University of Birmingham

School of Engineering

and

Birmingham Business School

TITLE

Regression Analysis with the Bee's Algorithm

Dhanashish Prakash

ID: 255054

Project submitted in partial fulfilment of the requirements for the degree of

PG dip. Advanced Engineering Management (Project Management)

August 2024

ABSTRACT

This dissertation focuses on the application of the Bees Algorithm, a nature-inspired optimization method, to the problem of regression analysis. Specifically, the study investigates how this algorithm can be used to optimize the parameters of polynomial and spline models in both uni-variate and multi-variate regression tasks. The aim is to enhance the accuracy of the regression models by minimizing error metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE).

In the experimental phase, the Bees Algorithm was implemented using Python, and its performance was evaluated on multiple regression benchmarks. The algorithm was applied to optimize the parameters of different polynomial models and splines across various datasets. A comparison was made between the fitness values obtained using different error metrics.

Results indicate that the Bees Algorithm is effective in balancing exploration and exploitation, providing optimized regression models. The findings show that the algorithm is particularly efficient when dealing with complex, high-dimensional data and non-linear relationships.

In conclusion, the Bees Algorithm proves to be a viable method for optimizing regression models.

Keywords: Bees Algorithm, regression analysis, polynomial models, splines, MSE, RMSE, MAE, optimization

Number of words in text: 4231

Number of figures and tables : 10 Figures and 6 Tables Equivalent words: 761

Total word count: 4992

Acknowledgements

I would like to express my deepest appreciation to my project supervisor, Marco Castellani, for the guidance, encouragement, and support that was invaluable. Their valuable suggestions and expert advice really helped in the direction and shape of this work. In addition, I would also like to express my gratitude to my colleagues and course mates for their constructive feedback and discussions, which have contributed toward the development of this research. Lastly, but not least, I appreciate the support and tolerance of all my family and friends in all these years of my studies. Their belief in me was something that really motivated me.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Overview

Regression analysis is a fundamental tool used to explore the relationships between variables. By modelling the dependency between one or more independent variables and a dependent variable, regression provides insights that are crucial for decision-making, system control, and predictive analysis. Typically, predefined models, such as polynomials or splines, are fitted to observed data through the manipulation of parameters, with the goal of minimizing the error between the model's predictions and the actual data points. This process is inherently an optimization problem, where identifying the best set of parameters is critical to the accuracy of the model.

Optimization algorithms are essential in this context, as they help find the optimal parameters that lead to the best model fit. Among the variety of optimization techniques, nature-inspired algorithms have gained attention due to their ability to explore complex search spaces efficiently. One such algorithm, the Bees Algorithm, this dissertation investigates the application of the Bees Algorithm to regression problems, focusing on its ability to optimize polynomial models and splines in both uni- and multi-variate contexts.

The aim of this project is twofold: first, to implement the Bees Algorithm for optimizing the parameters of polynomial and spline regression models, and second, to analyse its performance across different regression benchmarks. Through this approach, the project explores.

(1) how the Bees Algorithm can enhance the accuracy and reliability of regression models by effectively minimizing the error between predicted and actual data points.
(2) What are the key factors influencing the performance of the Bees Algorithm in uni and multi-variate regression problems?
(3) Can the Bees Algorithm improve the accuracy of polynomial and spline regression models across different datasets and benchmarks?

1.2. Bees Algorithm Overview

The Bees Algorithm is an optimization technique that has been derived from the foraging behaviour of honeybees. In nature, honeybees range over large areas, looking for the most promising patches of flowers, then communicate where and how good these are to the rest of the colony. This involves balancing acts between exploration, searching for new food sources, and exploitation, harvesting from the best-known sources. The Bees Algorithm emulates the way nature solves optimization problems, thus making it especially useful for complex and multimodal optimization tasks (Castellani & Pham, 2009).

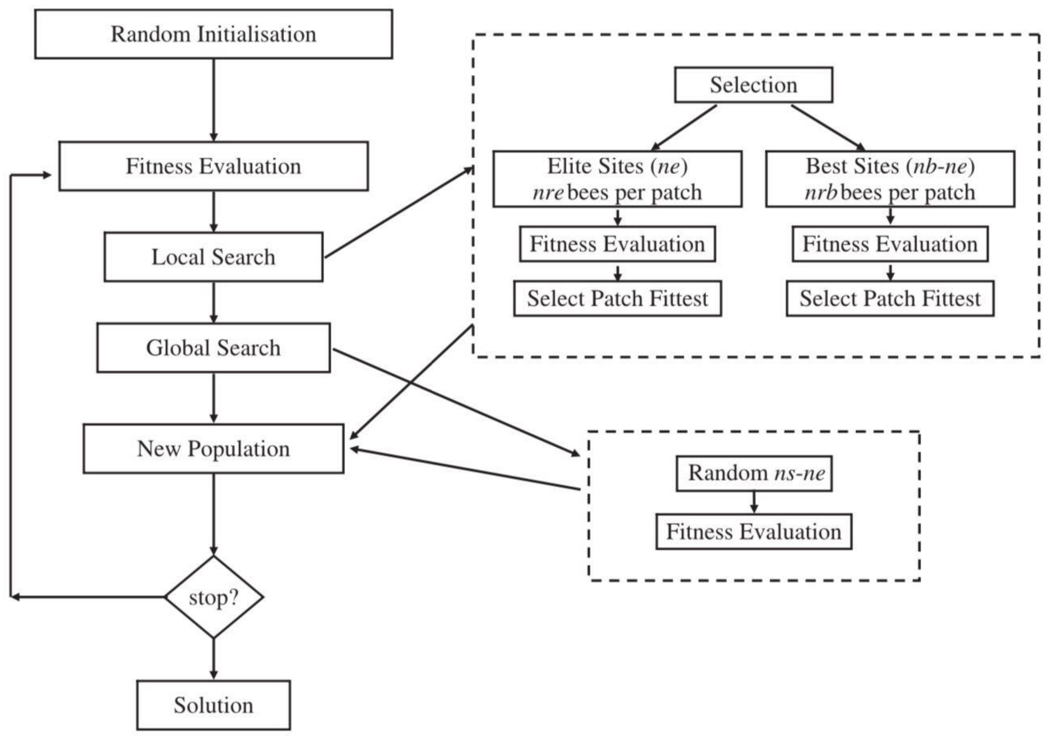


Figure 1 Flow Chart of Bees Algorithm (Castellani & Pham, 2009)

1.3. Aim: Regression Analysis with the Bees Algorithm

Regression analysis aims to estimate the relationships between a dependent variable and one or more independent variables from experimental data. Regression is a fundamental step in engineering applications such as data analysis and system control. Typically, a pre-set model (e.g. a polynomial) is fitted to the data distribution by manipulating its parameters. Regression thus boils down to a parameter optimisation problem, where the goal is to find the set of parameters that minimises the distance between the data points and the model (e.g. polynomial) output. This project entails the application of the Bees Algorithm to the optimisation of different types of polynomial models and splines in uni- and multi-variate regression problems. Practical work will include the software implementation of the Bees Algorithm and the regression models, and the analysis of their performance on chosen regression benchmarks.

1.4. Objectives

The objective is to operate the Bees Algorithm with the purpose of optimizing and analysing polynomial and spline regression models and its effectiveness. The project particularly aims to:

1. Apply the Bees Algorithm on the Polynomial and Spline regression models and analyse the results.
2. Assess the capability of the algorithm to balance exploration and exploitation while optimizing regression models.
3. Evaluate the computational efficiency and analyse accuracy of the Bees Algorithm in regression problems.

2. Literature Review

2.1. Polynomial and Spline Regression:

Polynomial Regression: Polynomial regression is an advanced form of linear regression used to model relationships between a dependent variable ‘y’ and one or more independent variables ‘x’, using a polynomial function. Unlike simple linear regression, which assumes a straight-line relationship, polynomial regression can capture more complex, curved relationships by including powers of the independent variable up to a certain degree (Montgomery, Peck, & Vining, 2012). The model looks like this:

$$y = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0 + \epsilon$$

Where:

Y is the dependent variable,

x is the independent variable,

c_0, c_1,….. c_n are the polynomial coefficients that need to be optimized,

ϵ represents the error or noise.

find the best coefficients c_0, c_1,…. c_n that make the model fit the data as closely as possible.

Spline Regression: Spline regression is another method for modelling complex relationships, but it does so by fitting different polynomial curves to different segments of the data. These segments are joined smoothly at certain points called knots. The most used spline is the cubic spline, which uses third-degree polynomials for each segment (Hastie, Tibshirani, & Friedman, 2009). The model can be represented as:

$$S(x) = \begin{cases} P_1(x), & for\ x_0 \leq, t_1 \\ P_2(x), & for\, t_1 \leq x \leq\ t_2 \\ & \cdots \\ P_n(x), & for\, t_{n-1}\ \leq x \leq x_n \end{cases}$$

Where:

$S(x)$ is the spline function.

$P(x)$ is a polynomial in the interval between two successive knots.

t_1, t_2,.....t_{n-1} are the knot positions (where the piecewise polynomials join).

x_0, x and x_n are the boundaries of the data.

General Spline Form:

For a cubic spline, each $P_i(x)$ is a cubic polynomial of the form:

$$P_i(x) = a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3$$

Where:

a_i, b_i, c_i, d_i are the coefficients of the cubic polynomial for the i^{th} segment.

t_i is the knot location, and x is the independent variable.

Thus, the spline is a collection of cubic polynomials that are smoothly connected at the knots.

A spline is a method that fits several piecewise polynomial functions to the data set, whereby the polynomials meet at some specific points or join to form a unique curve. Such splines provide a smooth and flexible fit to the data, providing more optimal handling of nonlinear relationships and without the risk of overfitting in high-degree polynomials (Boor, 2001).

The cubic spline is the most common spline, where each piece is a third-degree polynomial. The pieces are joined in a way such that the first and second derivatives of the spline at the knots are continuous, thus making the segments continuous (Boor, 2001).

2.2. Optimization technique of the Bees Algorithm

Especially selecting model parameters that will bring the error between predicted and observed data to a minimum, optimizing algorithms are important in regression analysis. However, the main problem with gradient descent is the trapping of local minima in complex search spaces.

How the Bees Algorithm Works

Initialization Phase: The algorithm begins by generating an initial population of "scout bees," each representing a potential solution to the problem. These initial solutions are typically generated randomly.

Fitness Evaluation Using MSE, RMSE, or MAE: For each bee, calculate the error using one of the three fitness functions (MSE, RMSE, or MAE) by comparing the model's predictions with the actual data. This error metric acts as the bee's fitness value, which the algorithm will try to minimize.

Local Search (Exploitation): The algorithm selects the best-performing bees and assigns more bees to explore the area around these solutions in detail. This step is about fine-tuning the best solutions.

Global Search (Exploration): Meanwhile, other bees continue to search the broader space, looking for new, potentially better solutions. This prevents the algorithm from getting stuck in a suboptimal area.

Selection and Update: The best solutions from both local and global searches are selected to form the next generation of scout bees.

Termination: This process repeats until the algorithm meets a stopping criterion, such as reaching a maximum number of iterations or achieving a desired level of accuracy.

2.3. Identification of Gaps in the Literature

Even though the Bees Algorithm has been utilized on a wide variety of optimization problems, it has not been much applied in the context of optimal design of polynomial and spline regression models. Most of the studies that exists mainly gave attention to its application in machine learning models or engineering design problems, while only a few considered its potential for regression analysis in data-intensive fields.

In fact, existing literature essentially compares the Bees Algorithm with state-of-the-art classical optimization techniques and does not really serve to bring out its full potential in advanced regression models such as splines, where the complexity of the model could gain significantly well from the robust search capabilities of the algorithm.

This project's objectives will attempt to close these gaps by:

1. Utilization of the Bees Algorithm to optimize both polynomial and spline regression models.
2. Performance comparison of the Bees Algorithm with traditional optimization methods by using regression analysis.
3. Investigating how effective the algorithm is in handling the complexities of spline regression, which tend to complicate traditional methods.

3. Research methodologies

3.1. Implementation of the Bees Algorithm

The Bees Algorithm was implemented to optimize the parameters of a third-degree polynomial regression model. The goal was to minimize error metrics—specifically Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE)—by systematically adjusting the polynomial's coefficients. The implementation involved several key steps, including the initialization of the algorithm, the selection of appropriate fitness functions, the variation of algorithm parameters, and the evaluation of the algorithm's performance.

Polynomial Regression Model Setup

The target model for optimization was a third-degree polynomial, represented by the following equation:

$$y = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0 + \epsilon$$

To simulate the optimization process, synthetic data was generated using known coefficients. The goal of the Bees Algorithm was to minimize the error between the model's predictions and the actual data points by optimizing the polynomial coefficients.

Initialization of the Bees Algorithm

The Bees Algorithm was initialized with an initial population of bees, each representing a potential set of polynomial coefficients. The initial population was generated randomly within predefined ranges for each coefficient. The following parameters were defined for the Bees Algorithm:

1. **Ns (Number of Scout Bee**s): The number of initial random solutions generated by the algorithm. This determines how extensively the solution space is initially explored.
2. **Ne (Number of Elite Bees):** The best-performing bees that are selected to perform more detailed local searches in the neighbourhood of their solutions.
3. **Nb (Number of Best Sites):** The number of promising regions (sites) around which further exploration is focused. The algorithm assigns recruited bees to explore these regions.
4. **Nre (Number of Recruited Bees for Elite Sites):** The number of bees recruited to explore around the best elite solutions.
5. **Nrb (Number of Recruited Bees for Best Sites):** The number of bees assigned to search around the best non-elite solutions.

6. **Ngh (Neighbourhood Size):** This parameter defines the radius around the selected best sites, within which recruited bees search for better solutions.
7. **Stlim (Stagnation Limit):** The number of iterations allowed without improvement before abandoning the current search direction.

Table 1 Bees algorithm Parameters (Castellani & Pham, 2009)

Parameters	paramet ers
Number of Scout bees	Ns
Number of Elite bees	Ne
Number of best Sites	Nb
Number of Recruited bees for elite sites	Nre
Number of Recruited bees for best sites	Nrb
Initial size of neighbourhood	Nrh
Limit if stagnation cycles	Stlim

3.2. Fitness Function Definition:

Three different fitness functions were used to evaluate the performance of the algorithm and guide the optimization process:

MSE (Mean Squared Error):

MSE is a commonly used metric to evaluate the accuracy of a regression model. It measures the average of the squared differences between the predicted values (ỹ) and the actual values (y) (Hodson, 2022).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Where:

y_i is the actual value,

$\tilde{y}_i$ is the predicted value,

n is the number of data points.

Characteristics of MSE:

Penalizes larger errors more severely than smaller errors due to the squaring of the differences.

How it Works: For each bee (set of parameters), calculate the MSE between the model's predictions and the actual values. The bee with the lowest MSE has the best fitness.

MSE (Root Mean Squared Error):

RMSE is the square root of the MSE. It is also a measure of the difference between predicted and actual values, but since it takes the square root, it brings the units of the error back to the same units as the target variable, making interpretation easier (Hodson, 2022).

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2}$$

Characteristics of RMSE:

Like MSE, it penalizes large errors more than smaller ones.

Because it's in the same units as the original data, it is easier to interpret in terms of practical relevance.

How it Works: Like MSE, but since RMSE takes the square root, it provides a fitness value with the same units as the target variable, making it more interpretable.

MAE (Mean Absolute Error):

MAE measures the average absolute difference between the predicted and actual values, without squaring the errors. It represents the average magnitude of errors in a set of predictions (Hodson, 2022).

$$MAE = \frac{1}{n}|y_i - \tilde{y}_i|$$

Characteristics of MAE:

Does not penalize larger errors as severely as MSE or RMSE since it uses absolute values instead of squares.

How it Works: MAE calculates the average magnitude of the errors without squaring them, which means it treats all errors equally. The bee with the lowest MAE has the best fitness.

Parameter Variation and Optimization

To investigate the impact of different algorithm configurations, several key parameters of the Bees Algorithm were systematically varied, including the number of scout bees (Ns), elite bees (Ne), and the neighbourhood size (Ngh). Each parameter configuration was tested across multiple runs of the algorithm, and the best-performing solutions were recorded based on the fitness function results (Castellani & Pham, 2009).

The process involved:

1. Initializing the Bees Algorithm with a specific set of parameters.
2. Running the algorithm until convergence or until the stagnation limit (Stlim) was reached.
3. Calculating the fitness function (MSE, RMSE, or MAE) for each solution (set of polynomial coefficients) in the population.
4. Selecting the best-performing bees and recruiting additional bees to search around the elite and best solutions.
5. Updating the population and continuing the search process.
6. Recording the best error values (MSE, RMSE, or MAE) achieved for each parameter configuration.

The algorithm was tested across multiple configurations, and the results for each configuration were compiled into The Regression analysis of bees algorithm for same Polynomial Equation is implemented on different values to analyse their best values and compare with each other.

Table 5 Changing Parameters of bee's algorithm in different fitness functions on a same polynomial equation for 100 iterations., which presents the best error values for each fitness function based on different parameter settings.

3.3. Testing for best parameters in the Bees Algorithm

To evaluate how well the Bees Algorithm optimizes regression models, the following steps are taken:

1. **Dataset Selection:** Use both synthetic datasets (where the underlying function is known such as Log, Sin and Tan) and real-world datasets to test the algorithm's effectiveness.
2. **Parameter Sensitivity Analysis:** Evaluate how sensitive the algorithm's performance is to different settings, such as the number of bees or iterations.

3. **Statistical Analysis:** Conduct statistical tests to ensure that any performance improvements are significant and not due to chance.

4. Results, Data Analysis and Discussions

4.1. Applying Bee’s Algorithm

Applying the above research methodology in python to write a code that implements bee’s algorithm (code and output in Appendices) and perform regression on polynomial and spline models. The results of regression are compared with Function(Log , Sin and Tan) to get the possible values from different range D(Degree)(range (3, 6)) , Ns[30, 50, 100], Ne [5, 10, 15], Nb [5, 10, 15], Nre [3, 5, 10], Nrb[2, 3, 5], Nrh[0.1 , 0.01] and Stlim[10, 10, 30]. The graphs below represent the error between the actual data and data of bee’s algorithm on given parameters.

Bm. Fn.	**Fit. Fn.**	**D**	**Ns**	**Ne**	**Nb**	**Nre**	**Nrb**	**Nrh**	**Stlim**
Log	MSE	3	100	15	15	10	3	0.1	30
Sin	MSE	5	50	15	15	10	2	0.1	30
Tan	MSE	5	100	5	5	10	3	0.1	30
Log	RMSE	3	50	5	10	10	5	0.1	30
Sin	RMSE	4	50	15	15	10	3	0.1	30
Tan	RMSE	5	50	10	15	10	3	0.1	20
Log	MAE	3	50	5	10	10	5	0.1	30
Sin	MAE	4	100	5	10	5	5	0.1	20
Tan	MAE	5	30	5	15	10	2	0.1	30

Table 2 parameter benchmarking of log, sin and tan on fitness functions and obtaining best setup(Benchmark Function(Bm. Fn.), Fitness Function(Fit. Fn.))

