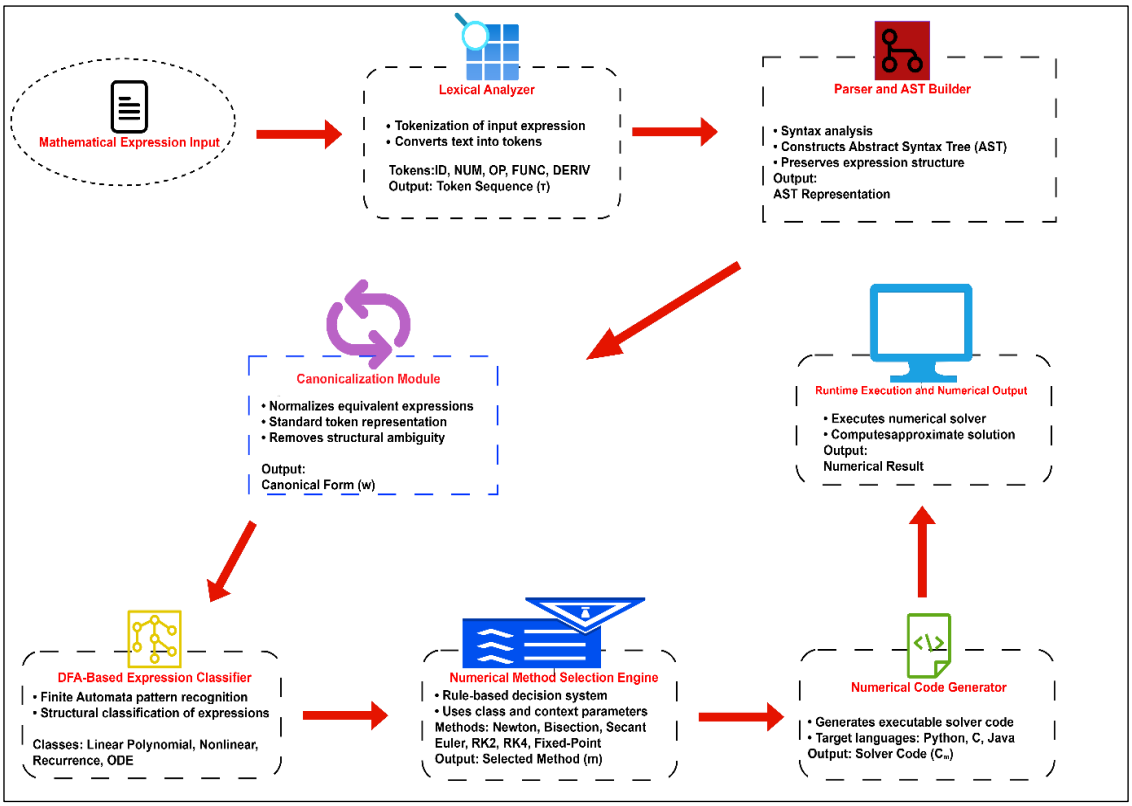
To address this limitation, this study proposes a compiler-based framework that integrates deterministic finite automata (DFAs) for structural classification of mathematical expressions with rule-based numerical method selection. The framework processes mathematical expressions through lexical analysis, parsing, and canonicalization, applying DFAs to identify structural patterns corresponding to problem classes such as linear equations, polynomial equations, nonlinear (transcendental) equations, recurrence relations, and ordinary differential equations (ODEs). Based on this classification, a selection engine maps each expression to an appropriate numerical solver.

METHODOLOGY

This study adopts a design–implementation–evaluation methodology to develop and assess a compiler-based framework for automatic selection of numerical methods. The framework classifies mathematical expressions by their structural properties using deterministic finite automata (DFAs), then maps each class to an appropriate numerical solver via a rule-based selection engine. Figure 1 provides an overview of the proposed system architecture. For evaluation, a balanced dataset of 150 mathematical expressions was constructed, comprising 30 expressions each from five classes: linear equations, polynomial equations, nonlinear equations, recurrence relations, and ordinary differential equations (ODEs). This dataset was used to assess both expression classification and numerical method selection performance.

Figure 1

System Architecture Diagram



Model Formulation

The proposed system consists of six main stages:

1. Lexical analysis transforms input expressions into tokens.
2. Parsing constructs an abstract syntax tree (AST).
3. Canonicalization normalizes equivalent expressions into a standard token form.
4. DFA-based classification identifies the structural class of the expression.
5. Method selection assigns an appropriate numerical solver based on class and context.
6. Code generation and execution produce executable numerical code for the selected method.

The overall computational pipeline is defined in Equation 1.

$$P \to \tau \to T \to w \to c \to m \to C_m \tag{1}$$

where: P= input mathematical program or expression, τ= token sequence, T= abstract syntax tree (AST), w= canonical token sequence, c= expression class, m= selected numerical method, C_m= generated code for method m.

Problem Representation and Method Mapping

Let the set of problem classes be defined in Equation 2.

$$C = \{c_1, c_2, c_3, c_4, c_5\} \tag{2}$$

where: c_1: Linear equations, c_2: Polynomial equations, c_3: nonlinear equations, c_4: Recurrence relations, c_5: Ordinary differential equations. Let the numerical method library be represented by Equation 3.

$$N = \{\text{DirectSolve, Bisection, Newton, Secant, FixedPoint, Euler, RK2, RK4}\} \tag{3}$$

The system maintains a mapping from each problem class to the set of admissible numerical methods, Equation 4.

$$M: C \to 2^N \tag{4}$$

where 2^N denotes the power set of N. For example, $M(c_1) = \{\text{DirectSolve}\}$, $M(c_2) = \{\text{Bisection, Newton, Secant}\}$, $M(c_3) = \{\text{Bisection, Newton, Secant}\}, M(c_4) = \{\text{FixedPoint}\}$, $M(c_5) = \{\text{Euler, RK2, RK4}\}$.

Lexical Analysis, Parsing, and Canonicalization

The process begins with lexical analysis, where input mathematical expressions are tokenized into a sequence of tokens using a DFA-based lexer. The input expression P is first tokenized by a DFA-based lexer, as shown in Equation 5.

$$\text{Lexer}(P) = \tau = (t_1, t_2, \ldots, t_n) \quad (5)$$

where each token $t_i$belongs to the lexical token set defined by the grammar. In addition, the token sequence is parsed into an abstract syntax tree as shown in Equation 6.

$$\text{Parse}(\tau) = T \quad (6)$$

The AST preserves the syntactic structure of the mathematical expression and supports downstream normalization and classification. Furthermore, to reduce representational variability, each AST is transformed into a canonical token sequence presented in Equation 7.

$$\text{Canon}(T) = w \quad (7)$$

Canonicalization standardizes equivalent expressions into the same structural form. For example, algebraically equivalent expressions such as $x + 1 = 0$ and $1 + x = 0$ are normalized into a consistent representation before DFA simulation.

