GenoPhysics - Report

A genetic programming approach for the discovery of physical laws

Tiago Ventura – 2019243695 Sancho Simões – 2019217590

Masters in Intelligent Systems University of Coimbra

Coimbra, 12th March 2023

Contents

1 Goal **3**

2 Introduction **4**

3 Datasets **5**

4 Baseline Approches **9**

5 Normalization **12**

6 Genetic Programming Approaches **15**
- 6.1 Hyperparameter optimization . . . 16
- 6.2 Tree-Based Approach . . . 18
  - 6.2.1 Hyperparameter optimization configuration . . . 20
  - 6.2.2 Variation operators . . . 20
- 6.3 Grammar-based Genetic Programming Approach (Grammatical Evolution) . . . 22
  - 6.3.1 Hyperparameter optimization configuration . . . 22
  - 6.3.2 Context-free Grammar . . . 23

7 Selection Mechanisms **24**
- 7.1 Parents Selection . . . 24
- 7.2 Survivors Selection . . . 24

8 Fitness Functions **25**

9 Implementation Notes **27**
- 9.1 Libraries Used . . . 28

10 Results **29**
- 10.1 Hyperparameter Optimization . . . 29
- 10.2 Approaches comparison . . . 31
- 10.3 Statistical Analysis . . . 35

11 Statistical inference and tests **40**

12 Conclusion **42**

1 Goal

The discovery of physical laws has been a fundamental goal of science since the time of Galileo and Newton. These laws describe the behavior of natural phenomena, and they are essential for understanding the world around us. However, the discovery of physical laws is not an easy task, and it often requires a significant amount of experimentation and observation.

In recent years, machine learning techniques have been used to discover physical laws. Genetic algorithms are a type of machine learning technique that has shown promising results in the discovery of physical laws. Genetic algorithms work by iteratively optimizing a population of candidate solutions, with the fittest individuals being selected for the next generation of solutions. In this way, the algorithm evolves toward a solution that satisfies a particular fitness function.

In this project, we aim to demonstrate the potential of genetic algorithms in discovering physical laws. Specifically, we will use genetic programming to (re)discover Kepler's Third Law, which relates the period of a planet's orbit to its distance from the Sun. We will use data from the planets in our solar system to test the effectiveness of two different variants of genetic programming in discovering this law: tree-based genetic programming and grammar-based genetic programming (grammatical evolution).

We will store quality measures for each variant, such as their performance at the end of the run, and perform a statistical analysis of these measures. Our ultimate goal is to draw conclusions about the effectiveness of genetic programming in discovering physical laws and to evaluate the potential for this technique to be used in other scientific fields.

Overall, this project represents an exciting opportunity to explore the use of machine learning techniques in scientific discovery and to advance our understanding of the world around us.

2 Introduction

In this project, we will use genetic programming, a machine learning technique, to discover physical laws that relate to the period and distance of planets in our solar system. Specifically, we will be using regression to predict the period of a planet's orbit based on its distance from the Sun. This relation is, as aforementioned, modeled by the 3rd Kepler's Law.

$$\frac{T^2}{a^3} = \mathrm{C}, \tag{1}$$

where T is the period of a planet's orbit around the Sun and a is the length of the semimajor axis of the planet's elliptical orbit.
This formula means that the square of the period of any planet's orbit is proportional to the cube of the semimajor axis of the orbit, where C is proportionality constant. Johannes Kepler accomplished this great discovery with techniques and technologies way more rudimentary than those we have today. 'It should not be difficult to rediscover this law with computational resources and knowledge we have at our disposal', we thought to ourselves. However, some obstacles arose but we still managed to overcome them.

Genetic programming for regression works by evolving a mathematical function that approximates the relationship between the input and output variables. In our case, the input variable is the distance of the planet from the Sun (semimajor axis), and the output variable is the period of the planet's orbit.

