BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF MATHEMATICS

EXPLAINABLE ARTIFICIAL INTELLIGENCE APPROACHES FOR SOLVING STOCHASTIC DIFFERENTIAL EQUATIONS

DIPLOMA THESIS

NAME SURNAME SHAHMIR KHAN

SUPERVISOR Assoc. Prof. JAKUB KŮDELA, Ph.D.

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Assignment Master's Thesis

Institut:	Institute of Mathematics
Student:	**Shahmir Khan**
Degree programm:	Applied and Interdisciplinary Mathematics
Branch:	no specialisation
Supervisor:	**doc. Ing. Jakub Kůdela, Ph.D.**
Academic year:	2024/25

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

Explainable artificial intelligence approaches for solving stochastic differential equations

Brief Description:

Explainable Artificial Intelligence (XAI) currently attracts great interest in the research community, motivated by the need for explanations in critical AI applications. Some recent advances in XAI are based on Evolutionary Computation techniques, such as Genetic Programming and Grammatical Evolution. The thesis will aim to investigate the utility of these XAI approaches for finding symbolic solutions to stochastic differential equations. The student will combine existing Python frameworks for Automatic Differentiation, Genetic Programming, and Grammatical Evolution to find classes of stochastic differential equations, for which these approaches give satisfactory results.

Master's Thesis goals:

Review of the theoretical background on stochastic differential equations, automatic differentiation, and numerical methods for stochastic differential equations.
Review of the Explainable Artificial Intelligence approaches, primarily focusing on Genetic Programming and Grammatical Evolution.
Implementation of existing Python frameworks for Automatic Differentiation, Genetic Programming, and Grammatical Evolution for finding symbolic solutions to stochastic differential equations.
Investigation of the utility of the implemented approaches for different classes of stochastic differential equations.

Recommended bibliography:

LOBAO, W. J. D.A., CAVALCANTI PACHECO, M. A., Dias, D. M. and ABREU, A. C. A. Solving stochastic differential equations through genetic programming and automatic differentiation. Engineering Applications of Artificial Intelligence 68 (2018): 110-120.

KLOEDEN, P. E. and E. PLATEN. Numerical Solution of Stochastic Differential Equations. Springer Berlin, Heidelberg, 1992.

LANGDON, W. B. and R. POLI. Foundations of genetic programming. Springer Science & Business Media, 2013.

RYAN, C., O'NEILL, M. and J. J. COLLINS, eds. Handbook of grammatical evolution. Vol. 1. Cham (Switzerland): Springer International Publishing, 2018.

BACARDIT, J., A. EI BROWNLEE, S. CAGNONI, G. IACCA, J. MCCALL and D. WALKER. The intersection of evolutionary computation and explainable AI. In Proceedings of the Genetic and Evolutionary Computation conference companion, pp. 1757-1762. 2022.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2024/25

In Brno,

L. S.

doc. Mgr. Petr Vašík, Ph.D. Director of the Institute	doc. Ing. Jiří Hlinka, Ph.D. FME dean

Abstract

This thesis presents a symbolic regression approach for solving stochastic differential equations (SDEs) using Grammatical Evolution (GE). The goal is to generate interpretable expressions that approximate the solution trajectories of these equations. Unlike traditional methods such as the Euler–Maruyama scheme, which only provide numerical approximations, the proposed approach aims to produce closed-form symbolic representations of complex stochastic models. Building on the motivation for explainable methods, we first review the GPAD (Genetic Programming and Automatic Differentiation) framework and its use of symbolic modeling for SDEs and their solutions. However, due to practical challenges in implementing this approach, we developed an alternative approach by expanding the Python-based PonyGE2 framework. The Proposed method learns symbolic expressions from simulated datasets composed of time and Wiener process values as inputs and the corresponding process values as outputs. We apply this framework to several well-known SDEs, including the Geometric Brownian Motion, Ornstein–Uhlenbeck process, and Cox–Ingersoll–Ross model. Both known and unknown realizations of the Wiener process are considered to assess the accuracy and generalization of the GE approach. To evaluate the performance of the symbolic models, we generate multiple trajectories for each process and compute the mean and variance over time. This statistical comparison helps us to determine how accurately the symbolic models represent the underlying dynamics of the original system. The results demonstrate that GE can recover concise and meaningful symbolic representations that reflect the underlying dynamics of the stochastic systems. This validation confirms that the symbolic models not only fit individual paths but also preserve the statistical properties of the processes. By integrating explainability into the modeling of SDEs, this work offers a new direction for transparent and flexible modeling in stochastic analysis.

Keywords

Explainable Artificial Intelligence, Symbolic Regression, Grammatical Evolution, Genetic Programming, Evolutionary Algorithms, Stochastic Calculus, Stochastic Differential Equations.

KHAN, Shahmir.: *Explainable Artificial Intelligence approaches for solving Stochastic Differential Equations*, Brno University of Technology, Faculty of Mechanical Engineering, 2025. Supervisor: Prof. Jakub KŮDELA.

I declare that I wrote the diploma thesis *Explainable Artificial Intelligence approaches for solving Stochastic Differential Equations* independently under the guidance of *Prof. Jakub KŮDELA* using the literature included in the list of references.

Shahmir Khan

I would like to sincerely thank Prof. Jakub KŮDELA for his invaluable guidance, continuous support, and generous investment of time throughout this journey. His insightful advice and encouraging mentorship have been instrumental in my work, and I am deeply grateful for his kindness and dedication.

Shahmir Khan

Contents

1 Introduction

Stochastic Differential Equations (SDEs) [18] are mathematical models used to describe systems influenced by random processes. Many real-world systems exhibit uncertainty and randomness that cannot be captured by classical deterministic models, whether in financial markets, physical systems, or biological phenomena. The foundations of this modeling approach were laid by Itô [11], who introduced stochastic calculus, and later expanded by Arnold [3], Schuss [21], and Ikeda and Watanabe [10], who formalized SDE theory and applications in diffusion processes. Numerical methods, such as those developed by Kloeden and Platen [12], further enabled practical approximation of solutions when closed-form expressions are not available. Despite these breakthroughs, exact analytical solutions remain limited to specific SDE classes, motivating the need for flexible, data-driven approaches capable of producing interpretable representations of stochastic systems.

Recently, data-driven approaches have become effective tools for approximating complicated mathematical functions, such as differential equation solutions. Genetic Programming (GP) [15] and Grammatical Evolution (GE) [20] are two examples of symbolic regression techniques that captured interest because they can generate expressions that are readable by humans. In addition to fitting data, these techniques also show interpretable mathematical relationships, which makes them optimal for cases where understanding a solution's structure is just as crucial as the solution itself.

However, the use of evolutionary algorithms in stochastic settings comes with unique challenges. Uncertain patterns may result from the presence of randomness in the data, which might lead to overfitting or unstable approximations. Therefore, integrating stochastic calculus into this modeling framework is essential to effectively apply symbolic methods to solve SDEs.

In this thesis, we will explore Explainable Artificial Intelligence (XAI) [4] methods to solve SDEs, with a primary focus on symbolic regression through the use of GP and GE. We will start by reviewing the GPAD (Genetic Programming and Automatic Differentiation) framework by Lobão et al. [16], which initially proposed a method to find symbolic solutions using Genetic Programming and then suggested a symbolic technique to estimate the drift and diffusion terms to model SDEs. However, we encountered many challenges with the implementation of this idea. The MATLAB codes were not initially accessible, and even when they were eventually made available, they were not fully functional. Furthermore, the whole analysis of the paper is based on a single straightforward instance of one-dimensional geometric Brownian motion (GBM).

To overcome these limitations, we looked towards the developments in GE, which has gained attention for its ability to model complex, nonlinear systems in noisy and evolving environments. For instance, in the paper [23], GE is used to evolve neural networks capable of capturing nonlinear patterns. In [1] GE is applied to dynamic financial forecasting, demonstrating its robustness in stochastic domains. Similarly, [2] introduced a good strategy to simplify evolved expressions, improving their generalization to real-world data. These studies show that GE is not only interpretable but also well-suited for handling uncertainty, which makes it a promising candidate for modeling SDEs. Building on this, we created a new framework using the PonyGE2 framework, which is built on Python and provides better flexibility and support for a wider range of SDEs.

The proposed method uses GE to learn symbolic representations of the solutions to SDEs directly from simulated data. The approach is designed to work with both known and unknown realizations of the Wiener process, offering flexibility in how the input data is structured. The primary goal is to identify symbolic expressions that accurately reflect the full trajectory of the stochastic process. To achieve this,

we simulate sample paths of the process X_t using either analytical solutions (when available) or the Euler–Maruyama method for cases without closed-form solutions. Each simulation provides a dataset consisting of time values t, corresponding Wiener process values W_t, and the resulting process values X_t, which together form the input-output pairs used in symbolic regression.

In the initial experiments, we considered two cases. In the first case, a fixed realization of the Wiener process was used both for generating the trajectory and for finding the symbolic solution. This allowed us to test whether GE could reconstruct the underlying structure of the solution when the randomness remained the same. In the second case, we introduced multiple different realizations of the Wiener processes during testing to explore how well the symbolic solution could replicate the trajectory with unknown Wiener inputs. These two cases, using both known and unknown noise, help assess the performance of the symbolic expressions produced by GE. In this work, we will also explore the impact of the drift and diffusion term parameters on the accuracy of the model.

To further confirm the reliability of the symbolic models, for every SDE, we simulate a large number of sample paths to do a statistical evaluation. We calculate the variance and mean over time from them and compare them with the results obtained using the symbolic approximations. This comparison serves as a crucial measure of how effectively the model captures the system's overall statistical behaviour in addition to individual trajectories.

We use this methodology on several benchmark SDEs, including Geometric Brownian Motion, the Ornstein–Uhlenbeck process, and the Cox–Ingersoll–Ross model. For each case, we examine GE's ability to identify clear symbolic solutions that accurately fit the actual behavior of the processes. Overall, these experiments demonstrate that symbolic regression with GE is an efficient tool to find understandable and interpretable symbolic solutions to complex stochastic systems.

In general, the goal of this thesis is to bridge the gap between stochastic modeling and explainable artificial intelligence by providing a reliable, flexible, and interpretable framework for approximating SDE solutions. The results demonstrate GE's potential as a transparent modeling tool in uncertain situations, giving a new direction for research that combines stochastic analysis and data-driven modeling.

2 Theoretical Foundations of SDEs

2.1 Brownian Motion

Let $(\Omega, \mathcal{F}, \mathcal{P})$ be a probability space, then Standard Wiener process or a standard Brownian motion [18] is a sequence of random variables denoted by $W(t,\omega)$ or $W(t)$ which is a continuous function of time on the interval $[0,T]$ for each $\omega \in \Omega$ and satisfies the following three conditions given below:

1. $W(0) = 0$ with probability 1.
2. For $0 \leq t_1 < t_2 \leq T$, the increment $W(t_2) - W(t_1)$ is a a random variable that follows the normal distribution with mean zero and variance $t_2 - t_1$; or $W(t_2) - W(t_1) \sim \sqrt{t_2 - t_1}N(0,1)$, where $N(0,1)$ denotes a normally distributed random variable with mean zero and variance one.
3. For $0 \leq t_1 \leq t_2 \leq t_3 \leq t_4 \leq T$, the increments given by $W(t_2) - W(t_1)$ and $W(t_4) - W(t_3)$ are independent random variables.

where for a fixed $\omega \in \Omega$, $W(t,\omega)$ is a sample path.

2.1.1 Simulation of Sample Paths of Brownian Motion

Although Brownian motion is defined in continuous time, we can only simulate it with a finite resolution due to computational limitations. To do this, we choose positive integers T and M, such that $dt = T/M$ is the time step of the simulation. The temporal mesh points given by $t_i = idt$, for $i = 0, \ldots, M$ partition the interval $[0,T]$ in the M number of sub-intervals. Let the discretized Brownian motion $W(t_i)$ be denoted by W_i, for each value of t_i. Then, from the definition of Brownian motion

$$W_i = W_{i-1} + dW_i, i = 1, \ldots, M. \tag{2.1}$$

where each $dW_i = \sqrt{dt}N(0,1)$ is a normally distributed random variable with mean zero and variance dt. Thus,

$$W_i = \sum_{k=1}^{i} dW_k. \tag{2.2}$$

The three different trajectories of Brownian motion are simulated. The **randn** function of the **random** module from the **NumPy** library generates pseudo-random numbers from the standard normal distribution. It produces different values for each call, therefore, the initial state of **randn**, a seed, is allowed to be set in Python to produce the same result each time the code is run. The fixed seeds 0, 1, and 2 are used respectively for each trajectory. Using the **randn** function, an array, N, of dimension $1 \times M$ is created that holds M standard normally distributed numbers. An array dW of the increments of Brownian motion, dW_i, is produced by taking the product of each element of N with dt. Finally, the cumulative sum given in (2.2) is computed using the **cumsum** function from the **NumPy** library. The three different Paths produced are shown in Figure 2.1. Moreover, the mean of the 1000 Brownian sample paths is also depicted in the Figure.

2.2 Stochastic Integrals

For a given suitable function $h(t)$, the integral $\int_0^T h(t)dt$ over the interval $[0,T]$ may be approximated by the Riemann sum

$$\sum_{j=0}^{N-1} h(t_j)(t_{j+1} - t_j) \tag{2.3}$$

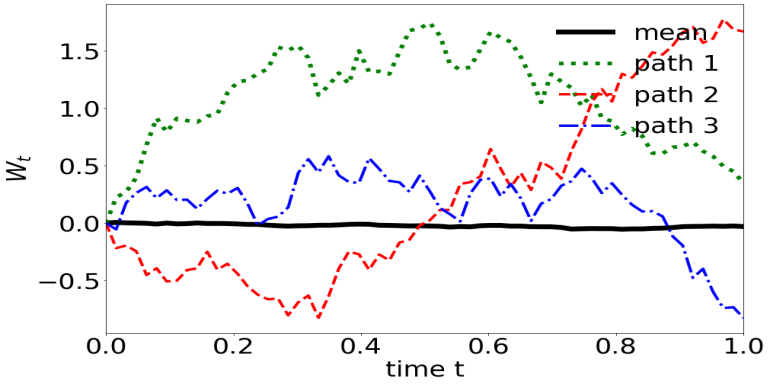


Figure 2.1: Simulation of Wiener processes $W(t)$: 3 individual sample paths and the mean of over 1000 Wiener sample paths are represented with a solid black line.

where $t_j = j\Delta t$, $\Delta t = T/N$. The integral is defined by taking the limit $\Delta t \to 0$ in (2.3). Similarly, consider the sum of the form

$$\sum_{j=0}^{N-1} h(t_j)\left(W(t_{j+1}) - W(t_j)\right), \tag{2.4}$$

which, by analogy with (2.3), may be regarded as an approximation to the stochastic integral $\int_0^T h(t)dW(t)$. Here, h is integrated with respect to Brownian motion. Moreover, alternatives to (2.3) are given by

$$\sum_{j=0}^{N-1} h(t_{j+1})(t_{j+1} - t_j) \tag{2.5}$$

and

$$\sum_{j=0}^{N-1} h\left(\frac{t_j + t_{j+1}}{2}\right)(t_{j+1} - t_j) \tag{2.6}$$

which are also a Riemann sum approximation to $\int_0^T h(t)dt$. The corresponding alternatives to (2.4) are given by

$$\sum_{j=0}^{N-1} h(t_j + 1)\left(W(t_{j+1}) - W(t_j)\right), \tag{2.7}$$

and

$$\sum_{j=0}^{N-1} h\left(\frac{t_j + t_{j+1}}{2}\right)\left(W(t_{j+1}) - W(t_j)\right) \tag{2.8}$$

So, when defining a stochastic integral as the limiting case of a Riemann sum, one must be precise about how the sum is formed. The **left-end-point** sum (2.4) gives rise to what is known as the Ito integral, the **right-end-point** sum(2.7) is known as the Backward-Ito integral, whereas the **midpoint** sum (2.8) produces the Stratonovich integral. More details on stochastic integral can be found in [12]. In the subsequent sections, the Ito version is used to define a stochastic differential equation.