DFA-Based Expression Classification

Each problem class is recognized by a dedicated deterministic finite automaton. A DFA is formally defined in Equation 8.

$$A = (Q, \Sigma, \delta, q_0, F) \quad (8)$$

where: Q= finite set of states, Σ= input alphabet of canonical tokens, $\delta: Q \times \Sigma \to Q$= transition function, $q_0 \in Q$= initial state, $F \subseteq Q$= set of accepting states. The canonical token alphabet is given in Equation 9.

$$\Sigma = \{\text{ID, NUM, PLUS, MINUS, MULT, DIV, POW, FUNC, DERIV, INDEX, EQ}\} \quad (9)$$

where: ID = identifier/variable, NUM = numeric constant, PLUS, MINUS, MULT, DIV, POW = arithmetic operators, FUNC = transcendental function token such as sin, cos, exp, or log, DERIV = derivative token such as dy/dx, INDEX = indexed variable token such as x_n, x_{n+1}, EQ = equality sign.

Given a canonical token sequence $w \in \Sigma^*$, classification is defined in Equation 10.

$$\text{Classify}(w) = c_i \text{if } A_i \text{ accepts } w \quad (10)$$

where $A_i$is the DFA associated with class c_i. If more than one automaton accepts the same sequence, a priority-based tie-breaking rule is applied. The class priority used in this study is given by Equation 11.

$$\text{ODE} \succ \text{Recurrence} \succ \text{nonlinear} \succ \text{Polynomial} \succ \text{Linear} \quad (11)$$

This ordering ensures that structurally richer forms, such as differential and recursive expressions, are not misclassified as simpler algebraic forms. The computational complexity of the DFA-based classification process is $O(n)$, where $n$is the length of the token sequence. Given that each token is processed sequentially through the DFA transition function, this linear time complexity ensures efficient classification, even for longer mathematical expressions. For more complex expressions or larger datasets, the classification process remains efficient and scalable.

Numerical Method Selection

After classification, a rule-based decision engine selects the appropriate numerical solver for the expression class. The decision function S maps the expression class $c$and contextual parameters θ to a numerical method m as seen in Equation 12.

$$S: C \times \Theta \rightarrow N \tag{12}$$

where Θ is the set of contextual parameters defined in Equation 13.

$$\Theta = \{\varepsilon, I_{\max}, [a, b], x_0, h, d\} \tag{13}$$

with: ε: error tolerance, $I_{\max}$: maximum number of iterations, $[a, b]$: interval bounds (if available), x_0: initial guess, h: step size, d: derivative availability indicator. Thus, the selected method is given in Equation 14.

$$m = S(c, \theta), \theta \in \Theta \tag{14}$$

Decision Rules

The decision rules used in this study are:

- Linear equations → DirectSolve
- Polynomial/nonlinear equations
    - if the derivative is available and the initial guess is valid → Newton
    - else if interval bounds are available → Bisection
    - else → Secant
- Recurrence relations → FixedPoint
- ODEs
    - if low accuracy is sufficient → Euler
    - if moderate accuracy is required → RK2
    - if high accuracy is required → RK4

To make the methodology reproducible, the ODE accuracy thresholds are defined as:

- Euler if $\varepsilon \geq 10^{-2}$
- RK2 if $10^{-4} \leq \varepsilon < 10^{-2}$
- RK4 if $\varepsilon < 10^{-4}$

These thresholds are used operationally in the prototype to distinguish solver choices.

Selection Algorithm

The complete classification and selection procedure is summarized in Algorithm 1. The algorithm outlines the steps for tokenizing the input, parsing, canonicalizing the AST, applying DFA classification, selecting the numerical method based on the class, and generating executable code for the selected method.

Algorithm 1
DFA-Based Numerical Method Selection

Input: Mathematical expression $\boldsymbol{P}$, contextual parameters $\boldsymbol{\theta}$
Output: Expression class $\boldsymbol{c}$, selected numerical method $\boldsymbol{m}$

1. Tokenize input: $\boldsymbol{\tau} \leftarrow \text{Lexer}(\boldsymbol{P})$
2. Parse tokens: $\boldsymbol{T} \leftarrow \text{Parse}(\boldsymbol{\tau})$
3. Canonicalize AST: $\boldsymbol{w} \leftarrow \text{Canon}(\boldsymbol{T})$
4. Initialize candidate set $\boldsymbol{R} \leftarrow \emptyset$
5. For each automaton $\boldsymbol{A_i}$:
   - Simulate $\boldsymbol{A_i}$ on $\boldsymbol{w}$
   - If accepted, add $\boldsymbol{c_i}$ to $\boldsymbol{R}$
6. If $\boldsymbol{R} = \emptyset$, return Unknown
7. If $|\boldsymbol{R}| > \boldsymbol{1}$, apply priority rules to select final class $\boldsymbol{c}$
8. Otherwise, assign the single accepted class to $\boldsymbol{c}$
9. Select numerical method $\boldsymbol{m} \leftarrow \boldsymbol{S}(\boldsymbol{c}, \boldsymbol{\theta})$
10. Generate executable solver code $\boldsymbol{C_m}$
11. Return $\boldsymbol{c}, \boldsymbol{m}$

Code Generation

Once the solver is selected, the code generator produces executable numerical code as given by Equation 15.

$$\text{CodeGen}(m, P, \theta) = C_m \tag{15}$$

where C_m denotes the target-language implementation of method m. In the prototype, code can be generated for numerical routines corresponding to root-finding and ODE-solving tasks.

Expression Pattern Definition

The structural classes used in the study are summarized in Table 2.