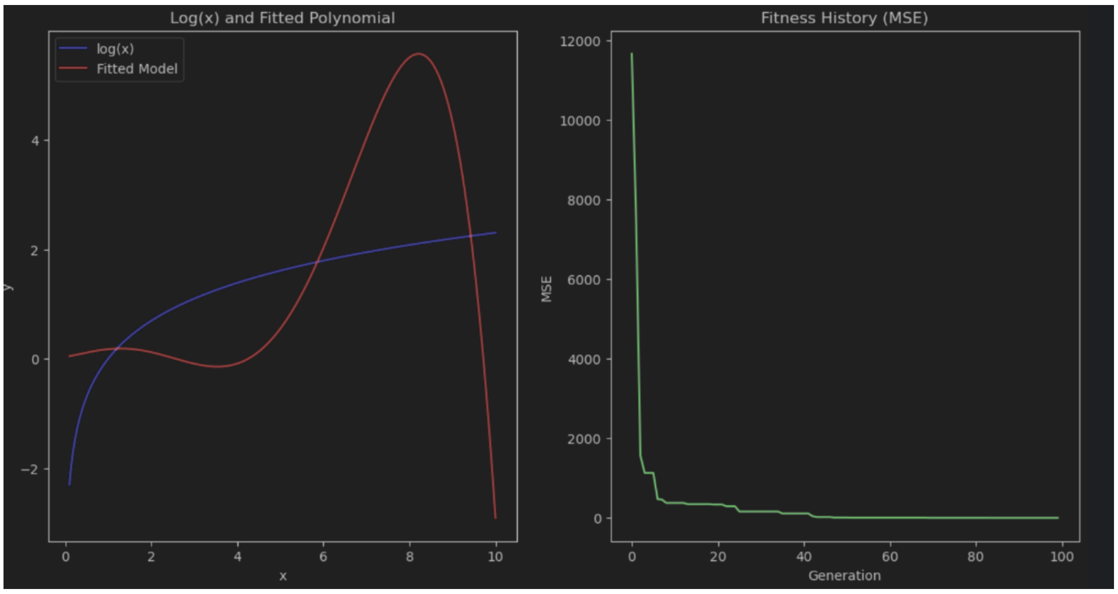


Figure 2 comparison of Log x and output of bees algorithm using fitness function MSE

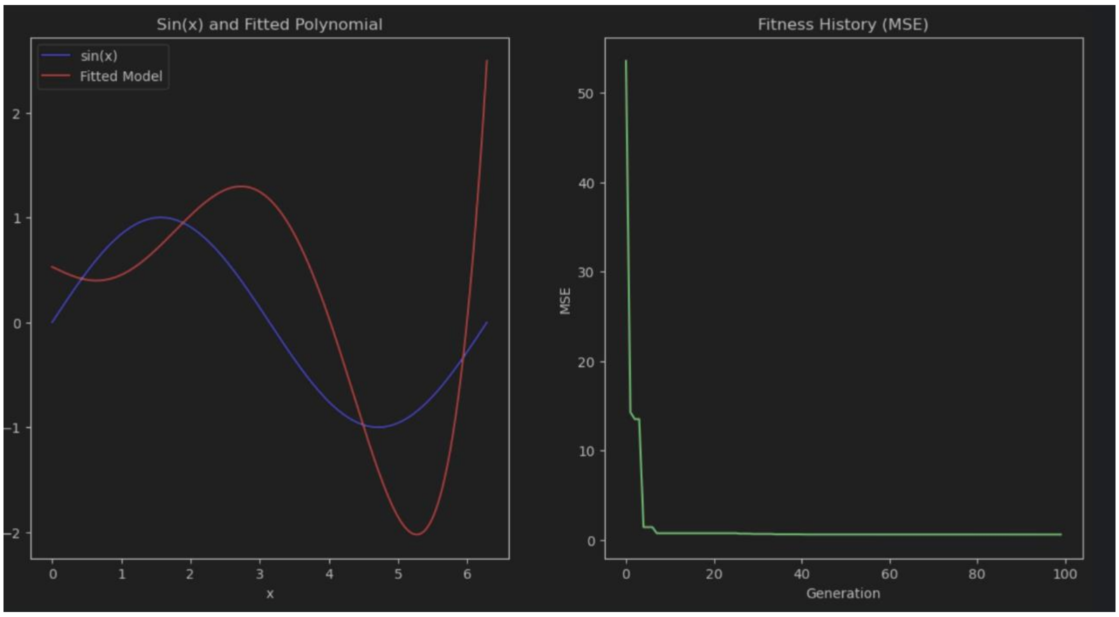


Figure 3 comparison of Sin x and output of bees algorithm using fitness function MSE

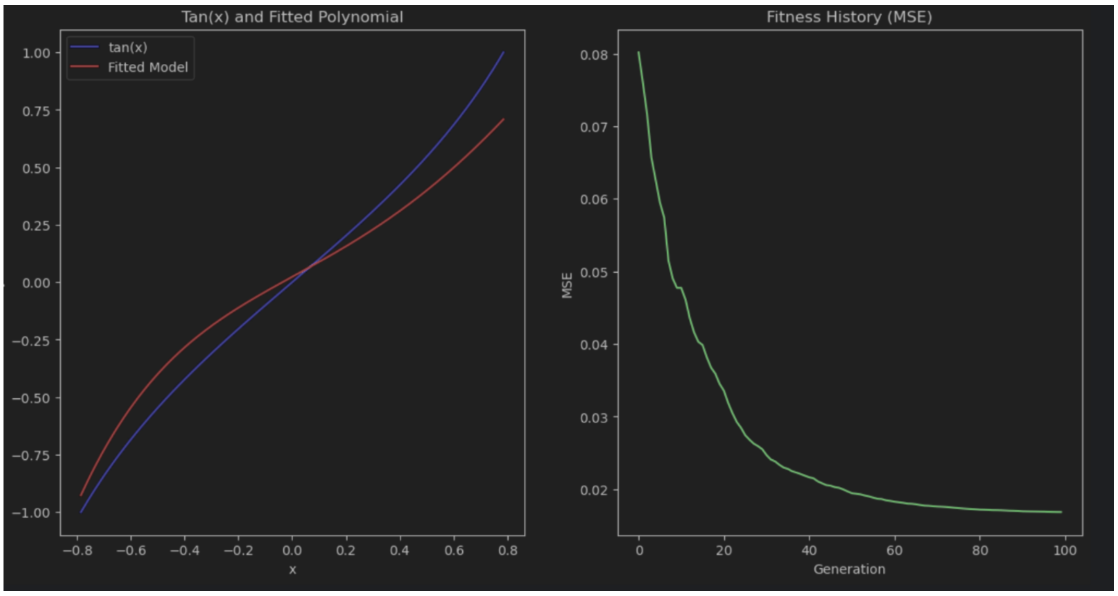


Figure 4 comparison of Tan x and output of bees algorithm using fitness function MSE

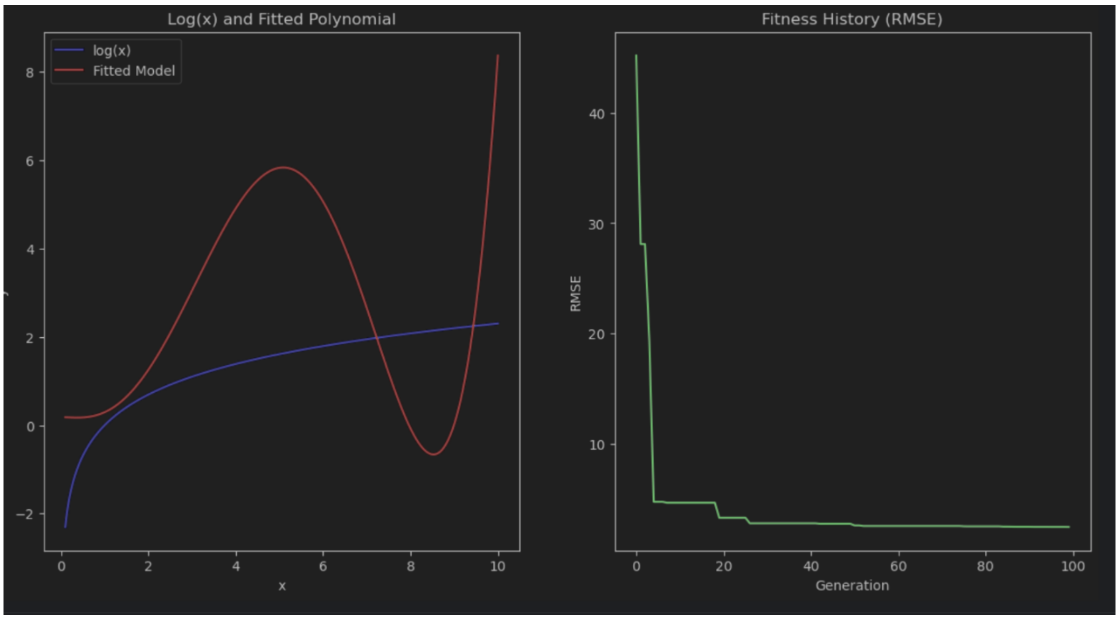


Figure 5 comparison of Log x and output of bees algorithm using fitness function RMSE

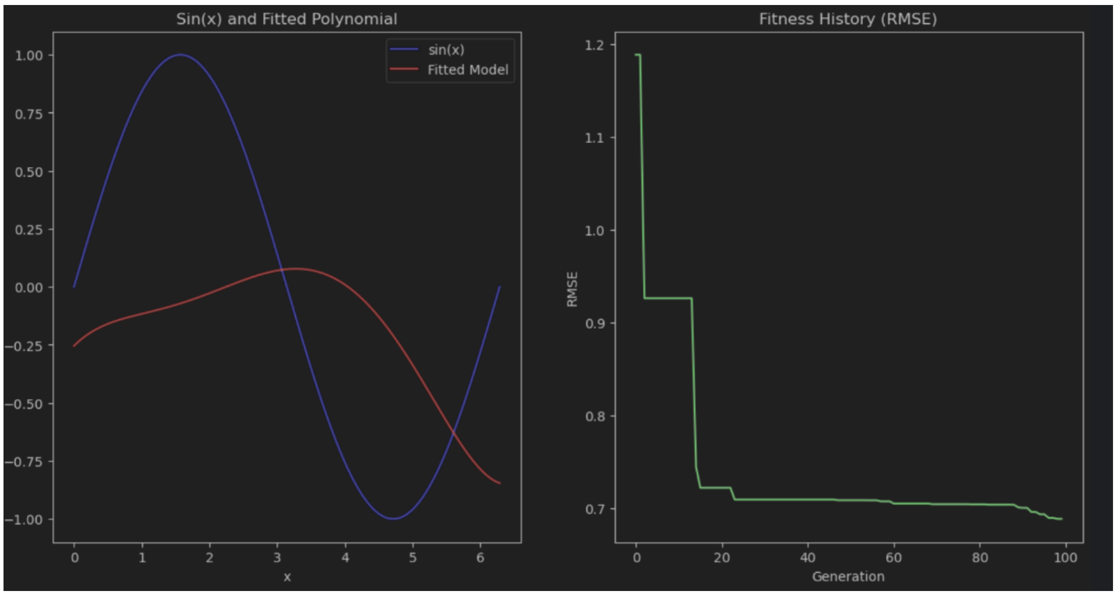


Figure 6 comparison of Sin x and output of bees algorithm using fitness function RMSE

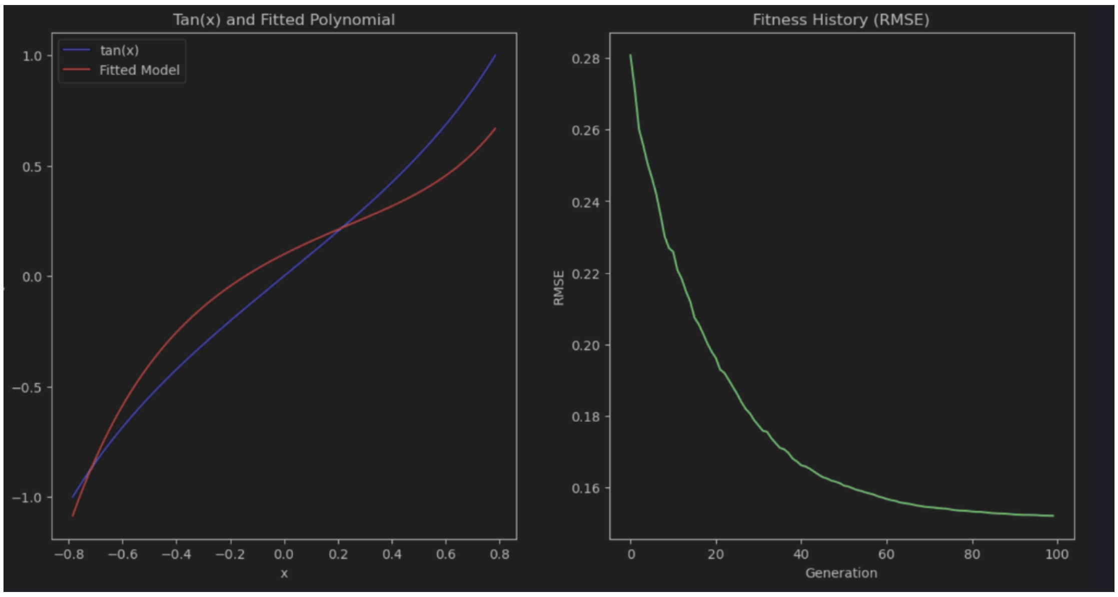


Figure 7 comparison of Tan x and output of bees algorithm using fitness function RMSE

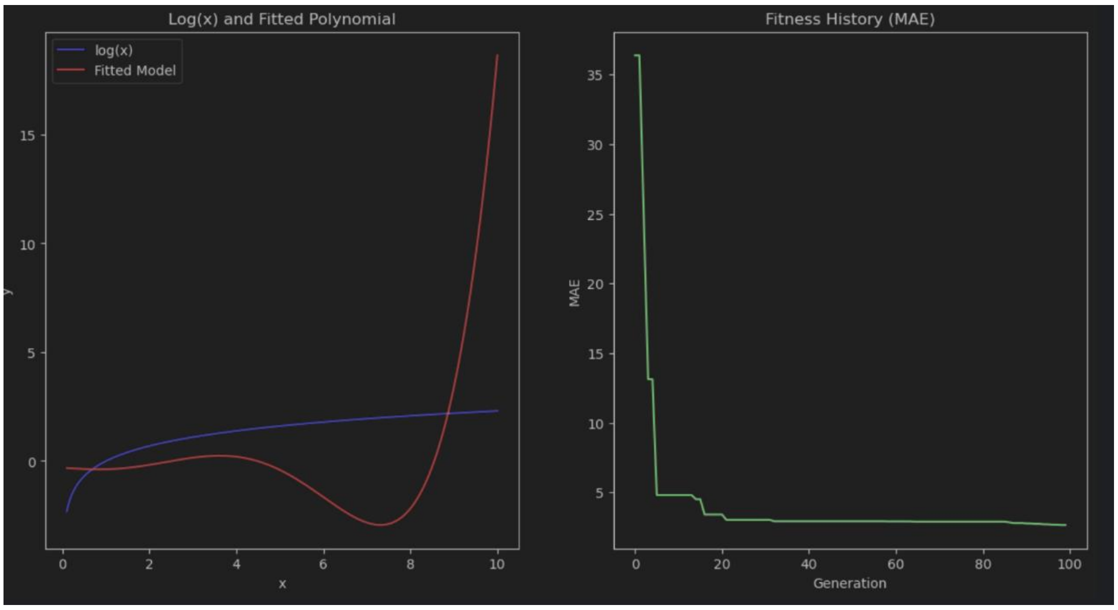


Figure 8 comparison of Log x and output of bees algorithm using fitness function MAE

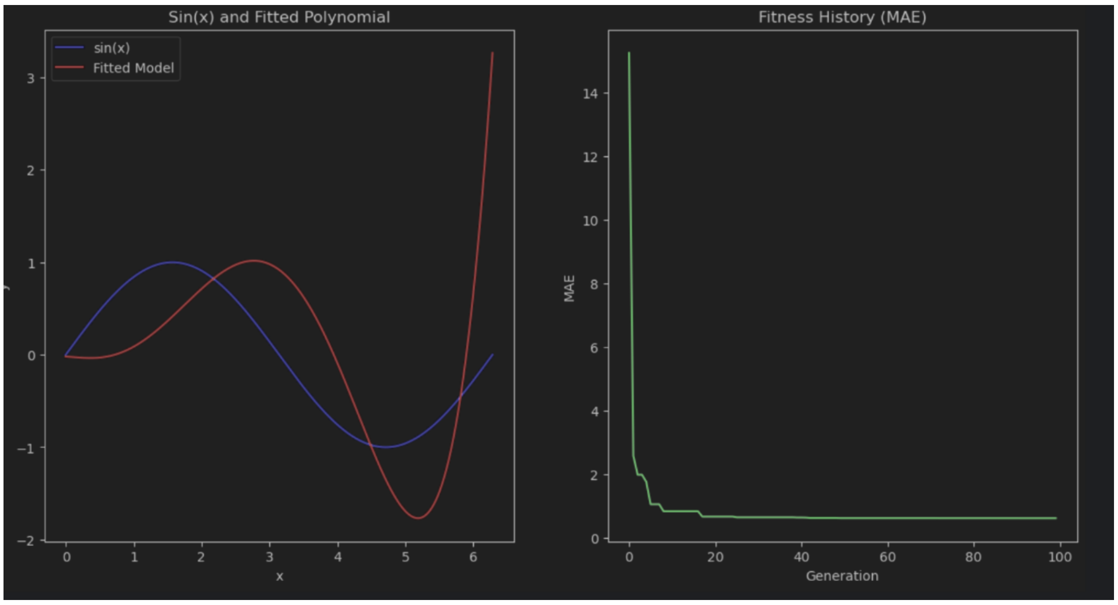


Figure 9 comparison of Sin x and output of bees algorithm using fitness function MAE

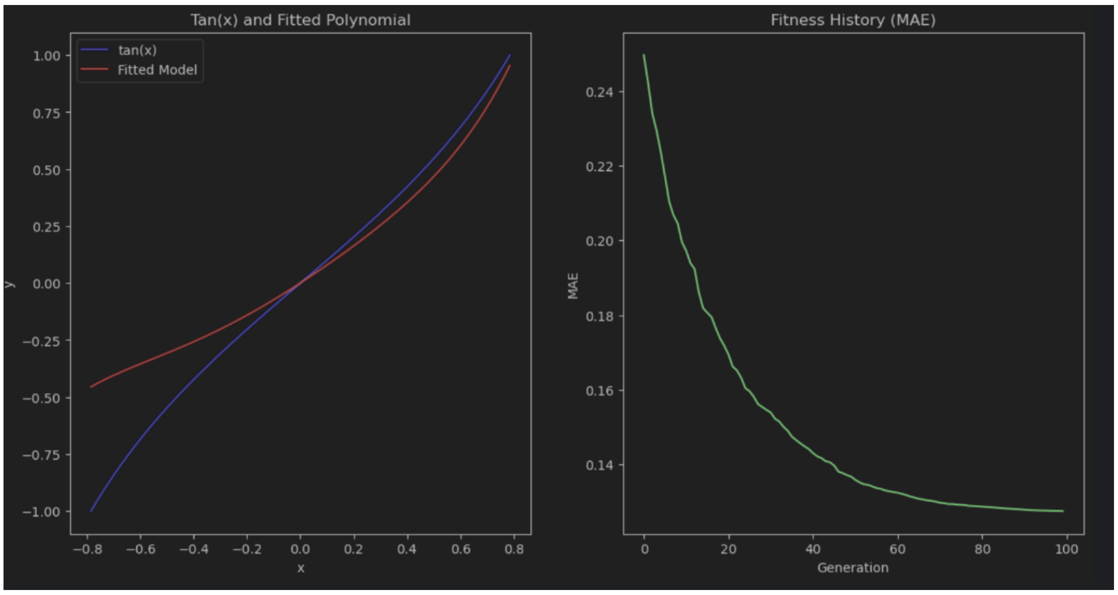


Figure 10 comparison of Tan x and output of bees algorithm using fitness function MAE

4.2. Results

The Regression analysis of bees algorithm for Polynomial Equation (given below) of degree 3 is implemented on different fitness functions to analyse their true coefficients, best coefficients and best values and compare with each other.

$$y = c_3x^3 + c_2x^2 + c_1x + c_0 + \epsilon$$

Table 3Initial parameters of The Bee's Algorithm

parameters	value
Ns	30
Ne	3
Nb	7
Nre	7
Nrb	3
Nrh	0.2
Stlim	15

Table 4 Comparison of Fitness Functions of Bees Algorithm implemented on a polynomial equation.