2.3 Ordinary Differential Equations (ODEs)

Ordinary Differential Equations (ODEs) are an essential tool in mathematical modeling, used to explain the variation in deterministic systems over time. A first-order ODE typically takes the form:

$$\frac{dx(t)}{dt} = f(t, x(t)), \tag{2.9}$$

where $x(t)$ is the unknown function of time, and $f(t, x(t))$ is a known function which represents the rate of change.

A solution to this ODE is a function $x(t)$ which satisfies the equation on a given specific interval, usually based on an initial condition:

$$x(0) = x_0. \tag{2.10}$$

Additionally, numerical schemes such as the Euler method are useful for approximation when analytical solutions to ODEs are not available. The Euler method discretizes time into small steps of size Δt, and iteratively computes:

$$x_{n+1} = x_n + \Delta t \cdot f(t_n, x_n), \tag{2.11}$$

starting from an initial value x_0. This method gives a simple but effective way to approximate the trajectory of $x(t)$ over a certain interval.

ODEs work well for simulating deterministic systems, but fully deterministic models cannot incorporate the random effects and uncertainties that influence many real-world phenomena. Examples of stochastic behaviour include biological systems that are inherently inconsistent, financial markets which are affected by unexpected happenings, and physical systems that are subjected to thermal noise. ODEs are extended into SDEs to incorporate randomness directly into the modeling framework. This is accomplished by including a noise term, which is usually represented by Brownian motion. Both deterministic patterns and random fluctuations found in dynamic systems are therefore explained by the resulting SDEs. In the next section, we formally introduce the concept of SDEs and their mathematical formulation.

2.4 Stochastic Differential Equations

In mathematical modeling, a lot of phenomena are inherently unpredictable and random. These intrinsic oscillations are often missed by traditional deterministic differential equations, which show the precise relationship between variables. To address such uncertainty, we use SDEs, a powerful extension of classical differential equations that explicitly include randomness in their formulation. [8,18]

Definition:. Let X_t be a continuous stochastic process. If small changes in the process X_t can be written as a linear combination of small changes in t and small increments of the Brownian motion W_t, we may write

$$dX_t = a(t, X_t)dt + b(t, X_t)dW_t \tag{2.12}$$

and call it a stochastic differential equation. In fact, this differential relation has the following integral meaning:

$$X_t = X_0 + \int_0^T a(s, X_s)ds + \int_0^T b(s, X_s)dW_s \tag{2.13}$$

where the last integral is taken in the Ito sense. Relation (2.13) is taken as the definition for the stochastic differential equation (2.12) [12].

The functions $a(t, X_t)$ and $b(t, X_t)$ are called the drift rate and the volatility of the process X_t, respectively. The drift component shows the average direction or trend of the process, while the diffusion captures variability or random shocks at each instant. For example, in financial mathematics, the drift may represent expected return, while diffusion shows market volatility. Given these two functions as input, one may find the solution X_t of the stochastic differential equation as an output.

2.4.1 Existence and Uniqueness of Solutions to SDEs

The existence of a solution and its uniqueness should be confirmed before attempting to model or simulate solutions of stochastic differential equations. To guarantee that the problem is well-posed, SDEs have to stick to certain regularity conditions, just like ordinary differential equations. Let's consider the general SDE:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t, \quad X_0 = x_0.$$

We define two important conditions that are sufficient to ensure the existence and uniqueness of a solution to this equation over the interval $[0, T]$:

- **Lipschitz Condition:** There exists a constant $L > 0$ such that for all $t \in [0, T]$ and all $x, y \in \mathbb{R}$,

$$|a(t,x) - a(t,y)| + |b(t,x) - b(t,y)| \leq L|x - y|.$$

- **Linear Growth Condition:** There exists a constant $K > 0$ such that for all $t \in [0, T]$ and all $x \in \mathbb{R}$,

$$|a(t,x)|^2 + |b(t,x)|^2 \leq K(1 + |x|^2).$$

According to traditional results in stochastic calculus [12, 18], the SDE permits a unique solution if both requirements are met. Verifying these requirements helps in preventing ill-posed conditions in practical modelling, which may result in multiple or no solutions. These presumptions also influence the choice of diffusion and drift functions in the building of models, whether implemented analytically or using machine learning techniques.

2.4.2 Itô's Formula for One-Dimensional SDEs

Itô's formula [8, 18] plays a fundamental role in stochastic calculus, as it allows us to compute the differential of a function of a stochastic process, analogous to the chain rule in classical calculus, but adapted to handle stochastic terms.

Let $X(t)$ be a stochastic process satisfying the SDE:

$$dX(t) = a(t, X(t))\, dt + b(t, X(t))\, dW(t),$$

and let $\phi(t, x)$ be a function such that $\phi \in C^{1,2}(\mathbb{R} \times \mathbb{R})$, i.e., ϕ is continuously differentiable in t and twice continuously differentiable in x. Then, the process $Y(t) = \phi(t, X(t))$ satisfies:

$$\begin{aligned} dY(t) = &\left(\frac{\partial \phi}{\partial t}(t, X(t)) + \frac{\partial \phi}{\partial x}(t, X(t)) \cdot a(t, X(t)) + \frac{1}{2} \frac{\partial^2 \phi}{\partial x^2}(t, X(t)) \cdot b^2(t, X(t)) \right) dt \\ &+ \frac{\partial \phi}{\partial x}(t, X(t)) \cdot b(t, X(t))\, dW(t). \end{aligned} \tag{2.14}$$

2.5 Stochastic Differential Equations Used in Simulation Framework

This section covers many significant SDEs that are applied in a variety of disciplines, including biology, economics, and physics. Both deterministic trends and random noise can be modeled with the help of these equations. Using common mathematical tools such as Itô's Lemma, we also investigate how to solve these equations analytically where feasible. For the experimental analysis carried out in this paper, the SDEs covered here will serve as the basis.

1. Geometric Brownian Motion (GBM).
Stock prices and other quantities that fluctuate in value are frequently represented by this model. The SDE is written as:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t, \tag{2.15}$$

where the drift and diffusion terms are:

$$a(t, X_t) = \mu X_t, \quad b(t, X_t) = \sigma X_t.$$

In order to solve this, apply Itô's Lemma to $Y_t = \ln(X_t)$. Then:

$$dY_t = \left(\mu - \frac{1}{2}\sigma^2\right) dt + \sigma dW_t.$$

Integrating both sides:

$$Y_t = \ln(X_t) = \ln(X_0) + \left(\mu - \frac{1}{2}\sigma^2\right) t + \sigma W_t.$$

Taking the exponential:

$$X_t = X_0 \exp\left[\left(\mu - \frac{1}{2}\sigma^2\right) t + \sigma W_t\right]. \tag{2.16}$$

This closed-form solution is useful in financial modeling, especially in the Black–Scholes formula [18].

2. Ornstein–Uhlenbeck Process (OU).
This process models systems that revert to a mean value, such as temperature, velocity, or interest rates. The SDE is:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t, \tag{2.17}$$

with

$$a(t, X_t) = \theta(\mu - X_t), \quad b(t, X_t) = \sigma.$$

To solve it, multiply both sides by $e^{\theta t}$ (an integrating factor):

$$d(e^{\theta t} X_t) = \theta\mu e^{\theta t} dt + \sigma e^{\theta t} dW_t.$$

Now integrate:

$$X_t = X_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) + \sigma \int_0^t e^{-\theta(t-s)} dW_s. \tag{2.18}$$

This gives the exact solution of the OU process [18].

3. Cox–Ingersoll–Ross (CIR) Model.
This model is used in finance to describe interest rates or stochastic volatility. The equation is:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t, \tag{2.19}$$

with

$$a(t, X_t) = \kappa(\theta - X_t), \quad b(t, X_t) = \sigma\sqrt{X_t}.$$

This SDE does not have a simple explicit solution.

4. General Nonlinear SDEs.
In many real-world applications, the drift and diffusion may depend nonlinearly on both time and the state variable. Such equations take the general form:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t.$$

Analytical solutions are often not available in these cases, and we rely on numerical methods like the Euler–Maruyama scheme or symbolic machine learning tools to approximate the solutions. These general models allow us to test the flexibility and accuracy of solvers in complex situations.

Together, these SDEs represent a wide range of behaviors and serve as benchmarks for evaluating both classical and modern solution techniques.

2.6 Numerical Scheme for Stochastic Differential Equations

In most cases, it is not possible to find exact solutions to SDEs. Therefore, numerical methods are often needed to approximate their solutions. In this section, we discuss a well-known numerical scheme: the Euler–Maruyama method [12], which is widely used to simulate SDEs when a closed-form solution is not available.

2.6.1 The Euler–Maruyama Method

The Euler–Maruyama method is a numerical scheme for approximating the solution of SDEs of Itô type. Consider the SDE:

$$dX_t = a(t, X_t)\, dt + b(t, X_t)\, dW_t, \quad X_0 = x_0, \tag{2.20}$$

The integral form of Eq. (2.20) over the interval $[t_n, t_{n+1}]$, where $t_{n+1} = t_n + \Delta t$, is:

$$X_{t_{n+1}} = X_{t_n} + \int_{t_n}^{t_{n+1}} a(s, X_s)\, ds + \int_{t_n}^{t_{n+1}} b(s, X_s)\, dW_s. \tag{2.21}$$

Since this expression is rarely solvable in closed form, we approximate the integrals by evaluating the integrands at the left endpoint t_n. We then have:

$$\int_{t_n}^{t_{n+1}} a(s, X_s)\, ds \approx a(t_n, X_{t_n}) \cdot \Delta t. \tag{2.22}$$

and,

$$\int_{t_n}^{t_{n+1}} b(s, X_s)\, dW_s \approx b(t_n, X_{t_n}) \cdot \Delta W_n, \tag{2.23}$$

where $\Delta W_n = W_{t_{n+1}} - W_{t_n} \sim \mathcal{N}(0, h)$ is the Brownian increment over the interval. By substituting the approximations into Eq. (2.21), we obtain the Euler–Maruyama update rule:

$$X_{n+1} = X_n + a(t_n, X_n)\, \Delta t + b(t_n, X_n)\, \Delta W_n. \tag{2.24}$$

So, given $t_n = n\Delta t$ with step size $\Delta t = \frac{T}{N}$, initial value $X_0 = x_0$, and increments $\Delta W_n \sim \mathcal{N}(0, h)$. The Euler–Maruyama method iteratively computes:

$$X_{n+1} = X_n + a(t_n, X_n)\, \Delta t + b(t_n, X_n)\, \sqrt{\Delta t} \cdot Z_n, \quad Z_n \sim \mathcal{N}(0, 1). \tag{2.25}$$

3 Automatic Differentiation

In many computational problems, derivatives are essential, particularly when doing optimization or solving numerical equations. Both numerical and symbolic differentiation are well-known strategies, but each has limitations. Overly complicated expressions can be produced via symbolic differentiation, and approximation errors are common in numerical techniques based on finite differences. However, a more accurate and effective substitute is provided by Automatic Differentiation (AD).

The concept behind AD is really simple: rather than treating a function as a formula that needs to be differentiated symbolically or approximated numerically, we break down its evaluation into simple steps and use calculus techniques like the chain rule alongside the computation. What makes AD different is that it computes derivatives with machine-level precision using only the same operations needed to evaluate the original function. This implies that we do not have to deal with symbolic complexity or lose accuracy as in numerical methods.

To illustrate how this works, consider the function the function $y = xe^{x^2}$, evaluated at $x = 3$, along with its derivative. A normal computation would proceed step-by-step, calculating intermediate values such as x^2, then e^{x^2}, and finally multiplying by x. Each of these processes is monitored in AD for both the value and its derivative. Thus, the chain rule is being applied at each stage, and derivative information is being propagated throughout the computation.

Table 3.1 shows how the function and its derivative evolve during the forward computation. The result is that both the function value and its derivative at $x = 3$ are available at the end of the process, with no approximation.

$x = 3$	$x' = 1$
$y_1 = x^2 = 9$	$y_1' = 2xx' = 6$
$y_2 = e^{y_1} = 8103.08$	$y_2' = e^{y_1} y_1' = 48618.5$
$y = xy_2 = 24309.24$	$y' = x'y_2 + xy_2' = 153958.58$

Table 3.1: Forward mode automatic differentiation for $y = xe^{x^2}$

There are two main ways to do automatic differentiation for a function f. The first is called **forward accumulation**. It works by following the steps of the function from the beginning (inputs) to the end (output), and calculates the derivative as it goes. This method is often done using dual numbers. The second method is **reverse accumulation**, which does the opposite: it first runs the function normally, then works backward from the output to the inputs to find the derivatives. Forward accumulation is better when the function has only a few inputs, while reverse accumulation works best when there are many inputs but only one output, like in training neural networks.

3.1 Forward Accumulation

In forward mode, each variable in the computation is augmented with its derivative using an algebraic structure called a **dual number**. A dual number is expressed as $x + y\epsilon$, where x is the function value, y is the derivative, and ϵ is an infinitesimal quantity satisfying $\epsilon^2 = 0$ [13]. This allows both the value and the derivative to be computed simultaneously as the function is evaluated.

We consider the following function as an example, where both x and y are functions of time t:

$$f(t) = e^{x(t)y(t)+(x(t)-y(t))^2}$$

To compute the total time derivative $\frac{df}{dt}$, we apply the chain rule:

$$\frac{df}{dt} = \frac{d}{dt}\left(e^{x(t)y(t)+(x(t)-y(t))^2}\right)$$

$$= e^{x(t)y(t)+(x(t)-y(t))^2} \cdot \frac{d}{dt}\left(x(t)y(t) + (x(t) - y(t))^2\right)$$

We now compute the derivative of the inner expression:

$$\frac{d}{dt}\left(x(t)y(t) + (x(t) - y(t))^2\right) = \frac{d}{dt}(x(t)y(t)) + \frac{d}{dt}((x(t) - y(t))^2)$$

$$= \dot{x}y + x\dot{y} + 2(x - y)(\dot{x} - \dot{y})$$

Therefore, the total time derivative of f is:

$$\frac{df}{dt} = e^{xy+(x-y)^2} \cdot (\dot{x}y + x\dot{y} + 2(x - y)(\dot{x} - \dot{y}))$$

where $\dot{x} = \frac{dx}{dt}$ and $\dot{y} = \frac{dy}{dt}$.