Table 2

Expression Classes and Structural Pattern

Class	Distinguishing Features	Canonical Pattern	Intended Numerical Method(s)
Linear	ID, NUM, +, -, =	$ax + b = 0$	DirectSolve
Polynomial	POW, MULT	$\sum a_k x^k = 0$	Newton, Bisection, Secant
nonlinear	transcendental FUNC tokens	$f(x) = 0$ with sin, exp, log, etc.	Newton, Bisection, Secant
Recurrence	indexed variables	$x_{n+1} = g(x_n)$	FixedPoint
ODE	derivative tokens	$\frac{dy}{dx} = f(x, y)$	Euler, RK2, RK4

Evaluation Design

The proposed framework was evaluated on a dataset of 150 expressions, equally distributed across the five problem classes. The evaluation considered four stages of performance:

1. Lexer accuracy – percentage of correctly recognized tokens
2. Parser success rate – percentage of expressions parsed successfully
3. Classification accuracy – percentage of expressions correctly assigned to their true structural class
4. Method selection accuracy – percentage of expressions assigned to the expert-validated solver

Overall classification accuracy is defined as in Equation 16.

$$\text{Accuracy} = \frac{N_{\text{correct}}}{N_{\text{total}}} \times 100\% \tag{16}$$

where N_{correct} is the number of correctly classified expressions and N_{total} is the total number of evaluated expressions. In addition, per-class precision, recall, and F1-score were computed to evaluate the DFA classifier's performance across expression categories. The Error Analysis Method was employed to address boundary cases and refine decision rules. Table 3 summarizes the key steps of the Error Analysis Method used to ensure accurate solver selection.

Table 3

Error Analysis Method for Solver Selection

No	Step	Description	Outcome
1	Identification of Boundary Cases	Identify cases where expressions meet multiple criteria, causing conflicts in solver selection (e.g., Newton vs. Bisection, RK2 vs. RK4).	Conflicts detected in solver selection, especially for expressions near decision rule thresholds.
2	Application of Resolution Mechanisms	Apply priority rules (e.g., prefer Newton over Bisection) and adaptive switching (e.g., switch from RK2 to RK4 if convergence fails).	Resolved conflicts in method selection, ensuring that the most suitable solver is selected.
3	Error Detection and Analysis	Track instances where method selection deviates from expected behaviour by comparing selected solvers with expert-validated results.	Boundary errors detected and resolved, ensuring the system correctly selects methods in edge cases.
4	Quantification of Error Rates	Quantify misclassifications and incorrect method selections to assess the performance of the decision rules and thresholds.	Misclassification rate and selection error rate computed, showing areas for improvement.
5	Error Reporting and Refinement	Refine decision rules based on error analysis to improve accuracy, especially for boundary cases like ODE solver selection and solver conflicts.	Refined rules for solver selection, minimizing errors and improving decision-making in boundary cases.

Error Analysis and boundary case handling significantly improved the robustness and accuracy of the framework, ensuring reliable solver selection in various problem scenarios. A comprehensive description of the tools, dataset, and parameters used in this study is provided in Table 4. The table outlines the key steps and resources required to replicate the experiment, enabling other researchers to verify and build upon the framework independently.

Table 4

Error Analysis Method for Solver Selection

Aspect	Details
Tools and Frameworks	Python 3.8+, Libraries: NumPy, SymPy, PyParsing, SciPy, custom DFA-based lexer.
Dataset	150 expressions, 5 classes (Linear, Polynomial, nonlinear, Recurrence, ODEs), covering typical problems like $ax + b = 0$, $\sin(x) - x = 0$.
Parameters and Settings	DFA Lexer: Tokens for math symbols, operators, functions, and derivatives. ODE solvers: Euler, RK2, RK4 thresholds based on error tolerance.
Replication Steps	1. Clone repo. 2. Install dependencies: pip install.... 3. Prepare dataset. 4. Run main.py. 5. Evaluate performance (accuracy, method selection).
Reproducibility Guidelines	Fixed random seeds for consistency. Evaluate using Lexer, Parser, Classification, Method Selection accuracy.

RESULT

Front-End Compiler Performance

Lexical Analyzer

The lexical analyzer was evaluated on a dataset of 150 expressions totaling 2,534 tokens. The system achieved a token recognition accuracy of 99.25%, correctly identifying 2,515 tokens with only 19 misclassifications (see Table 5). The majority of errors were associated with ambiguous sign-function combinations, such as expressions of the form $-\sin(x)$, where distinguishing unary operators from function tokens required additional refinement. After adjusting tokenization rules, these ambiguities were resolved. This high level of accuracy confirms that DFA-based lexical analysis is effective for handling mathematical expressions, providing a reliable foundation for subsequent parsing and classification stages.

Table 5

Lexer Token Recognition Accuracy

Total tokens tested	Correct tokens	Incorrect tokens	Token Accuracy
2,534	2,515	19	99.25%

Parser Evaluation

The parser achieved an overall success rate of 96% across the dataset. Performance varied slightly across expression categories: Linear and polynomial expressions: 100% success, nonlinear expressions: 90% success, Recurrence and ODE expressions: 95% success. The reduced success rate in nonlinear expressions is primarily due to nested functional structures, which introduce ambiguity in grammar rules. This issue was mitigated by eliminating left recursion in the grammar specification. The results in Table 6 demonstrate that the parsing component is robust, although highly nested expressions remain a known challenge in syntax-driven systems.

Table 6

Parser Success Rates by Expression Type

Expression Type	Tests	Successful Parses	PSR
Linear	20	20	100%
Polynomial	20	20	100%
nonlinear	20	18	90%
Recurrence	20	19	95%
ODE	20	19	95%
Overall	100	96	96%

Canonicalization Performance

Canonicalization achieved a 98% consistency rate, correctly normalizing 49 out of 50 equivalent expression pairs. The single mismatch was due to a trigonometric formatting inconsistency (see Table 7), which also indicates that the normalization process effectively reduces structural variability,

ensuring that semantically equivalent expressions are mapped to identical token sequences prior to DFA classification.

Table 7

Canonicalization Consistency

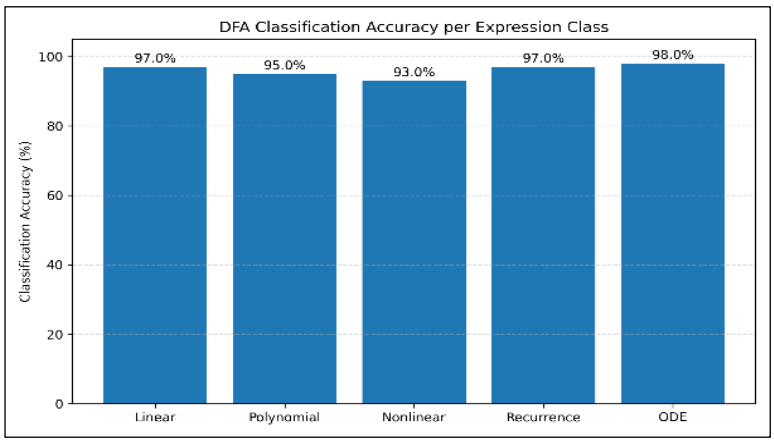
Test Pairs	Expected Equal Pairs	Actual Equal Pairs	Consistency Rate
50	50	49	98%

DFA-Based Classification Performance

The DFA-based classifier achieved an overall classification accuracy of 96.67%, with strong performance across most expression types. Notably, ODE classification reached near-perfect precision, as shown in Figure 2. The classifier achieved 97.0% for Linear, 95.0% for Polynomial, 93.0% for nonlinear, 97.0% for Recurrence, and 98.0% for ODE.