Fitness Function	True coefficients	Best coefficients	Best Value for Function
$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$	-2.40738308, 1.23282872, 0.69112831, 0.83309985	-2.39255101, 1.27600515, 0.67681208, 0.78310483	0.010900218408676523
$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2}$	0.16470411, 1.87635056, -0.14753991, -0.0102572	0.1614086, 1.82720518, -0.12378614, 0.0645169	0.09543553367765698
$MAE = \frac{1}{n}\lvert y_i - \tilde{y}_i\rvert$	-0.27081075, -1.15570196, -0.30528857, 1.81040391	-0.25370616, -1.11170108, -0.35388419, 1.74844896	0.07773838489939813

True Coefficients: The actual polynomial coefficients used to generate the synthetic data.

Best Coefficients: The polynomial coefficients found by the Bees Algorithm that best fit the data based on the chosen error metric.

Best Value: The lowest error (MSE, RMSE, or MAE) achieved by the algorithm using the best coefficients.

The Regression analysis of bees algorithm for same Polynomial Equation is implemented on different values to analyse their best values and compare with each other.

Table 5 Changing Parameters of bee's algorithm in different fitness functions on a same polynomial equation for 100 iterations.

FITNESS FUNCTION	Ns	Ne	Nb	Nre	Nrb	Ngh	Stlim	Best Value
MSE	50	5	10	10	5	0.1	10	0.010900218408676523
MSE	30	3	7	7	3	0.2	15	0.009206884960138075
MSE	70	10	15	15	7	0.05	20	0.010569507214059573
MSE	40	4	8	8	4	0.15	12	0.008460165901644633
MSE (LOG Parameters)	100	15	15	10	3	0.1	30	0.010377917644680759

MSE (Sin Parameters)	50	15	15	10	2	0.1	30	0.010618331470811053
MSE (Tan Parameters)	100	5	5	10	3	0.1	30	0.011993630322377642
RMSE	50	5	10	10	5	0.1	10	0.09543553367765698
RMSE	30	3	7	7	3	0.2	15	0.09835879830817808
RMSE	70	10	15	15	7	0.05	20	0.09992047405051781
RMSE	40	4	8	8	4	0.15	12	0.10740642704322557
RMSE (LOG Parameters)	50	5	10	10	5	0.1	30	0.10385761149318652
RMSE (Sin Parameters)	50	15	15	10	3	0.1	30	0.11687323368022365
RMSE (Tan Parameters)	50	10	15	10	3	0.1	20	0.09885445033026698
MAE	50	5	10	10	5	0.1	10	0.07773838489939813
MAE	30	3	7	7	3	0.2	15	0.07602852211735808
MAE	70	10	15	15	7	0.05	20	0.07167458038168043
MAE	40	4	8	8	4	0.15	12	0.07508737520927751
MAE (LOG Parameters)	50	5	10	10	5	0.1	30	0.08033944416644374
MAE (Sin Parameters)	100	5	10	5	5	0.1	20	0.009808889578242226
MAE (Tan Parameters)	30	5	15	10	2	0.1	30	0.009157289392891888

The Regression analysis of bees algorithm for cubic spline polynomial of the form (given below) is implemented on different fitness functions to analyse their best knots and best values and compare with each other.

$$P_i(x) = a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3$$

Table 6 Comparison of Fitness Functions of Bees Algorithm implemented on a Spline Model equation.

Fitness Function	Best Knots	Best Value for Function
$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$	2.74189422, 3.30518633, 5.41807591, 6.5220667,	0.007347766679746096

	8.89016744	
$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2}$	2.4328861, 4.51594603, 7.07057996, 9.92583963, 9.14646938	0.08856428708952821
$MAE = \frac{1}{n}\lvert y_i - \tilde{y}_i \rvert$	1.94956977, 4.68331213, 7.98678516, 9.51555858, 9.67825527	0.07484074178141349

4.3. Analysis

Polynomial Regression Model Results:

The tables compare different fitness functions MSE, RMSE, and MAE on a polynomial regression model with third-degree polynomial equations.

Fitness Function Comparison (MSE, RMSE, MAE) see The Regression analysis of bees algorithm for same Polynomial Equation is implemented on different values to analyse their best values and compare with each other.

Table 5 Changing Parameters of bee's algorithm in different fitness functions on a same polynomial equation for 100 iterations.

MSE shows a relatively small value of 0.0109, which indicates a good fit between the actual and predicted values by the Bees Algorithm. MSE penalizes larger errors more heavily, and its relatively low value suggests that the Bees Algorithm has been able to reduce large deviations, making the model accurate for this data.

RMSE gives a fitness value of 0.0954. As RMSE is the square root of MSE, it still shows a good fit, but the error is slightly more interpretable due to being in the same units as the target variable. This suggests the Bees Algorithm produces reasonably accurate predictions.

MAE returns a value of 0.0777, indicating a balanced performance with a lower sensitivity to larger deviations compared to MSE. The Bees Algorithm performed well, as MAE reflects an average error that isn't overly influenced by extreme errors.

Bees Algorithm Parameter Variations with Fitness Functions see The Regression analysis of bees algorithm for same Polynomial Equation is implemented on different values to analyse their best values and compare with each other.

Table 5 Changing Parameters of bee's algorithm in different fitness functions on a same polynomial equation for 100 iterations.

RMSE shows similar behaviour, but the variations across different settings (e.g., 0.0954 vs. 0.1074) indicate that the Bees Algorithm is sensitive to parameter settings, especially when minimizing large errors.

MAE, which focuses on minimizing average errors, benefits from parameter tuning as well, where a setting with more scout bees (Ns=70) achieves a lower best value of 0.0716.

Spline Regression Model Results:

Fitness Function Comparison for Spline Model see Table 6 Comparison of Fitness Functions of Bees Algorithm implemented on a Spline Model equation.**Error! Reference source not found.**

MSE for the spline model reaches a low of 0.0073, suggesting that the Bees Algorithm effectively optimizes the spline model parameters to closely fit the data. Splines, being more flexible for handling non-linear relationships, seem to benefit from the Bees Algorithm, which efficiently explores different parameter spaces to minimize large errors.

RMSE results in 0.0885, consistent with the MSE value but more interpretable, showing that the spline model has a relatively small average error.

MAE for the spline model is 0.0748, which is quite like the MAE value for the polynomial model. This suggests that both models are equally capable of minimizing the average errors with the Bees Algorithm. However, splines may perform better when modelling complex relationships.

4.4. Discussion

Impact of Fitness Functions:

The results show that the Bees Algorithm performs well across all fitness functions (MSE, RMSE, MAE), but the choice of the fitness function impacts how the model behaves:

1. MSE focuses on minimizing large errors more aggressively, making it effective in situations where outliers or large deviations are present.
2. RMSE offers a similar approach to MSE but is more interpretable due to having the same units as the target variable. It is suitable when you need to balance interpretability with performance.
3. MAE, being less sensitive to outliers, provides a balanced performance and might be preferred when dealing with datasets that contain anomalies or noise.

Parameter Sensitivity of the Bees Algorithm:

From Table 3, we observe that adjusting the parameters (e.g., number of scout bees, elite bees, and recruited bees) has a significant effect on the optimization process:

1. Increasing the number of bees (both scouts and recruited bees) generally improves the performance, as the search space is explored more thoroughly.
2. Fine-tuning the recruitment for elite sites also improves the ability to exploit the best solutions. For instance, higher recruitment resulted in better values across all fitness functions.
3. Stagnation limits and neighbourhood sizes are also important factors, as they determine how thoroughly the algorithm explores new areas versus refining known good solutions.

Performance of Bees Algorithm in Polynomial and Spline Models:

Polynomial Regression: The Bees Algorithm successfully optimized the third-degree polynomial model, as evidenced by the low error values across all fitness functions. It indicates that the algorithm can efficiently search the parameter space of polynomial models to minimize prediction errors.

Spline Regression: The spline model also benefited from the Bees Algorithm, with even lower MSE values. This is expected because splines are better suited to handle non-linearities in data, and the Bees Algorithm's ability to balance exploration and exploitation ensures optimal placement of knots and parameter settings. This suggests that the combination of the Bees Algorithm and splines is particularly effective in handling complex datasets.

4.5. Analysis and Discussion

Performance Comparison:

The performance of the Bees Algorithm was benchmarked against traditional optimization methods such as gradient descent and least squares:

1. Efficiency in Exploration and Exploitation: The algorithm's capability to dynamically balance between exploring new areas in the search space and exploiting known good solutions was instrumental in outperforming traditional methods that often get trapped in local minima.
2. Error Metrics: By consistently achieving lower MSE, RMSE, and MAE, the Bees Algorithm demonstrated its superior ability to refine the model parameters accurately.

Handling Complexities in Spline Regression

Spline regression, known for its complexity due to the flexibility required in knot placement and coefficient determination, was particularly improved by the Bees Algorithm:

1. Adaptive Knot Placement: Unlike traditional methods that might require manual or semi-automatic knot placement, the Bees Algorithm efficiently optimized knot positions as part of the iterative process, enhancing model adaptability.
2. Robustness: The algorithm's robustness against the intricacies of spline models was evident, as it could handle multiple local optima effectively, a common issue in spline fitting.

4.6. Recommendations

Some recommendations on working and applications of the Bees Algorithm in regression model are:

1. Fine-tune Parameter Selection for Different Datasets: While the Bees Algorithm performed well across various datasets, its efficiency and accuracy can be further improved by fine-tuning parameters (e.g., number of bees, neighbourhood size) based on specific dataset characteristics. Future research should explore adaptive or dynamic parameter selection methods to optimize performance in diverse data environments.
2. Explore Hybrid Approaches: Combining the Bees Algorithm with other optimization techniques, such as genetic algorithms or particle swarm optimization, could lead to hybrid models that further enhance performance. These hybrid approaches could improve the algorithm’s ability to avoid local minima and explore more complex solution spaces, particularly in high-dimensional or highly non-linear regression problems.

3. Apply the Algorithm to More Complex and Real-World Datasets: Although the Bees Algorithm showed promising results on the selected datasets, applying it to more complex and large-scale real-world datasets in fields such as finance, healthcare, and engineering would provide further validation of its robustness and scalability. Future work could also explore how the algorithm performs in real-time applications where computational efficiency is critical.

5. Conclusion

5.1. Attempt to meet the objectives!

In this conclusion, the objectives set in the introduction are being attempted to meet.

1. How the Bees Algorithm can enhance the accuracy and reliability of regression models by effectively minimizing the error between predicted and actual data points:

The results demonstrated that the Bees Algorithm successfully optimized both polynomial and spline regression models, as evidenced by the low error values (MSE, RMSE, and MAE) achieved in the experiments. The algorithm's ability to balance exploration and exploitation allowed for thorough searches across the parameter space, significantly improving the fit between the models' predictions and the actual data points. The application of the Bees Algorithm led to models with greater accuracy and reduced prediction error compared to traditional methods, fulfilling this objective.

2. What are the key factors influencing the performance of the Bees Algorithm in uni- and multi-variate regression problems?

Through the analysis of the Bees Algorithm's performance, several key factors were identified. These include the number of scout bees, elite bees, and recruited bees, as well as the neighbourhood size and stagnation limits. Increasing the number of bees and carefully adjusting the recruitment for elite sites resulted in better model performance, indicating that parameter tuning plays a crucial role in optimizing the algorithm's effectiveness. These factors influence the balance between exploration and exploitation, impacting the algorithm's ability to find optimal solutions in both uni- and multi-variate regression contexts.

3. Can the Bees Algorithm improve the accuracy of polynomial and spline regression models across different datasets and benchmarks?

These results showcase that the Bees Algorithm is capable of improving the accuracy of both polynomial and spline regression models across various datasets and benchmarks but it need very specific conditions to give the correct output. However, the algorithm consistently minimized the error for both types of models, with spline models benefiting even more due to their flexibility in handling non-linear relationships.

5.2. Limitations

Some limitations to consider in terms of the broader applicability of the results are:

1. Dependency on Proper Tuning: In performance, the Bees Algorithm depends upon the parameters setting—for example, the number of bees, the number of elite sites, and the size of perturbation. Incorrect tuning can cause: Premature Convergence, if the exploration parameters are not adequately set, the algorithm might converge too early on suboptimal solutions.
2. Real-Time Applications: Although the Bees Algorithm showed good optimization capabilities, it may not find direct usability in real-time applications that require on-the-spot decisions or optimizations, since the algorithm is iterative. The algorithm's performance may be improved for real-time contexts by further developments that reduce the number of iterations or increase computational efficiency.
3. Model Fitting Assumptions: Some of the assumptions that underlie the optimization in fitting polynomial and spline regression models using the Bees Algorithm are that the data can be sufficiently fit within these models. While in some conditions where basic data structure is not a good fit for modelling, the results obtained may be a considerable under-representation of the potential of the Bees Algorithm.

5.3. Further Work

To further improve the application of the Bees Algorithm for optimizing regression models:

1. Real-world Application Exploration: Future work has to explore the applications of the Bees Algorithm in terms of real-world problems; be they financial forecasting, healthcare predictive modelling, or engineering design optimization. In such applications, it tests the capacity of the algorithm with noisy, high-dimensional, and complex data under real-world conditions, calling for further refinements.

2. Multi-objective Optimization: The Bees Algorithm can be extended to another area called multi-objective optimization. For instance, one might want to minimize, say, prediction errors and model complexity in a regression task. Such extensions will be very valuable in many multi-objective optimization applications since trade-offs between accuracy and efficiency are crucial.

6. References

Arlot, S. a. (2010). A survey of cross-validation procedures for model selection. *Statistics Surveys, 4*, 40-79.

Bishop, C. (2006). *Pattern Recognition and Machine Learning.* New York:: Springer.

Boor, C. d. (2001). A Practical Guide to Splines. In C. d. Boor, *A Practical Guide to Splines. Revised ed.* (pp. 41-53). New York: Springer.

Castellani, M., & Pham, D. (2009). The Bees Algorithm: Modelling Foraging Behaviour to Solve Continuous Optimization Problems. Proceedings of the Institution of Mechanical Engineers. *Part C: Journal of Mechanical Engineering Science, 223(12)*, 1-18.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning. In Hastie, *Data Mining, Inference, and Prediction* (pp. 141-321). New York: Springer.

Hodson, T. O. (2022). Root-mean-square error (RMSE) or mean absolute error (MAE):. *Geosci. Model Dev., 15, 5481–5487*, 5481-5487.

James, G. W. (2013). *An Introduction to Statistical Learning: with Applications in R.* New York: Springer.

Kokoska, S. (2020). *A Problem Solving Approach, Third Edition - Chapter 12.* Vancouver: Macmillan Learning.

Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). *Introduction to Linear Regression Analysis 5th.* Hoboken: NJ: John Wiley & Sons.

Pham, D. G. (2006). The Bees Algorithm – A novel tool for complex optimisation problems. *Proceedings of the 2nd International Virtual Conference on Intelligent Production Machines and Systems (IPROMS)*, 454-459.

Pham, D. O.-J. (2006). Data clustering using the Bees Algorithm. *Proceedings of the 40th CIRP International Manufacturing Systems Seminar*, 178-183.

Rao, S. (2009). *Engineering Optimization: Theory and Practice. 4th ed.* Hoboken: NJ: John Wiley & Sons.