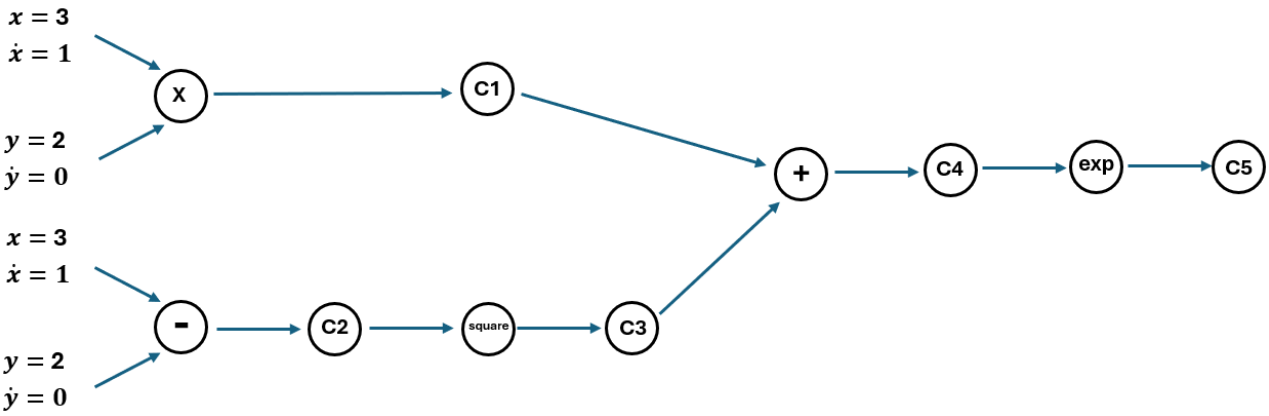


Figure 3.1: Initial structure for the computational graph of $e^{xy+(x-y)^2}$

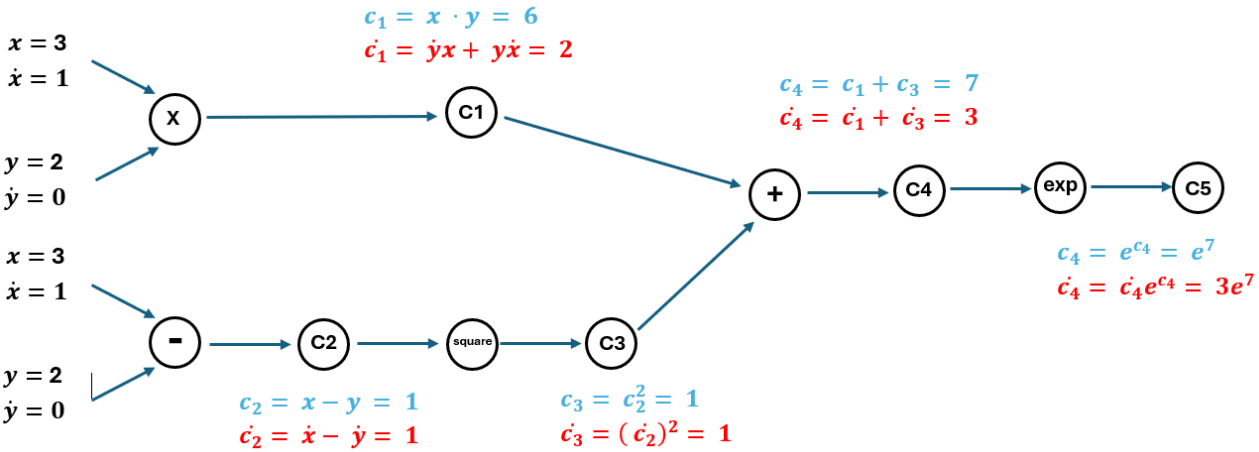


Figure 3.2: Forward-mode AD for $f(x, y) = \exp(xy + (x - y)^2)$ at $x = 3$, $y = 2$, with derivatives $\dot{x} = 1$, $\dot{y} = 0$.

While the process was quite lengthy doing the differenciation manually it becomes very easy when we evaluate this function and its derivative at $x = 3$, $y = 2$, with $\dot{x} = 1$, $\dot{y} = 0$, using forward accumulation.

We start with the first nodes in the computational graph, which are the function's inputs and any constant values. At each of these nodes, we record both the value and the partial derivative with respect to the variable of interest. The Initial structure is illustrated in Figure 3.1.

Next, we move down the graph one step at a time. At each step, we pick a node where all the input values have already been calculated. We find its value by using the values from the nodes before it. We also calculate how this node changes with respect to t, using both the values and derivatives from the earlier nodes. This process is shown in the Figure 3.2.

3.2 Reverse Accumulation

Reverse-mode AD works in two stages: first, a forward pass evaluates the function and stores intermediate values; then, a backward pass computes gradients by applying the chain rule in reverse. This approach is particularly efficient when dealing with scalar output functions with many inputs. In machine learning frameworks for training models with large parameter spaces, reverse accumulation is frequently used. [13].

In conclusion, Automatic Differentiation uses the chain rule at the elementary operation level to give a methodical and extremely precise way to compute derivatives. AD is a vital tool in optimization, scientific computing, and machine learning since it does away with the necessity for numerical approximations and symbolic differentiation, whether in forward or reverse mode.

4 Explainable Artificial Intelligence (XAI)

Artificial intelligence (AI) techniques, especially deep learning models, are often seen as **black boxes**. they can make very accurate predictions but don't clearly show how they reached those results. Even though they perform impressively in tasks like recognizing images or making forecasts, their lack of transparency makes it hard to understand, trust, or verify their decisions. This is especially a concern in important areas like self-driving cars, finance, and healthcare.

In order to address this issue, Explainable Artificial Intelligence (XAI) provides methods and tools for interpreting and understanding AI decision-making processes. As mentioned in [7] there are two main types of Explainable AI (XAI) methodologies. One is post-hoc explanation methods and other is intrinsically interpretable models. Intrinsically interpretable models are naturally interpretable and give users clear insights by immediately exposing their decision logic include decision trees and linear regression. However, these simpler models often fail to reach the high precision required for complex jobs.

To overcome this constraint, post-hoc explanation methods such as SHAP (SHapley Additive exPlanations) and LIME (Local Interpretable Model-agnostic Explanations) have been developed. Once choices have been made, these techniques analyse complex model outputs to offer local explanations of how certain predictions were made. SHAP calculates feature contributions to a forecast, for instance, while LIME uses more straightforward, interpretable models to locally approximate complex models, offering crucial information about choices that would otherwise be impossible to make.

However, implementing XAI successfully is still problematic. Creating standardized techniques to assess the quality of explanations, avoiding oversimplification, and making sure explanations correctly reflect the underlying decision logic are important concerns. Interdisciplinary cooperation is necessary to address these issues, integrating knowledge from computer science, cognitive psychology, ethics, and regulatory perspectives.

Finally, increasing explainability in AI systems facilitates responsible, moral application of cutting-edge AI technology in society, increases user trust, and ensures transparency. We primarily focused on two programming approaches in this study to implement XAI: Genetic Programming and Grammatical Evolution.

4.1 Genetic Programming (GP)

Genetic Programming (GP) is a type of evolutionary algorithm that simulates the process of natural selection to evolve computer programs. Similar to biological evolution, GP maintains a population of individuals, where each individual represents a candidate solution, in our case, a program. Through a cycle of selection, reproduction (i.e., crossover and mutation), and replacement (i.e., selecting the best fit), GP iteratively improves the population by promoting the most fit individuals while eliminating the less effective ones [15].

Every iteration, sometimes referred to as a generation, evaluates each Individual's fitness. This is usually done by running each program on a predefined set of inputs and comparing the program's outputs to the expected results. A program's fitness score increases with how closely its outputs match the desired values. Programs with higher levels of fitness have a higher chance of being chosen as

parents to produce individuals.

There are two primary population models in GP: the generational and steady-state models. The generational model replicates the natural cycles observed in species that reproduce seasonally by replacing the entire population at once. The steady-state model, on the other hand, gradually replaces older or weaker population members with new offspring.

In GP, individuals are usually represented by syntax trees, in which each leaf represents an input or constant value and each node represents a function or operation. These trees allow complicated behaviors to arise through combinations of simpler components, mirroring the structure of mathematical expressions.

The two primary genetic processes of crossover and mutation produce new individuals. Through arbitrary changes to an individual, such as replacing one function with another of the same class, mutation adds variation. For example, a terminal (a node without children) might be swapped out for another terminal, and multiplication could be used in place of a binary operation like addition. Crossover is the process of switching sub-trees between two parent programs and is modeled after biological recombination. By taking a sub-tree from one parent and inserting it into another, qualities from both parents are combined. This approach, which was first presented by [14], has gained broad acceptance since it effectively preserves constructive building pieces from both parents.

A continuous loop can be used to represent the entire GP process, which includes choosing individuals based on fitness, using variation operators, assessing the new individuals, and swapping out weaker ones. The population is gradually pushed towards programs that provide a better solution by this evolutionary cycle. Figure 4.1 illustrates this iterative process of genetic programming.

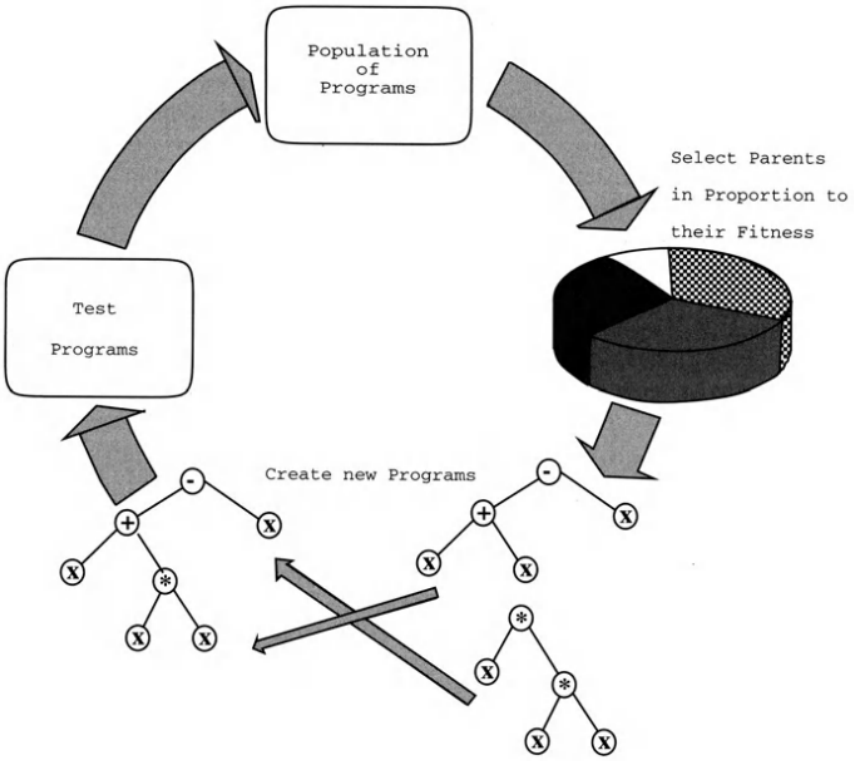


Figure 4.1: The genetic programming cycle: new programs are generated and evaluated for fitness. The best-fit programs succeed in creating offspring, while unfit ones are eliminated [15].

4.1.1 Example of Tree-Based Genetic Programming

To illustrate how genetic programming works in practice, consider the task of evolving a program that computes the function $y = x^2$. In this case, our genetic programming population may initially include individuals that produce different, but related, mathematical expressions.

For example, one program might evaluate the function $y = 2x - x$, and another might compute $y = \frac{x}{\frac{x}{x-x^3}} - x$. Both programs are selected based on their fitness because they produce answers similar to $y = x^2$ and therefore have relatively high fitness as shown in Figure 4.3. A new program can be created using the crossover operation, where a subtree from one parent replaces a subtree in the other. In this example, a subtree from the father Figure 4.2(b) is inserted into the mother Figure 4.2(a), producing the offspring shown in Figure 4.2(c). The resulting expression is $y = x + x^2 - x$, which simplifies to $y = x^2$, perfectly matching the target function.

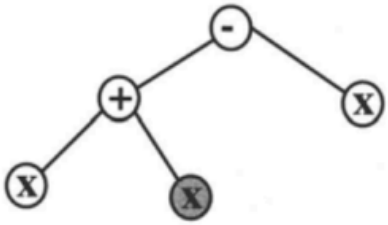


Figure a: Mum, fitness .64286, $2x - x$

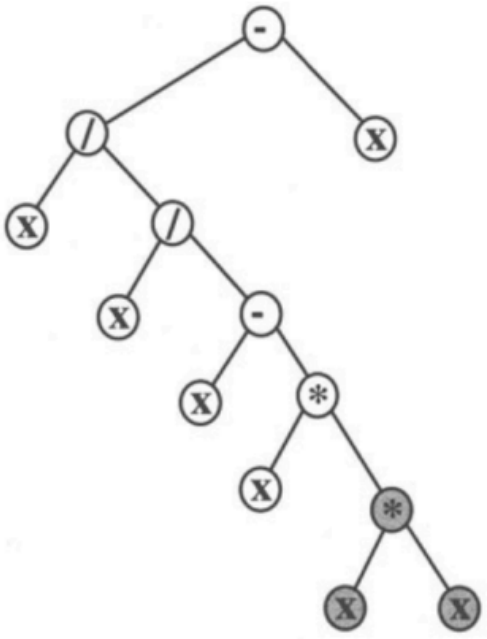


Figure b: Dad, fitness .70588, $\frac{x}{\frac{x}{x-x^3}} - x$

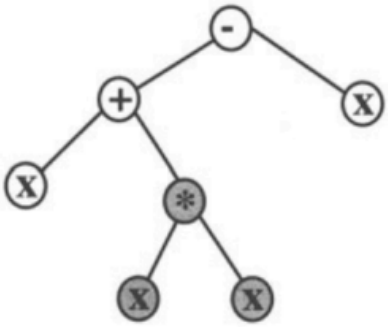


Figure c: Correct program, fitness 1.0, $x + x^2 - x$

Figure 4.2: Example of subtree crossover in GP. From left to right: Mum ($y = 2x - x$), Dad ($y = \frac{x}{\frac{x}{x-x^3}} - x$), and the resulting program ($y = x^2$) [15].

In this specific example, adapted from [15], the generated program evaluates $y = x^2$ and provides an accurate approximation. This process demonstrates how evolutionary operations like crossover help GP gradually build more accurate or optimal solutions by combining parts of promising individulas.

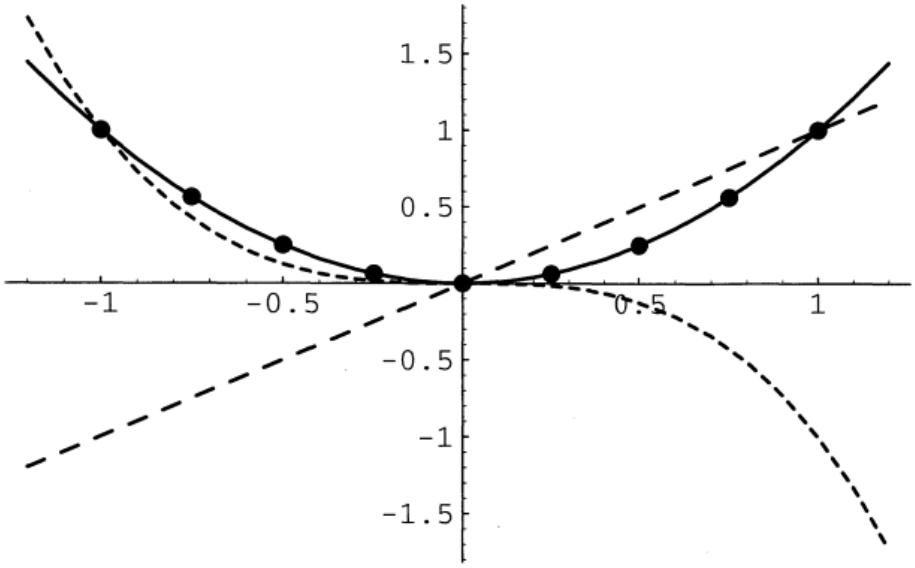


Figure 4.3: Plot of $y = x^2$ (solid), $y = 2x - x$ (Mum), and $y = \frac{x}{\frac{x}{x-x^3}} - x$ (Dad). [15]

4.2 Grammatical Evolution (GE)

Grammatical Evolution (GE), first introduced by O'Neill and Ryan [19, 20], is a form of evolutionary algorithm that generates computer programs or mathematical expressions using grammar rules. Unlike traditional genetic programming, which directly manipulates program structures (like syntax trees), GE evolves a binary string. This string is then mapped into a valid program by applying a predefined grammar, usually written in Backus–Naur Form (BNF).

The motivation behind GE is to provide more generality and flexibility than regular GP. Typically, standard GP is subject to strict limitations. For instance, demanding that the same data type be used by every function. By differentiating between the solution's representation (genotype) and ultimate structure (phenotype), GE overcomes these restrictions. As long as there is a grammar readily available, this enables growing solutions in any programming language or structure.

4.2.1 Underlying Process of GE

GE starts with a binary string, which is divided into smaller parts called **codons**. Each codon is an 8-bit segment and is interpreted as an integer. These integers are used to choose which grammar rule to apply during the construction of the final program. The grammar itself consists of production rules. These rules define how a start symbol (like <expr>) can be transformed into terminals (like numbers, operators, or functions).

For example, if the grammar for <expr> has four rules, and the codon has a value of 10, the rule selected will be 10 mod 4 = 2, meaning the second rule is applied. This process continues, codon by codon, until all nonterminals are expanded into terminals and the final program is complete. If

the codons run out before a full expansion is done, GE uses a technique called wrapping, where it starts reusing the codons from the beginning of the genome. This is inspired by biological systems, where overlapping genes are used efficiently to express multiple traits. The Figure 4.4 represents the complete flow of GE.