The genetic programming algorithm will start with an initial population of random candidate solutions, where each candidate solution represents a mathematical function that takes the distance of a planet from the Sun as input and returns an estimated period of its orbit. The fitness of each candidate solution will be evaluated based on how well it approximates the true relationship between the input and output variables. The fittest candidate solutions will be selected for breeding, and their genetic material will be combined to create new candidate solutions. These new candidate solutions will be mutated to introduce variation into the population, and the process will continue until a stopping criterion is met.

One advantage of using genetic programming for discovering physical laws is that it can handle complex relationships between variables, even when the relationship is non-linear. Additionally, genetic programming can discover new relationships that may not be apparent from traditional scientific approaches.

Overall, genetic programming for regression is a powerful technique that has the potential to discover physical laws that describe the behavior of natural phenomena. In our project, we aim to demonstrate the potential of genetic programming for the discovery of physical laws and evaluate its effectiveness in this domain.

3 Datasets

In this project, we used a dataset that includes the distances and periods of the planets in our solar system. This dataset provides a useful test bed for the genetic programming algorithm, as it contains known physical laws that relate the distance and period of a planet's orbit around the Sun. To evaluate the effectiveness of our genetic programming algorithm, we also used additional datasets for testing and benchmarking purposes. Specifically, we used three datasets: a sphere dataset, a sin dataset, and a TRAPPIST-1 dataset.

The sphere dataset is a synthetic dataset that contains random points uniformly distributed on the surface of a sphere. The goal of this dataset is to test the ability of the genetic programming algorithm to discover a simple mathematical relationship between input and output variables.

The sin dataset is another synthetic dataset that contains random points on a sine wave. The goal of this dataset is to test the ability of the genetic programming algorithm to discover a non-linear relationship between input and output variables.

The TRAPPIST-1 dataset contains information about the distances and periods of the seven planets orbiting the TRAPPIST-1 star system.

By using these different datasets, we can evaluate the effectiveness of our genetic programming algorithm and compare its performance to other regression techniques. Additionally, using synthetic datasets allows us to control the complexity and non-linearity of the relationship between input and output variables, which can help us understand the limitations and strengths of the genetic programming algorithm.

Overall, the combination of the solar system dataset and the additional datasets for testing and benchmarking allowed us to thoroughly evaluate the performance and effectiveness of our genetic programming algorithm for discovering physical laws.

Semimajor axis (AU)	**Period (years)**
5.79	0.241
10.8	0.615
15.0	1.0
22.8	1.88
77.8	11.9
143	29.5
297	84
19.926 ± 54.071	**25.053 ± 32.877**

Table 1: Semimajor axis and period of the seven planets of the solar system, along with the mean and standard deviation of each column.

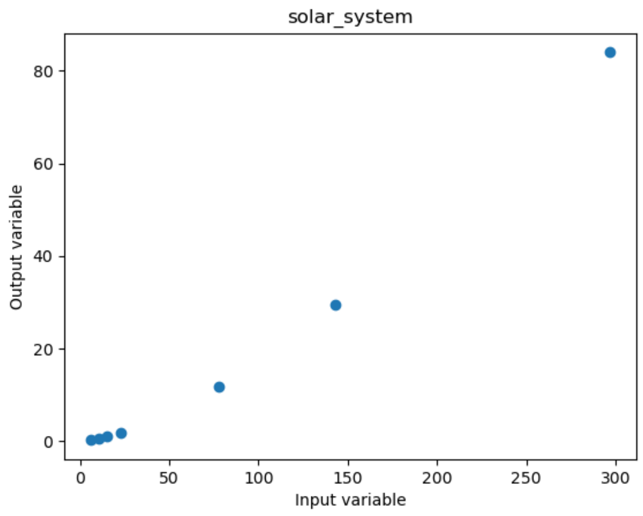

Figure 1: Solar system plot

As we can see from 3, the semimajor axis and period of the seven planets in the solar system exhibit a wide range of values. To better understand the relationship between these variables, we can plot them on a scatter plot, as shown in 1.