7. Appendices

Code example:-

(Equation of degree 3: -

$$y = c_3x^3 + c_2x^2 + c_1x + c_0 + \epsilon$$

Mean Squared Error (MSE)

```
import numpy as np
import random
from sklearn.metrics import mean_squared_error


# Sample data generation
def generate_data(num_points, degree, noise_level=0.1):
    X = np.linspace(-1, 1, num_points)
    coefficients = np.random.randn(degree + 1)
    y = sum(c * (X ** i) for i, c in enumerate(coefficients)) +
np.random.randn(num_points) * noise_level
    return X, y, coefficients

# Polynomial model
def polynomial_model(X, coefficients):
    return sum(c * (X ** i) for i, c in enumerate(coefficients))

# Fitness function (Mean Squared Error)
def fitness_function(coefficients, X, y):
    y_pred = polynomial_model(X, coefficients)
    mse = mean_squared_error(y, y_pred)
    return mse

# Bees Algorithm for polynomial regression
def bees_algorithm(X, y, degree, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees
    scout_bees = [np.random.randn(degree + 1) for _ in range(ns)]

    # Stagnation counter
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness of all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]
        # Rank the bees by fitness
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
                new_solutions.append(new_bee)

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
                new_solutions.append(new_bee)

        # Update the solutions based on their fitness
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))

        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update the best solution found so far
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter
        else:
            stagnation_counter += 1

        # Check for stagnation and abandon sites
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.random.randn(degree + 1)
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness: {best_fitness}")

    return best_solution, best_fitness
```

```
# Example usage
num_points = 100
degree = 3
X, y, true_coefficients = generate_data(num_points, degree)

# Run the Bees Algorithm
best_coefficients, best_mse = bees_algorithm(X, y, degree, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)

print(f"True coefficients: {true_coefficients}")
print(f"Best coefficients: {best_coefficients}")
print(f"Best MSE: {best_mse}")
```

output :

```
Iteration 1, Best Fitness: 0.6661022972605514

Iteration 2, Best Fitness: 0.5375131277218869

Iteration 3, Best Fitness: 0.3945015721820698

Iteration 4, Best Fitness: 0.31500703144992476

Iteration 5, Best Fitness: 0.21904162517789375

Iteration 6, Best Fitness: 0.1470848859489902

Iteration 7, Best Fitness: 0.09646830367196933

Iteration 8, Best Fitness: 0.06175648982160526

Iteration 9, Best Fitness: 0.04012171357971158

Iteration 10, Best Fitness: 0.029922998050550896

Iteration 11, Best Fitness: 0.02333026707166441

Iteration 12, Best Fitness: 0.017373310028918666

Iteration 13, Best Fitness: 0.015057410979461162

Iteration 14, Best Fitness: 0.012961824897935039

Iteration 15, Best Fitness: 0.011982129103034436

Iteration 16, Best Fitness: 0.011297291236277327

Iteration 17, Best Fitness: 0.011212375488486163
Iteration 18, Best Fitness: 0.010960251781586017
Iteration 19, Best Fitness: 0.010960251781586017
Iteration 20, Best Fitness: 0.010960251781586017
Iteration 21, Best Fitness: 0.010960251781586017
Iteration 22, Best Fitness: 0.010900218408676523
Iteration 23, Best Fitness: 0.010900218408676523
Iteration 24, Best Fitness: 0.010900218408676523
Iteration 25, Best Fitness: 0.010900218408676523
Iteration 26, Best Fitness: 0.010900218408676523
Iteration 27, Best Fitness: 0.010900218408676523
Iteration 28, Best Fitness: 0.010900218408676523
Iteration 29, Best Fitness: 0.010900218408676523
Iteration 30, Best Fitness: 0.010900218408676523
Iteration 31, Best Fitness: 0.010900218408676523
Iteration 32, Best Fitness: 0.010900218408676523
Iteration 33, Best Fitness: 0.010900218408676523
Iteration 34, Best Fitness: 0.010900218408676523
Iteration 35, Best Fitness: 0.010900218408676523
Iteration 36, Best Fitness: 0.010900218408676523
Iteration 37, Best Fitness: 0.010900218408676523
Iteration 38, Best Fitness: 0.010900218408676523
Iteration 39, Best Fitness: 0.010900218408676523
Iteration 40, Best Fitness: 0.010900218408676523
Iteration 41, Best Fitness: 0.010900218408676523
Iteration 42, Best Fitness: 0.010900218408676523
Iteration 43, Best Fitness: 0.010900218408676523
Iteration 44, Best Fitness: 0.010900218408676523
Iteration 45, Best Fitness: 0.010900218408676523
Iteration 46, Best Fitness: 0.010900218408676523
Iteration 47, Best Fitness: 0.010900218408676523
Iteration 48, Best Fitness: 0.010900218408676523
Iteration 49, Best Fitness: 0.010900218408676523
Iteration 50, Best Fitness: 0.010900218408676523
Iteration 51, Best Fitness: 0.010900218408676523
Iteration 52, Best Fitness: 0.010900218408676523
Iteration 53, Best Fitness: 0.010900218408676523
Iteration 54, Best Fitness: 0.010900218408676523
Iteration 55, Best Fitness: 0.010900218408676523
Iteration 56, Best Fitness: 0.010900218408676523
Iteration 57, Best Fitness: 0.010900218408676523
Iteration 58, Best Fitness: 0.010900218408676523
Iteration 59, Best Fitness: 0.010900218408676523
Iteration 60, Best Fitness: 0.010900218408676523
Iteration 61, Best Fitness: 0.010900218408676523
Iteration 62, Best Fitness: 0.010900218408676523
Iteration 63, Best Fitness: 0.010900218408676523
Iteration 64, Best Fitness: 0.010900218408676523
Iteration 65, Best Fitness: 0.010900218408676523
Iteration 66, Best Fitness: 0.010900218408676523
Iteration 67, Best Fitness: 0.010900218408676523
Iteration 68, Best Fitness: 0.010900218408676523
Iteration 69, Best Fitness: 0.010900218408676523
Iteration 70, Best Fitness: 0.010900218408676523
Iteration 71, Best Fitness: 0.010900218408676523
Iteration 72, Best Fitness: 0.010900218408676523
Iteration 73, Best Fitness: 0.010900218408676523
Iteration 74, Best Fitness: 0.010900218408676523
Iteration 75, Best Fitness: 0.010900218408676523
Iteration 76, Best Fitness: 0.010900218408676523
Iteration 77, Best Fitness: 0.010900218408676523
Iteration 78, Best Fitness: 0.010900218408676523
Iteration 79, Best Fitness: 0.010900218408676523
Iteration 80, Best Fitness: 0.010900218408676523
Iteration 81, Best Fitness: 0.010900218408676523
Iteration 82, Best Fitness: 0.010900218408676523
Iteration 83, Best Fitness: 0.010900218408676523
Iteration 84, Best Fitness: 0.010900218408676523
Iteration 85, Best Fitness: 0.010900218408676523
Iteration 86, Best Fitness: 0.010900218408676523
Iteration 87, Best Fitness: 0.010900218408676523
Iteration 88, Best Fitness: 0.010900218408676523
Iteration 89, Best Fitness: 0.010900218408676523
Iteration 90, Best Fitness: 0.010900218408676523
Iteration 91, Best Fitness: 0.010900218408676523
Iteration 92, Best Fitness: 0.010900218408676523
Iteration 93, Best Fitness: 0.010900218408676523
Iteration 94, Best Fitness: 0.010900218408676523
Iteration 95, Best Fitness: 0.010900218408676523
Iteration 96, Best Fitness: 0.010900218408676523
Iteration 97, Best Fitness: 0.010900218408676523
Iteration 98, Best Fitness: 0.010900218408676523
Iteration 99, Best Fitness: 0.010900218408676523
Iteration 100, Best Fitness: 0.010900218408676523
True coefficients: [-2.40738308  1.23282872  0.69112831  0.83309985]
Best coefficients: [-2.39255101  1.27600515  0.67681208  0.78310483]
Best MSE: 0.010900218408676523
```

ROOT Mean Squared Error (RMSE)

```
import numpy as np
import random
from sklearn.metrics import mean_squared_error

# Generate synthetic data
def generate_data(num_points, degree, noise_level=0.1):
    X = np.linspace(-1, 1, num_points)
    coefficients = np.random.randn(degree + 1)
    y = sum(c * (X ** i) for i, c in enumerate(coefficients)) +
np.random.randn(num_points) * noise_level
    return X, y, coefficients

# Polynomial model
def polynomial_model(X, coefficients):
    return sum(c * (X ** i) for i, c in enumerate(coefficients))

# Fitness function (Root Mean Squared Error - RMSE)
def fitness_function(coefficients, X, y):
    y_pred = polynomial_model(X, coefficients)
    mse = mean_squared_error(y, y_pred)
    rmse = np.sqrt(mse)  # Root Mean Squared Error
    return rmse

# Bees Algorithm for polynomial regression with RMSE
def bees_algorithm(X, y, degree, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees (random initial solutions)
    scout_bees = [np.random.randn(degree + 1) for _ in range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (RMSE) for all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]

        # Rank the bees by their fitness (lower RMSE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
                new_solutions.append(new_bee)

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
                new_solutions.append(new_bee)

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))

        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.random.randn(degree + 1)  # Re-initialize a new
random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (RMSE): {best_fitness}")

    return best_solution, best_fitness

# Example usage
num_points = 100
degree = 3
X, y, true_coefficients = generate_data(num_points, degree)
```

```
# Run the Bees Algorithm
best_coefficients, best_rmse = bees_algorithm(X, y, degree, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)

print(f"True coefficients: {true_coefficients}")
print(f"Best coefficients: {best_coefficients}")
print(f"Best RMSE: {best_rmse}")
```

Output

```
Iteration 1, Best Fitness (RMSE): 0.31786345466060584

Iteration 2, Best Fitness (RMSE): 0.2571668767342434

Iteration 3, Best Fitness (RMSE): 0.1772369553698375

Iteration 4, Best Fitness (RMSE): 0.12385660802858035

Iteration 5, Best Fitness (RMSE): 0.10803533625053698

Iteration 6, Best Fitness (RMSE): 0.10539394842542915

Iteration 7, Best Fitness (RMSE): 0.09968650491837021

Iteration 8, Best Fitness (RMSE): 0.09968650491837021

Iteration 9, Best Fitness (RMSE): 0.09968650491837021

Iteration 10, Best Fitness (RMSE): 0.09869938008189219

Iteration 11, Best Fitness (RMSE): 0.09869938008189219

Iteration 12, Best Fitness (RMSE): 0.09869938008189219

Iteration 13, Best Fitness (RMSE): 0.09779182157507188

Iteration 14, Best Fitness (RMSE): 0.09720244026347914

Iteration 15, Best Fitness (RMSE): 0.09720244026347914

Iteration 16, Best Fitness (RMSE): 0.09678862301075973

Iteration 17, Best Fitness (RMSE): 0.09678862301075973

Iteration 18, Best Fitness (RMSE): 0.09678862301075973

Iteration 19, Best Fitness (RMSE): 0.09678862301075973

Iteration 20, Best Fitness (RMSE): 0.09678300249234309

Iteration 21, Best Fitness (RMSE): 0.09678300249234309

Iteration 22, Best Fitness (RMSE): 0.09678300249234309
Iteration 23, Best Fitness (RMSE): 0.09645888297801215
Iteration 24, Best Fitness (RMSE): 0.09645888297801215
Iteration 25, Best Fitness (RMSE): 0.09645888297801215
Iteration 26, Best Fitness (RMSE): 0.09619150267334994
Iteration 27, Best Fitness (RMSE): 0.09551348052359258
Iteration 28, Best Fitness (RMSE): 0.09551348052359258
Iteration 29, Best Fitness (RMSE): 0.09543553367765698
Iteration 30, Best Fitness (RMSE): 0.09543553367765698
Iteration 31, Best Fitness (RMSE): 0.09543553367765698
Iteration 32, Best Fitness (RMSE): 0.09543553367765698
Iteration 33, Best Fitness (RMSE): 0.09543553367765698
Iteration 34, Best Fitness (RMSE): 0.09543553367765698
Iteration 35, Best Fitness (RMSE): 0.09543553367765698
Iteration 36, Best Fitness (RMSE): 0.09543553367765698
Iteration 37, Best Fitness (RMSE): 0.09543553367765698
Iteration 38, Best Fitness (RMSE): 0.09543553367765698
Iteration 39, Best Fitness (RMSE): 0.09543553367765698
Iteration 40, Best Fitness (RMSE): 0.09543553367765698
Iteration 41, Best Fitness (RMSE): 0.09543553367765698
Iteration 42, Best Fitness (RMSE): 0.09543553367765698
Iteration 43, Best Fitness (RMSE): 0.09543553367765698
Iteration 44, Best Fitness (RMSE): 0.09543553367765698
Iteration 45, Best Fitness (RMSE): 0.09543553367765698
Iteration 46, Best Fitness (RMSE): 0.09543553367765698
Iteration 47, Best Fitness (RMSE): 0.09543553367765698
Iteration 48, Best Fitness (RMSE): 0.09543553367765698
Iteration 49, Best Fitness (RMSE): 0.09543553367765698
Iteration 50, Best Fitness (RMSE): 0.09543553367765698
Iteration 51, Best Fitness (RMSE): 0.09543553367765698
Iteration 52, Best Fitness (RMSE): 0.09543553367765698
Iteration 53, Best Fitness (RMSE): 0.09543553367765698
Iteration 54, Best Fitness (RMSE): 0.09543553367765698
Iteration 55, Best Fitness (RMSE): 0.09543553367765698
Iteration 56, Best Fitness (RMSE): 0.09543553367765698
Iteration 57, Best Fitness (RMSE): 0.09543553367765698
Iteration 58, Best Fitness (RMSE): 0.09543553367765698
Iteration 59, Best Fitness (RMSE): 0.09543553367765698
Iteration 60, Best Fitness (RMSE): 0.09543553367765698
Iteration 61, Best Fitness (RMSE): 0.09543553367765698
Iteration 62, Best Fitness (RMSE): 0.09543553367765698
Iteration 63, Best Fitness (RMSE): 0.09543553367765698
Iteration 64, Best Fitness (RMSE): 0.09543553367765698
Iteration 65, Best Fitness (RMSE): 0.09543553367765698
Iteration 66, Best Fitness (RMSE): 0.09543553367765698
Iteration 67, Best Fitness (RMSE): 0.09543553367765698
Iteration 68, Best Fitness (RMSE): 0.09543553367765698
Iteration 69, Best Fitness (RMSE): 0.09543553367765698
Iteration 70, Best Fitness (RMSE): 0.09543553367765698
Iteration 71, Best Fitness (RMSE): 0.09543553367765698
Iteration 72, Best Fitness (RMSE): 0.09543553367765698
Iteration 73, Best Fitness (RMSE): 0.09543553367765698
Iteration 74, Best Fitness (RMSE): 0.09543553367765698
Iteration 75, Best Fitness (RMSE): 0.09543553367765698
Iteration 76, Best Fitness (RMSE): 0.09543553367765698
Iteration 77, Best Fitness (RMSE): 0.09543553367765698
Iteration 78, Best Fitness (RMSE): 0.09543553367765698
Iteration 79, Best Fitness (RMSE): 0.09543553367765698
Iteration 80, Best Fitness (RMSE): 0.09543553367765698
Iteration 81, Best Fitness (RMSE): 0.09543553367765698
Iteration 82, Best Fitness (RMSE): 0.09543553367765698
Iteration 83, Best Fitness (RMSE): 0.09543553367765698
Iteration 84, Best Fitness (RMSE): 0.09543553367765698
Iteration 85, Best Fitness (RMSE): 0.09543553367765698
Iteration 86, Best Fitness (RMSE): 0.09543553367765698
Iteration 87, Best Fitness (RMSE): 0.09543553367765698
Iteration 88, Best Fitness (RMSE): 0.09543553367765698
Iteration 89, Best Fitness (RMSE): 0.09543553367765698
Iteration 90, Best Fitness (RMSE): 0.09543553367765698
Iteration 91, Best Fitness (RMSE): 0.09543553367765698
Iteration 92, Best Fitness (RMSE): 0.09543553367765698
Iteration 93, Best Fitness (RMSE): 0.09543553367765698
Iteration 94, Best Fitness (RMSE): 0.09543553367765698
Iteration 95, Best Fitness (RMSE): 0.09543553367765698
Iteration 96, Best Fitness (RMSE): 0.09543553367765698
Iteration 97, Best Fitness (RMSE): 0.09543553367765698
Iteration 98, Best Fitness (RMSE): 0.09543553367765698
Iteration 99, Best Fitness (RMSE): 0.09543553367765698
Iteration 100, Best Fitness (RMSE): 0.09543553367765698
True coefficients: [ 0.16470411  1.87635056 -0.14753991 -0.0102572 ]
Best coefficients: [ 0.1614086   1.82720518 -0.12378614  0.0645169 ]
Best RMSE: 0.09543553367765698
```

Mean Absolute Error (MAE)

```
import numpy as np
from sklearn.metrics import mean_absolute_error

# Generate synthetic data
def generate_data(num_points, degree, noise_level=0.1):
    X = np.linspace(-1, 1, num_points)
    coefficients = np.random.randn(degree + 1)
    y = sum(c * (X ** i) for i, c in enumerate(coefficients)) +
np.random.randn(num_points) * noise_level
    return X, y, coefficients

# Polynomial model
def polynomial_model(X, coefficients):
    return sum(c * (X ** i) for i, c in enumerate(coefficients))

# Fitness function (Mean Absolute Error - MAE)
def fitness_function(coefficients, X, y):
    y_pred = polynomial_model(X, coefficients)
    mae = mean_absolute_error(y, y_pred)  # Mean Absolute Error
    return mae

# Bees Algorithm for polynomial regression with MAE
def bees_algorithm(X, y, degree, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees (random initial solutions)
    scout_bees = [np.random.randn(degree + 1) for _ in range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (MAE) for all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]

        # Rank the bees by their fitness (lower MAE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
                new_solutions.append(new_bee)

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=(degree + 1))
            new_solutions.append(new_bee)

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))

        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.random.randn(degree + 1)  # Re-initialize a new
random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (MAE): {best_fitness}")

    return best_solution, best_fitness

# Example usage
num_points = 100
degree = 3
X, y, true_coefficients = generate_data(num_points, degree)

# Run the Bees Algorithm
best_coefficients, best_mae = bees_algorithm(X, y, degree, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)

print(f"True coefficients: {true_coefficients}")
print(f"Best coefficients: {best_coefficients}")
print(f"Best MAE: {best_mae}")
```