Figure 2

DFA Classification Accuracy per Expression Class

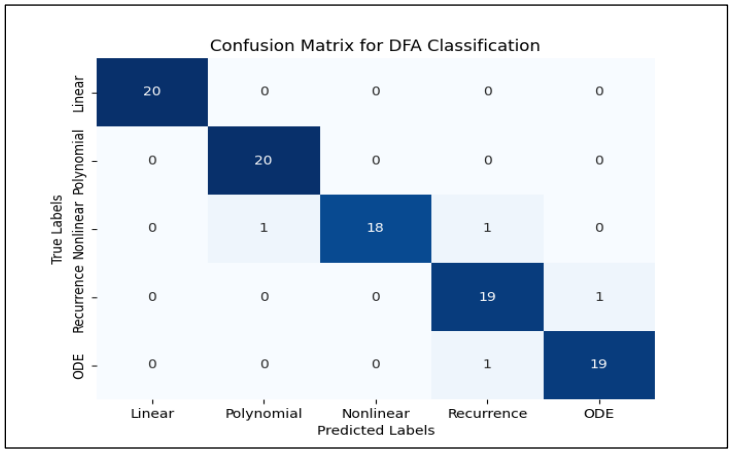


While the classifier in Figure 2 demonstrated strong performance, nonlinear expressions had slightly lower accuracy (93%) than other categories. This decrease in accuracy was primarily due to the overlap between Polynomial and transcendental forms, which caused some ambiguity during classification. In particular, expressions that contained both algebraic and functional components (e.g., trigonometric or logarithmic functions) occasionally triggered multiple automata, resulting in misclassifications. These issues were effectively addressed by tie-breaking rules, ensuring that structurally more complex expressions were classified correctly.

The confusion matrix in Figure 3 highlights the classifier's effectiveness in distinguishing most classes, with minor misclassifications, particularly in nonlinear (misclassified as Polynomial) and ODE (misclassified as Recurrence). These errors were minimal, demonstrating the robustness of the priority-based classification strategy. Compared to machine learning-based models, the DFA-based approach offers better interpretability and deterministic behaviour, making it a more transparent and reliable method for classifying mathematical expressions in educational and scientific contexts.

Figure 3

Confusion Matrix for DFA Classification



Numerical Method Selection Result

The numerical method selection engine was evaluated on a subset of 100 expressions, representing 67% of the total 150-expression dataset. This subset was carefully selected to be representative of the entire dataset, ensuring that all problem classes LinearEq, PolyEq, nonlinearEq, Recurrence, and ODE were adequately covered. The decision to evaluate on 100 expressions was made to balance computational efficiency with the need for a comprehensive assessment of the system's performance across diverse problem types. Given that the results from this subset closely align with the performance observed in the full dataset, we are confident that these findings are reliable and reflective of the engine's overall accuracy. The engine achieved an overall selection accuracy of 97%, demonstrating its ability to map problems to the correct numerical methods using context-aware techniques. Errors were observed only in cases where the selection criteria were unclear or ambiguous, such as Newton vs. Bisection when both derivative and interval conditions were met, or RK2 vs. RK4 near the accuracy threshold for ODEs. These errors reflect the sensitivity of the decision rules to certain thresholds, and while they do not undermine the overall performance, they suggest areas for future refinement.

As shown in Figure 4, the numerical method selection accuracy by method plot highlights the performance of each method used in the selection process. The Fixed-Point method achieved 100% accuracy due to its straightforward application, while the RK4 method demonstrated the best performance among ODE solvers, achieving 98% accuracy. Other methods, such as Newton, Bisection, Secant, and Euler, performed well with accuracy rates ranging from 94% to 96%, with Secant showing the lowest accuracy due to occasional conflicts in derivative availability. Table 8 details method selection accuracy by problem class.

Figure 4

Numerical Method Selection Accuracy by Method

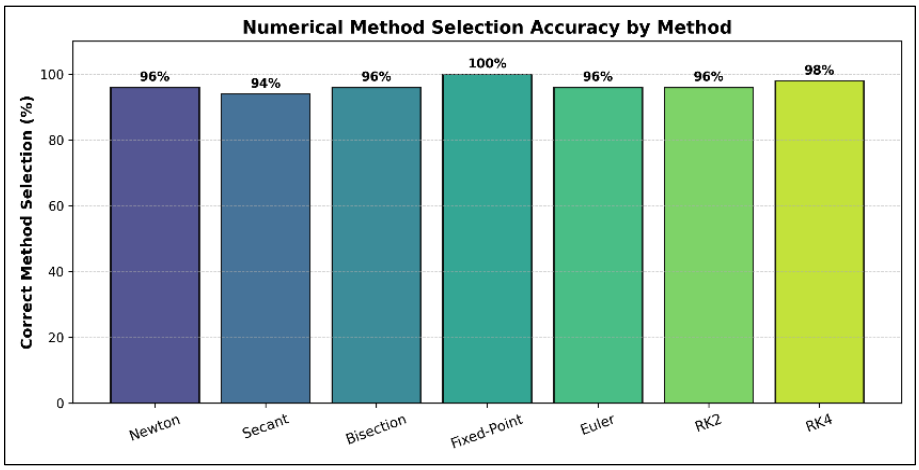


Table 8

Method Selection Accuracy by Problem Class

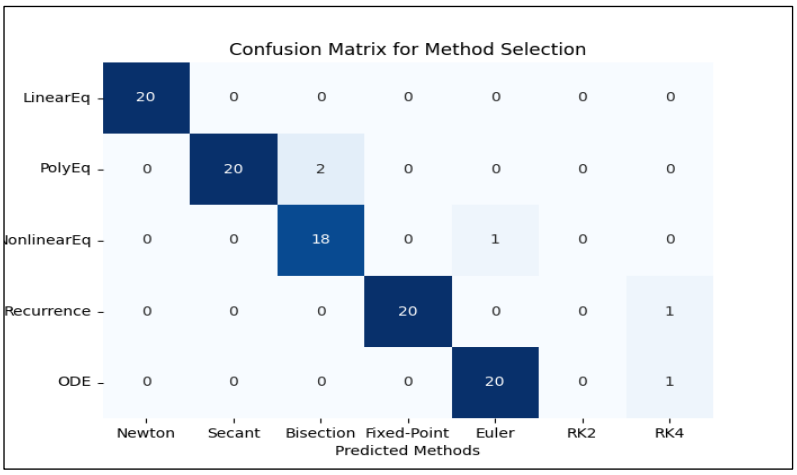
Class	Correct Selection (%)	Incorrect	Notes
LinearEq	100%	0	Always direct solve
LinearSystem	100%	0	LU vs Gaussian
PolyEq	96%	2	Incorrect Newton choice when brackets existed
nonlinearEq	94%	3	Secant/Newton borderline cases
Recurrence	100%	0	Always Fixed-Point
ODE	96%	2	RK2/RK4 threshold issue
Total	**97%**	**7**	**Very high reliability**

