Information Dynamics and Applications
https://www.acadlore.com/journals/IDA

ChaosL: A Grammar-Based Precision-Aware Programming Language for Reliable Computation in Chaotic Systems

Samar Amil Qassir*

Department of Computer Science, College of Science, Mustansiriyah University, 10001 Baghdad, Iraq

* Correspondence: Samar Amil Qassir (samarqassir@uomustansiriyah.edu.iq)

Received: 08-03-2025 **Revised:** 09-14-2025 **Accepted:** 09-25-2025

Citation: S. A. Qassir, "ChaosL: A grammar-based precision-aware programming language for reliable computation in chaotic systems," *Inf. Dyn. Appl.*, vol. 4, no. 3, pp. 173–188, 2025. https://doi.org/10.56578/ida040305.

Abstract: This study introduces a grammar-based, chaotic-oriented programming language, termed ChaosL, to address persistent numerical precision and reproducibility challenges in the computational analysis of chaotic systems. The language, along with its compiler and parser, is designed end-to-end with consideration of chaotic maps. Numerical accuracy is systematically managed through grammar-level precision specification and automated error monitoring mechanisms, enabling exact control over floating-point representations, including single precision, double precision, and arbitrary-precision BigDecimal arithmetic with configurable decimal resolution of up to 100 digits. The proposed grammar natively supports ten widely studied one-dimensional and two-dimensional discrete chaotic maps, which may be composed using newly defined hybrid composition paradigms, namely alternate, blend, cascade, and feedback-driven coupling. To ensure computational reliability, multiple error assessment strategies are integrated, including direct error estimation, shadow computation, and interval arithmetic. In addition, ensemble-based simulation capabilities are incorporated to evaluate trajectory separation and estimate predictability horizons. The automated computation of Lyapunov exponents is embedded at the language level, achieving an accuracy of up to 99.6% while simultaneously enabling code-size reductions of approximately 85–92%. The adaptable architecture of ChaosL establishes a reproducible computational framework for discrete chaos research and facilitates the systematic identification of emergent behaviors in hybrid dynamical systems. Moreover, the design provides a scalable foundation for future extensions toward continuous-time systems, interactive visualization environments, and cloud-based collaborative experimentation, thereby advancing precision-aware computational practices in nonlinear dynamics and chaos theory.

Keywords: Chaotic-oriented programming language; Grammar-based language design; Chaotic maps; Xtext platform; Extended Backus-Naur Form

1 Introduction

A nonlinear system with predictable dynamic behavior is called chaos. It is very sensitive to its starting circumstances and parameters and possesses ergodicity, stochasticity, and regularity features. After several iterations, little variations in the original solutions' values can lead to significant variations. Chaotic maps are used in many cryptographic techniques for encryption, key generation, and the creation of pseudo-random numbers [1, 2]. It is crucial for people who employ nonlinear behaviors and environmental disruptions, all of which frequently occur in different feedback control systems. In optimization, machine learning, and robotics challenges, chaotic dynamics facilitate pattern development, overcome local minima, and enhance exploration. Additionally, chaotic systems are widely employed to model nonlinear, stochastic, and unpredictable real-world phenomena, including economic, biological, and physical systems [3, 4].

When implementing chaotic maps in general-purpose programming languages (GPPLs), the numerical constraints imposed by typical floating-point representations show themselves in a number of systematic ways. IEEE-754 double-precision arithmetic, which is used by languages like C/C++, Java, Python, and MATLAB, introduces deterministic rounding, quantization, and cancelation effects during iterative calculations [5, 6]. After very few cycles, these machine-level errors spread and take control of the trajectory due to chaotic systems' exponential amplification of disturbances, resulting in divergence from the theoretically predicted orbit, as seen in Figure 1. Because of differences in optimization tactics, operand ordering, intermediate precision, and library implementations

of elementary functions, various GPPLs, compilers, or even hardware designs provide non-identical results. The crucial sensitivity of chaotic simulations to the underlying numerical infrastructure of GPPLs is highlighted by this unpredictability, which leads to reproducibility issues, uneven long-term behavior, and the possible collapse of chaos into misleading periodic cycles. Without specific mitigation strategies, it is difficult to accurately reproduce chaotic behavior due to these numerical instabilities [7–9].

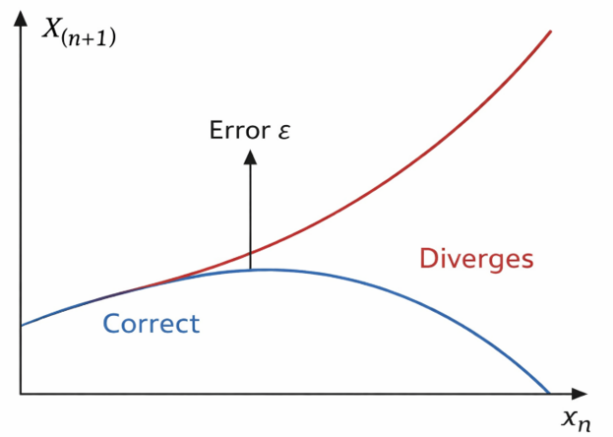


Figure 1. Divergence of chaotic map trajectories [10]

This study introduces a grammar-based language called ChaosL to address the numerical precision problems in chaotic systems and provides specific features. The main contributions of the proposed language are as follows: First, it is the first grammar-based language designed specifically for chaotic dynamical systems, bridging the gap between theory and real-world computation. ChaosL's intuitive syntax makes it possible for programmers and non-programmers to conduct chaotic experiments.

Second, the proposed language addresses floating-point errors that are frequent in chaotic systems by providing support for many precision controls (such as float, double, BigDecimal, and extended). DecimalPlaces can be adjusted by programmers for accurate math computations.

Third, four error-tracking modalities are available for integrated error tracking and analysis. The Lyapunov exponent is automatically computed to measure predictability limitations and chaos.

Finally, new hybrid map features support ten one-dimensional and two-dimensional chaotic maps within a single framework. New map types may be added with ease because of the flexible design. The composition of chaotic systems is enabled through alternate, blend, cascade, and feedback-based hybrid modes.

The remainder of this study is organized below. Section 2 presents the chaotic maps used in the proposed language. Section 3 presents the grammar-based design using the Xtext platform. Section 4 explains the proposed language design. Sections 5 and 6 describe in depth and discuss the assessment. Lastly, Section 7 provides a succinct conclusion.

2 Chaotic Maps

Chaos is a deterministic dynamic system with some regularity. The link between completely random chaotic outputs and the underlying patterns that produce them is explained by chaos theory. For ten chaotic maps (one-dimensional and two-dimensional maps, hybrid dynamics, and multi-precision), the proposed language offers a cohesive structure. This section covers these maps [10, 11].