Semimajor axis (AU)	**Period (days)**
1.65	0.041
2.47	0.065
3.28	0.103
4.91	0.156
6.54	0.236
8.17	0.315
12.4	0.479
5.211 ± 3.464	**0.190 ± 0.156**

Table 2: Semimajor axis and period of the seven planets in the TRAPPIST-1 solar system, along with the mean and standard deviation of each column.

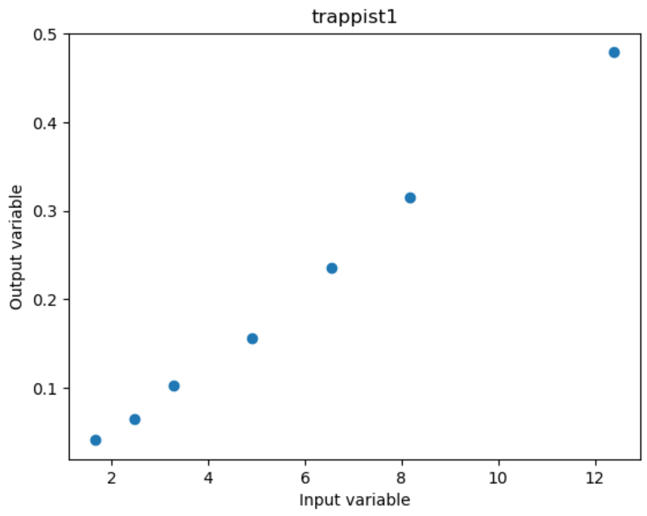

Figure 2: Solar system plot

The above statistical and graphical analysis might seem a bit pointless at first sight. However, it reveals the cause of the main issue we had when trying to apply a symbolic regression to the solar system dataset, using tree-based or grammar-based genetic programming. For now, the one thing to retain is that the data points in the referred dataset are much more sparse than the ones present in the TRAPPIST-1 dataset. They are so far apart that even the standard deviation surpasses the mean, for both columns. This does not happen in the second dataset. That's why we opted to have these and the other datasets as a baseline so that if we had some problems with the main one, we could always consult the others and perform some ad-hoc comparisons of the outputted results.

4 Baseline Approches

To establish a baseline solution for predicting the periods of the planets in our solar system based on their semi-major axis, we implemented a polynomial fit model of the third degree and the real Kepler law. The polynomial fit model was fitted and evaluated with the data using the NumPy library's `polyfit` and `polyval` functions, respectively. The real Kepler law was used to compute the theoretical values for the period axis based on the semi-major axis of each planet, using the gravitational constant and the mass of the Sun and each planet.

The performance of the polynomial fit model and the actual Kepler law was evaluated using the sum of squared errors (SSE) between the predicted and actual values of the semi-major axis for each planet. The SSE values for the two models are shown in Table 4.

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Model	SSE
Polyfit Model	0.033516385507059425
Real Kepler Law	2.948567271232605e-06

Table 3: SSE values for polynomial fit model and real Kepler law

The SSE values indicate that the real Kepler law provides a much better fit to the data than the polynomial fit model. However, we still included the polynomial fit model in our baseline solution to provide a simple and computationally efficient alternative that can be used as a reference point for comparison to more complex models.

Figure 3 shows a plot of the polynomial fit model, fitted to the data using the `polyfit` function. Figure 4 shows a plot of the real Kepler law, computed using the gravitational constant and the mass of the Sun and each planet. In both plots, the blue dashed line represents the predicted values of the semi-major axis based on the period of each planet, and the blue dots represent the actual values. As observable, there is no visible between both plots, and that's expectable since the first method applies a polynomial fit of the same degree as the equation defined by Kepler's 3rd Law and the second simply applies the actual law to the input values (distance).