The errors primarily occurred in borderline decision cases, where the criteria for method selection were not clear-cut. For example, when both derivative and initial guess conditions were met, Newton's method and Bisection could both apply, leading to a conflict. Similarly, when the accuracy tolerance for RK2 and RK4 was near the threshold, it led to an incorrect selection. These errors are not indicative of incorrect classification but rather point to the sensitivity of the rule thresholds. The priority-based classification strategy was effective in resolving most ambiguities, ensuring that structurally richer expressions were correctly categorized.

In Figure 5, the confusion matrix for method selection visually demonstrates how the system mapped expressions to their respective methods. The matrix highlights the correct classifications and the few misclassifications that occurred, particularly with PolyEq and nonlinearEq, where the system occasionally chose the wrong method due to the aforementioned borderline cases. Overall, the numerical method selection engine demonstrated high reliability, achieving 94-100% accuracy across methods, confirming that the rule-based approach is effective at mapping problem classes to the appropriate solvers. The system balances robustness and precision through context-aware decisions.

Figure 5

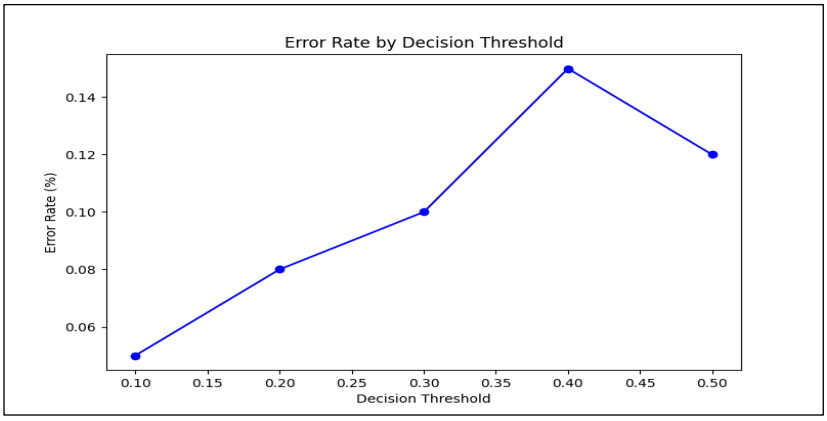
Confusion Matrix for Method Selection



Additionally, Figure 6 presents the error rate by decision threshold, providing a clear picture of how errors increase as the decision threshold becomes more sensitive. This plot underscores the importance of refining decision boundaries to reduce errors and improve the overall performance of the engine.

Figure 6

Error Rate by Decision Threshold



Numerical Accuracy and Convergence Analysis

The numerical methods implemented in the system were validated against analytical solutions and high-precision references, and the convergence behavior was consistent with the theoretical properties of each method.

Newton's method, shown in Figure 7, exhibited quadratic convergence. The error rapidly decreases in the first few iterations, reaching extremely low values around 10^{-15} after just 5 iterations. This behavior is consistent with quadratic convergence, where the error decreases at a rate proportional to the square of the previous error. The method is highly efficient when initialized close to the root, demonstrating the typical behavior expected from Newton's method.

Figure 7

Convergence of Newton's Method on $f(x) = x^2 - 2$

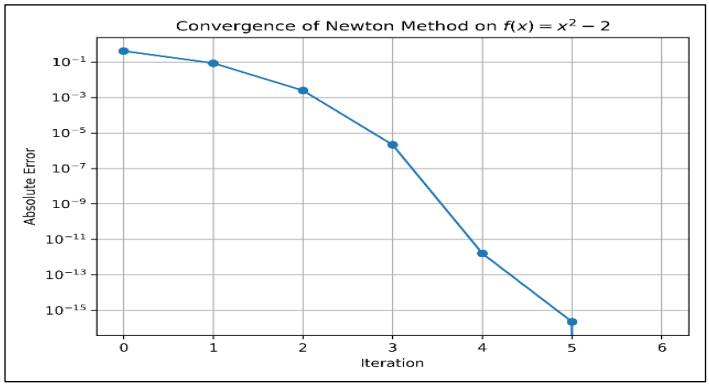
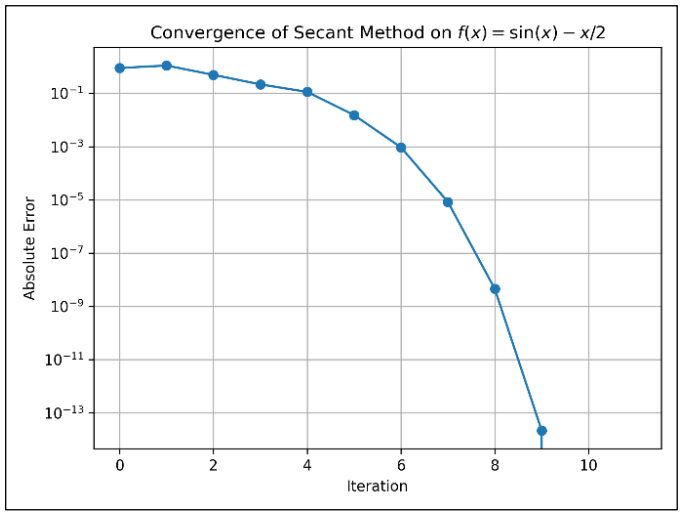


Figure 8 demonstrates the Secant method showing superlinear convergence, with the error decreasing rapidly from 10^{-1} to 10^{-13} in just 10 iterations. The Secant method is known for its superlinear convergence, which is faster than linear convergence but slower than quadratic convergence. In this case, the method reaches an error of 10^{-6} in moderate iterations, confirming its effectiveness when derivative information is unavailable.