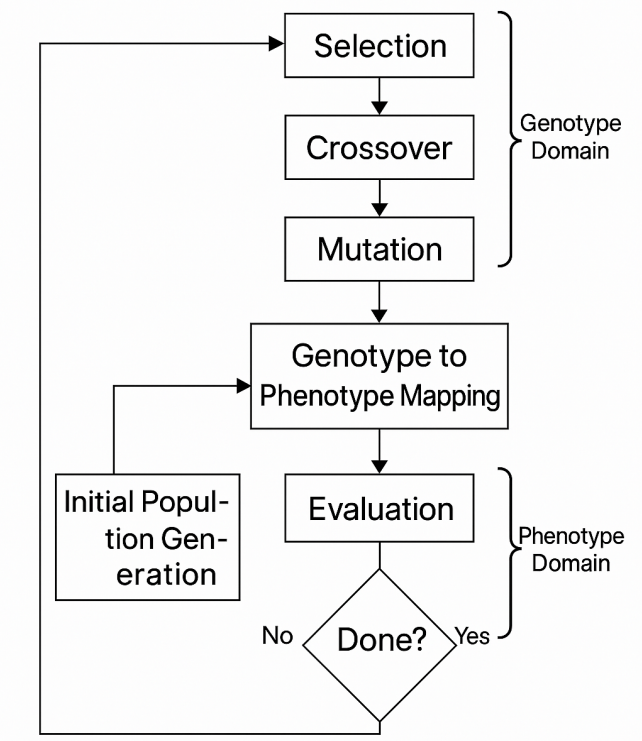


Figure 4.4: Flowchart of the Grammatical Evolution Process

4.2.2 Example

Imagine we want to evolve mathematical expressions. We can define a simple grammar like:

```
<expr> ::= <expr> + <expr>      (0)
         | <expr> - <expr>      (1)
         | a                    (2)
         | b                    (3)
```

Let's say GE starts with a binary string that is converted into these integers (codons):

$$60,\ 13,\ 62,\ 123,\ 74,\ 128,\ 233$$

The mapping process begins with the start symbol `<expr>` and uses each codon to expand the leftmost non-terminal using modular arithmetic over the number of production choices. There are four rules, so $60 \mod 4 = 0$, meaning we apply the first rule: `<expr> + <expr>`. Then, the system continues by expanding each of these `<expr>` symbols using the next codons, applying the same logic. After processing all codons, we might get a final expression like:

$$(a + b) + b$$

This shows how GE can automatically construct complete, meaningful expressions using binary strings and grammars. The full step-by-step expansion process is detailed in the Table 4.1.

Step	**Codon**	**Codon** mod 4	**Action / Structure**
1	60	0	$\langle expr \rangle \rightarrow \langle expr \rangle + \langle expr \rangle$
2	13	1	$\langle expr \rangle \rightarrow \langle expr \rangle - \langle expr \rangle$
Current structure			$(\langle expr \rangle - \langle expr \rangle) + \langle expr \rangle$
3	62	2	$\langle expr \rangle \rightarrow a$
Current structure			$(a - \langle expr \rangle) + \langle expr \rangle$
4	123	3	$\langle expr \rangle \rightarrow b$
Current structure			$(a - b) + \langle expr \rangle$
5	72	0	$\langle expr \rangle \rightarrow b$
Final Phenotype			$(a - b) + b$

Table 4.1: Step-by-step codon decoding process with production rules and expression building

The process of mapping a binary genome to a final program in GE is inspired by biological genetics. Just as DNA sequences are transcribed and translated into proteins through codons, GE uses codons (8-bit chunks) to guide the construction of a program.

5 GPAD Approach

In the paper [16], the authors propose a method for obtaining symbolic solutions to stochastic differential equations (2.12). Their approach combines Genetic Programming (4.1) with Automatic Differentiation (3), forming what they refer to as the GPAD framework. The method enables us to discover continuous-time symbolic solutions by evolving candidate functions that satisfy Itô's lemma (2.14) and optimizing them based on statistical consistency and fitness measures. Specifically, their method offers a flexible and interpretable substitute for conventional numerical methods by estimating the dynamics of SDEs directly from actual data without the need for prior knowledge of the functional form.

5.1 Methodology for GPAD

The approach used in their work is divided into two primary phases, with a final step outlining the composition and operation of the entire GPAD algorithm. In the first stage, the goal is to specify the SDE by estimating the drift and diffusion coefficients, denoted by $a(t, X_t)$ and $b(t, X_t)$, respectively. These estimates are obtained through symbolic regression guided by statistical validation techniques. The second stage focuses on obtaining a symbolic solution to the SDE using the estimated coefficients from the first stage. This involves solving the transformed equation under Itô calculus and recovering the original process through inversion of the transformation. Lastly, the structure and operation of the entire GPAD method are explained, emphasizing the interaction between Automatic Differentiation and Genetic Programming in both phases to produce a data-driven, symbolic solution.

5.1.1 Stage 1: Specification of the SDE

In the first step, the SDE requirements that control the dynamics of the underlying process are explained. Two different possible scenarios are presented here. In the first scenario, SDE's functional form is known, which outlines the modeling procedure. However, in many real-life circumstances and practical settings, this knowledge is unavailable, and the only known process trajectories are observable ones. The objective is to determine the SDE's structure in these situations using only the data.

To handle this challenge, Lobão et al. [16] developed an algorithmic procedure that integrates GPAD. The approach begins by considering a discretized form of the continuous-time SDE. Since continuous-time data is generally unavailable, observations are assumed to be collected at discrete time intervals. This discretization is handled using methods such as the Euler–Maruyama or Milstein Schemes.

The discritization of the SDE (2.12) is given by :

$$X_{t_i} - X_{t_{i-1}} = a(t_{i-1}, X_{t_{i-1}})\Delta + b(t_{i-1}, X_{t_{i-1}})(W_{t_i} - W_{t_{i-1}}) \tag{5.1}$$

where $\{t_i\}$ is a partition of the interval $[0, T]$ and $\Delta = t_i - t_{i-1}$ is the step size, assumed to be constant. Now to solve it analytically using Itô calculus, a transformation $Y_t = U(t, X_t)$ is used. This leads to a corresponding transformed SDE of the form:

$$Y_{t_i} - Y_{t_{i-1}} = \tilde{a}(t_{i-1}, X_{t_{i-1}})\Delta + \tilde{b}(t_{i-1}, X_{t_{i-1}})(W_{t_i} - W_{t_{i-1}}) \tag{5.2}$$

However, the coefficients $\tilde{a}$ and $\tilde{b}$ as well as the trajectory of the Wiener process, are unknown. Using the statistical properties of Brownian motion (2.1), the wiener increment $(W_{t_i} - W_{t_{i-1}})$ is considered as a normal random variable:

$$\frac{W_{t_i} - W_{t_{i-1}}}{\sqrt{\Delta}} = \varepsilon_{t_i} \sim \mathcal{N}(0, 1)$$

Using this, the equation becomes:

$$\frac{Y_{t_i} - Y_{t_{i-1}} - \tilde{a}(t_{i-1}, X_{t_{i-1}})\Delta}{\tilde{b}(t_{i-1}, X_{t_{i-1}})\sqrt{\Delta}} = \varepsilon_{t_i} \tag{5.3}$$

This relation enables us to estimate the coefficients $\tilde{a}$ and $\tilde{b}$ using the maximum likelihood principle. The GPAD algorithm searches for symbolic expressions for these coefficients such that the resulting residuals ε_{t_i} behave as independent standard normal distributions. The function that needs to be maximized is:

$$\mathcal{L}(\tilde{a}, \tilde{b} \mid t, Y, X) = -\frac{n}{2}\log(2\pi) - \sum_{i=1}^{n} \frac{(Y_{t_i} - Y_{t_{i-1}} - \tilde{a}(t_{i-1}, X_{t_{i-1}})\Delta)^2}{2\tilde{b}(t_{i-1}, X_{t_{i-1}})^2\Delta}$$

Once an optimal symbolic form for $\tilde{a}$ and $\tilde{b}$ is found, they are transformed back into the original coefficients a and b using relations derived from Itô's lemma. After that, replacing the original SDE coefficients a and b with them in the likelihood framework gives a new optimization problem which is:

$$(\hat{a}, \hat{b}) = \arg\max_{a,b} \{\mathcal{L}(a, b \mid t, Y, X)\}$$

In particular, it looks like:

$$\mathcal{L}(a, b \mid t, Y, X) = -\frac{n\log(2\pi)}{2} - \sum_{i=1}^{n} \frac{\left[Y_{t_i} - Y_{t_{i-1}} - \left(\frac{\partial U(t_{i-1}, X_{t_{i-1}})}{\partial t_{i-1}} + \frac{\partial U(t_{i-1}, X_{t_{i-1}})}{\partial X_{t_{i-1}}} a(t_{i-1}, X_{t_{i-1}}) + \frac{\partial^2 U(t_{i-1}, X_{t_{i-1}})}{2\partial X_{t_{i-1}}^2} b^2(t_{i-1}, X_{t_{i-1}})\right)\Delta\right]^2}{2\left[\frac{\partial U(t_i, X_{t_i})}{\partial X_{t_i}} b(t_i, X_{t_i})\right]^2\Delta}$$

and hence equation (5.3) becomes:

$$\frac{Y_{t_i} - Y_{t_{i-1}} - \hat{a}\left(t_{i-1}, X_{t_{i-1}}\right)\Delta}{\hat{b}\left(t_{i-1}, X_{t_{i-1}}\right)\sqrt{\Delta}} = \hat{\varepsilon}_{t_i} \sim \text{Normal}(0, 1), \quad \forall t_i. \tag{5.4}$$

Once the residuals $\hat{\varepsilon}_{t_i}$ are computed using the estimated coefficients, the methodology performs a series of tests to validate the normality and independence assumptions essential for a correct specification. Firstly, the standardized residuals must satisfy the following properties:

$$\bar{\varepsilon} = \frac{1}{n}\sum_{i=1}^{n} \hat{\varepsilon}_{t_i} = 0, \quad S_\varepsilon = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\hat{\varepsilon}_{t_i} - \bar{\varepsilon})^2} = 1$$

Here, $\bar{\varepsilon}$ is the sample mean, and S_ε is the sample standard deviation of the residuals. These conditions ensure that the residuals are centered around zero and have unit variance, as expected under the standard normal distribution. In addition, the methodology applies the Jarque–Bera (JB) test to formally assess the normality of the residuals:

$$JB = \frac{n}{6}\left(A^2 + \frac{(K-3)^2}{4}\right)$$

where A is the sample estimate and K is the sample kurtosis of the residuals. If the computed JB statistic is less than the critical chi-squared value (e.g., 5.9915 for 5% significance with 2 degrees of freedom), then the null hypothesis of normality is not rejected. Further, to test for serial independence, the Ljung–Box (QLB) test is used:

$$Q_{LB} = T(T+1)\sum_{j=1}^{k} \frac{r_j^2}{T-j}$$

Where r_j is the autocorrelation at lag j, and the statistic is compared to the chi-squared distribution with k degrees of freedom. A low Q_{LB} value (below the tabulated critical value) confirms that the residuals are not autocorrelated up to lag k.

The estimations of the residuals are considered valid when the QLB and JB tests both support the null hypotheses of white noise behaviour and normality, respectively. After validation, the scaled noise increments are added up to these residuals, which are then used to recreate the Wiener process' trajectory:

$$\hat{W}_{t_i} = \sqrt{\Delta} \sum_{k=1}^{i} \hat{\varepsilon}_{t_k}$$

This formulation is based on the fact that the increment $(W_{t_i} - W_{t_{i-1}})$ is approximated by $\sqrt{\Delta}\,\hat{\varepsilon}_{t_i}$. Thus, a consistent estimate of the Wiener process is obtained recursively, starting from $\hat{W}_{t_0} = 0$.

This brings an end to the first step, which involves using discretized observations to identify the structure of the SDE and the associated Wiener process. The methodology's second step, which determines the symbolic solution to the SDE, then uses the validated estimates as inputs.

5.1.2 Stage 2: Symbolic Solution of the SDE

The objective of the second stage of the process is to use the estimated coefficients from the first stage to symbolically reconstruct the solution to the original SDE. This reconstruction is done through a transformation $Y_t = U(X_t)$, leading to a new formulation of the problem where a solution to the transformed SDE is obtained.
The transformed SDE takes the form:

$$dY(t) = \tilde{a}(t, X(t))\, dt + \tilde{b}(t, X(t))\, dW_t$$

In the integral form, we have:

$$\int_0^t dY(s) = \int_0^t \tilde{a}(s, X(s))\, ds + \int_0^t \tilde{b}(s, X(s))\, dW(s)$$

The left-hand side simplifies to:

$$Y(t) - Y(0) = U(X(t)) - U(X(0))$$

Where $X(0)$ is the known initial value of the original process. The integrals on the right-hand side involve stochastic components and are not analytically solvable in general. Therefore, numerical approximations are employed. The integral of the drift term is approximated using the trapezoidal rule:

$$\int_0^t \tilde{a}(s, X(s))\, ds \approx \sum_{i=1}^{n} \frac{1}{2} \left[\tilde{a}(t_i, X_{t_i}) + \tilde{a}(t_{i-1}, X_{t_{i-1}})\right] \Delta$$

The integral involving the diffusion term is approximated by Ito integrals (2.2) :

$$\int_0^t \tilde{b}(s, X(s))\, dW(s) \approx \sum_{i=1}^{n} \tilde{b}(t_{i-1}, X_{t_{i-1}})(W_{t_i} - W_{t_{i-1}})$$

By combining both approximations, the discrete-time representation of the solution Y_t is obtained as:

$$Y_{t_i} = Y_0 + \sum_{j=1}^{i} \frac{1}{2} \left[\tilde{a}(t_j, X_{t_j}) + \tilde{a}(t_{j-1}, X_{t_{j-1}})\right] \Delta + \sum_{j=1}^{i} \tilde{b}(t_{j-1}, X_{t_{j-1}})(W_{t_j} - W_{t_{j-1}})$$

Once the symbolic expression for $Y(t)$ is derived, the final step is to recover $X(t)$ by inverting the transformation:

$$X(t) = U^{-1}(Y(t)) = U^{-1}\left(Y_0 + \int_0^t \tilde{a}(s, X(s))\, ds + \int_0^t \tilde{b}(s, X(s))\, dW(s)\right)$$

This symbolic inversion yields the closed-form representation of the original process:

$$X(t) = U^{-1}\left(Y_0 + f(t, X(0), W)\right) = h(t, X(0), W)$$

Thus, the function h represents the symbolic solution to the SDE derived through the GPAD algorithm, based on both deterministic and stochastic components identified in the earlier stages [16].

5.1.3 GPAD Framework

This section explains the structure and workflow of the GPAD framework, which is used to generate symbolic solutions to stochastic differential equations. The implementation was built using MATLAB and adheres to the methodology suggested by [16]. It builds on prior basic work in automated differentiation and genetic programming.

In this framework, tree topologies are used to encode candidate mathematical expressions, sometimes referred to as individuals. This representation enhances the interpretability of the models and permits symbolic modification. Because of the framework's great flexibility, the user can change both the structural and functional aspects. These include the functional and terminal sets used to create expressions, population size, tree depth, and genetic operator settings (such as mutation and crossover rates).

The function set typically includes elementary arithmetic operators ($+, -, \times, \div$) and common mathematical functions ($\sin, \cos, \exp, \log, x^y, \tan$). The terminal set includes constants (e.g., π, real and complex numbers, or even random values generated through *rand*) and the variables that define over SDE. These sets were defined according to conventional procedures in the genetic programming literature and the parameterization used in the GPLAB Toolbox, a widely adopted framework for genetic programming developed by Silva [22].

This approach is essential for the algorithm to function successfully, even though functions and constants can be freely selected. The approach may converge early and produce unwanted answers if the sets have few items because of a lack of diversity. Longer processing times for locating a suitable solution could result from an excessive number of elements in the solution space. According to the genetic programming literature [16], the function and terminal sets are thus chosen in a balanced way to provide both closure and sufficiency.