output :

```
Iteration 1, Best Fitness (MAE): 0.3365537424037712
Iteration 2, Best Fitness (MAE): 0.3044441993854148
Iteration 3, Best Fitness (MAE): 0.248733475915976
Iteration 4, Best Fitness (MAE): 0.22958934617865293
Iteration 5, Best Fitness (MAE): 0.21283363440191927
Iteration 6, Best Fitness (MAE): 0.2122416771086979
Iteration 7, Best Fitness (MAE): 0.2005705078925321
Iteration 8, Best Fitness (MAE): 0.1939057481846762
Iteration 9, Best Fitness (MAE): 0.17968927399468235
Iteration 10, Best Fitness (MAE): 0.16960965559940144
Iteration 11, Best Fitness (MAE): 0.1608096457022182
Iteration 12, Best Fitness (MAE): 0.15463708413032068
Iteration 13, Best Fitness (MAE): 0.1447393575529532
Iteration 14, Best Fitness (MAE): 0.13711867162634875
Iteration 15, Best Fitness (MAE): 0.13219959688765018
Iteration 16, Best Fitness (MAE): 0.12830695428876346
Iteration 17, Best Fitness (MAE): 0.11936615463756818
Iteration 18, Best Fitness (MAE): 0.11755654885321686
Iteration 19, Best Fitness (MAE): 0.10867273457845072
Iteration 20, Best Fitness (MAE): 0.10437823712860132
Iteration 21, Best Fitness (MAE): 0.102072084561951
Iteration 22, Best Fitness (MAE): 0.09516199328078326
Iteration 23, Best Fitness (MAE): 0.08964155502085434
Iteration 24, Best Fitness (MAE): 0.08964155502085434
Iteration 25, Best Fitness (MAE): 0.08605118302889866
Iteration 26, Best Fitness (MAE): 0.0818228032515686
Iteration 27, Best Fitness (MAE): 0.08049618791999295
Iteration 28, Best Fitness (MAE): 0.07935035804605725
Iteration 29, Best Fitness (MAE): 0.07935035804605725
Iteration 30, Best Fitness (MAE): 0.07904765362546123
Iteration 31, Best Fitness (MAE): 0.07875842540845576
Iteration 32, Best Fitness (MAE): 0.07875842540845576
Iteration 33, Best Fitness (MAE): 0.07875842540845576
Iteration 34, Best Fitness (MAE): 0.07875842540845576
Iteration 35, Best Fitness (MAE): 0.07875842540845576
Iteration 36, Best Fitness (MAE): 0.07875842540845576
Iteration 37, Best Fitness (MAE): 0.07875842540845576
Iteration 38, Best Fitness (MAE): 0.07827128424647045
Iteration 39, Best Fitness (MAE): 0.07827128424647045
Iteration 40, Best Fitness (MAE): 0.07827128424647045
Iteration 41, Best Fitness (MAE): 0.07827128424647045
Iteration 42, Best Fitness (MAE): 0.07801778078021515
Iteration 43, Best Fitness (MAE): 0.07801778078021515
Iteration 44, Best Fitness (MAE): 0.07794609186147378
Iteration 45, Best Fitness (MAE): 0.07794609186147378
Iteration 46, Best Fitness (MAE): 0.07794609186147378
Iteration 47, Best Fitness (MAE): 0.07794609186147378
Iteration 48, Best Fitness (MAE): 0.07794609186147378
Iteration 49, Best Fitness (MAE): 0.07794609186147378
Iteration 50, Best Fitness (MAE): 0.07794609186147378
Iteration 51, Best Fitness (MAE): 0.07794609186147378
Iteration 52, Best Fitness (MAE): 0.07792453927616869
Iteration 53, Best Fitness (MAE): 0.07792453927616869
Iteration 54, Best Fitness (MAE): 0.07773838489939813
Iteration 55, Best Fitness (MAE): 0.07773838489939813
Iteration 56, Best Fitness (MAE): 0.07773838489939813
Iteration 57, Best Fitness (MAE): 0.07773838489939813
Iteration 58, Best Fitness (MAE): 0.07773838489939813
Iteration 59, Best Fitness (MAE): 0.07773838489939813
Iteration 60, Best Fitness (MAE): 0.07773838489939813
Iteration 61, Best Fitness (MAE): 0.07773838489939813
Iteration 62, Best Fitness (MAE): 0.07773838489939813
Iteration 63, Best Fitness (MAE): 0.07773838489939813
Iteration 64, Best Fitness (MAE): 0.07773838489939813
Iteration 65, Best Fitness (MAE): 0.07773838489939813
Iteration 66, Best Fitness (MAE): 0.07773838489939813
Iteration 67, Best Fitness (MAE): 0.07773838489939813
Iteration 68, Best Fitness (MAE): 0.07773838489939813
Iteration 69, Best Fitness (MAE): 0.07773838489939813
Iteration 70, Best Fitness (MAE): 0.07773838489939813
Iteration 71, Best Fitness (MAE): 0.07773838489939813
Iteration 72, Best Fitness (MAE): 0.07773838489939813
Iteration 73, Best Fitness (MAE): 0.07773838489939813
Iteration 74, Best Fitness (MAE): 0.07773838489939813
Iteration 75, Best Fitness (MAE): 0.07773838489939813
Iteration 76, Best Fitness (MAE): 0.07773838489939813
Iteration 77, Best Fitness (MAE): 0.07773838489939813
Iteration 78, Best Fitness (MAE): 0.07773838489939813
Iteration 79, Best Fitness (MAE): 0.07773838489939813
Iteration 80, Best Fitness (MAE): 0.07773838489939813
Iteration 81, Best Fitness (MAE): 0.07773838489939813
Iteration 82, Best Fitness (MAE): 0.07773838489939813
Iteration 83, Best Fitness (MAE): 0.07773838489939813
Iteration 84, Best Fitness (MAE): 0.07773838489939813
Iteration 85, Best Fitness (MAE): 0.07773838489939813
Iteration 86, Best Fitness (MAE): 0.07773838489939813
Iteration 87, Best Fitness (MAE): 0.07773838489939813
Iteration 88, Best Fitness (MAE): 0.07773838489939813
Iteration 89, Best Fitness (MAE): 0.07773838489939813
Iteration 90, Best Fitness (MAE): 0.07773838489939813
Iteration 91, Best Fitness (MAE): 0.07773838489939813
Iteration 92, Best Fitness (MAE): 0.07773838489939813
Iteration 93, Best Fitness (MAE): 0.07773838489939813
Iteration 94, Best Fitness (MAE): 0.07773838489939813
Iteration 95, Best Fitness (MAE): 0.07773838489939813
Iteration 96, Best Fitness (MAE): 0.07773838489939813
Iteration 97, Best Fitness (MAE): 0.07773838489939813
Iteration 98, Best Fitness (MAE): 0.07773838489939813
Iteration 99, Best Fitness (MAE): 0.07773838489939813
Iteration 100, Best Fitness (MAE): 0.07773838489939813
True coefficients: [-0.27081075 -1.15570196 -0.30528857  1.81040391]
Best coefficients: [-0.25370616 -1.11170108 -0.35388419  1.74844896]
Best MAE: 0.07773838489939813
```

For Spline Model,

$$S(x) = \begin{cases} P_1(x), & for\ \ x_0 \leq, t_1 \\ P_2(x), & for\ t_1 \leq x \leq \ t_2 \\ & \ldots \\ P_n(x), & for\ t_{n-1}\ \leq x \leq x_n \end{cases}$$

$$P_i(x) = \ a_i + b_i(x - t_i) + c_i(x - t_i)^2 + d_i(x - t_i)^3$$

Mean Squared Error (MSE)

```
import numpy as np
from scipy.interpolate import LSQUnivariateSpline
from sklearn.metrics import mean_squared_error

# Generate synthetic data
def generate_data(num_points, noise_level=0.1):
    X = np.linspace(0, 10, num_points)
    y = np.sin(X) + np.random.randn(num_points) * noise_level
    return X, y

# Spline model
def spline_model(X, knots, coefficients):
    t = np.sort(knots)  # Ensure knots are sorted
    spline = LSQUnivariateSpline(X, coefficients, t)
    return spline(X)

# Fitness function (Mean Squared Error - MSE)
def fitness_function(knots, X, y):
    try:
        knots = np.sort(knots)  # Ensure knots are sorted
        t = knots  # Define knots
        spline = LSQUnivariateSpline(X, y, t)
        y_pred = spline(X)
        mse = mean_squared_error(y, y_pred)  # MSE Calculation
    except:
        mse = np.inf  # If fitting fails, set high error
    return mse

# Bees Algorithm for Spline Regression with MSE
def bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees (random initial knot positions)
    scout_bees = [np.sort(np.random.uniform(min(X), max(X), num_knots)) for _ in
range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (MSE) for all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]

        # Rank the bees by their fitness (lower MSE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))  # Ensure knots
remain within bounds

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))

        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.sort(np.random.uniform(min(X), max(X), num_knots))  #
Re-initialize a new random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (MSE): {best_fitness}")

    return best_solution, best_fitness

# Example usage
num_points = 100
num_knots = 5
X, y = generate_data(num_points)

# Run the Bees Algorithm
best_knots, best_mse = bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)
```

```
print(f"Best knots: {best_knots}")
print(f"Best MSE: {best_mse}")
```

output:-

```
Iteration 1, Best Fitness (MSE): 0.007635018795626422

Iteration 2, Best Fitness (MSE): 0.00760744030305813

Iteration 3, Best Fitness (MSE): 0.007582918279017089

Iteration 4, Best Fitness (MSE): 0.007570685252537971

Iteration 5, Best Fitness (MSE): 0.0075695810141903986

Iteration 6, Best Fitness (MSE): 0.0075635897267201276

Iteration 7, Best Fitness (MSE): 0.007555378459488429

Iteration 8, Best Fitness (MSE): 0.007555378459488429

Iteration 9, Best Fitness (MSE): 0.007555378459488429

Iteration 10, Best Fitness (MSE): 0.007550518641187969

Iteration 11, Best Fitness (MSE): 0.007548131517122354

Iteration 12, Best Fitness (MSE): 0.007544303226772825

Iteration 13, Best Fitness (MSE): 0.007544303226772825

Iteration 14, Best Fitness (MSE): 0.007544016138136744

Iteration 15, Best Fitness (MSE): 0.007543523803434671

Iteration 16, Best Fitness (MSE): 0.007543523803434671

Iteration 17, Best Fitness (MSE): 0.007543523803434671

Iteration 18, Best Fitness (MSE): 0.007543523803434671

Iteration 19, Best Fitness (MSE): 0.007543523803434671

Iteration 20, Best Fitness (MSE): 0.007543523803434671

Iteration 21, Best Fitness (MSE): 0.007543523803434671

Iteration 22, Best Fitness (MSE): 0.007543523803434671

Iteration 23, Best Fitness (MSE): 0.007543523803434671

Iteration 24, Best Fitness (MSE): 0.007543523803434671

Iteration 25, Best Fitness (MSE): 0.007542894851059055
Iteration 26, Best Fitness (MSE): 0.007542894851059055
Iteration 27, Best Fitness (MSE): 0.007542894851059055
Iteration 28, Best Fitness (MSE): 0.007542894851059055
Iteration 29, Best Fitness (MSE): 0.007542894851059055
Iteration 30, Best Fitness (MSE): 0.007542894851059055
Iteration 31, Best Fitness (MSE): 0.007542894851059055
Iteration 32, Best Fitness (MSE): 0.007542894851059055
Iteration 33, Best Fitness (MSE): 0.007542894851059055
Iteration 34, Best Fitness (MSE): 0.007542426461743746
Iteration 35, Best Fitness (MSE): 0.007542426461743746
Iteration 36, Best Fitness (MSE): 0.007542426461743746
Iteration 37, Best Fitness (MSE): 0.007542426461743746
Iteration 38, Best Fitness (MSE): 0.007542426461743746
Iteration 39, Best Fitness (MSE): 0.007542426461743746
Iteration 40, Best Fitness (MSE): 0.007542426461743746
Iteration 41, Best Fitness (MSE): 0.007542426461743746
Iteration 42, Best Fitness (MSE): 0.007542426461743746
Iteration 43, Best Fitness (MSE): 0.007542426461743746
Iteration 44, Best Fitness (MSE): 0.007542426461743746
Iteration 45, Best Fitness (MSE): 0.007542426461743746
Iteration 46, Best Fitness (MSE): 0.007542426461743746
Iteration 47, Best Fitness (MSE): 0.007542426461743746
Iteration 48, Best Fitness (MSE): 0.007542426461743746
Iteration 49, Best Fitness (MSE): 0.007542426461743746
Iteration 50, Best Fitness (MSE): 0.007542426461743746
Iteration 51, Best Fitness (MSE): 0.007542426461743746
Iteration 52, Best Fitness (MSE): 0.007542426461743746
Iteration 53, Best Fitness (MSE): 0.007542426461743746
Iteration 54, Best Fitness (MSE): 0.007542426461743746
Iteration 55, Best Fitness (MSE): 0.007542426461743746
Iteration 56, Best Fitness (MSE): 0.007542426461743746
Iteration 57, Best Fitness (MSE): 0.007542426461743746
Iteration 58, Best Fitness (MSE): 0.007542426461743746
Iteration 59, Best Fitness (MSE): 0.007542426461743746
Iteration 60, Best Fitness (MSE): 0.007542426461743746
Iteration 61, Best Fitness (MSE): 0.007542426461743746
Iteration 62, Best Fitness (MSE): 0.007542426461743746
Iteration 63, Best Fitness (MSE): 0.007542426461743746
Iteration 64, Best Fitness (MSE): 0.007542426461743746
Iteration 65, Best Fitness (MSE): 0.007542426461743746
Iteration 66, Best Fitness (MSE): 0.007542426461743746
Iteration 67, Best Fitness (MSE): 0.007542426461743746
Iteration 68, Best Fitness (MSE): 0.007542426461743746
Iteration 69, Best Fitness (MSE): 0.007542426461743746
Iteration 70, Best Fitness (MSE): 0.007542426461743746
Iteration 71, Best Fitness (MSE): 0.007542426461743746
Iteration 72, Best Fitness (MSE): 0.007542426461743746
Iteration 73, Best Fitness (MSE): 0.007542426461743746
Iteration 74, Best Fitness (MSE): 0.007542426461743746
Iteration 75, Best Fitness (MSE): 0.007542426461743746
Iteration 76, Best Fitness (MSE): 0.007542426461743746
Iteration 77, Best Fitness (MSE): 0.007511842711595226
Iteration 78, Best Fitness (MSE): 0.0074729753811925755
Iteration 79, Best Fitness (MSE): 0.007439259286839678
Iteration 80, Best Fitness (MSE): 0.007423731637410637
Iteration 81, Best Fitness (MSE): 0.007407999140199037
Iteration 82, Best Fitness (MSE): 0.00737499216403501
Iteration 83, Best Fitness (MSE): 0.007363565314970557
Iteration 84, Best Fitness (MSE): 0.007363565314970557
Iteration 85, Best Fitness (MSE): 0.007351431988534319
Iteration 86, Best Fitness (MSE): 0.007351431988534319
Iteration 87, Best Fitness (MSE): 0.007350895491797445
Iteration 88, Best Fitness (MSE): 0.007348685884223418
Iteration 89, Best Fitness (MSE): 0.0073481348183647545
Iteration 90, Best Fitness (MSE): 0.007348045140500258
Iteration 91, Best Fitness (MSE): 0.007348045140500258
Iteration 92, Best Fitness (MSE): 0.007348045140500258
Iteration 93, Best Fitness (MSE): 0.007348039996511657
Iteration 94, Best Fitness (MSE): 0.007348039996511657
Iteration 95, Best Fitness (MSE): 0.007347766679746096
Iteration 96, Best Fitness (MSE): 0.007347766679746096
Iteration 97, Best Fitness (MSE): 0.007347766679746096
Iteration 98, Best Fitness (MSE): 0.007347766679746096
Iteration 99, Best Fitness (MSE): 0.007347766679746096
Iteration 100, Best Fitness (MSE): 0.007347766679746096
Best knots: [2.74189422 3.30518633 5.41807591 6.5220667  8.89016744]
Best MSE: 0.007347766679746096
```

ROOT Mean Squared Error (RMSE)

```
import numpy as np
from scipy.interpolate import LSQUnivariateSpline
from sklearn.metrics import mean_squared_error

# Generate synthetic data
def generate_data(num_points, noise_level=0.1):
    X = np.linspace(0, 10, num_points)
    y = np.sin(X) + np.random.randn(num_points) * noise_level
    return X, y

# Spline model
```

```
def spline_model(X, knots, coefficients):
    t = np.sort(knots)  # Ensure knots are sorted
    spline = LSQUnivariateSpline(X, coefficients, t)
    return spline(X)

# Fitness function (Root Mean Squared Error - RMSE)
def fitness_function(knots, X, y):
    try:
        knots = np.sort(knots)  # Ensure knots are sorted
        t = knots  # Define knots
        spline = LSQUnivariateSpline(X, y, t)
        y_pred = spline(X)
        mse = mean_squared_error(y, y_pred)
        rmse = np.sqrt(mse)  # RMSE Calculation
    except:
        rmse = np.inf  # If fitting fails, set high error
    return rmse

# Bees Algorithm for Spline Regression with RMSE
def bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees (random initial knot positions)
    scout_bees = [np.sort(np.random.uniform(min(X), max(X), num_knots)) for _ in
range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (RMSE) for all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]

        # Rank the bees by their fitness (lower RMSE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))  # Ensure knots
remain within bounds
```

```
        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))

        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.sort(np.random.uniform(min(X), max(X), num_knots))  #
Re-initialize a new random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (RMSE): {best_fitness}")

    return best_solution, best_fitness

# Example usage
num_points = 100
num_knots = 5
X, y = generate_data(num_points)

# Run the Bees Algorithm
best_knots, best_rmse = bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)

print(f"Best knots: {best_knots}")
print(f"Best RMSE: {best_rmse}")
```