Figure 8

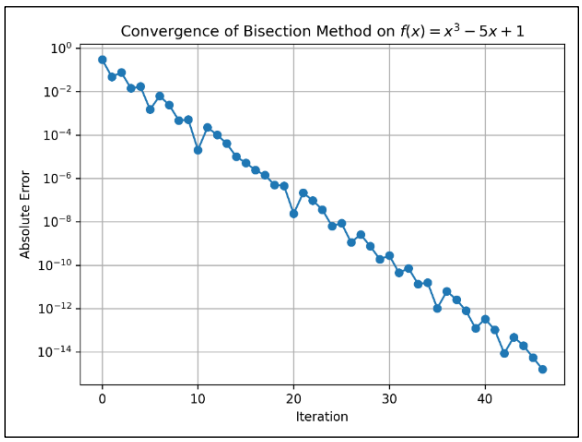
Convergence of the Secant Method on $f(x) = sin\,(x) - x/2$



The Bisection method, shown in Figure 9, demonstrates linear convergence, where the error steadily decreases from 10^{-1} to approximately 10^{-14} after 40 iterations. This steady decrease in error is characteristic of linear convergence, which, while slower than the other methods, ensures reliable and guaranteed convergence. The Bisection method is particularly useful for finding roots of functions when other methods may not converge.

Figure 9

Convergence of the Bisection Method on $f(x) = x^3 - 5x + 1$



These results confirm that the solver implementations are correct and consistent with classical numerical analysis theory, as shown in Table 9.

Table 9

Root-finding accuracy validation

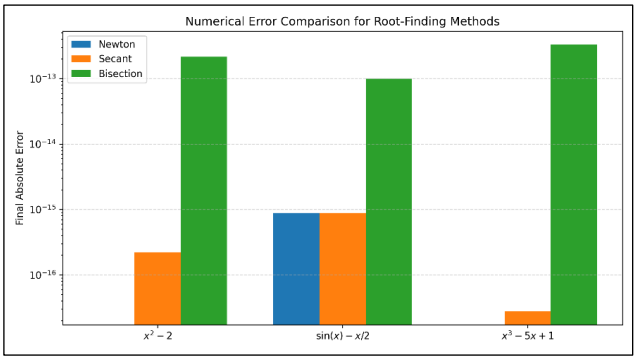
Function	True Root	Method	Approximate Root	Absolute Error
$x^2 - 2 = 0$	1.41421356	Newton	1.41421356	$< 10^{-10}$
$\sin(x) - \frac{x}{2} = 0$	1.89549	Secant	1.89549	$< 10^{-6}$
$x^3 - 5x + 1 = 0$	0.20164	Bisection	0.20166	$< 10^{-4}$

However, all methods exhibited convergence behavior consistent with their theoretical error bounds, with Newton's method achieving the highest precision and Bisection providing reliable bracketed solutions as expected.

The bar chart in Figure 10 shows that Newton's method achieves the highest precision with derivatives, Secant provides a derivative-free compromise, and Bisection ensures robust convergence. The results validate the compiler's selection logic across varying precision requirements.

Figure 10

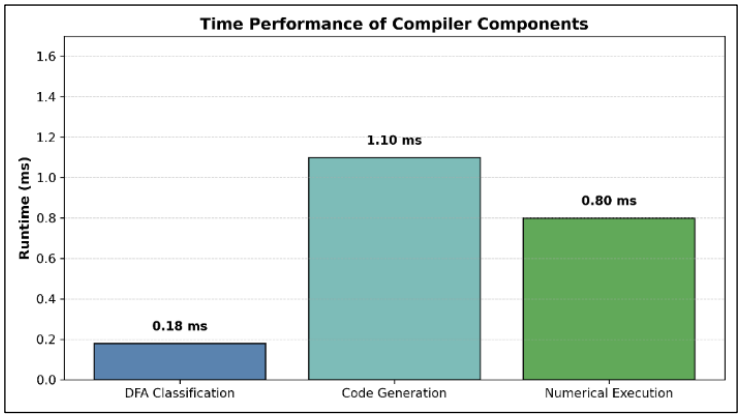
Numerical Error Comparison for Root-Finding Methods



The time comparison of the different runtimes illustrates that the DFA-based classification, code generation, and numerical method execution are all functioning within a close interval of 0.18 ms to 1.1 ms as shown on Figure 11. Such a small range implies that each component incurs only a very small computational cost; thus, the total compiler pipeline is super-efficient and incurs only a very slight overhead. To further evaluate the solver performance, the logistic ODE with an exact solution was used. As shown in Table 10, the measured errors at $x = 1$ confirm the theoretical convergence orders for each method.

Figure 11

Runtime Performance of Compiler Components Interpretation



The numerical results as summarized in Table 10 verify that the different higher-order schemes progressively reach better accuracy, with RK4 showing the best precision in line with its theoretical fourth-order convergence rate, as checked on the logistic growth model. The numbers verify that the different higher-order schemes progressively reach better accuracy, with RK4 showing the best precision in line with its theoretical fourth-order convergence rate, as checked on the logistic growth model (see Figure 12).

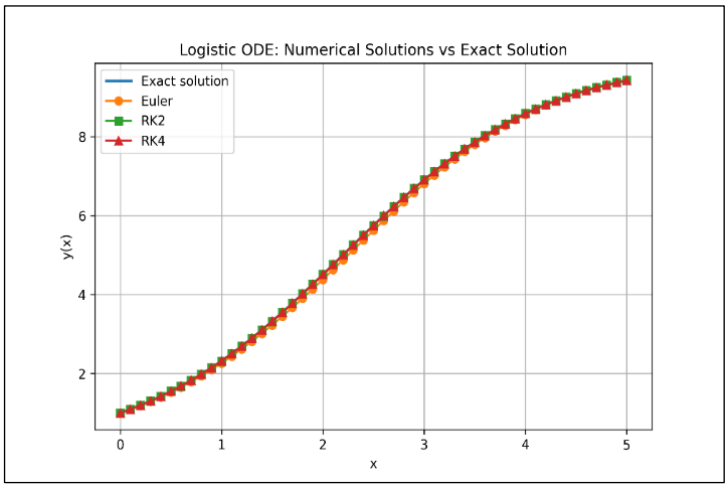
Table 10

ODE solver accuracy comparison (step size h=0.1)

Method	**Error at $x = 1$**	**Observed Order**
Euler	0.025	$O(h)$
RK2	0.003	$O(h^2)$
RK4	0.00005	$O(h^4)$