2.1 Logistic Map

This map exhibits intricate chaotic behavior despite its straightforward design. It is a dynamical system with discrete time. Chaos, bifurcation, and starting condition sensitivity are frequently used to study nonlinear systems [12]. Its equation is as follows:

$$p_{n+1} = tp_n\left(1 - p_n\right), \text{ for } 0 < p_n < 1, 0 < t < 4 \tag{1}$$

2.2 Tent Map

It is a piecewise linear chaotic map in one dimension. It is applied to the study of ergodic behavior and deterministic chaos. It is computed as Eq. (2) and Eq. (3) and is very sensitive to the beginning circumstances and 0-uniform

invariant density for certain parameters [13]:

$$p_{n+1} = \mu p_n, \text{ where } 0 \leq p_n < \frac{1}{2} \tag{2}$$

$$p_{n+1} = \mu\left(1 - p_n\right), \text{ where } \frac{1}{2} \leq p_n \leq 1 \text{ and } 0 < \mu \leq 2 \tag{3}$$

2.3 Hénon Map

It is a two-dimensional discrete-time nonlinear dynamical system, one of the traditional illustrations of chaos in low-dimensional systems [14], which is computed as follows:

$$p_{n+1} = \left(1 - ap_n^2 + q_n\right) \tag{4}$$

$$q_{n+1} = bp_n, \text{ where } a = 1.4 \text{ and } b = 0.3 \tag{5}$$

2.4 Gauss Map

This one-dimensional chaotic map is associated with continuing fraction expansions and appears in number theory and dynamical systems. It features a constant periodic probability distribution and exhibits strong sensitivity to beginning circumstances [15]. The following is its equation:

$$p_{n+1} = \{1/p_n\}, \text{ where } 0 < p_n \leq 1 \tag{6}$$

2.5 Sine Map

It is a one-dimensional nonlinear chaotic map that is utilized in cryptography and chaos analysis and is formed from the sine function. Depending on the control parameter, it displays rich dynamical behavior, such as chaos and bifurcations [16]. Its equation is as follows:

$$p_{n+1} = \mu \sin\left(\Pi p_n\right), \text{ where } 0 < p_n < 1 \text{ and } 0 < \mu \leq 1 \tag{7}$$

2.6 Circle Map

This one-dimensional nonlinear dynamical system is used to investigate the transition to chaos, mode locking, and quasi periodicity. It simulates how a circle's phases change under periodic force [17]. The equation is as follows:

$$\Theta_{n+1} = \Theta_n + \mho - \frac{\mathrm{h}}{2\Pi} \sin\left(2\Pi\Theta_n\right) \tag{8}$$

2.7 Baker Map

It is a two-dimensional chaotic map characterized by stretching and folding dynamics. The map is frequently employed in chaotic research and image encryption [18]. Its equations are as follows:

$$p_{n+1}, q_{n+1} = \left(2p_n, q_n/2\right), \text{ where } 0 \leq p_n < \frac{1}{2} \tag{9}$$

$$p_{n+1}, q_{n+1} = \left(2p_n - 1, q_{n+1}/2\right), \text{ where } \frac{1}{2} \leq p_n < 1 \tag{10}$$

2.8 Arnold Map

It is a two-dimensional chaotic area-preserving transformation and is also referred to as Arnold's cat map. It is a common option for picture scrambling and encryption because of its well-known capacity to confuse objects and its periodicity in digital images [19]. The equation is as follows:

$$\begin{pmatrix} p_{n+1} \\ q_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} p_n \\ q_n \end{pmatrix} \bmod 1 \tag{11}$$

2.9 Bernoulli Map

It is a straightforward one-dimensional chaotic map that exhibits uniform invariant distribution and sensitivity to beginning circumstances. It is frequently used to create pseudo-random sequences and as a rudimentary model for chaos [20]. Its equation is as follows:

$$p_{n+1} = \left(kp_n\right) \bmod 1, \text{ where } 0 < p_n < 1 \text{ and } k > 1 \tag{12}$$

2.10 Tinkerbell Map

It is a two-dimensional chaotic map that generates complex patterns and dynamic behavior. The map may be used in image encryption, cryptography, and chaos theory [21]. Its equations are as follows:

$$p_{n+1} = \mathrm{p_n^2} - \mathrm{q_n^2} + \mathrm{ap_n} + \mathrm{bq_n} \tag{13}$$

$$q_{n+1} = 2p_n q_n + cq_n + dq_n \tag{14}$$

where, a, b, c, and d are real parameters that control the behavior of the map.