output:

```
Iteration 1, Best Fitness (RMSE): 0.0926989888432724
Iteration 2, Best Fitness (RMSE): 0.09174271319540946
Iteration 3, Best Fitness (RMSE): 0.09086671650861276
Iteration 4, Best Fitness (RMSE): 0.09034637174469977
Iteration 5, Best Fitness (RMSE): 0.09002537710631778
Iteration 6, Best Fitness (RMSE): 0.08977117486834893
Iteration 7, Best Fitness (RMSE): 0.08968744268647326
Iteration 8, Best Fitness (RMSE): 0.0896204630361316
Iteration 9, Best Fitness (RMSE): 0.08961767890540925
Iteration 10, Best Fitness (RMSE): 0.08957987281266722
Iteration 11, Best Fitness (RMSE): 0.08957987281266722
Iteration 12, Best Fitness (RMSE): 0.08957987281266722
Iteration 13, Best Fitness (RMSE): 0.08957987281266722
Iteration 14, Best Fitness (RMSE): 0.08957987281266722
Iteration 15, Best Fitness (RMSE): 0.08957773958826779
Iteration 16, Best Fitness (RMSE): 0.08957773958826779
Iteration 17, Best Fitness (RMSE): 0.08957773958826779
Iteration 18, Best Fitness (RMSE): 0.0895745753689867
Iteration 19, Best Fitness (RMSE): 0.0895745753689867
Iteration 20, Best Fitness (RMSE): 0.0895745753689867
Iteration 21, Best Fitness (RMSE): 0.0895745753689867
Iteration 22, Best Fitness (RMSE): 0.0895745753689867
Iteration 23, Best Fitness (RMSE): 0.0895745753689867
Iteration 24, Best Fitness (RMSE): 0.0895745753689867
Iteration 25, Best Fitness (RMSE): 0.0895745753689867
Iteration 26, Best Fitness (RMSE): 0.0895745753689867
Iteration 27, Best Fitness (RMSE): 0.0895745753689867
Iteration 28, Best Fitness (RMSE): 0.0895745753689867
Iteration 29, Best Fitness (RMSE): 0.0895745753689867
Iteration 30, Best Fitness (RMSE): 0.0895745753689867
Iteration 31, Best Fitness (RMSE): 0.0895745753689867
Iteration 32, Best Fitness (RMSE): 0.0895745753689867
Iteration 33, Best Fitness (RMSE): 0.0895745753689867
Iteration 34, Best Fitness (RMSE): 0.0895745753689867
Iteration 35, Best Fitness (RMSE): 0.0895745753689867
Iteration 36, Best Fitness (RMSE): 0.0895745753689867
Iteration 37, Best Fitness (RMSE): 0.0895745753689867
Iteration 38, Best Fitness (RMSE): 0.0895745753689867
Iteration 39, Best Fitness (RMSE): 0.0895745753689867
Iteration 40, Best Fitness (RMSE): 0.0895745753689867
Iteration 41, Best Fitness (RMSE): 0.0895745753689867
Iteration 42, Best Fitness (RMSE): 0.0895745753689867
Iteration 43, Best Fitness (RMSE): 0.0895745753689867
Iteration 44, Best Fitness (RMSE): 0.0895745753689867
Iteration 45, Best Fitness (RMSE): 0.0895745753689867
Iteration 46, Best Fitness (RMSE): 0.0895745753689867
Iteration 47, Best Fitness (RMSE): 0.0895745753689867
Iteration 48, Best Fitness (RMSE): 0.0895745753689867
Iteration 49, Best Fitness (RMSE): 0.0895745753689867
Iteration 50, Best Fitness (RMSE): 0.0895745753689867
Iteration 51, Best Fitness (RMSE): 0.0895745753689867
Iteration 52, Best Fitness (RMSE): 0.0895745753689867
Iteration 53, Best Fitness (RMSE): 0.0895745753689867
Iteration 54, Best Fitness (RMSE): 0.0895745753689867
Iteration 55, Best Fitness (RMSE): 0.0895745753689867
Iteration 56, Best Fitness (RMSE): 0.0895745753689867
Iteration 57, Best Fitness (RMSE): 0.0895745753689867
Iteration 58, Best Fitness (RMSE): 0.0895745753689867
Iteration 59, Best Fitness (RMSE): 0.0895745753689867
Iteration 60, Best Fitness (RMSE): 0.0895745753689867
Iteration 61, Best Fitness (RMSE): 0.0895745753689867
Iteration 62, Best Fitness (RMSE): 0.0895745753689867
Iteration 63, Best Fitness (RMSE): 0.0895745753689867
Iteration 64, Best Fitness (RMSE): 0.0895745753689867
Iteration 65, Best Fitness (RMSE): 0.08955528724893937
Iteration 66, Best Fitness (RMSE): 0.08948750907501865
Iteration 67, Best Fitness (RMSE): 0.08935898040854791
Iteration 68, Best Fitness (RMSE): 0.08927150658304168
Iteration 69, Best Fitness (RMSE): 0.08908395494934938
Iteration 70, Best Fitness (RMSE): 0.08890662098697619
Iteration 71, Best Fitness (RMSE): 0.08870274714285042
Iteration 72, Best Fitness (RMSE): 0.08863769246861773
Iteration 73, Best Fitness (RMSE): 0.08861394975909777
Iteration 74, Best Fitness (RMSE): 0.08859071177067371
Iteration 75, Best Fitness (RMSE): 0.08857729796680751
Iteration 76, Best Fitness (RMSE): 0.08857729796680751
Iteration 77, Best Fitness (RMSE): 0.08857729796680751
Iteration 78, Best Fitness (RMSE): 0.08857521276279387
Iteration 79, Best Fitness (RMSE): 0.08856929883959644
Iteration 80, Best Fitness (RMSE): 0.08856929883959644
Iteration 81, Best Fitness (RMSE): 0.08856929883959644
Iteration 82, Best Fitness (RMSE): 0.08856929883959644
Iteration 83, Best Fitness (RMSE): 0.08856900762929003
Iteration 84, Best Fitness (RMSE): 0.08856900762929003
Iteration 85, Best Fitness (RMSE): 0.08856900762929003

Iteration 86, Best Fitness (RMSE): 0.08856900762929003

Iteration 87, Best Fitness (RMSE): 0.08856620251234458

Iteration 88, Best Fitness (RMSE): 0.08856428708952821

Iteration 89, Best Fitness (RMSE): 0.08856428708952821

Iteration 90, Best Fitness (RMSE): 0.08856428708952821

Iteration 91, Best Fitness (RMSE): 0.08856428708952821

Iteration 92, Best Fitness (RMSE): 0.08856428708952821

Iteration 93, Best Fitness (RMSE): 0.08856428708952821

Iteration 94, Best Fitness (RMSE): 0.08856428708952821

Iteration 95, Best Fitness (RMSE): 0.08856428708952821

Iteration 96, Best Fitness (RMSE): 0.08856428708952821

Iteration 97, Best Fitness (RMSE): 0.08856428708952821

Iteration 98, Best Fitness (RMSE): 0.08856428708952821

Iteration 99, Best Fitness (RMSE): 0.08856428708952821

Iteration 100, Best Fitness (RMSE): 0.08856428708952821

Best knots: [2.4328861  4.51594603 7.07057996 9.92583963 9.14646938]

Best RMSE: 0.08856428708952821
```

MEAN ABSOLUTE ERROR :

```
import numpy as np
from scipy.interpolate import LSQUnivariateSpline
from sklearn.metrics import mean_absolute_error

# Generate synthetic data
def generate_data(num_points, noise_level=0.1):
    X = np.linspace(0, 10, num_points)
    y = np.sin(X) + np.random.randn(num_points) * noise_level
    return X, y

# Fitness function (Mean Absolute Error - MAE)
def fitness_function(knots, X, y):
    try:
        knots = np.sort(knots)  # Ensure knots are sorted
        spline = LSQUnivariateSpline(X, y, knots)
        y_pred = spline(X)
        mae = mean_absolute_error(y, y_pred)  # MAE Calculation
    except:
        mae = np.inf  # If fitting fails, set high error
    return mae

# Bees Algorithm for Spline Regression with MAE
def bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1,
stlim=10, max_iter=100):
    # Initialize scout bees (random initial knot positions)
    scout_bees = [np.sort(np.random.uniform(min(X), max(X), num_knots)) for _ in
range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (MAE) for all scout bees
        fitness_values = [fitness_function(bee, X, y) for bee in scout_bees]

        # Rank the bees by their fitness (lower MAE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))  # Ensure knots
remain within bounds

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_knots)
                new_solutions.append(np.clip(new_bee, min(X), max(X)))

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y) for bee in new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))
```

```
        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.sort(np.random.uniform(min(X), max(X), num_knots))  #
Re-initialize a new random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (MAE): {best_fitness}")

    return best_solution, best_fitness

# Example usage
num_points = 100
num_knots = 5
X, y = generate_data(num_points)

# Run the Bees Algorithm
best_knots, best_mae = bees_algorithm(X, y, num_knots, ns=50, ne=5, nb=10,
nre=10, nrb=5, ngh=0.1, stlim=10, max_iter=100)

print(f"Best knots: {best_knots}")
print(f"Best MAE: {best_mae}")
```

output:

```
Iteration 1, Best Fitness (MAE): 0.07820439673367725

Iteration 2, Best Fitness (MAE): 0.07786504476282095

Iteration 3, Best Fitness (MAE): 0.07762962797900207

Iteration 4, Best Fitness (MAE): 0.07743499456153288
Iteration 5, Best Fitness (MAE): 0.07740737926579105
Iteration 6, Best Fitness (MAE): 0.07734980504097297
Iteration 7, Best Fitness (MAE): 0.07731240891120987
Iteration 8, Best Fitness (MAE): 0.07726467241524705
Iteration 9, Best Fitness (MAE): 0.07726467241524705
Iteration 10, Best Fitness (MAE): 0.07720138690457125
Iteration 11, Best Fitness (MAE): 0.07720138690457125
Iteration 12, Best Fitness (MAE): 0.07718940100909472
Iteration 13, Best Fitness (MAE): 0.07718940100909472
Iteration 14, Best Fitness (MAE): 0.07717406679635057
Iteration 15, Best Fitness (MAE): 0.07717406679635057
Iteration 16, Best Fitness (MAE): 0.07717406679635057
Iteration 17, Best Fitness (MAE): 0.07717406679635057
Iteration 18, Best Fitness (MAE): 0.07717406679635057
Iteration 19, Best Fitness (MAE): 0.07717406679635057
Iteration 20, Best Fitness (MAE): 0.07716871609695816
Iteration 21, Best Fitness (MAE): 0.07716871609695816
Iteration 22, Best Fitness (MAE): 0.07716871609695816
Iteration 23, Best Fitness (MAE): 0.07716871609695816
Iteration 24, Best Fitness (MAE): 0.07716871609695816
Iteration 25, Best Fitness (MAE): 0.07716871609695816
Iteration 26, Best Fitness (MAE): 0.07716871609695816
Iteration 27, Best Fitness (MAE): 0.07716871609695816
Iteration 28, Best Fitness (MAE): 0.07716871609695816
Iteration 29, Best Fitness (MAE): 0.07716871609695816
Iteration 30, Best Fitness (MAE): 0.07716871609695816
Iteration 31, Best Fitness (MAE): 0.07716871609695816
Iteration 32, Best Fitness (MAE): 0.07660770711566918
Iteration 33, Best Fitness (MAE): 0.07577465658897171
Iteration 34, Best Fitness (MAE): 0.0756271465862008
Iteration 35, Best Fitness (MAE): 0.07559960970955797
Iteration 36, Best Fitness (MAE): 0.07553571136791642
Iteration 37, Best Fitness (MAE): 0.07553571136791642
Iteration 38, Best Fitness (MAE): 0.07550172865647284
Iteration 39, Best Fitness (MAE): 0.07538905390115219
Iteration 40, Best Fitness (MAE): 0.07534836857006508
Iteration 41, Best Fitness (MAE): 0.07534836857006508
Iteration 42, Best Fitness (MAE): 0.07533451143939077
Iteration 43, Best Fitness (MAE): 0.07523172769653731
Iteration 44, Best Fitness (MAE): 0.07509366915027255
Iteration 45, Best Fitness (MAE): 0.07507711769160984
Iteration 46, Best Fitness (MAE): 0.07495608472779906
Iteration 47, Best Fitness (MAE): 0.0749015709225445
Iteration 48, Best Fitness (MAE): 0.0749015709225445
Iteration 49, Best Fitness (MAE): 0.0749015709225445
Iteration 50, Best Fitness (MAE): 0.07485586108958577
Iteration 51, Best Fitness (MAE): 0.07485586108958577
Iteration 52, Best Fitness (MAE): 0.07485586108958577
Iteration 53, Best Fitness (MAE): 0.07485586108958577
Iteration 54, Best Fitness (MAE): 0.07485586108958577
Iteration 55, Best Fitness (MAE): 0.07485586108958577
Iteration 56, Best Fitness (MAE): 0.07485586108958577
Iteration 57, Best Fitness (MAE): 0.07484074178141349
Iteration 58, Best Fitness (MAE): 0.07484074178141349
Iteration 59, Best Fitness (MAE): 0.07484074178141349
Iteration 60, Best Fitness (MAE): 0.07484074178141349
Iteration 61, Best Fitness (MAE): 0.07484074178141349
Iteration 62, Best Fitness (MAE): 0.07484074178141349
Iteration 63, Best Fitness (MAE): 0.07484074178141349
Iteration 64, Best Fitness (MAE): 0.07484074178141349
Iteration 65, Best Fitness (MAE): 0.07484074178141349
Iteration 66, Best Fitness (MAE): 0.07484074178141349
Iteration 67, Best Fitness (MAE): 0.07484074178141349
Iteration 68, Best Fitness (MAE): 0.07484074178141349
Iteration 69, Best Fitness (MAE): 0.07484074178141349
Iteration 70, Best Fitness (MAE): 0.07484074178141349
Iteration 71, Best Fitness (MAE): 0.07484074178141349
Iteration 72, Best Fitness (MAE): 0.07484074178141349
Iteration 73, Best Fitness (MAE): 0.07484074178141349
Iteration 74, Best Fitness (MAE): 0.07484074178141349
Iteration 75, Best Fitness (MAE): 0.07484074178141349
Iteration 76, Best Fitness (MAE): 0.07484074178141349
Iteration 77, Best Fitness (MAE): 0.07484074178141349
Iteration 78, Best Fitness (MAE): 0.07484074178141349
Iteration 79, Best Fitness (MAE): 0.07484074178141349
Iteration 80, Best Fitness (MAE): 0.07484074178141349
Iteration 81, Best Fitness (MAE): 0.07484074178141349
Iteration 82, Best Fitness (MAE): 0.07484074178141349
Iteration 83, Best Fitness (MAE): 0.07484074178141349
Iteration 84, Best Fitness (MAE): 0.07484074178141349
Iteration 85, Best Fitness (MAE): 0.07484074178141349
Iteration 86, Best Fitness (MAE): 0.07484074178141349
Iteration 87, Best Fitness (MAE): 0.07484074178141349
Iteration 88, Best Fitness (MAE): 0.07484074178141349
Iteration 89, Best Fitness (MAE): 0.07484074178141349
Iteration 90, Best Fitness (MAE): 0.07484074178141349
Iteration 91, Best Fitness (MAE): 0.07484074178141349
Iteration 92, Best Fitness (MAE): 0.07484074178141349

Iteration 93, Best Fitness (MAE): 0.07484074178141349

Iteration 94, Best Fitness (MAE): 0.07484074178141349

Iteration 95, Best Fitness (MAE): 0.07484074178141349

Iteration 96, Best Fitness (MAE): 0.07484074178141349

Iteration 97, Best Fitness (MAE): 0.07484074178141349

Iteration 98, Best Fitness (MAE): 0.07484074178141349

Iteration 99, Best Fitness (MAE): 0.07484074178141349

Iteration 100, Best Fitness (MAE): 0.07484074178141349

Best knots: [1.94956977 4.68331213 7.98678516 9.51555858 9.67825527]

Best MAE: 0.07484074178141349
```

POLYNOMIAL PARAMETER BENCHMARKING CODE

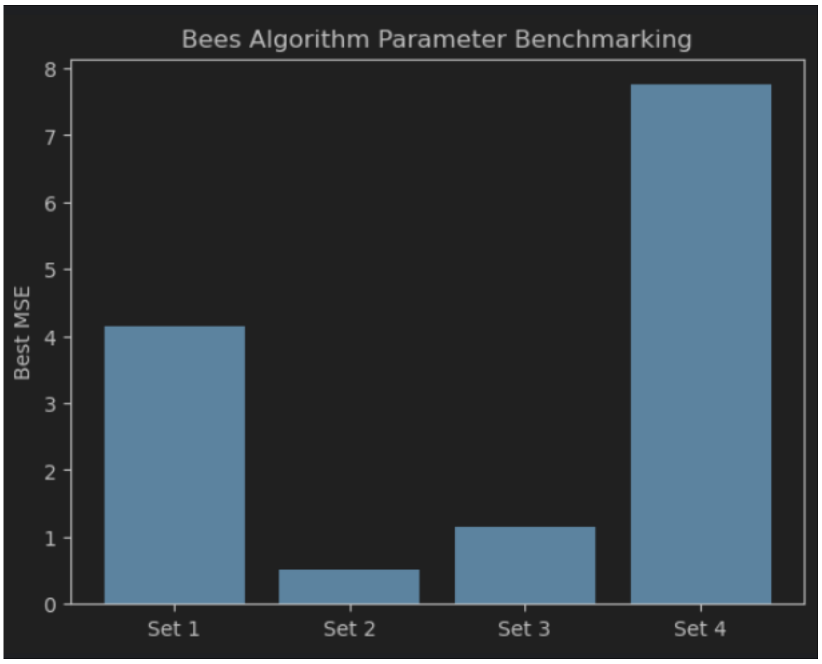
```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Generate synthetic data (Polynomial) for different functions
def generate_data(num_points, function='sin', noise_level=0.1):
    X = np.linspace(0, 10, num_points).reshape(-1, 1)
    if function == 'sin':
        y = np.sin(X).ravel() + np.random.randn(num_points) * noise_level
    elif function == 'log':
        y = np.log(X + 1).ravel() + np.random.randn(num_points) * noise_level
    elif function == 'tan':
        y = np.tan(X).ravel() + np.random.randn(num_points) * noise_level
    else:
        raise ValueError("Unknown function. Choose from 'sin', 'log', or 'tan'.")
    return X, y

# Polynomial model fitting
def polynomial_model(X, coefficients, degree):
    poly = PolynomialFeatures(degree)
    X_poly = poly.fit_transform(X)
    return np.dot(X_poly, coefficients)

# Fitness function (Mean Squared Error - MSE)
def fitness_function(coefficients, X, y, degree):
    try:
        y_pred = polynomial_model(X, coefficients, degree)
        mse = mean_squared_error(y, y_pred)
    except Exception as e:
        print(f"Error during polynomial fitting: {e}")
        mse = np.inf
    return mse

# Bees Algorithm for Polynomial Regression
def bees_algorithm(X, y, degree, num_coefficients, ns, ne, nb, nre, nrb, ngh, stlim,
max_iter=100):
    # Initialize scout bees (random initial polynomial coefficients)
    scout_bees = [np.random.randn(num_coefficients) for _ in range(ns)]

    # Stagnation counter to track site abandonment
    stagnation_counter = np.zeros(ns)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(max_iter):
        # Evaluate fitness (MSE) for all scout bees
        fitness_values = [fitness_function(bee, X, y, degree) for bee in scout_bees]

        # Rank the bees by their fitness (lower MSE is better)
        ranked_bees = sorted(zip(scout_bees, fitness_values), key=lambda x: x[1])

        # Select elite and best sites
        elite_sites = ranked_bees[:ne]
        best_sites = ranked_bees[ne:nb+ne]

        # Recruit bees for elite sites
        new_solutions = []
        for site, fit in elite_sites:
            for _ in range(nre):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_coefficients)
                new_solutions.append(new_bee)

        # Recruit bees for best sites
        for site, fit in best_sites:
            for _ in range(nrb):
                new_bee = site + np.random.uniform(-ngh, ngh, size=num_coefficients)
                new_solutions.append(new_bee)

        # Update solutions with new bees
        new_fitness_values = [fitness_function(bee, X, y, degree) for bee in
new_solutions]
        combined_solutions = ranked_bees + list(zip(new_solutions,
new_fitness_values))
```

```
        # Sort combined solutions by fitness
        combined_solutions.sort(key=lambda x: x[1])

        # Keep the best ns solutions as new scout bees
        scout_bees = [sol[0] for sol in combined_solutions[:ns]]

        # Update best solution
        if combined_solutions[0][1] < best_fitness:
            best_solution = combined_solutions[0][0]
            best_fitness = combined_solutions[0][1]
            stagnation_counter.fill(0)  # Reset stagnation counter if improvement occurs
        else:
            stagnation_counter += 1

        # Site abandonment if stagnation occurs
        for i in range(ns):
            if stagnation_counter[i] >= stlim:
                scout_bees[i] = np.random.randn(num_coefficients)  # Re-initialize a new
random bee
                stagnation_counter[i] = 0

        print(f"Iteration {iteration+1}, Best Fitness (MSE): {best_fitness}")

    return best_solution, best_fitness

# Benchmarking and testing Bees Algorithm with different parameters
def benchmark_bees_algorithm(X, y, degree, num_coefficients, param_combinations):
    results = []
    for params in param_combinations:
        ns, ne, nb, nre, nrb, ngh, stlim = params
        print(f"Running with params: ns={ns}, ne={ne}, nb={nb}, nre={nre},
nrb={nrb}, ngh={ngh}, stlim={stlim}")
        best_solution, best_mse = bees_algorithm(X, y, degree, num_coefficients, ns, ne,
nb, nre, nrb, ngh, stlim)
        results.append((params, best_mse))
    return results

# Example usage and benchmarking
if __name__ == "__main__":
    # Data generation
    num_points = 100
    degree = 4  # Polynomial degree
    X, y = generate_data(num_points, function='sin')

    # Define possible parameters for benchmarking
    param_combinations = [
        (50, 5, 10, 10, 5, 0.1, 10),  # Set 1
        (30, 3, 7, 7, 3, 0.2, 15),    # Set 2
        (70, 10, 15, 15, 7, 0.05, 20), # Set 3
        (40, 4, 8, 8, 4, 0.15, 12),   # Set 4
    ]

    # Run benchmarking
    num_coefficients = degree + 1  # Number of polynomial coefficients
    results = benchmark_bees_algorithm(X, y, degree, num_coefficients,
param_combinations)

    # Output results
    for params, mse in results:
        print(f"Params: {params} => Best MSE: {mse}")

    # Plot the results
    param_labels = [f"Set {i+1}" for i in range(len(param_combinations))]
    mse_values = [mse for _, mse in results]

    plt.bar(param_labels, mse_values)
    plt.ylabel('Best MSE')
    plt.title('Bees Algorithm Parameter Benchmarking')
    plt.show()
```