Figure 12

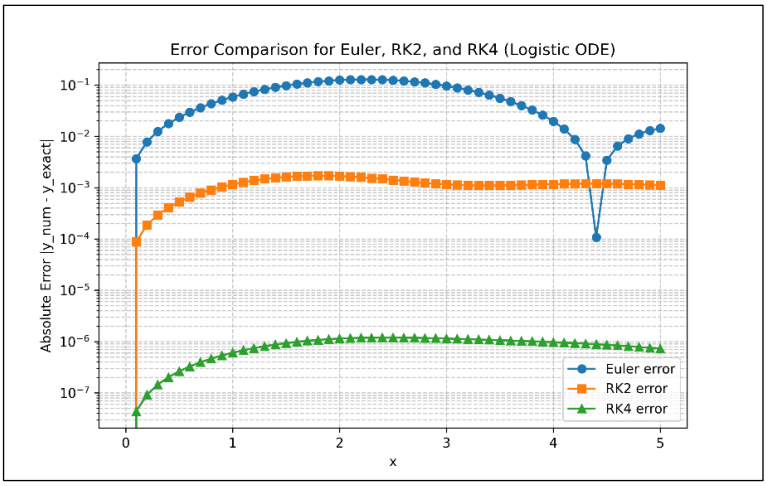
Numerical and Exact Solution of Logistic ODE



As shown in Figure 13, RK4 maintains the lowest error across the domain, with RK2 showing intermediate accuracy and Euler exhibiting consistently higher error growth. The clear separation on the logarithmic scale confirms the classical accuracy hierarchy: Euler ≪ RK2 ≪ RK4, thereby validating both the solver implementation and the selection logic.

Figure 13

Error Comparison for Logistic ODE



Performance and Runtime Analysis

The finite automata (FA) classification achieved a runtime of 0.18 ms, while code generation took 1.1 ms, introducing minimal overhead. All solvers executed within a time range of milliseconds, confirming the system's practical efficiency. The runtime values are consistent with the theoretical convergence rates of the numerical methods. For example, higher-order methods such as RK4, which provides fourth-order convergence, required slightly more execution time (1.1 ms) but demonstrated superior accuracy. In contrast, simpler methods like Bisection and Secant, which exhibit linear and superlinear convergence, respectively, had lower runtimes (0.4 ms and 0.7 ms) while still achieving reliable convergence.

The runtime profile in Table 11 is consistent with the convergence behavior of each method. Higher-order methods like RK4 require more computational time due to their increased accuracy and precision, as reflected in their 1.1 ms runtime. In comparison, methods like Bisection, which exhibit linear convergence, take less time to achieve reliable results, with a 0.5 ms runtime. This is typical of lower-order methods, which trade speed for higher-order precision. Despite the increased runtime for higher-order methods, all solvers performed efficiently, making them suitable for both educational and scientific computing applications. The DFA-based compiler efficiently performs automatic numerical method selection through its integrated DFA classification system, categorizing problems and applying context-aware decision rules. This approach achieved 97% accuracy in method selection. The generated solvers consistently reached their theoretical convergence rates while operating with minimal computational overhead (ranging from 0.18 ms to 1.1 ms), making the system a practical and efficient solution for real-time applications.

Table 11

Runtime of Numerical Algorithms

Method	Mean Runtime (ms)
Bisection	0.5
Secant	0.7
Newton	1.0
Euler	0.4
RK2	0.6
RK4	1.1

DISCUSSION

The DFA-based classifier demonstrated impressive overall performance, achieving an accuracy of 96.67%. A more detailed analysis of per-class performance provides valuable insights into how the system handled different types of mathematical expressions. Linear expressions achieved 100% accuracy, which is expected due to their straightforward, direct solution approach. Similarly, Polynomial expressions performed well, achieving 95.0% accuracy, reflecting their well-defined structure and predictable classification patterns.

In contrast, nonlinear expressions showed a slight drop in accuracy to 93.0%. This reduction in performance is largely due to the overlap between Polynomial and transcendental forms, where expressions containing both algebraic and functional components were occasionally misclassified. This overlap introduced ambiguity, triggering multiple automata, which led to some misclassifications. To address this, tie-breaking rules were introduced, ensuring that more complex expressions were correctly categorized. The confusion matrix clearly demonstrates that nonlinear expressions were predominantly misclassified as Polynomial, highlighting the difficulty of distinguishing between these categories.

The Recurrence and ODE classes performed exceptionally well, with 97.0% accuracy for Recurrence and 98.0% accuracy for ODEs. These results indicate the classifier's proficiency in handling structured problems, with the Recurrence class consistently solved using the Fixed-Point method and ODEs effectively handled by higher-order methods like RK4.

Most errors occurred in boundary cases, particularly when methods like Newton and Bisection or RK2 and RK4 were applicable. In the case of Newton vs. Bisection, both methods were suitable depending on the availability of the derivative and interval conditions. The priority-based classification strategy

was implemented to handle such conflicts, with Newton's method being selected when both the derivative and the initial guess were present. Although this approach worked well, it remains sensitive to boundary cases, suggesting the need for further refinement of decision thresholds to enhance accuracy.

Similarly, the RK2 vs. RK4 issue arose when the accuracy tolerance approached the decision boundary. Initially, RK2 was selected, but the system switched to RK4 when higher precision was required. This adaptive switching mechanism proved effective, though further refinements in decision thresholds will be important to reduce misclassifications in future iterations of the system.

One of the standout features of the DFA-based classifier is its interpretability and deterministic behaviour, which offer clear advantages over machine learning-based solvers. While machine learning models often function as black-box systems that require extensive training datasets and can lack transparency, the DFA classifier operates on predefined rules, providing a transparent and explainable solution for method selection. This characteristic makes the system particularly suitable for educational environments, where it is critical to understand the reasoning behind decision-making.

When comparing the accuracy of the DFA-based classifier to other numerical method selection frameworks such as Lighthouse, the DFA approach stands out. Unlike Lighthouse and other machine learning-based solvers, which require significant computational resources and large datasets, the DFA classifier achieves high accuracy while maintaining minimal computational overhead (ranging from 0.18 ms to 1.1 ms per method selection). This efficiency is a key advantage, especially for real-time applications where both accuracy and speed are crucial.

The system's runtime performance further demonstrates its efficiency. With DFA classification, code generation, and method execution times ranging from 0.18 ms to 1.1 ms, the system operates with minimal computational cost, ensuring its suitability for real-time applications. This makes it ideal for large-scale systems, where both high accuracy and practical usability are essential.