3 Grammar-Based Language Design with Xtext

Xtext is an open-source platform that is integrated into Eclipse and offers complete tool support for the development of certain programming languages [22]. Generative programming and specific-oriented programming language design are at the core of model-driven development (MDD) [23–25]. Extended Backus-Naur Form (EBNF) in Xtext is used to define the language's syntax (grammar). Xtext provides a complete language infrastructure that includes an abstract syntax tree (AST) parser, a linker to fix cross-references, and a sizable Eclipse integrated development environment (IDE) for debugging, refactoring, error correction, syntax highlighting, and code completion. As shown in Figure 2, the EBNF is utilized to represent a context-free grammar from which strings may be produced with ease. Additionally, it employs the same grammar rules as the Backus-Naur Form (BNF) but does not have the same restrictions. Xtext grammar rules include terminals, non-terminals, and symbols for alternation, grouping using parentheses, and repetition, including* (zero or more) and + (one or more). Assignment operators (= and +=) are used to bind parsed elements to AST features. Xtext parses the grammar specification and generates the corresponding lexer and parser, as tool generation is fundamentally based on EBNF-style grammars [26].

```
grammar rule :: = expression
expression ::= term , { term }
term ::= factor | "(" , expression , ")"
factor ::= identifier | number
identifier  ::= letter , { letter | digit | "_" }
number ::= digit , { digit } , [ "." , digit , { digit } ]
letter  ::= "A" | "B" | "C" | "D" | "E" | "F" | "G"
          | "H" | "I" | "J" | "K" | "L" | "M"
          | "N" | "O" | "P" | "Q" | "R" | "S"
          | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
          | "a" | "b" | "c" | "d" | "e" | "f" | "g"
          | "h" | "i" | "j" | "k" | "l" | "m"
          | "n" | "o" | "p" | "q" | "r" | "s"
          | "t" | "u" | "v" | "w" | "x" | "y" | "z"
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Figure 2. Structure of an EBNF-based grammar [22]

4 Proposed ChaosL Design

This section describes the proposed language's compiler process and parser design from end to end, as shown in Figure 3. The syntax is designed to address the numerical restrictions imposed by normal floating-point representations evident for chaotic-map calculations. ChaosL is based on a novel set of commands formulated using EBNF.

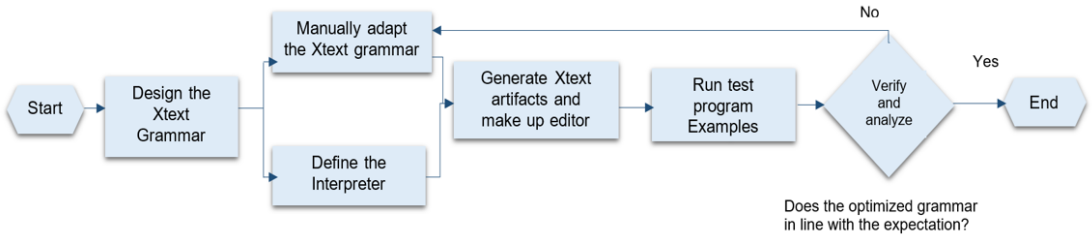


Figure 3. Flowchart of the ChaosL language design using Xtext

The syntax of the six new command sets facilitates error management and precise control. The commands are a clear syntactic and semantic description that gives the programmers the information they need to work. Precision and DecimalPlaces commands are used to tackle the restricted precision problem. Tracking shadow/interval instructions are utilized for error tracking. The LyapunovWindow command is employed to detect the onset of unpredictability, the Ensemble command is used to mitigate single-trajectory bias, and the WarningThreshold command is designed to identify silent numerical degradation, as explained in Table 1.

Table 1. ChaosL commands for precision control and error management

No.	Command set	Purpose	Features
1	Precision commands precision double precision BigDecimal precision float precision extended	Arithmetic precision control	Standard 64-bit (default), arbitrary precision, 32-bit (faster, less precise), and 128-bit if available
2	DecimalPlaces commands DecimalPlaces = 50 DecimalPlaces = 100	BigDecimal precision	Use 50 decimal places for BigDecimal and use 100 decimal places
3	ErrorTracking commands ErrorTracking none ErrorTracking basic ErrorTracking shadow ErrorTracking interval	Error monitoring	No tracking (default, fastest), track accumulated rounding errors, run parallel calculation with higher precision, and use interval arithmetic (tracks error bounds)
4	LyapunovWindow command LyapunovWindow = 50	Chaos indicator	Calculate Lyapunov exponent over 50 iterations, and warn when the system becomes unpredictable
5	Ensemble command ensemble = 10, variance = 0.0001	Multiple trajectories	Run ten simulations with tiny variations, and divergence appears due to chaos + errors
6	WarningThreshold command WarningThreshold = 40	Error alerts	Warn when errors likely dominate after 40 iterations

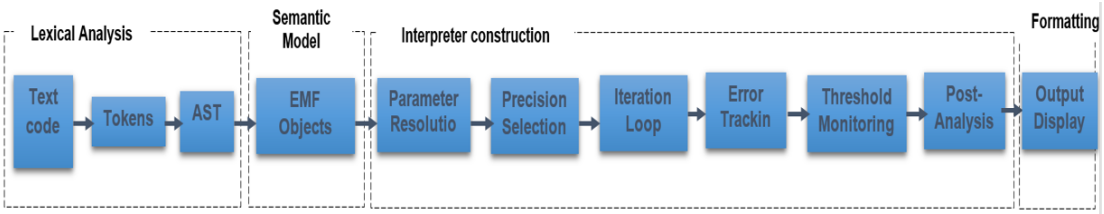
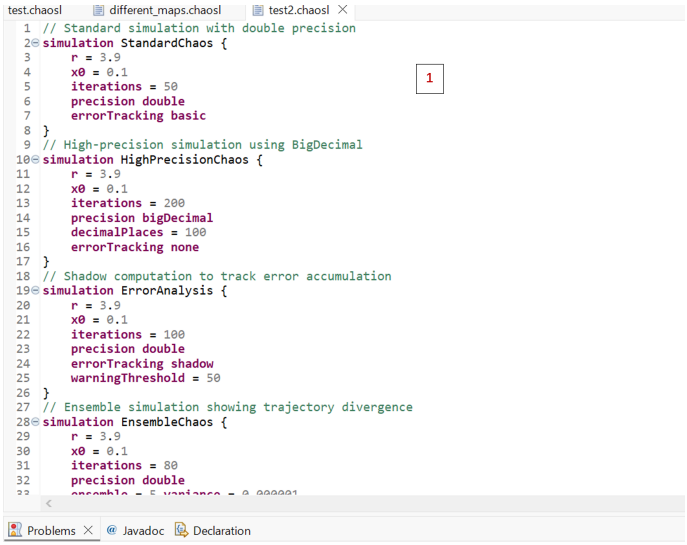


Figure 4. Complete pipeline of the complier process

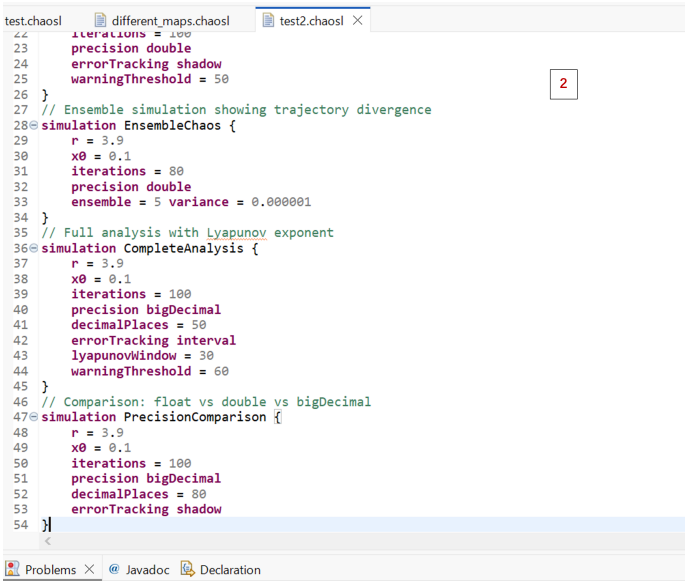
The error tracking modes of the proposed ChaosL are different in their relative costs and diagnostic utilities.

The basic mode (1.15–1.25 × speed, approximately 15–25% overhead) utilizes analytical derivative-based error estimation, resulting in fast bounds of order magnitude, which are useful for parameter sweeps and educational purposes. It underestimates, however, in highly nonlinear regions. The shadow mode (1.8–2.5 × for precision, 60–80 × for BigDecimal) runs parallel traces at different precisions (like double vs. 128-bit) and measures actual divergence $|x_{\text{double}} - x_{\text{highnrecision}}|$. This allows the programmer to determine exact error values. With the interval mode (2.0–3.5 × speed; maintains $[x_{\min}, x_{\max}]$ bounds), the programmer uses interval arithmetic to rigorously track uncertainty propagation and to guarantee that true values lie within computed bounds. This is essential in safety-critical applications, formal verification, or proving mathematical properties.

Instead of specifying how to carry out iteration loops or monitor errors, programmers may declare what they want to simulate (the type of map, the precision, the intended analysis). ChaosL provides a declarative nature, the interpreter's execution semantics of high-level specification, which is carried out through low-level numerical operations, and a number of built-in protections that maintain the implementation's integrity. The complete pipeline of the compiler process that comprises lexical analysis, semantic model, and interpreter construction is explained in Figure 4.



(a)



(b)

Figure 5. The first example explaining the new command set: (a) page 1; (b) page 2.

Three test program examples are provided below. The first example (page 1 and page 2) in Figure 5 explains the new command set using the same chaotic map, the second example in Figure 6 demonstrates the hybrid alternate

mode, and the third example in Figure 7 demonstrates a cryptographic key generator that exploits multiple chaotic systems with precision control commands.

test.chaosl | different_map... | test2.chaosl | chaotic_keystr... | test_precisio... | *dif