We shall then describe the GPAD procedure. First, an initial population is created, with each member being a randomly built expression tree. In [16], the **Grow method** was preferred among the many initialization strategies because it usually produces asymmetrical trees that more closely resemble mathematical models.

Next, each individual is evaluated using a fitness function. In the case of parameter estimation, the fitness reflects how well the expression maximizes a log-likelihood function while meeting certain statistical requirements (such as the normality and independence of residuals). Selection for reproduction is then performed. Among several available methods, including tournament and roulette selection, the **lexictour** strategy was preferred in [16] for its balance between accuracy and model simplicity.

Following selection, new individuals are created by applying the genetic operators, crossover and mutation. These offspring need to meet structural requirements, such as staying within maximum depth or node restrictions. Only eligible individuals move on to their next generation. Which individuals will be kept for future generations is then decided by the replacement plan. The **total elitism** approach was applied in this instance in [16], guaranteeing that the top performers, whether new or existing, are retained per their fitness assessments.

The implementation of the GPAD framework is divided into two operational phases, each handled by a pair of separate codes:

1. **Model Identification Phase:** In this phase, one algorithm explores symbolic possibilities using genetic programming, and the other evaluates their fitness by comparing statistical criteria and log-likelihood. Both the rebuilt Wiener process and the approximated SDE structure are included in the result.
2. **Symbolic Solution Phase:** Here, the focus shifts to solving the identified SDE. One algorithm again generates symbolic candidates, and the second evaluates them by comparing their outputs to a reference solution $Y(t)$. The sum of squared errors (SQE) and penalty terms for violating parameters like initial values or boundary constraints are included in the fitness score.

$$\text{Fitness Score} = \text{SQE} + \text{Penalty Terms}$$

where,

$$\text{SQE} = \sum_{i=1}^{n} \left[Y_{t_i} - \hat{Y}_{t_i} \right]^2 = \sum_{i=1}^{n} \left[Y_{t_i} - Y_0 - f(t_i, X_{t_i}, \hat{W}_{t_i}) \right]^2$$

And the function f representing the numerical integration of the SDE is defined as:

$$f(t_i, X_{t_i}, \hat{W}_{t_i}) \approx \sum_{s=1}^{i} \frac{1}{2} \left[\hat{a}(t_s, X_{t_s}) + \hat{a}(t_{s-1}, X_{t_{s-1}}) \right] \Delta + \sum_{s=1}^{i} \hat{b}(t_{s-1}, X_{t_{s-1}}) \left(\hat{W}_{t_s} - \hat{W}_{t_{s-1}} \right)$$

Now, after all these steps, once the symbolic solution is obtained through the genetic programming phase, the next step is to verify its consistency with Itô calculus. This is achieved by applying an Automatic Differentiation (AD) algorithm (3) that evaluates whether the derived symbolic expression has the same differential properties as the original SDE.

A practical example of automatic differentiation applied to numerical problems is provided by Fink, who developed MATLAB implementations named `myAD` and `myA2D` [6]. These implementations are publicly available via MATLAB Central File Exchange. They compute first and second-order partial derivatives accurately within MATLAB environments.

To demonstrate automatic differentiation, an illustrative MATLAB example using `myAD` and `myA2D` is provided in [6, 16]. These packages allow for the accurate calculation of first and second-order derivatives. The usage for computing derivatives is presented exactly as indicated:

```
x = myAD(x0);
result = myFunc(x);
functionValue = getvalues(result);
derivatives = getderivs(result);
```

For second-order derivatives:

```
x = myA2D(x0);
result = myFunc(x);
functionValue = getvalues(result);
derivatives = getderivs(result);
secondDerivatives = getsecderivs(result);
```

The limitation of these packages is that they cannot handle matrix multiplication, power, or division operations directly. Therefore, concatenation needs to be utilized instead. For instance, instead of:

```
dx(1) = dx_1;
dx(2) = dx_2;
```

use:

```
dx = [dx_1; dx_2];
```

as recommended in [16].

To check the accuracy, the solution is assessed using the mean absolute percentage error (MAPE) between the analytical derivative $dX(t)$ and its automatically computed counterpart $d\hat{X}(t)$. If the resulting MAPE exceeds a predefined tolerance threshold (set at 0.01% in the original study), the current solution is discarded, and a new iteration of the genetic search is initiated [16].

This procedure guarantees that only mathematically coherent and statistically valid symbolic solutions are chosen as final products. The GPAD method provides a thorough and comprehensible modeling approach for stochastic differential equations [16] by combining symbolic regression, stochastic integration, and differential verification.

5.2 Discussion and Analysis on GPAD

In [16], the author has considered a Geometric Brownian Motion (2.15) as an example to show the simulations with $\mu = 0.2$, $\sigma = 0.5$ and $X(0) = 20$ over $n = 1000$ time steps of size $\Delta t = 0.004$. The general analytical solution to the equation (2.15) is known and can be written as:

$$X(t) \;=\; X(0)\,\exp\!\left(\left(\mu - \tfrac{\sigma^2}{2}\right) t \;+\; \sigma\, W(t)\right) \tag{5.5}$$

However, the author has considered it in its raw form and without even the specification of the SDE. After applying the two stages of the GPAD procedure (5.1), the following estimates for the coefficients are obtained as:

$$\hat{a}(t, X(t)) = 0.198\, X(t), \quad \hat{b}(t, X(t)) = 0.501996\, X(t)$$

and hence the specification of SDE is:

$$dX(t) = 0.198\, X(t)\, dt + 0.501996\, X(t)\, dW(t),$$

or in transformed form, using the transformation identified in the process, that is, $Y_t = \log\left(X_t\right)$ we have:

$$dY(t) = 0.072\, dt + 0.501996\, dW(t)$$

Note that the expression for the normal white noise was obtained as:

$$\begin{aligned}\hat{\varepsilon}_{t_i} &= \log\!\left(\left(\frac{X_{t_i}}{1.000288\, X_{t_{i-1}}}\right)^{31.4971}\right) \\ &= \frac{\log X_{t_i} - \log X_{t_{i-1}} - 0.000288}{0.031749}\end{aligned}$$

consequently, the Wiener process that corresponds to this is

$$\hat{W}_{t_i} = \sqrt{\Delta} \sum_{k=1}^{i} \hat{\varepsilon}_{t_k} = \sqrt{0,004} \sum_{k=1}^{i} \hat{\varepsilon}_{t_k}$$

Once the function $Y_{t_i} = \log\left(X_{t_i}\right)$ is identified and the trajectory of the Wiener process $\left(\hat{W}_{t_i}\right)$ is estimated, the final solution of the SDE $dY(t)$ is obtained,

$$\hat{Y}(t) = Y(0) + f\left(t,\, X(t),\, W(t)\right) = \log(20) + 0.0724\, t + 0.50198\, W(t)$$

and by applying the inverse transformation, we obtained the solution to the original SDE:

$$\hat{X}(t) = 20 \exp\left\{0.0724\, t + 0.50198\, \hat{W}(t)\right\}$$

The performance statistics provided in [16] show that the solution matched the simulated data very closely. It had a very high R^2 value of 0.99999, which means the model was able to describe the behavior of the random process accurately. The error values, such as Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), and Root Sum of Squared Errors (RSSE), were extremely small, around 10^{-8} and 10^{-9}, showing the model's high accuracy. The trend coefficient was estimated as 0.07240 with a standard error of 0.00287. The volatility coefficient was 0.50198 with a standard error of 0.00645. Both values were statistically very significant, as shown by high t-values (25.24 for trend and 77.80 for volatility) and p-values close to zero. When these results are compared to the true values used in the simulation (0.075 for trend and 0.5 for volatility), the difference is small, around 0.0026 for trend and 0.00198 for volatility. These small differences represent relative errors of 3.5% and 0.4%, which are likely caused by random variation in the data, especially due to the random nature of the Wiener process.

Even though the study presents an innovative and interesting approach, there are still some problems that need to be resolved before it can be used or tested. According to the authors, they created **two pairs of codes** in MATLAB: one for symbolically solving the SDEs and another for configuring them. At first, no download links or public repositories were offered, and these codes were not distributed. After a considerable amount of time, they finally replied and sent the codes by email. However, because of missing dependencies or components, the shared files were not complete and could not be used correctly. The codes were therefore unusable, particularly because they depended on specialised tools like the GPLAB toolbox and custom differentiation packages like myAD or myA2D.

Additionally, the paper's whole analysis is predicated on a single, straightforward instance of one-dimensional geometric Brownian motion (GBM). They didn't test their approach on real-world data, greater dimensions, or other kinds of SDEs. This raises questions about how well their approach performs in more intricate or practical contexts. Furthermore, the study doesn't explain what happens if the input data is somewhat noisy, if the time step is altered, or even if the method is performed several times using various random seeds. There is no way to verify if the approach consistently finds the same answer or occasionally runs into problems.

These problems with the non-functional and incomplete codes prevented us from replicating the original analysis and results or from expanding their methodology. This made it challenging to investigate its efficacy or use it in more intricate situations. Consequently, we used a Python-based framework and Grammatical Evolution (4.4) to propose a new method for solving stochastic differential equations.

6 GE Approach

After studying the GPAD approach and facing issues with its implementation, especially due to the incomplete MATLAB codes, we decided to modify the approach. We used GE to approximate the solution of SDEs directly from simulated data. Unlike GPAD, our goal was not to separately estimate the drift and diffusion functions, but instead to find a symbolic expression that closely fits the entire trajectory of the solution.

We started by simulating the solution X_t of the SDE using the exact solution of Brownian motion, and later we used the Euler–Maruyama method for different SDEs whose exact and explicit solution is not possible. For this, we generated a Wiener process W_t and used it to simulate the trajectory. This gave us a dataset containing time values t, the corresponding Wiener values W_t, and the simulated process values X_t. These were then used as input-output pairs for symbolic regression using GE. At first, we used the same Wiener process for both simulation and regression. That is, the GE model was given access to the exact Wiener path used during simulation. This helped us verify whether the symbolic model could recover the structure of the solution when the stochastic input was known.

Once this baseline was established, we moved to a more difficult setting. We tried to approximate the same trajectory, but this time without using the original Wiener process. Instead, we used a different realization of the Wiener path during training. Here, the main goal was to test sensitivity of the GE approach towards the changing in the stochastic input and to check the ability of the model to find a good approximation of the trajectory when experimented with different noise.

We then performed statistical validation to make sure the models were accurate. For each SDE, we specifically created a large number of sample pathways and calculated the variance and mean over time. The relevant values obtained from the symbolic models were then contrasted with these statistical metrics. Through this study, we were able to assess how well the symbolic expressions represented the stochastic system's overall behaviour rather than just its individual trajectories.

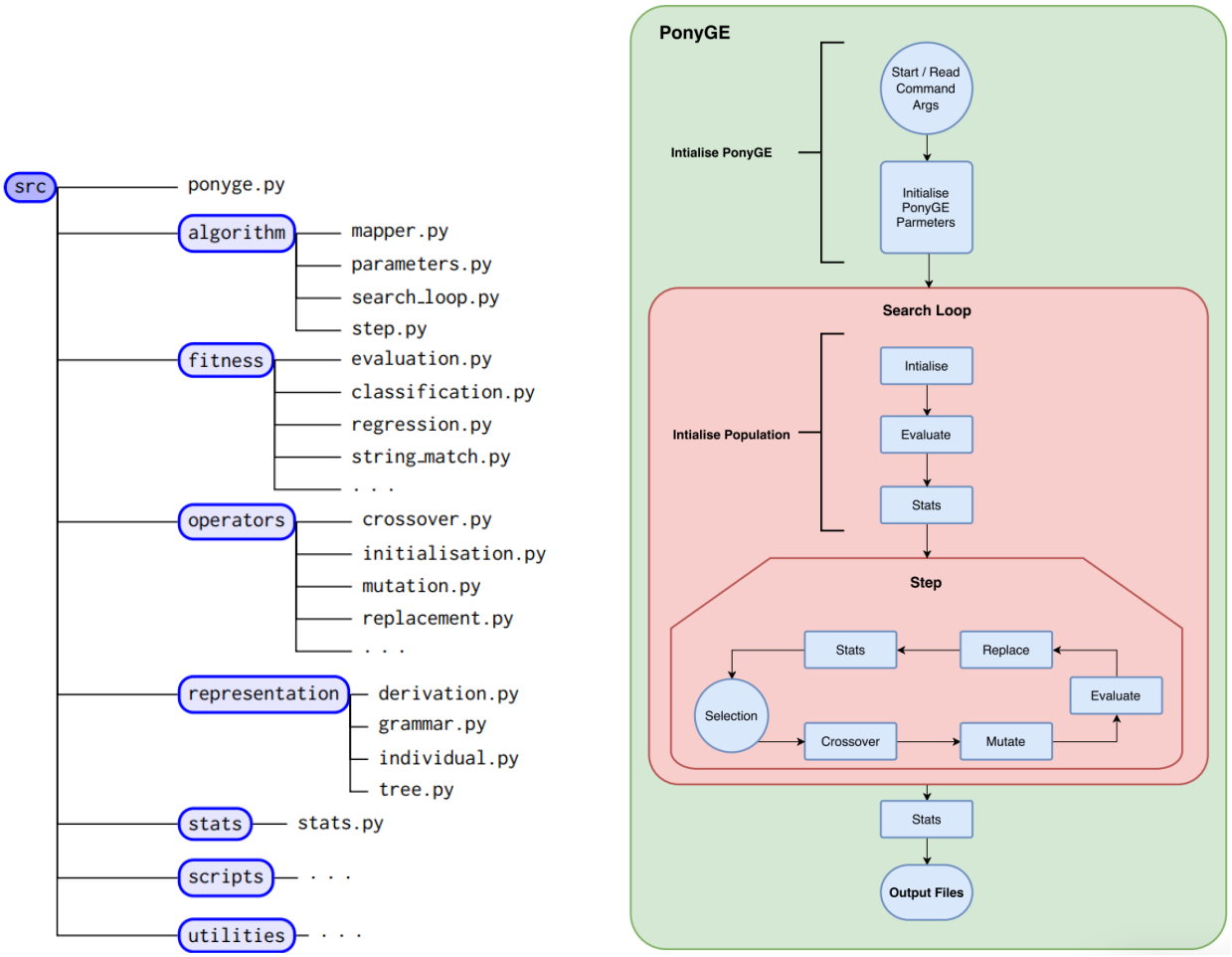
We utilised the Python framework PonyGE2 [5] for implementation because it was flexible and simpler to extended. PonyGE2's framework enabled us to directly alter the fitness and regression modules as needed, develop custom grammars, and modify evolutionary parameters. This flexibility was important because we needed to control how the symbolic expressions were built, which variables were allowed (like time and Wiener process values), and how the error between the predicted and true trajectories was computed. Unlike black-box machine learning libraries, PonyGE2 gave us full access to the internals of the evolutionary process, making it easier to track the process.

6.1 PonyGE2 framework

PonyGE2 is a Python-based framework specifically developed for grammatical evolution, first introduced in the literature [5] as an improved and user-friendly platform for evolutionary computation. It provides a straightforward Python environment designed to overcome limitations found in earlier implementations, such as complex code structures and limited flexibility.

One of the core reasons we selected PonyGE2 is its modular and transparent architecture. The framework clearly separates components responsible for initialization, grammar definition, genetic operations (mutation and crossover), fitness evaluation, and genotype-to-phenotype mapping. This modularity simplified our experimental workflow because we could directly customize and optimize each module separately. It also made debugging significantly easier, as we could isolate and quickly resolve issues in

individual components without needing to modify unrelated parts. Complete control flow diagram of PonyGE2 is shown in Figure. 6.1(b)



(a) Codebase structure of PonyGE2 [5]

(b) Control flow diagram for a typical GE setup [5]

Figure 6.1: Overview of PonyGE2: (a) Codebase structure, (b) Execution flow.

PonyGE2 has a special way of showing solutions, called a dual representation. This means every solution is described in two ways at once. The first way is as a simple list of numbers, known as codons. These numbers tell the system how to choose rules from a grammar. The second way is as a tree structure, called a derivation tree. This tree is built step-by-step by following the grammar rules chosen by the codons. Having these two ways is helpful because lists of numbers are easy to change using evolutionary methods like crossover and mutation, while trees make it easier to see and understand the final solution.