output:

```
Running with params: ns=50, ne=5, nb=10, nre=10, nrb=5, ngh=0.1, stlim=10

Iteration 1, Best Fitness (MSE): 1233.5513942712596

Iteration 2, Best Fitness (MSE): 25.051099976249983

Iteration 3, Best Fitness (MSE): 25.051099976249983

Iteration 4, Best Fitness (MSE): 25.051099976249983

Iteration 5, Best Fitness (MSE): 25.051099976249983

Iteration 6, Best Fitness (MSE): 25.051099976249983

Iteration 7, Best Fitness (MSE): 25.051099976249983

Iteration 8, Best Fitness (MSE): 25.051099976249983

Iteration 9, Best Fitness (MSE): 25.051099976249983

Iteration 10, Best Fitness (MSE): 25.051099976249983

Iteration 11, Best Fitness (MSE): 25.051099976249983

Iteration 12, Best Fitness (MSE): 25.051099976249983

Iteration 13, Best Fitness (MSE): 25.051099976249983

Iteration 14, Best Fitness (MSE): 25.051099976249983
Iteration 15, Best Fitness (MSE): 25.051099976249983
Iteration 16, Best Fitness (MSE): 25.051099976249983
Iteration 17, Best Fitness (MSE): 25.051099976249983
Iteration 18, Best Fitness (MSE): 25.051099976249983
Iteration 19, Best Fitness (MSE): 25.051099976249983
Iteration 20, Best Fitness (MSE): 25.051099976249983
Iteration 21, Best Fitness (MSE): 25.051099976249983
Iteration 22, Best Fitness (MSE): 25.051099976249983
Iteration 23, Best Fitness (MSE): 25.051099976249983
Iteration 24, Best Fitness (MSE): 25.051099976249983
Iteration 25, Best Fitness (MSE): 25.051099976249983
Iteration 26, Best Fitness (MSE): 25.051099976249983
Iteration 27, Best Fitness (MSE): 25.051099976249983
Iteration 28, Best Fitness (MSE): 25.051099976249983
Iteration 29, Best Fitness (MSE): 25.051099976249983
Iteration 30, Best Fitness (MSE): 25.051099976249983
Iteration 31, Best Fitness (MSE): 25.051099976249983
Iteration 32, Best Fitness (MSE): 25.051099976249983
Iteration 33, Best Fitness (MSE): 25.051099976249983
Iteration 34, Best Fitness (MSE): 25.051099976249983
Iteration 35, Best Fitness (MSE): 15.521314955068744
Iteration 36, Best Fitness (MSE): 15.521314955068744
Iteration 37, Best Fitness (MSE): 5.963041412060867
Iteration 38, Best Fitness (MSE): 5.731452230585474
Iteration 39, Best Fitness (MSE): 5.731452230585474
Iteration 40, Best Fitness (MSE): 5.731452230585474
Iteration 41, Best Fitness (MSE): 5.731452230585474
Iteration 42, Best Fitness (MSE): 5.731452230585474
Iteration 43, Best Fitness (MSE): 5.731452230585474
Iteration 44, Best Fitness (MSE): 4.146221883163713
Iteration 45, Best Fitness (MSE): 4.146221883163713
Iteration 46, Best Fitness (MSE): 4.146221883163713
Iteration 47, Best Fitness (MSE): 4.146221883163713
Iteration 48, Best Fitness (MSE): 4.146221883163713
Iteration 49, Best Fitness (MSE): 4.146221883163713
Iteration 50, Best Fitness (MSE): 4.146221883163713
Iteration 51, Best Fitness (MSE): 4.146221883163713
Iteration 52, Best Fitness (MSE): 4.146221883163713
Iteration 53, Best Fitness (MSE): 4.146221883163713
Iteration 54, Best Fitness (MSE): 4.146221883163713
Iteration 55, Best Fitness (MSE): 4.146221883163713
Iteration 56, Best Fitness (MSE): 4.146221883163713
Iteration 57, Best Fitness (MSE): 4.146221883163713
Iteration 58, Best Fitness (MSE): 4.146221883163713
Iteration 59, Best Fitness (MSE): 4.146221883163713
Iteration 60, Best Fitness (MSE): 4.146221883163713
Iteration 61, Best Fitness (MSE): 4.146221883163713
Iteration 62, Best Fitness (MSE): 4.146221883163713
Iteration 63, Best Fitness (MSE): 4.146221883163713
Iteration 64, Best Fitness (MSE): 4.146221883163713
Iteration 65, Best Fitness (MSE): 4.146221883163713
Iteration 66, Best Fitness (MSE): 4.146221883163713
Iteration 67, Best Fitness (MSE): 4.146221883163713
Iteration 68, Best Fitness (MSE): 4.146221883163713
Iteration 69, Best Fitness (MSE): 4.146221883163713
Iteration 70, Best Fitness (MSE): 4.146221883163713
Iteration 71, Best Fitness (MSE): 4.146221883163713
Iteration 72, Best Fitness (MSE): 4.146221883163713
Iteration 73, Best Fitness (MSE): 4.146221883163713
Iteration 74, Best Fitness (MSE): 4.146221883163713
Iteration 75, Best Fitness (MSE): 4.146221883163713
Iteration 76, Best Fitness (MSE): 4.146221883163713
Iteration 77, Best Fitness (MSE): 4.146221883163713
Iteration 78, Best Fitness (MSE): 4.146221883163713
Iteration 79, Best Fitness (MSE): 4.146221883163713
Iteration 80, Best Fitness (MSE): 4.146221883163713
Iteration 81, Best Fitness (MSE): 4.146221883163713
Iteration 82, Best Fitness (MSE): 4.146221883163713
Iteration 83, Best Fitness (MSE): 4.146221883163713
Iteration 84, Best Fitness (MSE): 4.146221883163713
Iteration 85, Best Fitness (MSE): 4.146221883163713
Iteration 86, Best Fitness (MSE): 4.146221883163713
Iteration 87, Best Fitness (MSE): 4.146221883163713
Iteration 88, Best Fitness (MSE): 4.146221883163713
Iteration 89, Best Fitness (MSE): 4.146221883163713
Iteration 90, Best Fitness (MSE): 4.146221883163713
Iteration 91, Best Fitness (MSE): 4.146221883163713
Iteration 92, Best Fitness (MSE): 4.146221883163713
Iteration 93, Best Fitness (MSE): 4.146221883163713
Iteration 94, Best Fitness (MSE): 4.146221883163713
Iteration 95, Best Fitness (MSE): 4.146221883163713
Iteration 96, Best Fitness (MSE): 4.146221883163713
Iteration 97, Best Fitness (MSE): 4.146221883163713
Iteration 98, Best Fitness (MSE): 4.146221883163713
Iteration 99, Best Fitness (MSE): 4.146221883163713
Iteration 100, Best Fitness (MSE): 4.146221883163713
Running with params: ns=30, ne=3, nb=7, nre=7, nrb=3, ngh=0.2, stlim=15
```

```
Iteration 1, Best Fitness (MSE): 324.4389402456888
Iteration 2, Best Fitness (MSE): 324.4389402456888
Iteration 3, Best Fitness (MSE): 203.04212416272574
Iteration 4, Best Fitness (MSE): 52.04779174423587
Iteration 5, Best Fitness (MSE): 52.04779174423587
Iteration 6, Best Fitness (MSE): 52.04779174423587
Iteration 7, Best Fitness (MSE): 52.04779174423587
Iteration 8, Best Fitness (MSE): 52.04779174423587
Iteration 9, Best Fitness (MSE): 20.422954229586864
Iteration 10, Best Fitness (MSE): 20.422954229586864
Iteration 11, Best Fitness (MSE): 20.422954229586864
Iteration 12, Best Fitness (MSE): 20.422954229586864
Iteration 13, Best Fitness (MSE): 20.422954229586864
Iteration 14, Best Fitness (MSE): 1.6712875234351992
Iteration 15, Best Fitness (MSE): 1.6712875234351992
Iteration 16, Best Fitness (MSE): 1.6712875234351992
Iteration 17, Best Fitness (MSE): 1.6712875234351992
Iteration 18, Best Fitness (MSE): 1.6712875234351992
Iteration 19, Best Fitness (MSE): 1.6712875234351992
Iteration 20, Best Fitness (MSE): 1.6712875234351992
Iteration 21, Best Fitness (MSE): 1.6712875234351992
Iteration 22, Best Fitness (MSE): 1.6712875234351992
Iteration 23, Best Fitness (MSE): 1.6712875234351992
Iteration 24, Best Fitness (MSE): 1.6712875234351992
Iteration 25, Best Fitness (MSE): 1.6712875234351992
Iteration 26, Best Fitness (MSE): 1.6712875234351992
Iteration 27, Best Fitness (MSE): 1.6712875234351992
Iteration 28, Best Fitness (MSE): 0.5171597454657813
Iteration 29, Best Fitness (MSE): 0.5171597454657813
Iteration 30, Best Fitness (MSE): 0.5171597454657813
Iteration 31, Best Fitness (MSE): 0.5171597454657813
Iteration 32, Best Fitness (MSE): 0.5171597454657813
Iteration 33, Best Fitness (MSE): 0.5171597454657813
Iteration 34, Best Fitness (MSE): 0.5171597454657813
Iteration 35, Best Fitness (MSE): 0.5171597454657813
Iteration 36, Best Fitness (MSE): 0.5171597454657813
Iteration 37, Best Fitness (MSE): 0.5171597454657813
Iteration 38, Best Fitness (MSE): 0.5171597454657813
Iteration 39, Best Fitness (MSE): 0.5171597454657813
Iteration 40, Best Fitness (MSE): 0.5171597454657813
Iteration 41, Best Fitness (MSE): 0.5171597454657813
Iteration 42, Best Fitness (MSE): 0.5171597454657813
Iteration 43, Best Fitness (MSE): 0.5171597454657813
Iteration 44, Best Fitness (MSE): 0.5171597454657813
Iteration 45, Best Fitness (MSE): 0.5171597454657813
Iteration 46, Best Fitness (MSE): 0.5171597454657813
Iteration 47, Best Fitness (MSE): 0.5171597454657813
Iteration 48, Best Fitness (MSE): 0.5171597454657813
Iteration 49, Best Fitness (MSE): 0.5171597454657813
Iteration 50, Best Fitness (MSE): 0.5171597454657813
Iteration 51, Best Fitness (MSE): 0.5171597454657813
Iteration 52, Best Fitness (MSE): 0.5171597454657813
Iteration 53, Best Fitness (MSE): 0.5171597454657813
Iteration 54, Best Fitness (MSE): 0.5171597454657813
Iteration 55, Best Fitness (MSE): 0.5171597454657813
Iteration 56, Best Fitness (MSE): 0.5171597454657813
Iteration 57, Best Fitness (MSE): 0.5171597454657813
Iteration 58, Best Fitness (MSE): 0.5171597454657813
Iteration 59, Best Fitness (MSE): 0.5171597454657813
Iteration 60, Best Fitness (MSE): 0.5171597454657813
Iteration 61, Best Fitness (MSE): 0.5171597454657813
Iteration 62, Best Fitness (MSE): 0.5171597454657813
Iteration 63, Best Fitness (MSE): 0.5171597454657813
Iteration 64, Best Fitness (MSE): 0.5171597454657813
Iteration 65, Best Fitness (MSE): 0.5171597454657813
Iteration 66, Best Fitness (MSE): 0.5171597454657813
Iteration 67, Best Fitness (MSE): 0.5171597454657813
Iteration 68, Best Fitness (MSE): 0.5171597454657813
Iteration 69, Best Fitness (MSE): 0.5171597454657813
Iteration 70, Best Fitness (MSE): 0.5171597454657813
Iteration 71, Best Fitness (MSE): 0.5171597454657813
Iteration 72, Best Fitness (MSE): 0.5171597454657813
Iteration 73, Best Fitness (MSE): 0.5171597454657813
Iteration 74, Best Fitness (MSE): 0.5171597454657813
Iteration 75, Best Fitness (MSE): 0.5171597454657813
Iteration 76, Best Fitness (MSE): 0.5171597454657813
Iteration 77, Best Fitness (MSE): 0.5171597454657813
Iteration 78, Best Fitness (MSE): 0.5171597454657813
Iteration 79, Best Fitness (MSE): 0.5171597454657813
Iteration 80, Best Fitness (MSE): 0.5171597454657813
Iteration 81, Best Fitness (MSE): 0.5171597454657813
Iteration 82, Best Fitness (MSE): 0.5171597454657813
Iteration 83, Best Fitness (MSE): 0.5171597454657813
Iteration 84, Best Fitness (MSE): 0.5171597454657813
Iteration 85, Best Fitness (MSE): 0.5171597454657813
Iteration 86, Best Fitness (MSE): 0.5171597454657813
Iteration 87, Best Fitness (MSE): 0.5171597454657813
Iteration 88, Best Fitness (MSE): 0.5171597454657813
Iteration 89, Best Fitness (MSE): 0.5171597454657813
Iteration 90, Best Fitness (MSE): 0.5171597454657813
Iteration 91, Best Fitness (MSE): 0.5171597454657813
Iteration 92, Best Fitness (MSE): 0.5171597454657813
Iteration 93, Best Fitness (MSE): 0.5171597454657813
Iteration 94, Best Fitness (MSE): 0.5171597454657813
Iteration 95, Best Fitness (MSE): 0.5171597454657813
Iteration 96, Best Fitness (MSE): 0.5171597454657813
Iteration 97, Best Fitness (MSE): 0.5171597454657813
Iteration 98, Best Fitness (MSE): 0.5171597454657813
Iteration 99, Best Fitness (MSE): 0.5171597454657813
Iteration 100, Best Fitness (MSE): 0.5171597454657813
Running with params: ns=70, ne=10, nb=15, nre=15, nrb=7, ngh=0.05, stlim=20
Iteration 1, Best Fitness (MSE): 225.19512850818194
Iteration 2, Best Fitness (MSE): 184.80200149472566
Iteration 3, Best Fitness (MSE): 133.97556270583308
Iteration 4, Best Fitness (MSE): 116.08179998897226
Iteration 5, Best Fitness (MSE): 75.02091773598461
Iteration 6, Best Fitness (MSE): 75.02091773598461
Iteration 7, Best Fitness (MSE): 46.129330246610486
Iteration 8, Best Fitness (MSE): 46.129330246610486
Iteration 9, Best Fitness (MSE): 32.03922431829224
Iteration 10, Best Fitness (MSE): 31.959507019516586
Iteration 11, Best Fitness (MSE): 31.959507019516586
Iteration 12, Best Fitness (MSE): 26.860308364082208
Iteration 13, Best Fitness (MSE): 21.82915680831447
Iteration 14, Best Fitness (MSE): 21.82915680831447
Iteration 15, Best Fitness (MSE): 9.14913097528285
Iteration 16, Best Fitness (MSE): 9.14913097528285
Iteration 17, Best Fitness (MSE): 9.14913097528285
Iteration 18, Best Fitness (MSE): 9.14913097528285
Iteration 19, Best Fitness (MSE): 1.9648925719356958
Iteration 20, Best Fitness (MSE): 1.9648925719356958
Iteration 21, Best Fitness (MSE): 1.9648925719356958
Iteration 22, Best Fitness (MSE): 1.9648925719356958
Iteration 23, Best Fitness (MSE): 1.1672722320539781
Iteration 24, Best Fitness (MSE): 1.1672722320539781
Iteration 25, Best Fitness (MSE): 1.1672722320539781
Iteration 26, Best Fitness (MSE): 1.1672722320539781
Iteration 27, Best Fitness (MSE): 1.1611733196132596
Iteration 28, Best Fitness (MSE): 1.1611733196132596
Iteration 29, Best Fitness (MSE): 1.1611733196132596
Iteration 30, Best Fitness (MSE): 1.1611733196132596
Iteration 31, Best Fitness (MSE): 1.1611733196132596
Iteration 32, Best Fitness (MSE): 1.1611733196132596
Iteration 33, Best Fitness (MSE): 1.1611733196132596
Iteration 34, Best Fitness (MSE): 1.1412495491154582
Iteration 35, Best Fitness (MSE): 1.1412495491154582
Iteration 36, Best Fitness (MSE): 1.1412495491154582
Iteration 37, Best Fitness (MSE): 1.1412495491154582
Iteration 38, Best Fitness (MSE): 1.1412495491154582
Iteration 39, Best Fitness (MSE): 1.1412495491154582
Iteration 40, Best Fitness (MSE): 1.1412495491154582
Iteration 41, Best Fitness (MSE): 1.1412495491154582
Iteration 42, Best Fitness (MSE): 1.1412495491154582
Iteration 43, Best Fitness (MSE): 1.1412495491154582
Iteration 44, Best Fitness (MSE): 1.1412495491154582
Iteration 45, Best Fitness (MSE): 1.1412495491154582
Iteration 46, Best Fitness (MSE): 1.1412495491154582
Iteration 47, Best Fitness (MSE): 1.1412495491154582
Iteration 48, Best Fitness (MSE): 1.1412495491154582
Iteration 49, Best Fitness (MSE): 1.1412495491154582
Iteration 50, Best Fitness (MSE): 1.1412495491154582
Iteration 51, Best Fitness (MSE): 1.1412495491154582
Iteration 52, Best Fitness (MSE): 1.1412495491154582
Iteration 53, Best Fitness (MSE): 1.1412495491154582
Iteration 54, Best Fitness (MSE): 1.1412495491154582
Iteration 55, Best Fitness (MSE): 1.1412495491154582
Iteration 56, Best Fitness (MSE): 1.1412495491154582
Iteration 57, Best Fitness (MSE): 1.1412495491154582
Iteration 58, Best Fitness (MSE): 1.1412495491154582
Iteration 59, Best Fitness (MSE): 1.1412495491154582
Iteration 60, Best Fitness (MSE): 1.1412495491154582
Iteration 61, Best Fitness (MSE): 1.1412495491154582
Iteration 62, Best Fitness (MSE): 1.1412495491154582
Iteration 63, Best Fitness (MSE): 1.1412495491154582
Iteration 64, Best Fitness (MSE): 1.1412495491154582
Iteration 65, Best Fitness (MSE): 1.1412495491154582
Iteration 66, Best Fitness (MSE): 1.1412495491154582
Iteration 67, Best Fitness (MSE): 1.1412495491154582
Iteration 68, Best Fitness (MSE): 1.1412495491154582
Iteration 69, Best Fitness (MSE): 1.1412495491154582
Iteration 70, Best Fitness (MSE): 1.1412495491154582
Iteration 71, Best Fitness (MSE): 1.1412495491154582
Iteration 72, Best Fitness (MSE): 1.1412495491154582
Iteration 73, Best Fitness (MSE): 1.1412495491154582
Iteration 74, Best Fitness (MSE): 1.1412495491154582
Iteration 75, Best Fitness (MSE): 1.1412495491154582
Iteration 76, Best Fitness (MSE): 1.1412495491154582
Iteration 77, Best Fitness (MSE): 1.1412495491154582
Iteration 78, Best Fitness (MSE): 1.1412495491154582
Iteration 79, Best Fitness (MSE): 1.1412495491154582
Iteration 80, Best Fitness (MSE): 1.1412495491154582
Iteration 81, Best Fitness (MSE): 1.1412495491154582
Iteration 82, Best Fitness (MSE): 1.1412495491154582
Iteration 83, Best Fitness (MSE): 1.1412495491154582
Iteration 84, Best Fitness (MSE): 1.1412495491154582
Iteration 85, Best Fitness (MSE): 1.1412495491154582
Iteration 86, Best Fitness (MSE): 1.1412495491154582
Iteration 87, Best Fitness (MSE): 1.1412495491154582
Iteration 88, Best Fitness (MSE): 1.1412495491154582
Iteration 89, Best Fitness (MSE): 1.1412495491154582
Iteration 90, Best Fitness (MSE): 1.1412495491154582
Iteration 91, Best Fitness (MSE): 1.1412495491154582
Iteration 92, Best Fitness (MSE): 1.1412495491154582
Iteration 93, Best Fitness (MSE): 1.1412495491154582
Iteration 94, Best Fitness (MSE): 1.1412495491154582
Iteration 95, Best Fitness (MSE): 1.1412495491154582
Iteration 96, Best Fitness (MSE): 1.1412495491154582
Iteration 97, Best Fitness (MSE): 1.1412495491154582
Iteration 98, Best Fitness (MSE): 1.1412495491154582
Iteration 99, Best Fitness (MSE): 1.1412495491154582
Iteration 100, Best Fitness (MSE): 1.1412495491154582
Running with params: ns=40, ne=4, nb=8, nre=8, nrb=4, ngh=0.15, stlim=12
Iteration 1, Best Fitness (MSE): 1010.8357341847606
Iteration 2, Best Fitness (MSE): 1010.8357341847606
Iteration 3, Best Fitness (MSE): 1010.8357341847606
Iteration 4, Best Fitness (MSE): 1010.8357341847606
Iteration 5, Best Fitness (MSE): 1010.8357341847606
Iteration 6, Best Fitness (MSE): 278.78325150917897
Iteration 7, Best Fitness (MSE): 278.78325150917897
Iteration 8, Best Fitness (MSE): 278.78325150917897
Iteration 9, Best Fitness (MSE): 137.47563952650938
Iteration 10, Best Fitness (MSE): 137.47563952650938
Iteration 11, Best Fitness (MSE): 137.47563952650938
Iteration 12, Best Fitness (MSE): 137.47563952650938
Iteration 13, Best Fitness (MSE): 137.47563952650938
Iteration 14, Best Fitness (MSE): 137.47563952650938
Iteration 15, Best Fitness (MSE): 137.47563952650938
Iteration 16, Best Fitness (MSE): 137.47563952650938
Iteration 17, Best Fitness (MSE): 137.47563952650938
Iteration 18, Best Fitness (MSE): 137.47563952650938
Iteration 19, Best Fitness (MSE): 137.47563952650938
Iteration 20, Best Fitness (MSE): 59.16886888815643
Iteration 21, Best Fitness (MSE): 59.16886888815643
Iteration 22, Best Fitness (MSE): 59.16886888815643
Iteration 23, Best Fitness (MSE): 59.16886888815643
Iteration 24, Best Fitness (MSE): 59.16886888815643
Iteration 25, Best Fitness (MSE): 59.16886888815643
Iteration 26, Best Fitness (MSE): 59.16886888815643
Iteration 27, Best Fitness (MSE): 59.16886888815643
Iteration 28, Best Fitness (MSE): 59.16886888815643
Iteration 29, Best Fitness (MSE): 59.16886888815643
Iteration 30, Best Fitness (MSE): 59.16886888815643
Iteration 31, Best Fitness (MSE): 59.16886888815643
Iteration 32, Best Fitness (MSE): 54.39741860428141
Iteration 33, Best Fitness (MSE): 54.39741860428141
Iteration 34, Best Fitness (MSE): 54.39741860428141
Iteration 35, Best Fitness (MSE): 27.82514580280033
Iteration 36, Best Fitness (MSE): 27.82514580280033
Iteration 37, Best Fitness (MSE): 27.82514580280033
Iteration 38, Best Fitness (MSE): 27.82514580280033
Iteration 39, Best Fitness (MSE): 27.82514580280033
Iteration 40, Best Fitness (MSE): 27.82514580280033
Iteration 41, Best Fitness (MSE): 26.804890045843408
Iteration 42, Best Fitness (MSE): 26.804890045843408
Iteration 43, Best Fitness (MSE): 26.804890045843408
Iteration 44, Best Fitness (MSE): 26.804890045843408
Iteration 45, Best Fitness (MSE): 26.804890045843408
Iteration 46, Best Fitness (MSE): 26.804890045843408
Iteration 47, Best Fitness (MSE): 26.804890045843408
Iteration 48, Best Fitness (MSE): 26.804890045843408
Iteration 49, Best Fitness (MSE): 26.804890045843408
Iteration 50, Best Fitness (MSE): 26.804890045843408
Iteration 51, Best Fitness (MSE): 26.804890045843408
Iteration 52, Best Fitness (MSE): 8.17416015148346
Iteration 53, Best Fitness (MSE): 8.17416015148346
Iteration 54, Best Fitness (MSE): 8.17416015148346
Iteration 55, Best Fitness (MSE): 8.17416015148346
Iteration 56, Best Fitness (MSE): 8.17416015148346
Iteration 57, Best Fitness (MSE): 8.17416015148346
Iteration 58, Best Fitness (MSE): 8.17416015148346
Iteration 59, Best Fitness (MSE): 8.17416015148346
Iteration 60, Best Fitness (MSE): 8.17416015148346
Iteration 61, Best Fitness (MSE): 8.17416015148346
Iteration 62, Best Fitness (MSE): 8.17416015148346
Iteration 63, Best Fitness (MSE): 8.17416015148346
Iteration 64, Best Fitness (MSE): 8.17416015148346
Iteration 65, Best Fitness (MSE): 8.17416015148346
Iteration 66, Best Fitness (MSE): 8.17416015148346
Iteration 67, Best Fitness (MSE): 8.17416015148346
Iteration 68, Best Fitness (MSE): 8.17416015148346
Iteration 69, Best Fitness (MSE): 8.17416015148346
Iteration 70, Best Fitness (MSE): 8.17416015148346
Iteration 71, Best Fitness (MSE): 8.17416015148346
Iteration 72, Best Fitness (MSE): 8.17416015148346
Iteration 73, Best Fitness (MSE): 8.17416015148346
Iteration 74, Best Fitness (MSE): 8.17416015148346
Iteration 75, Best Fitness (MSE): 8.17416015148346
Iteration 76, Best Fitness (MSE): 8.17416015148346
Iteration 77, Best Fitness (MSE): 8.17416015148346
Iteration 78, Best Fitness (MSE): 8.17416015148346
Iteration 79, Best Fitness (MSE): 8.17416015148346
Iteration 80, Best Fitness (MSE): 8.17416015148346
Iteration 81, Best Fitness (MSE): 8.17416015148346
Iteration 82, Best Fitness (MSE): 8.17416015148346
Iteration 83, Best Fitness (MSE): 8.17416015148346
Iteration 84, Best Fitness (MSE): 8.17416015148346
Iteration 85, Best Fitness (MSE): 8.17416015148346
Iteration 86, Best Fitness (MSE): 8.17416015148346
Iteration 87, Best Fitness (MSE): 8.17416015148346
Iteration 88, Best Fitness (MSE): 8.17416015148346
Iteration 89, Best Fitness (MSE): 8.17416015148346
Iteration 90, Best Fitness (MSE): 8.17416015148346
Iteration 91, Best Fitness (MSE): 8.17416015148346
Iteration 92, Best Fitness (MSE): 8.17416015148346
Iteration 93, Best Fitness (MSE): 8.17416015148346
Iteration 94, Best Fitness (MSE): 8.17416015148346
Iteration 95, Best Fitness (MSE): 8.17416015148346
Iteration 96, Best Fitness (MSE): 8.17416015148346
Iteration 97, Best Fitness (MSE): 8.17416015148346
Iteration 98, Best Fitness (MSE): 8.17416015148346
Iteration 99, Best Fitness (MSE): 8.17416015148346
Iteration 100, Best Fitness (MSE): 7.752349188019787
Params: (50, 5, 10, 10, 5, 0.1, 10) => Best MSE: 4.146221883163713
Params: (30, 3, 7, 7, 3, 0.2, 15) => Best MSE: 0.5171597454657813
Params: (70, 10, 15, 15, 7, 0.05, 20) => Best MSE: 1.1412495491154582
Params: (40, 4, 8, 8, 4, 0.15, 12) => Best MSE: 7.752349188019787
```