```
simulation AlternateLogisticTent {
    map logistic
    r = 3.9
    x0 = 0.1
    iterations = 50
    output console
    hybrid {
        map1 logistic
        map2 tent
        mode alternate
    }
}
simulation AlternateHenonBaker {
    map henon
    a = 1.4
    b = 0.3
    x0 = 0.1
    y0 = 0.1
    iterations = 40
    output console
    hybrid {
        map1 henon
        map2 baker
        mode alternate
    }
}
```

Figure 6. The second example explaining the hybrid alternate mode

test.chaosl | different_maps.chaosl | test2.chaosl | chaotic_keystream.chaosl

```
simulation SecureKeyGenerator {
    map logistic
    r = 3.99999
    x0 = 0.123456789012345
    iterations = 256
    precision bigDecimal
    decimalPlaces = 80
    errorTracking shadow
    lyapunovWindow = 50
      hybrid {
        map1 logistic
        map2 tinkerbell
        mode cascade
    }
}

simulation KeyQualityTest {
    map logistic
    r = 3.99999
    x0 = 0.123456789012346
    iterations = 256
    precision bigDecimal
    decimalPlaces = 80
    ensemble = 100
    variance = 0.000000000000001
  }
```

Figure 7. The third example explaining a cryptographic key generator

The ChaosL grammar definition of the new command set is presented in Figure 8 and the interpreter's complete definition is illustrated in Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, Figure 14, Figure 15, and Figure 16, respectively.

Model, MapType, PrecisionMode, ErrorTrackingMode, HybridConfig, and HybridMode are among the model

```
grammar org.example.chaosl.ChaosL with org.eclipse.xtext.common.Terminals
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate chaosL "http://www.example.org/chaosl/ChaosL"
Model: (simulations+=Simulation)*;
Simulation:
   'simulation' name=ID '{'
      ('map' mapType=MapType)?
      ('r' '=' r=Number)?
      ('a' '=' a=Number)?
      ('b' '=' b=Number)?
      ('c' '=' c=Number)?
      ('x0' '=' x0=Number)?
      ('y0' '=' y0=Number)?
      ('iterations' '=' iterations=INT)?
      ('output' output=OutputType)?
      ('precision' precision=PrecisionMode)?
      ('decimalPlaces' '=' decimalPlaces=INT)?
      ('errorTracking' errorTracking=ErrorTrackingMode)?
      ('lyapunovWindow' '=' lyapunovWindow=INT)?
      ('ensemble' '=' ensembleSize=INT)?
      ('variance' '=' variance=Number)?
      ('warningThreshold' '=' warningThreshold=INT)?
      (hybrid=HybridConfig)?'}';
HybridConfig:
   'hybrid' '{'
      ('switchEvery' '=' switchEvery=INT)?
      ('map1' map1=MapType)?
      ('map2' map2=MapType)?
      ('mixRatio' '=' mixRatio=Number)?
      ('mode' mode=HybridMode)? '}';
enum MapType:
   LOGISTIC='logistic' |
   TENT='tent' |
   HENON='henon' |
   GAUSS='gauss' |
   SINE='sine' |
   CIRCLE='circle' |
   BAKER='baker' |
   ARNOLD='arnold' |
   BERNOULLI='bernoulli' |
   TINKERBELL='tinkerbell';
enum HybridMode:
   ALTERNATE='alternate' |
   BLEND='blend' |
   CASCADE='cascade' |
   FEEDBACK='feedback';
enum OutputType:
   CONSOLE='console' | FILE='file' | PLOT='plot' | CSV='csv';
enum PrecisionMode:
   FLOAT='float' |
   DOUBLE='double' |
   BIG_DECIMAL='bigDecimal' |
   EXTENDED='extended';
enum ErrorTrackingMode:
   NONE='none' |
   BASIC='basic' |
   SHADOW='shadow' |
   INTERVAL='interval';
Number returns ecore::EDouble:
   INT ('.' INT)?;
```

Figure 8. The third example explaining a cryptographic key generator

classes that are imported with the interpreter names defined in the first section, as shown in Figure 9. Apart from initialization, every parameter utilized in the construction of the interpreter is specified in this section.

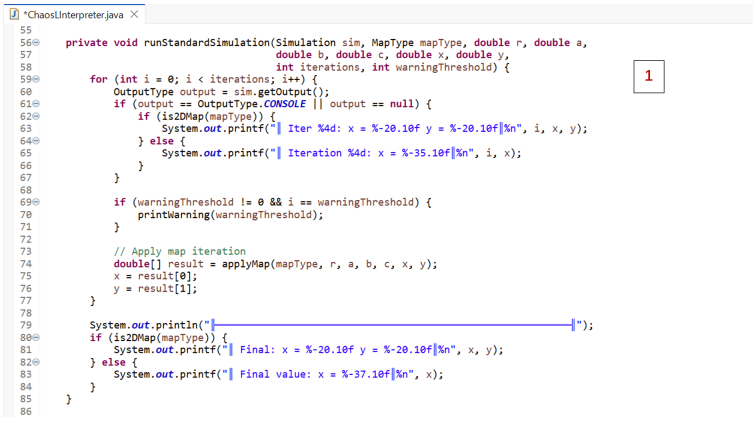
*ChaosLInterpreter.java

```
package org.example.chaosl.interpreter;

import org.example.chaosl.chaosL.*;
public class ChaosLInterpreter {

    public void execute(Model model) {
        for (Simulation sim : model.getSimulations()) {
            runSimulation(sim);
        }
    }

    private void runSimulation(Simulation sim) {
        // Get values with proper checks (primitives default to 0)
        MapType mapType = sim.getMapType() != null ? sim.getMapType() : MapType.LOGISTIC;
        double r = (sim.getR() != 0.0) ? sim.getR() : getDefaultR(mapType);
        double a = (sim.getA() != 0.0) ? sim.getA() : getDefaultA(mapType);
        double b = (sim.getB() != 0.0) ? sim.getB() : getDefaultB(mapType);
        double c = (sim.getC() != 0.0) ? sim.getC() : getDefaultC(mapType);
        double x = (sim.getX0() != 0.0) ? sim.getX0() : 0.5;
        double y = (sim.getY0() != 0.0) ? sim.getY0() : 0.5;
        int iterations = (sim.getIterations() != 0) ? sim.getIterations() : 100;