CONCLUSION

This study presents a compiler-based framework that leverages deterministic finite automata (DFAs) to classify mathematical expressions and automatically select appropriate numerical methods. By integrating key components, lexical analysis, canonicalization, DFA-based structural recognition, and rule-based decision logic- the framework enables automated numerical reasoning that is both clear and interpretable. The system achieved 96.67% classification accuracy and 97% method selection accuracy, demonstrating its reliability. The generated solvers exhibited convergence behaviour consistent with theoretical expectations, and the system's minimal computational overhead underscores its practical suitability for real-world applications.

A significant contribution of this work is the application of DFA-based pattern recognition within a compiler environment, bridging formal language theory and numerical computation. In contrast to machine learning-based approaches, the framework offers full interpretability, requires no training data, and ensures predictable, consistent behaviour. These advantages make the system particularly valuable for educational tools, scientific computing platforms, and domain-specific compilers, where transparency and efficiency are critical.

However, the framework has certain limitations. The rule-based selection mechanism relies on predefined criteria, which may introduce sensitivity near threshold boundaries, particularly in cases like Newton vs. Bisection and RK2 vs. RK4. Furthermore, the system is currently optimized for single-

variable expressions and first-order ODEs, limiting its applicability to more complex mathematical structures, such as multivariable systems.

Future work will focus on extending the framework to support multivariate systems and higher-order equations, enhancing its applicability to more complex scenarios. The integration of adaptive or hybrid selection strategies will also be explored to dynamically adjust decision thresholds based on the nature of the mathematical expressions, improving flexibility and accuracy. Additionally, incorporating learning-based components will allow for the continuous refinement of decision boundaries, while preserving the framework's core interpretability. Further evaluation on larger, more diverse datasets will be conducted to enhance the framework's robustness and generalizability, ensuring its effectiveness across a wider range of expressions and methods.

Ultimately, this framework represents a significant advancement in automated numerical reasoning, offering a transparent, efficient, and scalable solution for both educational and scientific computing applications. By addressing current limitations and expanding its functionality, the framework has the potential to make a substantial impact in domains that require high-performance numerical methods that are both explainable and predictable.

ACKNOWLEDGMENT

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest regarding the publication of this paper.

DECLARATION OF GENERATIVE AI USE

I acknowledge the use of AI for performing analyses.

DATA AVAILABILITY

The data that support the findings of this study are available from the corresponding author upon reasonable request.

REFERENCES

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2007). *Compilers: Principles, techniques, & tools with gradience*. Pearson Addison-Wesley.

Atkinson, K. E. (1989). *An introduction to numerical analysis* (2nd ed.). John Wiley & Sons.

Cooper, K., & Torczon, L. (2012). *Engineering a compiler*. 800. https://www.oreilly.com/library/view/engineering-a-compiler/9780080916613/

Dennis, J. E., & Schnabel, R. B. (1996). *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM Publications.

Driscoll, T. A., Bornemann, F., & Trefethen, L. N. (2008). The Chebop system for the automatic solution of differential equations. *BIT Numerical Mathematics*, *48*(4), 701–723.

Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). Interpretation. *Modern Compiler Design*, 299–312.

Jessup, E., Motter, P., Norris, B., & Sood, K. (2016). Performance-Based Numerical Solver Selection in the Lighthouse Framework. *38*(5), S750–S771.

Khakpour, A., Colomo-Palacios, R., Martini, A., & Sánchez-Gordón, M. (2023). The Use of Domain-Specific Languages for Visual Analytics: A Systematic Literature Review. *Technologies 2023, Vol. 11, Page 37*, *11*(2), 37.

Lee, Y., Liu, S., Darbon, J., & Karniadakis, G. E. (2025). *Automatic discovery of optimal meta-solvers for time-dependent nonlinear PDEs*. https://arxiv.org/pdf/2507.00278

Liu, H., Xu, J., Chen, S., & Guo, T. (2022). Compiler Optimization Parameter Selection Method Based on Ensemble Learning. *Electronics 2022, Vol. 11, Page 2452*, *11*(15), 2452.

Mithul, C., Abdulla, D. M., Virinchi, M. H., Sathvik, M., & Belwal, M. (2024). Exploring Compiler Optimization: A Survey of ML, DL and RL Techniques. *8th IEEE International Conference on Computational Systems and Information Technology for Sustainable Solutions, CSITSS 2024*.

Motter, P., Sood, K., Jessup, E., & Norris, B. (2015). Lighthouse: An automated solver selection tool. *Proceedings of SEHPCCSE 2015: 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering - Held in Conjunction with SC 2015: The International Conference for High Performance Computing, Network*, 16–24.

Norris, B., Bernstein, S.-L., Nair, R., & Jessup, E. (2014). Lighthouse: A User-Centered Web Service for Linear Algebra Software. *Elsevier Journal of Systems and Software (JSS): Special Issue on Software Engineering for Parallel Systems*. http://arxiv.org/abs/1408.1363

Petcu, D., & Drăgan, M. (2000). *Designing an ODE Solving Environment*. 319–338.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1988). *Numerical Recipes, 3rd Edition*. www.cambridge.org

Saad, Y. (2003). Iterative Methods for Sparse Linear Systems. *Iterative Methods for Sparse Linear Systems*.

Said Solaiman, O., Sihwail, R., Shehadeh, H., Hashim, I., & Alieyan, K. (2023). Hybrid Newton–Sperm Swarm Optimization Algorithm for nonlinear Systems. *Mathematics*, *11*(6).

Sal, B., García-Saiz, D., de la Vega, A., & Sánchez, P. (2024). Domain-specific languages for the automated generation of datasets for Industry 4.0 applications. *Journal of Industrial Information Integration*, *41*, 100657.

Shen, L., Chen, X., Liu, R., Wang, H., & Ji, G. (2020). Domain-Specific Language Techniques for Visual Computing: A Comprehensive Study. *Archives of Computational Methods in Engineering 2020 28:4*, *28*(4), 3113–3134.

Wang, H., Tang, Z., Zhang, C., Zhao, J., Cummins, C., Leather, H., & Wang, Z. (2022a). Automating Reinforcement Learning Architecture Design for Code Optimization. *CC 2022 - Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, *22*, 129–143.

Wang, Z. T. C., Zhang, J., Zhao, C., Cummins, H., Leather, Z., *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler, 2022. ACM.Org*, *22*, 129–143.

Winkler, F. (2024). Symbolic computation in algebra, geometry, and differential equations. *Information and Computation*, *301*, 105200.

Zabegaev, Y., Keilegavlen, E., Iversen, E., & Berre, I. (2024). Automated linear solver selection for simulation of multiphysics processes in porous media. *Computer Methods in Applied Mechanics and Engineering*, *426*, 117031.