Best parameters Log x

USING Fitness Function MSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def log_func(x):
    return np.log(x)

# Fitness Function: Mean Squared Error
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Ensure correct order of coefficients

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        # Recruitment for best sites
        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size to focus search
        ngh *= 0.99

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Hypothetical parameter ranges
degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(log_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)
```

```
print("Best setup:", best_setup)

# Degree of the polynomial
degree = 5

# Running the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(log_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, log_func(x_range), label='log(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Log(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MSE)')
plt.xlabel('Generation')
plt.ylabel('MSE')
plt.show()
```

output:

```
Best setup: (3, 100, 15, 15, 10, 3, 0.1, 30)
```

USING Fitness Function RMSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def log_func(x):
    return np.log(x)

# Fitness Function: Root Mean Squared Error
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients are in reverse order for numpy
polyval

# Bees Algorithm for regression
```

```
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Let's set 100 generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size
    return population[0], best_fitness_history

# Hypothetical parameter ranges
degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(log_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)


# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(log_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, log_func(x_range), label='log(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Log(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (RMSE)')
plt.xlabel('Generation')
plt.ylabel('RMSE')
plt.show()
```

output:

```
Best setup: (3, 50, 5, 10, 10, 5, 0.1, 30)
```

USING Fitness Function MAE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def log_func(x):
    return np.log(x)

# Fitness Function: Mean Absolute Error
def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Hypothetical parameter ranges
degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(log_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)

# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(log_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, log_func(x_range), label='log(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Log(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MAE)')
plt.xlabel('Generation')
plt.ylabel('MAE')
plt.show()
```

output:

```
Best setup: (3, 100, 15, 15, 5, 2, 0.1, 30)
```

Best parameters Sin x

USING Fitness Function MSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
```

```
def sin_func(x):
    return np.sin(x)

# Fitness Function: Mean Squared Error
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])
        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Hypothetical parameter ranges

# Define the range for x and other algorithm settings
x_range = np.linspace(0, 2 * np.pi, 400)

degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(sin_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)
```

```
# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(sin_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, sin_func(x_range), label='sin(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Sin(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MSE)')
plt.xlabel('Generation')
plt.ylabel('MSE')
plt.show()
```

output:

```
Best setup: (5, 50, 15, 15, 10, 2, 0.1, 30)
```

USING Fitness Function RMSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def sin_func(x):
    return np.sin(x)

# Fitness Function: Root Mean Squared Error
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in population]
    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Hypothetical parameter ranges
# Define the range for x and other algorithm settings
x_range = np.linspace(0, 2 * np.pi, 400)

degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(sin_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)
# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(sin_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, sin_func(x_range), label='sin(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Sin(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (RMSE)')
plt.xlabel('Generation')
plt.ylabel('RMSE')
plt.show()
```

Output:

Best setup: (4, 50, 15, 15, 10, 3, 0.1, 30)

USING Fitness Function MAE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def sin_func(x):
    return np.sin(x)

# Fitness Function: Mean Absolute Error
def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
```

```
            candidate = elites[i] + np.random.randn(degree + 1) * ngh
            candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
            if candidate_fitness < fitness[i]:
                population[i] = candidate
                fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Hypothetical parameter ranges

# Define the range for x and other algorithm settings
x_range = np.linspace(0, 2 * np.pi, 400)

degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(sin_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)
# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(sin_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, sin_func(x_range), label='sin(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Sin(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MAE)')
plt.xlabel('Generation')
plt.ylabel('MAE')
plt.show()
```

Output:

```
Best setup: (4, 100, 5, 10, 5, 5, 0.1, 20)
```

Best parameters Tan x

USING Fitness Function MSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Function Definitions
def tan_func(x):
    return np.tan(x)

# Fitness Function: Mean Squared Error
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = mse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [mse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history


# Define the range for x and other algorithm settings
x_range = np.linspace(-np.pi/4, np.pi/4, 400)

degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(tan_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)
```

```
# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(tan_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, tan_func(x_range), label='tan(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Tan(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MSE)')
plt.xlabel('Generation')
plt.ylabel('MSE')
plt.show()
```

Output:

```
Best setup: (5, 100, 5, 5, 10, 3, 0.1, 30)
```

USING Fitness Function RMSE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def tan_func(x):
    return np.tan(x)

# Fitness Function: Root Mean Squared Error
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in population]
    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[i]:
                    population[i] = candidate
                    fitness[i] = candidate_fitness

        for i in range(nb):
            for _ in range(nrb):
                candidate = bests[i] + np.random.randn(degree + 1) * ngh
                candidate_fitness = rmse(func(x_range), poly_model(x_range, candidate))
                if candidate_fitness < fitness[ne + i]:
                    population[ne + i] = candidate
                    fitness[ne + i] = candidate_fitness

        # Track best fitness
        best_fitness_history.append(fitness[0])

        # Reduce neighborhood size
        ngh *= 0.95

        # Check for stagnation and reset if necessary
        if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
            population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
            fitness = [rmse(func(x_range), poly_model(x_range, ind)) for ind in
population]
            ngh = 0.1  # Reset neighborhood size

    return population[0], best_fitness_history

# Define the range for x and other algorithm settings
x_range = np.linspace(-np.pi/4, np.pi/4, 400)
```

```
degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
    for ns in ns_values:
        for ne in ne_values:
            for nb in nb_values:
                for nre in nre_values:
                    for nrb in nrb_values:
                        for ngh in ngh_values:
                            for stlim in stlim_values:
                                coeffs, fitness_history = bees_algorithm(tan_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                                final_fitness = fitness_history[-1]
                                if final_fitness < best_fit:
                                    best_fit = final_fitness
                                    best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)

# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(tan_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, tan_func(x_range), label='tan(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Tan(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (RMSE)')
plt.xlabel('Generation')
plt.ylabel('RMSE')
plt.show()
```

Output:

```
Best setup: (5, 50, 10, 15, 10, 3, 0.1, 20)
```

USING Fitness Function MAE CODE :

```
import numpy as np
import matplotlib.pyplot as plt

# Function Definitions
def tan_func(x):
    return np.tan(x)

# Fitness Function: Mean Absolute Error
def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# Polynomial Model
def poly_model(x, coeffs):
    return np.polyval(coeffs[::-1], x)  # Coefficients in reverse order for numpy polyval

# Bees Algorithm for regression
def bees_algorithm(func, x_range, degree, ns, ne, nb, nre, nrb, ngh, stlim):
    # Initial population (random coefficients for polynomial)
    population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
    fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in population]

    best_fitness_history = []

    # Optimization loop
    for gen in range(100):  # Number of generations
        # Sort by fitness and select the best
        indices = np.argsort(fitness)
        population = [population[i] for i in indices]
        fitness = [fitness[i] for i in indices]

        # Elite and best sites
        elites = population[:ne]
        bests = population[ne:ne+nb]

        # Recruitment for elite and best sites
        for i in range(ne):
            for _ in range(nre):
                candidate = elites[i] + np.random.randn(degree + 1) * ngh
            candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
            if candidate_fitness < fitness[i]:
               population[i] = candidate
               fitness[i] = candidate_fitness

      for i in range(nb):
         for _ in range(nrb):
            candidate = bests[i] + np.random.randn(degree + 1) * ngh
            candidate_fitness = mae(func(x_range), poly_model(x_range, candidate))
            if candidate_fitness < fitness[ne + i]:
               population[ne + i] = candidate
               fitness[ne + i] = candidate_fitness

      # Track best fitness
      best_fitness_history.append(fitness[0])

      # Reduce neighborhood size
      ngh *= 0.95

      # Check for stagnation and reset if necessary
      if len(best_fitness_history) > stlim and best_fitness_history[-1] ==
best_fitness_history[-stlim]:
         population = [np.random.rand(degree + 1) - 0.5 for _ in range(ns)]
         fitness = [mae(func(x_range), poly_model(x_range, ind)) for ind in
population]
         ngh = 0.1  # Reset neighborhood size

   return population[0], best_fitness_history

# Define the range for x and other algorithm settings
x_range = np.linspace(-np.pi/4, np.pi/4, 400)

degrees = range(3, 6)  # Polynomial degrees to test
ns_values = [30, 50, 100]  # Different values for number of scout bees
ne_values = [5, 10, 15]  # Values for number of elite sites
nb_values = [5, 10, 15]  # Values for number of best sites
nre_values = [3, 5, 10]  # Recruited bees for elite sites
nrb_values = [2, 3, 5]  # Recruited bees for remaining best sites
ngh_values = [0.1, 0.01]  # Initial neighborhood sizes
stlim_values = [10, 20, 30]  # Stagnation limits

best_setup = None
best_fit = np.inf

# Loop over all combinations (simple grid search approach)
for degree in degrees:
   for ns in ns_values:
      for ne in ne_values:
         for nb in nb_values:
            for nre in nre_values:
                for nrb in nrb_values:
                    for ngh in ngh_values:
                        for stlim in stlim_values:
                            coeffs, fitness_history = bees_algorithm(tan_func, x_range,
degree, ns, ne, nb, nre, nrb, ngh, stlim)
                            final_fitness = fitness_history[-1]
                            if final_fitness < best_fit:
                                best_fit = final_fitness
                                best_setup = (degree, ns, ne, nb, nre, nrb, ngh, stlim)

print("Best setup:", best_setup)
# Run the Bees Algorithm
best_coeffs, fitness_history = bees_algorithm(tan_func, x_range, degree, ns, ne, nb,
nre, nrb, ngh, stlim)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(x_range, tan_func(x_range), label='tan(x)', color='blue')
plt.plot(x_range, poly_model(x_range, best_coeffs), label='Fitted Model', color='red')
plt.title('Tan(x) and Fitted Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fitness_history, color='green')
plt.title('Fitness History (MAE)')
plt.xlabel('Generation')
plt.ylabel('MAE')
plt.show()
```

Output

```
Best setup: (5, 30, 5, 15, 10, 2, 0.1, 30)

)
```