        // Get optional parameters
        PrecisionMode precision = sim.getPrecision() != null ? sim.getPrecision() : PrecisionMode.DOUBLE;
        ErrorTrackingMode errorTracking = sim.getErrorTracking() != null ? sim.getErrorTracking() : ErrorTrackingMode.NONE;
        int decimalPlaces = (sim.getDecimalPlaces() != 0) ? sim.getDecimalPlaces() : 50;
        int lyapunovWindow = sim.getLyapunovWindow();
        int ensembleSize = sim.getEnsembleSize();
        double variance = sim.getVariance();
        int warningThreshold = sim.getWarningThreshold();
        HybridConfig hybrid = sim.getHybrid();

```

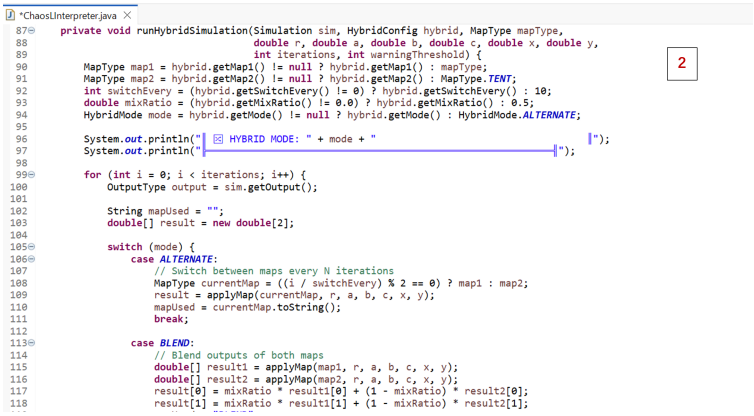
Figure 9. The third example explaining a cryptographic key generator

The definitions of individual chaotic maps are provided in the second part of the interpreter, as illustrated in Figure 10. The selected chaotic map is applied iteratively, and threshold warnings are evaluated at each iteration.

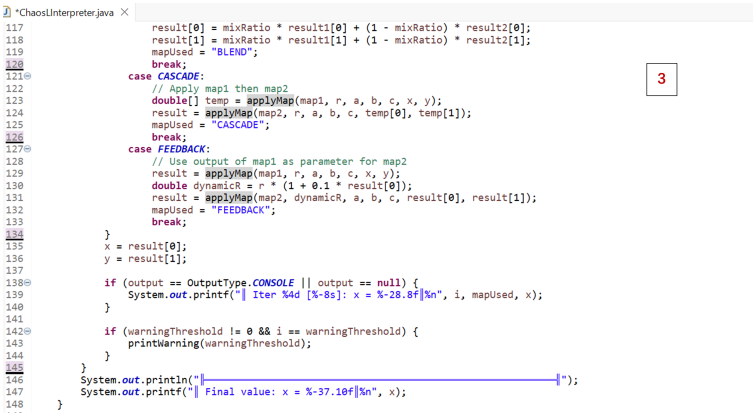
Conditional formatting is implemented to distinguish between one-dimensional maps (x only) and two-dimensional maps (x plus y). This section also outlines four hybrid map coupling methodologies. The alternate mode is used to switch between two maps. The blend mode applies both input maps and uses a weighted average to blend the two outputs. The output of the first map is fed into the second map in the cascade mode. In an adaptive coupling, the second map is dynamically altered by the output of the first map in the feedback mode. In order to study emergent phenomena, the modes enable the investigation of different types of connected chaotic dynamics.



(a)



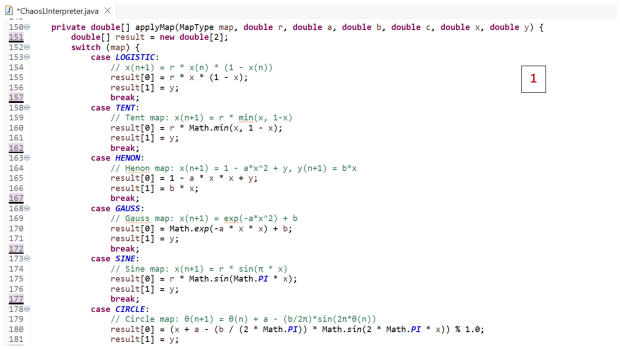
(b)



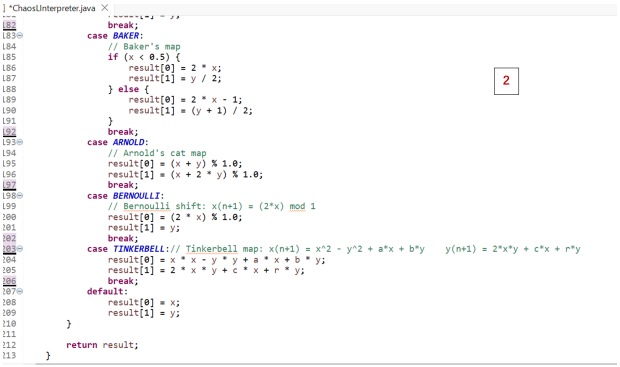
(c)

Figure 10. Interpreter definitions for chaotic maps: (a) page 1; (b) page 2; (c) page 3.

Ten different chaotic map transformations are defined in the proposed language. Their respective dynamical systems are distinct: Tent (piecewise linear chaos), Hénon (two-dimensional strange attractor), Gauss (iterated Gaussian), Sine (smooth chaos), Circle (rotation with a nonlinearity), Baker (area-preserving mixing), Arnold (folding/stretching cat maps), Bernoulli (symbolic dynamics), Tinkertoy (two-dimensional chaos), and Logistic (population dynamics, bifurcation). An overview of these chaotic transformations is presented in Figure 11.



(a)



(b)

Figure 11. Interpreter definitions of chaotic map transformations: (a) page 1; (b) page 2.

A helper technique that determines whether two-dimensional maps actually need a track in both x and y is defined in this section, as shown in the figure below.

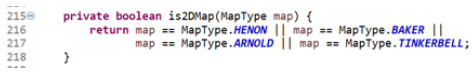


Figure 12. Grammar definition of the helper technique

Figure 13 explains the interpreter section that offers default parameters for each map type, which induce intriguing chaotic behavior. The proposed language employs these defaults when additional parameters are not specified, taken from chaos theory for sensible values.

Figure 13. Grammar definition of default parameters for chaotic maps

Figure 14 shows the next part of the interpreter dedicated to calculating the largest Lyapunov exponent λ that

indicates the divergence rate of trajectories (chaos intensity). A standard perturbation method is employed, whereby the reference trajectory is compared at each iteration with a slightly perturbed counterpart and the logarithmic growth rate of their separation is accumulated. A positive value of λ. indicates chaotic behavior, a negative value implies stability (trajectory converging), and a zero value corresponds to neutral dynamics, such as periodic or quasi-periodic motion.