PonyGE2 also provides two special operations: fixed two-point crossover and codon-level mutation. These methods help to keep solutions valid after applying genetic changes. Because of this, it reduces the risk of creating incorrect solutions that don't follow the grammar rules. In addition, PonyGE2 uses something called Backus-Naur Form (BNF) grammar, which clearly defines how solutions should be built. This makes it flexible and easy to adapt PonyGE2 to many different problems.

Another advantage of PonyGE2 is reproducibility. This means all details of the experiments, such as random seeds, grammar rules, and evolutionary settings, can be clearly saved in parameter files.

Because of this, each experiment can be repeated exactly in the same way. This was very important for our analysis and evaluation

PonyGE2 requires Python 3.5 or higher and depends on standard packages like `matplotlib`, `numpy`, `scipy`, `scikit-learn`, and `pandas`. The main file `ponyge.py` is located in the `src` folder. Additional scripts are provided in the `scripts` folder, which can run independently or together with PonyGE2. These include tools like an experiment manager and a statistics parser for multiple runs. More details are available in [5]. The `utilities` folder contains useful functions for file input/output, plotting, command-line parsing, protected math operations, and error metrics. Figure 6.1(a) shows the overall codebase structure.

6.2 Modifications in PonyGE2

To use PonyGE2 for solving SDEs, we made several changes to its original structure. These changes allowed the system to work with data that included both time and Wiener process values. Below is a detailed explanation of the main modifications we introduced:

Dataset Generation: To train our symbolic models using GE, we first created datasets by simulating the solution X_t of a SDE using the geometric Brownian motion model. We considered two cases: one with a known Wiener process (single path) and another with multiple independent Wiener processes (multiple paths).

In the first case, we generated data using a single Wiener process. This was used in our baseline experiments, where the same Wiener process was used during both generating X_t and in computing the symbolic solution for X_t. The code below shows how the dataset was created:

```
T = 1.0
N = 2**10
dt = T / N
lam = 2.0
mu = 0.5
X0 = 1.0

t = np.linspace(0, T, N+1)

np.random.seed(100)
dW = np.sqrt(dt) * np.random.randn(N)
W = np.concatenate(([0], np.cumsum(dW)))

X_exact = X0 * np.exp((lam - 0.5 * mu**2) * t + mu * W)

with open('train.txt', 'w') as data:
    data.write("x0\tx1\tresponse\n")
    for t_val, w_val, x_val in zip(t, W, X_exact):
        data.write(f"{t_val:.8f}\t{w_val:.8f}\t{x_val:.8f}\n")
```

This dataset includes columns for time t, Wiener process W_t, and the corresponding solution X_t, and was saved in a tab-separated format as a text file.

In the second case, we wanted to find a Wiener process that could approximate our original trajectory in symbolic form. For this, we used multiple independent Wiener paths. The target solution X_t was

computed using a fixed Wiener process, while the dataset included only Wiener processes that were independent of this fixed path.
We first generated the true X_t using a fixed Wiener process:

```
np.random.seed(100)
dW_xt = np.sqrt(dt) * np.random.randn(N - 1)
W_xt = np.concatenate(([0], np.cumsum(dW_xt)))

X_t = X0 * np.exp((lam - 0.5 * mu**2) * t + mu * W_xt)
```

After generating the true solution, we initially considered giving multiple Wiener processes as separate input columns to the framework. However, due to how genotype-to-phenotype mapping works in PonyGE2, the expressions were constructed using combinations of all available input columns. This led to symbolic expressions that combined different Wiener paths, which were not useful in finding a Wiener process that approximates over trajectory.

To overcome this issue, we restructured the dataset so that each row used only one Wiener path at a time, repeated across multiple training runs. Each training dataset included the same time vector t, the same target X_t, and a unique Wiener realization $W^{(i)}$. We then trained a separate GE model for each dataset and evaluated which Wiener path produced the best symbolic approximation of the true solution. This approach gave us more control over the input and allowed us to stay within the technical constraints of PonyGE2. The dataset looked conceptually like this:

$$\begin{array}{ccc} t & W_t^{(1)} & X_t \\ t & W_t^{(2)} & X_t \\ t & W_t^{(3)} & X_t \\ \vdots & \vdots & \vdots \end{array}$$

Here, $W_t^{(1)}$,$W_t^{(2)}$, etc., represent different realizations of the Wiener process, while the same t and X_t are repeated for each trajectory. This was done to test whether the model could find an expression that approximates the original solution even when the Wiener process is not known. To implement this, we made multiple copies of the time vector and solution array, and generated new Wiener paths for each copy.

Listing 1: Generating multiple Wiener paths

```
M = 1000
wiener_list = []
time_list = []
X_t_list = []

np.random.seed(999)
for m in range(M):
    dW = np.sqrt(dt) * np.random.randn(N - 1)
    W = np.concatenate(([0], np.cumsum(dW)))
    wiener_list.append(W)
    time_list.append(t)
    X_t_list.append(X_t)
```

We selected different seed 999 to ensure that the wiener proceses should be indepented of the orignal one which had seed value of 100. Finally, the data was flattened and saved:

Listing 2: Saving the dataset with multiple Wiener paths

```
time_array   = np.array(time_list).flatten()
wiener_array = np.array(wiener_list).flatten()
X_t_array    = np.array(X_t_list).flatten()

data = np.column_stack((time_array, wiener_array, X_t_array))

np.savetxt("train.txt", data, fmt="%.8f", delimiter="\t",
           header="time\tWiener\tX_t", comments='')
```

Custom Fitness Function: The original fitness function in PonyGE2 was designed for general supervised learning tasks, such as classification or basic regression. However, it was not suitable for data generated from SDEs involving Wiener processes. To address this, we made changes in two steps. First, we handled a simple case without grouping, where we used time t and a single Wiener process W_t to simulate the solution X_t and tried to approximate it using symbolic expressions. Then, we extended the function to handle a more complex case with multiple Wiener paths. The goal in this second stage was to find symbolic expressions that could approximate the solution of the SDE using a Wiener process different from the one used during simulation. This helped us understand how sensitive the model was to changes in the stochastic input. To evaluate the quality of the approximation, we used the following fitness criterion:

$$\text{Fitness} = \frac{1}{n}\sum_{i=1}^{n}\frac{\left|X_t^{(i)} - \hat{X}_t^{(i)}\right|}{\sqrt{t^{(i)}}}, \tag{6.1}$$

where X_t is the true solution, $\hat{X}_t$ is the approximated value, and t is the corresponding time point.

In the first case, the predicted values from the symbolic expression were compared directly to the true values using an error function 6.2. If the predicted output was a single constant, we made sure it was broadcasted to match the shape of the actual data. The code for this part is shown below:

```
yhat = eval(ind.phenotype)
assert np.isrealobj(yhat)

if np.ndim(yhat) != 0:
    if y.shape != yhat.shape:
        raise ValueError("Shape mismatch.")
else:
    yhat = np.full_like(y, yhat)

return params['ERROR_METRIC'](y, yhat, x[:, 0])
```

This code checks if the model's output has the correct shape and type to match the target values. If it's a constant, it creates an array of the same shape before computing the error. Then, it returns the fitness value by computing the error using the chosen error metric.

Later, we extended the function to handle multiple different Wiener processes. We added a new parameter called `NUM_WIENER_PROCESSES` to tell the code how many separate trajectories were used. The idea was to group the time-series data by trajectory and check the model's performance on each one separately. We first reshaped the outputs into groups:

```
n_w = params['NUM_WIENER_PROCESSES']
total_points = y.shape[0]
n_t = total_points // n_w

y_grouped = y.reshape((n_w, n_t))
yhat_grouped = yhat.reshape((n_w, n_t))
t_grouped = x[:, 0].reshape((n_w, n_t))
```

Then we calculated the error for each Wiener path one by one:

```
errors = []
for i in range(n_w):
    group_y = y_grouped[i, :]
    group_yhat = yhat_grouped[i, :]
    group_t = t_grouped[i, :]

    error_i = params['ERROR_METRIC'](group_y, group_yhat, group_t)
    errors.append(error_i)
```

Instead of taking the average, we returned the 1st percentile (very best score) from the list of errors:

```
    best_error = np.quantile(errors, q=0.01)
return best_error
```

This helped the model focus on the most promising patterns, even if it did not perform well on all paths.

Moreover, the error metric was changed to (6.1). Unlike the standard Mean Absolute Error (MAE), this fitness criterian normalizes the absolute error by the square root of time. This is because in a Wiener process, the variance grows with time, so normalizing by $\sqrt{t}$ helps balance early and late time steps. The implemented version is:

```
def mae(y, yhat, x0):
    if np.isscalar(yhat):
        result = np.mean(np.abs(y[1:] - yhat) / np.sqrt(x0[1:]))
    else:
        yhat = np.array(yhat)
        result = np.mean(np.abs(y[1:] - yhat[1:]) / np.sqrt(x0[1:]))
    return result
```

This function works with both scalar and array outputs and gives a better measure of how well the symbolic model fits the SDE solution. In a nutshell, these changes to the fitness function and error metric made PonyGE2 more suitable for symbolic regression on data coming from stochastic simulations.