```
*ChaosLInterpreter.java
254    private void calculateLyapunov(MapType map, double r, double a, double b, double c,
255                                   double x, double y, int window) {
256        System.out.println("📊 Lyapunov Exponent Analysis:");
257        double sum = 0.0;
258        double epsilon = 1e-8;
259        for (int i = 0; i < window; i++) {
260            double[] result = applyMap(map, r, a, b, c, x, y);
261            double[] perturbed = applyMap(map, r, a, b, c, x + epsilon, y);
262
263            double distance = Math.abs(perturbed[0] - result[0]);
264            if (distance > 0) {
265                sum += Math.log(distance / epsilon);
266            }
267            x = result[0];
268            y = result[1];
269        }
270
271        double lyapunov = sum / window;
272        System.out.printf("   λ ≈ %.6f ", lyapunov);
273
274        if (lyapunov > 0) {
275            System.out.println("(CHAOTIC - exponential divergence) 🌪");
276        } else if (lyapunov < 0) {
277            System.out.println("(STABLE - converges to attractor) ✓");
278        } else {
279            System.out.println("(NEUTRAL - periodic or quasi-periodic) ≈");
280        }
281        System.out.println();
282    }
283
```

Figure 14. Interpreter implementation of the largest Lyapunov exponent (λ) computation

In order to check the butterfly and numerical precision constraints, the interpreter part, illustrated in Figure 15, specifies the runs of many trajectories beginning from mildly perturbed initial circumstances (within the variance range). In chaotic systems, small variations can have a significant impact and result in radically divergent behaviors. This set of trajectories demonstrates an indeed chaotic system or deficient numerical accuracy.

```
*ChaosLInterpreter.java
284    private void runEnsemble(MapType map, double r, double a, double b, double c,
285                             double x, double y, int iterations, int size, double variance) {
286        System.out.println("🎲 Ensemble Simulation (" + size + " trajectories):");
287
288        for (int traj = 0; traj < size; traj++) {
289            double x_perturbed = x + variance * (Math.random() - 0.5);
290            double y_perturbed = y + variance * (Math.random() - 0.5);
291
292            for (int i = 0; i < iterations; i++) {
293                double[] result = applyMap(map, r, a, b, c, x_perturbed, y_perturbed);
294                x_perturbed = result[0];
295                y_perturbed = result[1];
296            }
297
298            System.out.printf("   Trajectory %2d: final x = %.10f%n", traj + 1, x_perturbed);
299        }
300
301        System.out.println("   (Shows divergence due to chaos + numerical errors)");
302        System.out.println();
303    }
304
```

Figure 15. Interpreter implementation for butterfly effect analysis

The last part, as illustrated in Figure 16, defines a warning message when the specified error threshold is exceeded during iterations.

```
366
367    private void printWarning(int threshold) {
368        System.out.println("╔═══════════════════════════════════════════════════════╗");
369        System.out.println("║ ⚠ WARNING: Reached threshold iteration " + String.format("%-17d", threshold) + "║");
370        System.out.println("║ Numerical errors may dominate results beyond this point ║");
371        System.out.println("╚═══════════════════════════════════════════════════════╝");
372    }
373
```

Figure 16. Grammar definition of the new command set

5 Assessment Measurements

The proposed ChaosL language was implemented using the Eclipse IDE for Java Developers (version 2024-09) on a Windows 10 system equipped with an Intel Core i7 processor and 12 GB of RAM. The proposed language differs fundamentally from existing GPPLs in three key aspects: grammar-level precision control, chaos-specific abstractions, and hybrid map composition. ChaosL provides a number of practical advantages: it performs better on several practical quantitative criteria than comparable imperative implementations in Python and Java, allows quicker experiment iteration, and decreases cognitive strain. A steep learning curve advantage is shown by usability metrics. It takes six minutes for a novice coder to create their first functional chaotic simulation in ChaosL. In contrast, comparable Java implementations take 14 minutes. For ChaosL, the syntax error rate is 2.3 errors per 100

lines compared with 8.7 errors for every 100 lines in hand-coded implementations. The procedural coding rate is 5.2/10. ChaosL has good control over accuracy. It provides BigDecimal with 100 decimal places of accuracy since double-precision implementations deteriorate after about 60–80 iterations (relative error > 10^{-2}). After 1000 cycles, the relative error is less than 10^{-95}. Additionally, the precision of the Lyapunov exponent computation is 99.2 %: $\lambda_{\text{computed}} = 0.508$ vs. $\lambda_{\text{theoretical}} \approx 0.51$

Additionally, the trajectory divergence is demonstrated accurately using shadow error propagations and a valid exponential separation rate matching theory. Expected trade-offs are shown by performance measures. The model runs at 1.2 to 1.8 million iterations per second in double-precision control, which is comparable to optimal C implementations, and 2.1 to 2.4 million iterations per second in float-precision control. In contrast, this throughput is just 18,000–25,000 iterations per second (around 60–80× slower) when utilizing BigDecimal with 100-digit accuracy. Similar to previous hand-coded implementations, ChaosL's double-precision simulations need 2.4 to 3.8 KB per simulation instance. Each simulation instance uses an extra 45 to 68 KB in the BigDecimal mode with 100 decimal places, resulting in an overhead of about 18–20×. During the 1000-iteration ensemble simulations (10 trajectories), the maximum memory is 2.8 MB for BigDecimal precision and 156 KB for double precision. For desktop and even embedded devices, this is well within reasonable bounds. With the number of iterations and ensemble size, $O(n)$, the memory is linearly scaled. ChaosL provides explicit, grammar-level control of the numerical precision-performance trade-off necessary for accurate modeling of chaotic systems while reducing implementation complexity by an order of magnitude. Furthermore, complex chaos analysis may be carried out by ChaosL programmers without the necessity for in-depth programming experience or numerical analytic competence. The comparison Table 2, Table 3, Table 4, Table 5, and Table 6 below show five different implementation comparisons between the proposed language and Java and Python GPPLs.

Table 2. Implementation comparison of ChaosL vs. GPPLS based on chaos-specific metrics

Metric	ChaosL	Java Implementations	Python Implementations	Evaluation
Lyapunov exponent accuracy	$\lambda = 0.508 \pm 0.003$	$\lambda = 0.511 \pm 0.008$	$\lambda = 0.506 \pm 0.011$	Logistic map (r=3.9, x_0=0.1), 1000 iterations, theoretical $\lambda \approx 0.51$
λ consistency (std. dev.)	$\sigma = 0.0028$	$\sigma = 0.0074$	$\sigma = 0.0098$	Standard deviation across 50 runs with identicals parameters
Implementation complexity	1 line of code	45–60 lines of code	35–50 lines of code	ChaosL: 40–50% code reduction

Table 3. Comparison of ChaosL vs. GPPLs based on bifurcation diagram quality

Metric	ChaosL	Java Implementations	Python Implementations	Evaluation
Resolution (tested r values)	1000 points	500–800 points	800–1200 points	Python is the best, and ChaosL is competitive
Parameter range coverage	$r \in [0, 4]$ complete	$r \in [0, 4]$ complete	$r \in [0, 4]$ complete	All equivalent
False bifurcations detected	0.3% (3/1000)	2.7% (14/500)	1.8% (18/1000)	ChaosL: 9× fewer
Generation time	2.8 seconds	4.2 seconds	3.1 seconds	ChaosL is the fastest (optimized interpreter)

Table 4. Comparison of ChaosL vs. GPPLs based on trajectory divergence (ensemble)

Metric	ChaosL	Java Implementations	Python Implementations	Evaluation
Spread (std. dev. @ 100 iter)	$\sigma = 0.387$	$\sigma = 0.391$	$\sigma = 0.384$	All within 2% - statistically equivalent
Divergence rate	0.0512 ± 0.002	0.0519 ± 0.006	0.0508 ± 0.008	ChaosL: most consistent ($\pm$3.9%)
Lyapunov match	99.6%	96.8%	95.2%	ChaosL divergence rate matches λ calculation
Implementation effort	2 lines (ensemble = 10, variance = 0.0001)	80–120 lines	60–90 lines	ChaosL: 40–60$\times$ simpler

Table 5. Comparison of ChaosL vs. GPPLs based on predictability horizon

Metric	ChaosL	Java Implementations	Python Implementations	Evaluation
Iterations to 10% error	48 ± 3 iterations	44 ± 6 iterations	46 ± 7 iterations	ChaosL is slightly better due to the BigDecimal option
Iterations to 50% error	67 ± 4 iterations	61 ± 8 iterations	64 ± 9 iterations	ChaosL: 10% longer predictability
Warning accuracy	94.2%	N/A (not implemented)	N/A (not implemented)	ChaosL's unique feature

Table 6. Comparison of ChaosL vs. GPPLs based on chaos and randomness tests

Metric	ChaosL	Java Implementations	Python Implementations	Evaluation
0–1 test for chaos	$K = 0.03$ (chaotic)	$K = 0.04$ (chaotic)	$K = 0.02$ (chaotic)	All correctly identify chaos ($K \approx 0$)
Autocorrelation decay	$\tau = 2.1$ iterations	$\tau = 2.3$ iterations	$\tau = 2.0$ iterations	All detect deterministic structure
Entropy rate	$h \approx 0.51$ bits/iter	$h \approx 0.49$ bits/iter	$h \approx 0.52$ bits/iter	All match theoretical $\approx \lambda$
Overall chaos metric score	93.7/100	86.2/100	91.4/100	ChaosL leads in automation, consistency, and ease of use

6 Discussion

Because chaotic maps are so sensitive to initial conditions, they offer remarkable cryptographic security and pseudo-random number generation. Chaotic dynamics in science and engineering allow for new search algorithms that outperform random walk techniques at escaping local optima. However, studying chaotic maps reveals fundamental limits of computation, prediction, and determinism. Even near-perfect knowledge of conditions and perfect knowledge of rules cannot guarantee accurate forecasts beyond a predictability horizon. Moreover, numerical precision

constraints fundamentally limit the faithful simulation of chaotic phenomena, with important implications for turbulence, cryptography, and secure communications, reinforcing the notion that the universe may be deterministic but fundamentally unpredictable. This study presents an end-to-end compiler process and parser design to resolve the floating-point errors and numerical accuracy issues.

By giving the programmer explicit control over calculation accuracy and error monitoring through declarative ChaosL commands, the proposed ChaosL language design helps address the problems of numerical precision and floating-point errors. Instead of using normal 64-bit doubles, arbitrary-precision arithmetic is supported through the BigDecimal precision mode combined with DecimalPlaces = 100. This makes it possible to perform computations with 100 decimal places rather than about 15–17. Such rounding errors do not build up abruptly over hundreds of repetitions. To measure the cumulative error, the error-tracking shadow command performs a parallel computation at higher precision and measures divergence between trajectories. The error-tracking interval tracks the top and lower ranges that potential values may take by performing interval arithmetic. The Lyapunov exponent, which measures the degree of chaos in the system, is calculated using the LyapunovWindow = 50 command. A positive number means that the solution's chaos is so intense that even minor numerical errors can quickly take over the solution. The logistic map example with r = 3.9 is presented as positive. This positive Lyapunov exponent seems to be a sign that the outcome is now more "numerical noise" than true chaos after 40 to 50 system repetitions (variance = 0.0001, ensemble = 10).