Grammar: We defined a new grammar in Backus-Naur Form (BNF) to build expressions that include both time and Wiener process values. Generally, the grammar consists of basic operations like addition, subtraction, multiplication, and division, and mathematical functions like sine, cosine, exponential, and logarithm. It also includes the variables t and w, representing time and the Wiener process. For example, the grammar used in finding the symbolic solution for the Brownian motion was defined as follows:

```
<expr> ::= <const> * <expr>
         | (<expr> + <expr>)
         | (<expr> - <expr>)
```

```
          | (<expr> * <expr>)
          | (<expr> / <expr>)
          | np.exp(<expr>)
          | x[:, <varidx>]
          | <const>

<varidx> ::= GE_RANGE:dataset_n_vars

<const> ::= <digit>.<digit><digit>
          | -<digit>.<digit><digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

These were the main changes we made to the PonyGE2 framework so that it could work with data from stochastic differential equations. Besides these main changes, we also made some smaller adjustments during different experiments. For example, we tried different parameter values like population size, mutation rate, crossover rate, and grammar depth to see how they affected the accuracy of the results. These small changes helped us improve the performance of the model and find better settings for symbolic regression when working with noisy data.

6.3 Results and Discussion

In this section, we present the results obtained from applying GE to approximate the solution of SDEs using both known and unknown Wiener processes. We also discuss how the changes made to the PonyGE2 framework affected the performance and interpretability of the evolved symbolic models. All implementation codes, simulation results, and additional plots for the experiments discussed in this section are provided in the supplement. These resources are there to support transparency, reproducibility, and allow deeper visual inspection of the methodology and results.

6.3.1 Experiment 1: Geometric Brownian Motion (GBM)

In this experiment, we worked with the well-known Geometric Brownian Motion (GBM) model(2.15), which is often used to describe the evolution of financial variables such as stock prices. The exact analytical solution of this SDE is known and is given in (2.16). This made GBM a suitable choice for testing the ability of GE to recover meaningful symbolic expressions from simulated data. In this experiment, we considered $\mu = 0.5$ and $\lambda = 2.0$, with the number of time steps set to be $N = 2^{10}$ for first case and $N = 2^9$ for second case because of time and storage complexity. The time interval was set to be $[0, 1]$. Our grammar was the same as we defined earlier in (6.2).

Case 1 (Single Known Wiener Process): In the first case, where the same Wiener process was used to simulate X_t and to train the GE model, the symbolic expressions generated were able to closely match the original trajectory. Since the model had access to the exact Wiener path, it was easier to discover expressions that aligned well with the simulated data.

In our experiments, we obtained two symbolic approximations that performed particularly well. The first solution was:

$$X_t \approx \exp(1.85 \cdot t + 0.5 \cdot W_t) \tag{6.2}$$

and the second, slightly more complex but more accurate, was:

$$X_t \approx \exp\left(1.86(t + 0.27027W_t)\right) + 0.0818\, t^2 \exp\left(-0.0936t\right) \tag{6.3}$$

Both expressions produced good fits to the original simulated path, as shown in Figures 6.2 and 6.3. These figures display the exact solution in red and the GE-generated approximation in black. In both cases, the GE model was able to follow the shape of the true path closely.

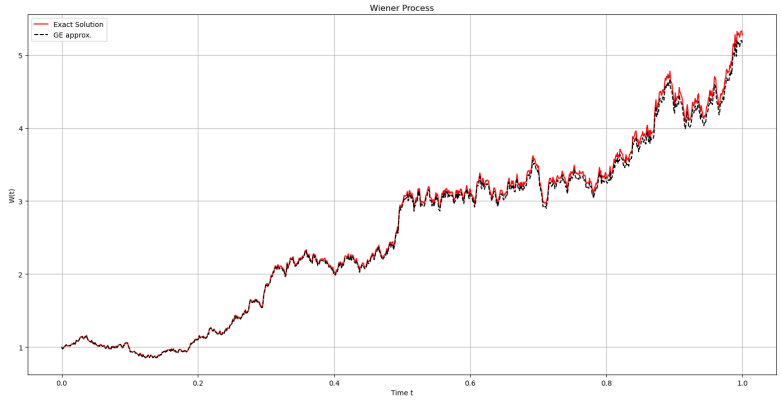


Figure 6.2: The figure shows Exact vs Symbolic solution for the first symbolic expression.

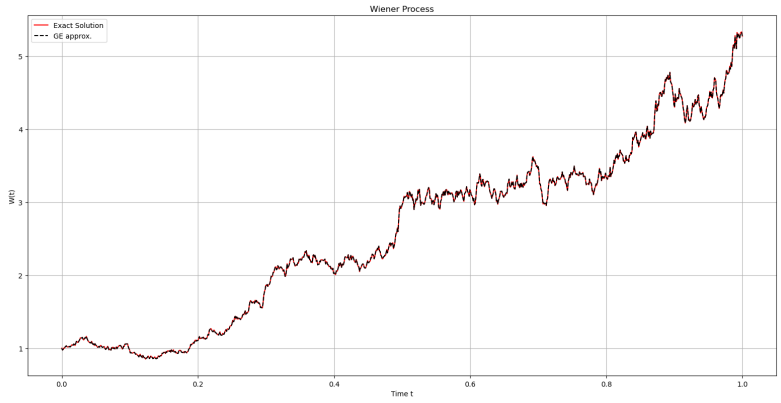
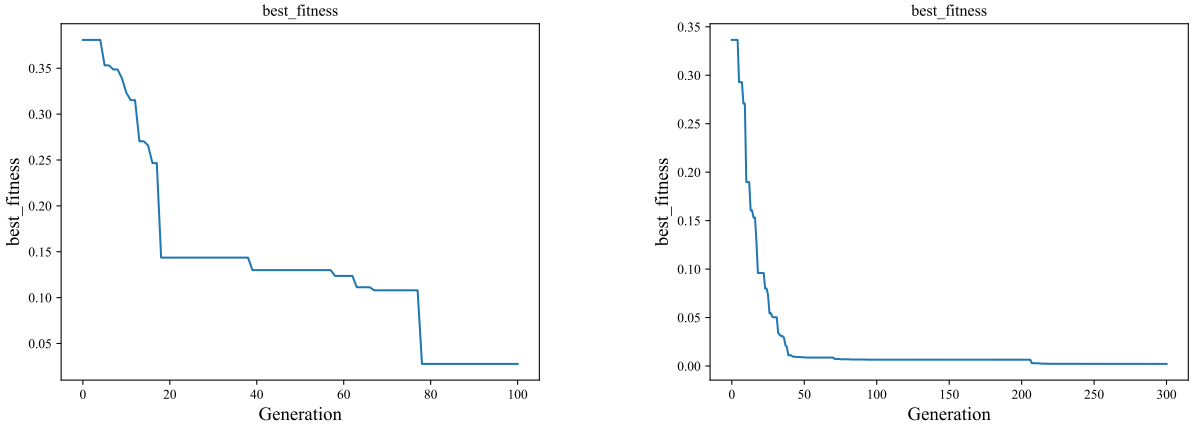


Figure 6.3: The figure shows Exact vs Symbolic solution for the second symbolic expression.

The first expression was generated using a simpler configuration: maximum tree depth was limited to 8, the model was trained for 100 generations with a population size of 500, a tournament size of 5, and a crossover probability of 0.75. The fitness value for this run was approximately 0.02776. The simplicity of this expression made it highly interpretable and easy to handle for further analysis.

The second expression, which provided a better numerical fit, was obtained using a deeper tree depth of 9, with 300 generations, a population size of 1500, a tournament size of 5, and the same crossover probability of 0.75. Although the structure was more complex, the expression achieved a much lower fitness value of approximately 0.00216.

In both runs, the codon size was set to 100,000, and the initial genome length was 200. The maximum genome length followed the default setting in PonyGE2. These configurations helped control the complexity of the generated expressions while still allowing the search space to be wide enough for



(a) First Execution (Simple Expression)

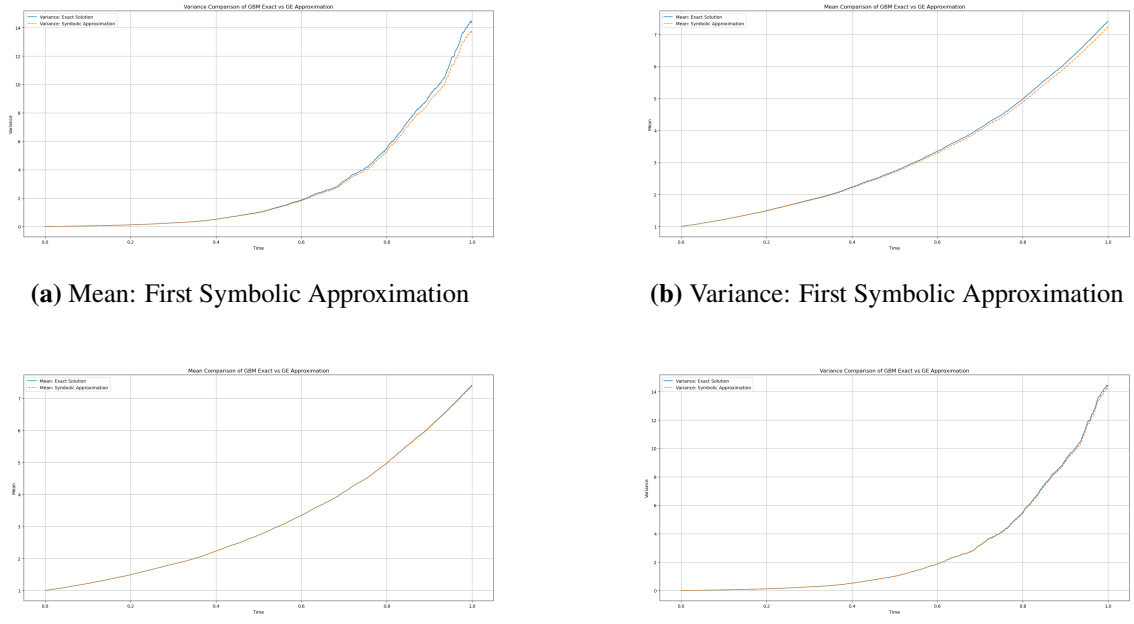
(b) Second Execution (Complex Expression)

Figure 6.4: Fitness vs. Generations: (a) Simple symbolic expression, (b) Complex symbolic expression.

exploration.

Figure 6.4 shows the best fitness value at each generation for both configurations. In the first case, the fitness values improved gradually and stabilized around generation 80. In the second case, the error decreased more significantly and continued to improve until generation 110, after which it remained steady.

Now we check the model validation, we compare the mean and variance of 1000 sample trajectories from the true process with those generated by the symbolic approximation. This comparison is shown in the Figure 6.5



(a) Mean: First Symbolic Approximation

(b) Variance: First Symbolic Approximation

(c) Mean: Improved Symbolic Approximation

(d) Variance: Improved Symbolic Approximation

Figure 6.5: Comparison of GBM Exact Solution vs Symbolic Approximations: (a, b) First symbolic form; (c, d) Improved symbolic form.

Figure 6.5(a) shows how the mean of the GBM process changes over time. The solid blue line represents the mean from the exact solution, while the dashed orange line shows the mean from the symbolic approximation. The symbolic model closely follows the exact solution, especially at the beginning, with only small differences appearing later. Figure 6.5(b) shows the variance comparison. Again, the blue line is the exact variance, and the orange dashed line is from the symbolic model. The symbolic approximation captures the overall growth pattern of variance well, though it slightly underestimates it towards the end.

For the second equation, a more accurate symbolic expression was used to approximate the GBM process. Figure 6.5(c) shows how the mean changes over time. In this case, the symbolic model almost perfectly matches the exact solution from start to end, with no visible deviation. This shows a clear improvement compared to the first version. Figure 6.5(d) shows the variance comparison. Again, the symbolic model follows the exact variance very closely throughout the entire time range. Even at later times, where small gaps appeared in the earlier model, this improved version stays well-aligned with the true variance. Overall, the second equation gives a more precise and reliable approximation of the GBM process.

Case 2 (Multiple Unknown Wiener Processes): In the second case, we tested the model with different Wiener paths that were not used to generate the original X_t. These new paths were used as input during training, and the goal was to see whether GE could still find a good approximation of the solution. The results showed that GE was able to produce expressions that followed the general shape of X_t, even though the input noise was different, as shown in Figure 6.6. The symbolic form for the SDE is given as:

$$X_t \approx \exp\left(t \cdot (\exp(W_t) - t)\right) + t$$

This showed that the method has the potential to find a good approximation using an unkown Wiener process, though the error was generally higher than in the first case.

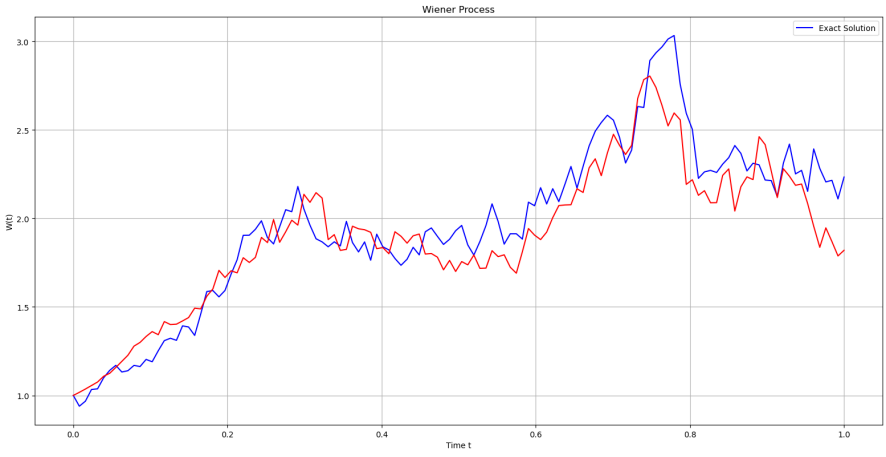


Figure 6.6: GE approximation (black) vs. exact solution (red) – Case 2: Multiple unknown Wiener processes.

For this experiment, the model is trained using a population size of 500, 250 generations, a tournament size of 3, and a maximum tree depth of 9. The number of independent Wiener paths was set to 1000. The results were promising. As shown in Figure 6.6, the approximation closely follows the original trajectory. The fitness curve over generations is shown in Figure 6.7, and it shows consistent improvement.

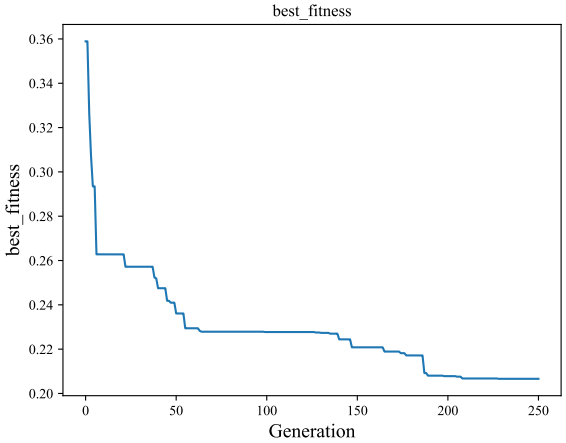


Figure 6.7: Fitness vs. generations – Case 2 experiment with 1000 unknown Wiener paths.

We also wanted to see how changes in drift (λ) and diffusion (μ) affected the model's ability to learn. To do this, we performed 90 additional experiments: 9 experiments where we varied μ and kept $\lambda = 1$, and another 9 where we varied λ and kept $\mu = 1$ with 5 times repetation of each setup. In all of these runs, we used a population size of 350, 200 generations, 1000 Wiener processes, and a tree depth of 10.

Table 6.1: Effect of increasing μ (with $\lambda = 1$) on fitness

λ	μ	Average Fit	Worst Fit	Best Fit
1	0	0.000	0.000	0.000
1	0.5	0.138	0.139	0.136
1	1	0.217	0.232	0.209
1	1.5	0.246	0.253	0.242
1	2	0.246	0.255	0.239
1	2.5	0.220	0.248	0.186
1	3	0.237	0.259	0.210
1	3.5	0.214	0.251	0.167
1	4	0.195	0.229	0.168

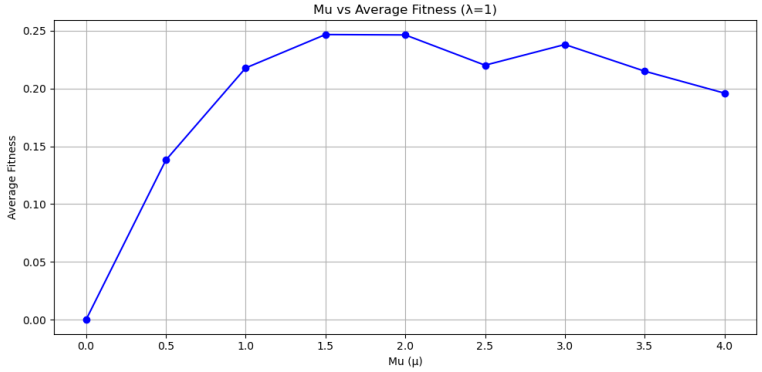
Table 6.2: Effect of increasing λ (with $\mu = 1$) on fitness

λ	μ	Average Fit	Worst Fit	Best Fit
0	1	0.153	0.159	0.148
0.5	1	0.178	0.184	0.173
1	1	0.225	0.234	0.216
1.5	1	0.268	0.287	0.249
2	1	0.319	0.339	0.268
2.5	1	0.404	0.438	0.341
3	1	0.471	0.499	0.442
3.5	1	0.597	0.652	0.548
4	1	0.725	0.744	0.709

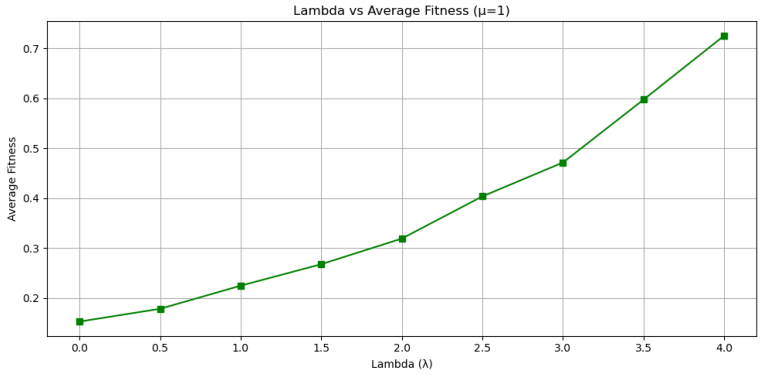
The results of these experiments are shown in Table 6.1 and Figure 6.2. Each row gives the average, worst, and best fitness values across the runs for each pair of λ and μ.

The graph in Figure 6.8(a) shows how the average fitness changes as we increase the diffusion coefficient μ, while keeping the drift coefficient $\lambda = 1$. We observed that when $\mu = 0$, the fitness was very low, which is expected since the system behaves deterministically in this case. As μ increased, the fitness error also increased due to higher noise in the data. However, beyond a certain point (around $\mu = 2$), the average fitness values started to stabilize or slightly decrease. This suggests that the GE framework is still able to learn useful patterns even when the level of stochasticity increases.

Figure 6.8(b) shows the effect of increasing the drift coefficient λ, while keeping $\mu = 1$. In this case, the fitness values increased more consistently as λ increased. This indicates that stronger drift makes the dynamics harder to approximate with symbolic expressions, possibly due to the increasing exponential growth in the solution. Overall, both graphs highlight how the model's performance is sensitive to the SDE parameters, and that the symbolic approximation becomes more challenging as the underlying process becomes more complex.



(a) Effect of changing diffusion coefficient μ (with $\lambda = 1$)



(b) Effect of changing drift coefficient λ (with $\mu = 1$)

Figure 6.8: Influence of parameters on average fitness: (a) Variation in μ, (b) Variation in λ.

6.3.2 Experiment 2: Ornstein–Uhlenbeck (OU) Process

In this experiment, we focused on the Ornstein–Uhlenbeck (OU) process (2.17), a mean-reverting stochastic process commonly used in financial modeling and physics. While the OU process has an exact analytical solution (2.18), it includes a stochastic integral that cannot be solved in closed form. Because of this, we simulated the process numerically using the Euler–Maruyama method. We used the following parameters to simulate the OU process:

$$\theta = 2.0, \quad \mu = 1.0, \quad \sigma = 0.3, \quad X_0 = 0.5, \quad T = 1.0, \quad N = 2^9$$

Here, θ controls the rate of mean reversion, μ is the long-term mean, σ is the volatility, X_0 is the initial condition, T is the final time, and N is the number of time steps. Next, for the symbolic regression, we implemented the GE framework using the following evolutionary parameters:

$$\text{Population size} = 1500, \quad \text{Generations} = 300, \quad \text{Tournament size} = 5$$

The symbolic expression recovered by GE was:

$$X_t \approx 0.10 \cdot \Big((1.64 - t) \cdot e^{1.99} \cdot t + W \cdot (3.21 - t) + 4.90 \Big)$$

Although this is not the exact solution of the stochastic differential equation, but it follows the pattern of the process quite well. It includes both the effect of time t and the Wiener process W, which are key parts of the original equation. This shows that the symbolic regression method was able to build a clear and useful expression that gives a good idea of how the system behaves over time.

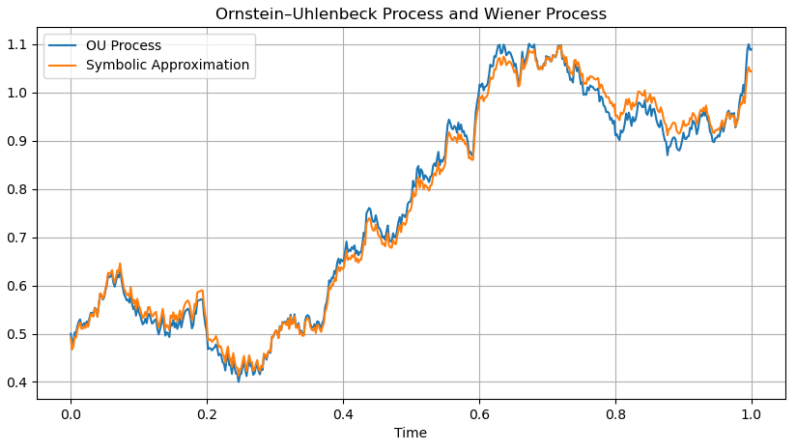


Figure 6.9: GE approximation (orange) vs. true OU process (blue) using Euler–Maruyama simulation.

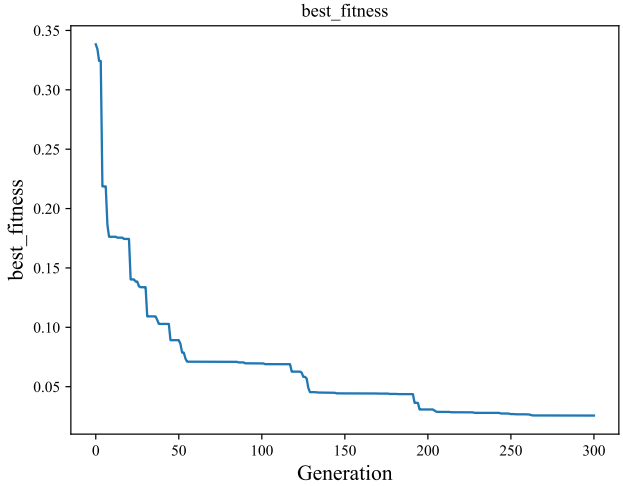
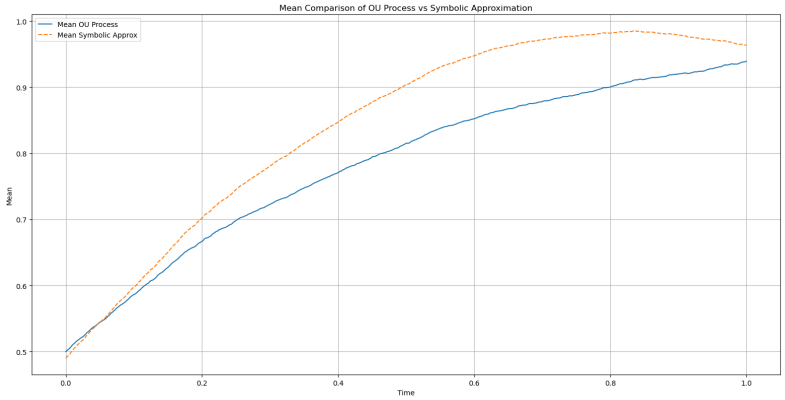


Figure 6.10: Best fitness value over generations – OU process experiment.

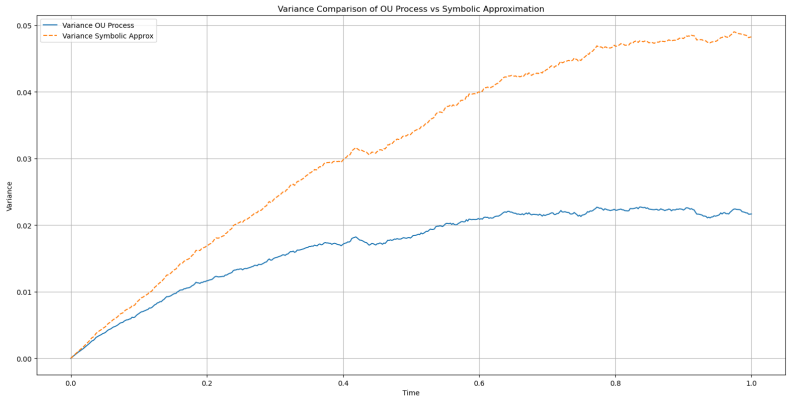
Figure 6.9 shows the comparison between the true OU trajectory and the GE-generated symbolic solution. The framework successfully produced a best-fit solution that almost accurately approximated the trajectory of the original process and tracked the mean-reverting behavior over time.

The best fitness values across generations are shown in Figure 6.10. The graph shows that the model continuously improved its approximation throughout the training process, with the best fitness score decreasing steadily until it stabilized near the final generations with a fitness value of 0.02566.

To validate the model, we compared the mean and variance of 1000 sample trajectories from the true OU process with those generated by the symbolic approximation. This comparison is shown in the Figure 6.11



(a) Mean Comparison



(b) Variance Comparison

Figure 6.11: Comparison of OU Process vs Symbolic Approximation: (a) Mean, (b) Variance.

The Figure 6.11(a) presents **mean** over time. The blue line shows the actual mean of the OU process, and the dashed orange line represents the symbolic model's mean. The symbolic regression closely follows the trend of the mean-reverting behavior and remains within a reasonable range of the true mean. Figure 6.11(b) shows the **variance** over time. The approximation tracks the qualitative growth in variance but tends to overestimate the magnitude as time progresses.

6.3.3 Experiment 3: Cox–Ingersoll–Ross (CIR) Process

In this experiment, we studied the Cox–Ingersoll–Ross (CIR) process (2.19), which is commonly used in financial mathematics to model interest rates and other mean-reverting processes. A key challenge with the CIR process is that it does not have a closed-form analytical solution. Because of this, we simulated the trajectory using the Euler–Maruyama method. We used the following parameters to simulate the process:

$$\kappa = 0.5, \quad \theta = 1.0, \quad \sigma = 0.1, \quad X_0 = 0.5, \quad T = 1.0, \quad N = 1000$$

The symbolic regression task was performed using the following evolutionary settings:

$$\text{Population size} = 1500, \quad \text{Generations} = 300, \quad \text{Tournament size} = 5$$

After training, GE returned one of the best symbolic expressions as:

$$X_t \approx 0.09\,(5.64 + 2.2636\,t + 0.8587\,W_t)$$

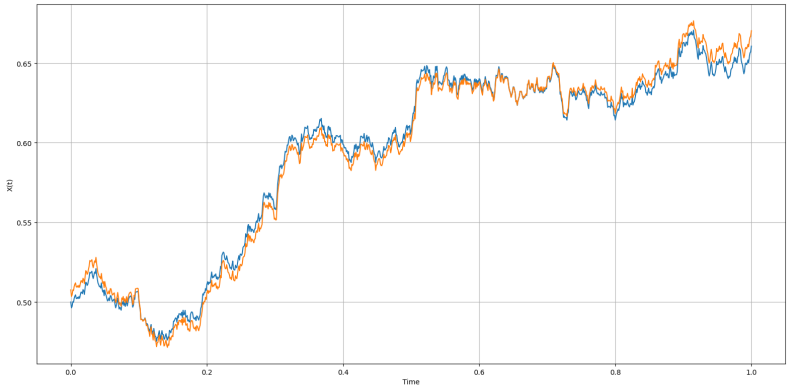


Figure 6.12: GE approximation (orange) vs. simulated CIR process (blue).

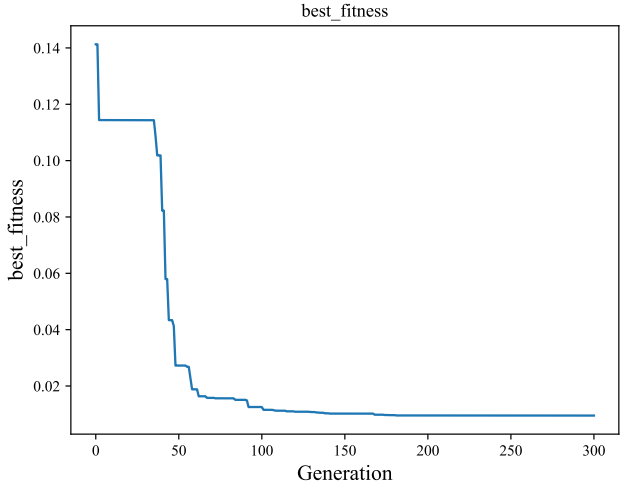
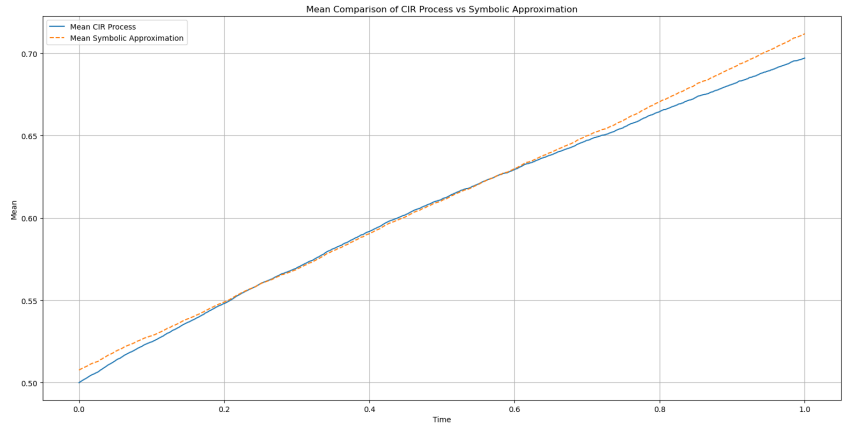


Figure 6.13: Best fitness value over generations – CIR process experiment.

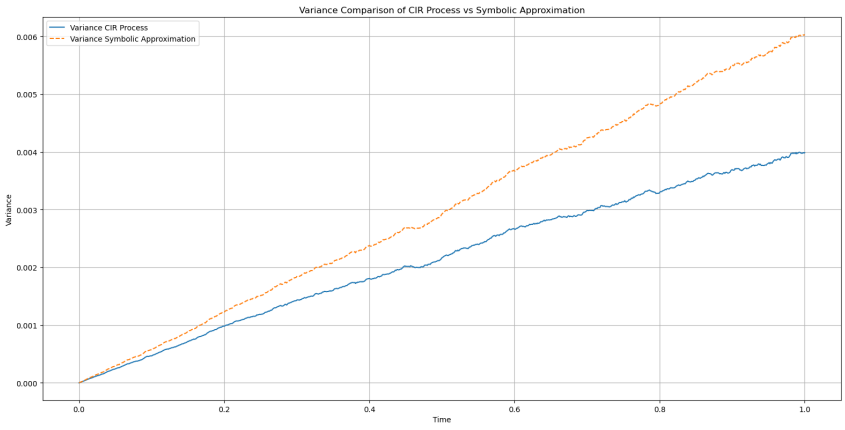
Figure 6.12 shows a visual comparison between the symbolic solution generated by GE and the simulated CIR process generated by Euler Euler-Maryama scheme. It can be clearly seen from the figure that the technique was able to follow the pattern closely.

The fitness graph in Figure 6.13 shows the best error at each generation. There was a clear improvement during the early generations, with the fitness stabilizing after generation 200. The results indicate that GE was able to find a strong approximation for the CIR process, even though the exact solution is not available.

Finally, to evaluate the validity of the symbolic approximation for the Cox–Ingersoll–Ross (CIR) process, we analyzed both the mean and variance based on 1000 simulated trajectories from the true process and its symbolic solution.



(a) Mean Comparison of CIR Process vs Symbolic Approximation



(b) Variance Comparison of CIR Process vs Symbolic Approximation

Figure 6.14: Comparison of CIR Process vs Symbolic Approximation: (a) Mean, (b) Variance.

Figure 6.14(a) presents the evolution of the mean over time. The two curves are closely aligned throughout the time interval, indicating that the symbolic model effectively captures the expected value dynamics of the CIR process. Figure 6.14(b) illustrates the variance behavior. The variance from the true process grows more gradually compared to the symbolic model, which shows a slightly steeper

increase. While the symbolic approximation slightly overestimates the spread, especially in later stages, it still tracks the general trend well.

6.4 Limitations and Future Work

As with all computational studies, the experiments and setup could be further expanded. The relatively high computational cost, particularly in the case of experiments involving the unknown Wiener process, posed a constraint. For the known Wiener process, each run took approximately 10 to 15 minutes on average. In contrast, experiments using the unknown Wiener process based on $1,000$ trajectories with a time discretization step of 2^{-7} required nearly 12 hours per run. These computational demands did not permit us to conduct a more extensive computational study, especially with respect to the OU and CIR processes. Additionally, running the proposed framework on more diverse SDEs would be interesting.

Another limitation of this study is the extent of the grammar used in the GE and the overall computational setup (population size, number of iterations, tree depth, etc.). These should be further investigated, and the appropriate settings, utilizing some sort of hyperheuristic, such as SMAC [9] or irace [17], could be found. However, these types of studies are extremely computationally demanding.

7 Conclusion

In this thesis, we investigated the use of Explainable Artificial Intelligence (XAI) approaches to solve SDEs, with a focus on symbolic regression using Grammatical Evolution. Starting from a review of the GPAD framework by [16], we analyzed its strengths and limitations. Although their method provided a symbolic approach to estimating drift and diffusion terms, we faced several challenges in reproducing their results due to incomplete MATLAB codes. This motivated us to develop a modified, Python-based framework using the PonyGE2 library.

Unlike the original GPAD approach, which estimates components of the SDE separately, our method aims to discover a single symbolic expression that captures the entire solution trajectory. We applied this approach to several standard SDEs, including Geometric Brownian Motion, the Ornstein–Uhlenbeck process, and the Cox–Ingersoll–Ross model. Simulated data was used to train the GE framework under different settings, with both known and unknown Wiener paths, to evaluate the accuracy and reliability of the method. The unknown process case was only considered for the first example. A similar analysis could be extended to the remaining SDEs; however, since explicit symbolic solutions are not available for those, our primary focus was on generating symbolic expressions that accurately approximate the full trajectory.

Our results show that GE can successfully generate interpretable symbolic solutions that approximate the behavior of SDEs. When the exact Wiener process is known, GE is able to recover expressions that closely match the true solution. Even when the noise input is unknown or varied, the method still finds reasonable approximations. This suggests that symbolic regression using GE offers a promising alternative to typical numerical solvers, especially in situations where interpretability is important.

Overall, this work bridges the gap between data-driven modeling and stochastic analysis. By combining simulation techniques with symbolic regression, we present a transparent, flexible framework for approximating SDE solutions. Future work can focus on extending this method to high-dimensional systems, noisy real-world data, and adaptive grammars for better performance.

References

[1] *Adaptive Trading Using Grammatical Evolution*, pages 193–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[2] Muhammad Sarmad Ali, Meghana Kshirsagar, Enrique Naredo, and Conor Ryan. Towards automatic grammatical evolution for real-world symbolic regression. In *IJCCI*, pages 68–78, 2021.

[3] Ludwig Arnold. Stochastic differential equations: theory and applications. 1974.

[4] Jaume Bacardit, Alexander EI Brownlee, Stefano Cagnoni, Giovanni Iacca, John McCall, and David Walker. The intersection of evolutionary computation and explainable ai. In *Proceedings of the Genetic and Evolutionary Computation conference companion*, pages 1757–1762, 2022.

[5] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. Ponyge2: Grammatical evolution in python. *arXiv preprint arXiv:1703.08535*, March 2017.

[6] J. Fink. Automatic differentiation for matlab. MATLAB Central File Exchange, 2007. `https://www.mathworks.com/matlabcentral/fileexchange/`.

[7] Amit Ganatra, Brijeshkumar Panchal, Devarshi Doshi, Devanshi Bhatt, Jesal Desai, Bijal Talati, Neha Soni, and Apurva Shah. *Introduction to Explainable AI*, pages 1–31. July 2024.

[8] Thomas Gard. Introduction to stochastic differential equations. *Journal of the American Statistical Association*, 84, 12 1989.

[9] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and intelligent optimization: 5th international conference, LION 5, Rome, Italy, January 17-21, 2011*, pages 507–523. Springer, 2011.

[10] N. Ikeda and S. Watanabe. *Stochastic Differential Equations and Diffusion Processes*. North-Holland/Kodansha, 1989.

[11] Kiyosi Itô. *On stochastic differential equations*. Number 4. American Mathematical Soc., 1951.

[12] Peter Kloeden and Eckhard Platen. *The Numerical Solution of Stochastic Differential Equations*, volume 23. Springer, 2011.

[13] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.

[14] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4:87–112, 1994.

[15] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer, 2013.

[16] W. J. A. Lobão, M. A. C. Pacheco, D. M. Dias, and A. C. Alves Abreu. Solving stochastic differential equations through genetic programming and automatic differentiation. *Engineering Applications of Artificial Intelligence*, 68:110–120, 2018.

[17] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[18] Bernt K. Oksendal. *Stochastic Differential Equations: An Introduction with Applications*. Springer, 5 edition, 2002.

[19] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[20] Conor Ryan, Michael O'Neill, and JJ Collins. Introduction to 20 years of grammatical evolution. *Handbook of grammatical evolution*, pages 1–21, 2018.

[21] Zeev Schuss. Theory and applications of stochastic differential equations. 1980.

[22] S. Silva. *Genetic Programming Toolbox for MATLAB*. Universidade Nova de Lisboa, PO Box 127, 2780-156 Oeiras, Portugal, 2009.

[23] Ioannis G Tsoulos, Alexandros Tzallas, and Dimitris Tsalikakis. Nnc: A tool based on grammatical evolution for data classification and differential equation solving. *SoftwareX*, 10:100297, 2019.