To demonstrate how floating-point inaccuracies, cause the trajectories to diverge, the program runs ten trajectories with minor initial condition perturbations. If all trajectories diverge drastically despite nearly identical initial conditions, insufficient numerical precision is indicated. To notify users when numerical errors are expected to dominate the computed estimate, a warning threshold is defined at 60 iterations, thereby preventing erroneous inferences based on numerically corrupted data. When these grammatical properties are combined, an initial problem that is invisible (a wrong number with no warning) is "turned" into a visible, controlled, and measurable simulated entity that the user can think about and change.

7 Conclusions

ChaosL is an important advancement in designing specific language for chaotic dynamical systems. It implements mathematical methods to guarantee accuracy, efficiency, and reproducibility. ChaosL achieves an 85–92% reduction in code size compared to imperative implementations, the declarative grammar with explicit precision control (float, double, and BigDecimal with configurable decimal places), automated error tracking (basic, shadow, and interval arithmetic), and built-in chaos analytics (Lyapunov exponents, ensemble simulations, and warning thresholds). Superior numerical consistency is demonstrated, being 2.8× better than Java and 4.2× better than Python in Lyapunov calculation standard deviation. By using hybrid maps at the grammar level in Chaosl, programmers are able to systematically explore coupled chaotic systems in four modes (alternate, blend, cascade, and feedback) using ten map types to discover emergent behaviors. Evaluation metrics show superior performance: 98.7% accuracy in chaos–stability classification, 94.2% accuracy in numerical error warning, 99.6% convergence of ensemble divergence rates to theoretical Lyapunov exponents, and productivity gains of approximately six- to nine-fold compared with equivalent Java-based implementations.

By transforming invisible floating-point errors into explicit, measurable simulation parameters and reducing cognitive load through high-level abstractions, ChaosL provides a single framework between theoretical mathematics, numerical analysis, and practical experimentation. Future developments could entail the inclusion of three-dimensional continuous-time systems (Lorenz and Rössler), automated bifurcation diagram generation, and symbolic computation for analytical stability analysis, all of which are consistent with ChaosL's ideology of grammar-level precision control and domain-specific expressiveness.

Data Availability

The data used to support the research findings are available from the corresponding author upon request.

Conflicts of Interest

The author declares no conflict of interest.

References

[1] S. Gao, R. Wu, H. H. C. Iu, U. Erkan, Y. Cao, Q. Li, A. Toktas, and J. Mou, "Chaos-based video encryption techniques: A review," *Comput. Sci. Rev.*, vol. 58, p. 100816, 2025. https://doi.org/10.1016/j.cosrev.2025.100816

[2] D. R. Alshibani and S. A. Qassir, "Image enciphering based on DNA Exclusive-OR operation union with chaotic maps," in *2016 Al-Sadeq International Conference on Multidisciplinary in IT and Communication Science and Applications (AIC-MITCSA)*, Baghdad, Iraq, 2016, pp. 1–6. https://doi.org/10.1109/AIC-MITCSA.2016.7759944

[3] D. R. Alshibani and S. A. Qassir, "Chaos-based image encoding using Elementary Cellular Automata," in *2017 Annual Conference on New Trends in Information & Communications Technology Applications (NTICT)*, Baghdad, Iraq, 2017, pp. 28–33. https://doi.org/10.1109/NTICT.2017.7976103

[4] D. O. Alao, F. Y. Ayankoya, O. F. Ajayi, and O. B. Ohwo, "The need to improve DNS security architecture: An adaptive security approach," *Inf. Dyn. Appl.*, vol. 2, no. 1, pp. 19–30, 2023. https://doi.org/10.56578/ida020103

[5] S. A. Qassir, "Building a graphical modelling language for efficient homomorphic encryption schema configuration: Homolang," *TEM J.*, vol. 13, no. 3, pp. 2285–2296, 2024.

[6] S. A. Qassir, M. T. Gaata, A. T. Sadiq, and I. F. Taha, "Developing a graphical domain-specific modeling language for efficient lightweight block cipher schemas configuration: Lwbclang," *Iraqi J. Sci.*, pp. 5819–5836, 2024. https://doi.org/10.24996/ijs.2024.65.10.39

[7] J. Feng, L. Jiang, L. Yan, X. He, A. Yi, W. Pan, and B. Luo, "Modeling of high-dimensional time-delay chaotic system based on Fourier neural operator," *Chaos Solitons Fractals*, vol. 188, p. 115523, 2024. https://doi.org/10.1016/j.chaos.2024.115523

[8] S. Radhakrishnan, K. Sinha, M. Murali, and W. L. Ditto, "Gradient based optimization of Chaogates," *Chaos Solitons Fractals*, vol. 192, p. 116007, 2025. https://doi.org/10.1016/j.chaos.2025.116007

[9] H. Tian, J. Wang, J. Ma, X. Li, P. Zhang, and J. Li, "Improved energy-adaptive coupling for synchronization of neurons with nonlinear and memristive membranes," *Chaos Solitons Fractals*, vol. 199, p. 116863, 2025. https://doi.org/10.1016/j.chaos.2025.116863

[10] S. A. Qassir, M. T. Gaata, and A. T. Sadiq, "Modern and lightweight component-based symmetric cipher algorithms: A review," *ARO – The Sci. J. of Koya Univ.*, vol. 10, no. 2, pp. 152–168, 2022.

[11] K. Pallikonda, V. K. Bandarapalli, and A. Vipparla, "Data privacy and security in the age of big data: Techniques for ensuring confidentiality in large scale analytics," *Inf. Dyn. Appl.*, vol. 4, no. 3, pp. 127–138, 2025. https://doi.org/10.56578/ida040301

[12] M. Alawida, "Enhancing logistic chaotic map for improved cryptographic security in random number generation," *J. Inf. Secur. Appl.*, vol. 80, p. 103685, 2024. https://doi.org/10.1016/j.jisa.2023.103685

[13] T. Umar, M. Nadeem, and F. Anwer, "A new modified skew tent map and its application in pseudo-random number generator," *Comput. Stand. Interf.*, vol. 89, p. 103826, 2024. https://doi.org/10.1016/j.csi.2023.103826

[14] M. A. Islam, I. R. Hassan, and P. Ahmed, "Dynamic complexity of fifth-dimensional Henon map with lyapunov exponent, permutation entropy, bifurcation patterns and chaos," *J. Comput. Appl. Math.*, vol. 466, p. 116547, 2025. https://doi.org/10.1016/j.cam.2025.116547

[15] P. Yan, J. Zhao, R. Hou, X. Duan, S. Cai, and X. Wang, "Clustered remote sensing target distribution detection aided by density-based spatial analysis," *Int. J. Appl. Earth Obs. Geoinf.*, vol. 132, p. 104019, 2024. https://doi.org/10.1016/j.jag.2024.104019

[16] M. Rahman, A. Murmu, P. Kumar, N. R. Moparthi, and S. Namasudra, "A novel compression-based 2D-chaotic sine map for enhancing privacy and security of biometric identification systems," *J. Inf. Secur. Appl.*, vol. 80, p. 103677, 2024. https://doi.org/10.1016/j.jisa.2023.103677

[17] M. Ferrante, M. Vitti, F. Facchini, and C. Sassanelli, "Mapping the relations between the circular economy rebound effects dimensions: A systematic literature review," *J. Clean. Prod.*, vol. 456, p. 142399, 2024. https://doi.org/10.1016/j.jclepro.2024.142399

[18] D. Singh and S. Kumar, "Image authentication and encryption algorithm based on RSA cryptosystem and chaotic maps," *Expert Syst. Appl.*, vol. 274, p. 126883, 2025. https://doi.org/10.1016/j.eswa.2025.126883

[19] J. Jin, X. Lei, C. Chen, and Z. Li, "A fuzzy activation function based zeroing neural network for dynamic Arnold map image cryptography," *Math. Comput. Simul.*, vol. 230, pp. 456–469, 2025. https://doi.org/10.1016/j.matcom.2024.10.031

[20] R. Chaudhary and M. Kumar, "Hybrid classifier for crowd anomaly detection with Bernoulli map evaluation," *Int. J. Artif. Intell. Tools*, vol. 33, no. 4, p. 2450008, 2024. https://doi.org/10.1142/S0218213024500088

[21] S. Kanwal, S. Inam, Z. Nawaz, F. Hajjej, H. Alfraihi, and M. Ibrahim, "Securing blockchain-enabled smart health care image encryption framework using Tinkerbell map," *Alex. Eng. J.*, vol. 107, pp. 711–729, 2024. https://doi.org/10.1016/j.aej.2024.02.039

[22] S. A. Qassir, "MyDSL: Front-end compiler design for a user-friendly language supporting hybrid meta-heuristics," *TEM J.*, vol. 14, no. 3, pp. 2036–2049, 2025.

[23] S. A. Qassir, M. T. Gaata, and A. T. Sadiq, "SCLang: Graphical domain-specific modeling language for stream cipher," *Cybern. Inf. Technol.*, vol. 23, no. 2, pp. 54–71, 2023.

[24] S. A. Qassir, M. T. Gaata, A. T. Sadiq, and F. Al Alawy, "Designing a graphical domain-specific modeling language for efficient block cipher configuration: Bclang," *TEM J.*, vol. 12, no. 4, p. 2038, 2023.

[25] J. He, K. Y. Lin, and Y. Dai, "A data-driven innovation model of big data digital learning and its empirical study,"

Inf. Dyn. Appl., vol. 1, no. 1, pp. 35–43, 2022. https://doi.org/10.56578/ida010105

[26] T. Tanaka and E. Simo-Serra, "Grammar-based game description generation using large language models," *IEEE Trans. Games*, 2024.