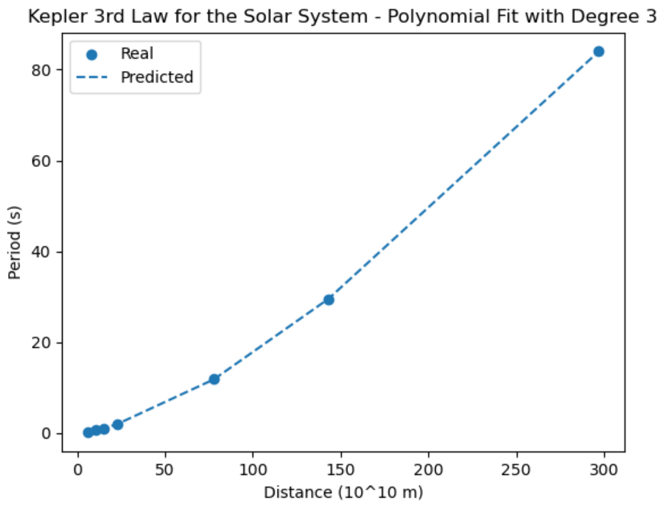

Figure 3: Polyfit model of third-degree fitted to solar system data

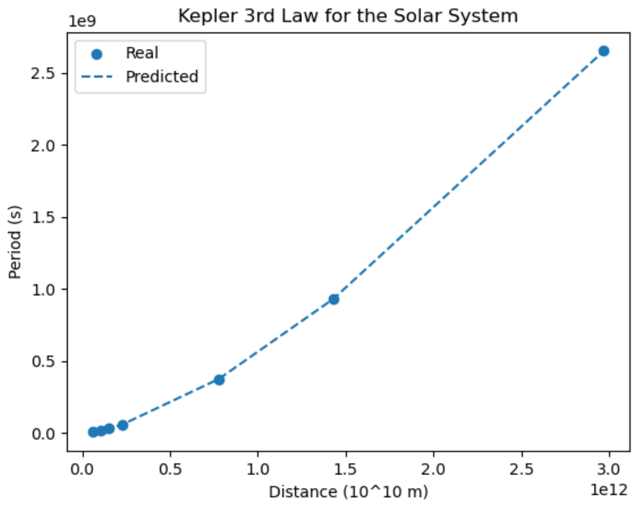

Figure 4: Real Kepler law applied to solar system data

We will use the SSE values for the polynomial fit model and the real Kepler law as reference point for evaluating the performance of our genetic programming algorithms. The goal of these algorithms is to find a mathematical expression that provides a better fit to the data than the baseline models. The tree-based and grammar-evolution algorithms are described in detail in the following sections.

5 Normalization

We discussed that the data points in the solar system dataset are much more sparse than the ones present in the TRAPPIST-1 dataset. This means that the data in the solar system dataset is more spread out and has a wider range of values compared to the TRAPPIST-1 dataset. This can cause issues when applying symbolic regression using tree-based or grammar-based genetic programming because these algorithms may not be able to accurately capture the relationships between variables in the data.

To overcome this issue, the data needs to be normalized. Normalization is a technique that scales the data to a specific range so that all variables have the same influence on the analysis. In this case, min-max normalization is used, which scales the data to a range between 0 and 1. This helps in bringing the data in the same range, making it easier to perform ad-hoc comparisons of the outputted results. By using normalized data, we can ensure that the range of values for each variable is consistent and that the algorithms can accurately capture the relationships between variables in the data, leading to more reliable and accurate results, as well as faster convergence. By doing so, no information is lost, and the normalization can be easily reversed to obtain the predicted values of the dataset in the original scale.

$$x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{2}$$

where:

- x is the original value of the feature
- x_{min} is the minimum value of the feature in the dataset
- x_{max} is the maximum value of the feature in the dataset
- x_{norm} is the normalized value of the feature, ranging between 0 and 1

Although we did not perform a statistical analysis to prove the aforementioned fact, it was evident that the performance of the genetic programming algorithms was much higher when the data was normalized. This ad-hoc comparison of performances is shown in the graphs below, which represent the fit of the outputted expression by a grammar-based GP with the same configuration and with or not normalization active.

Parameter	**Value**
RNG Seed	1
Generations	50
Population Size	900
Genotype Size	512
Mutation rate	0.0340
Crossover rate	0.7150
Elite Size	0.1660
Tournament Size	80

Table 4: Configuration of Grammar-based GP algorithm to test the effect of normalization/non-normalization

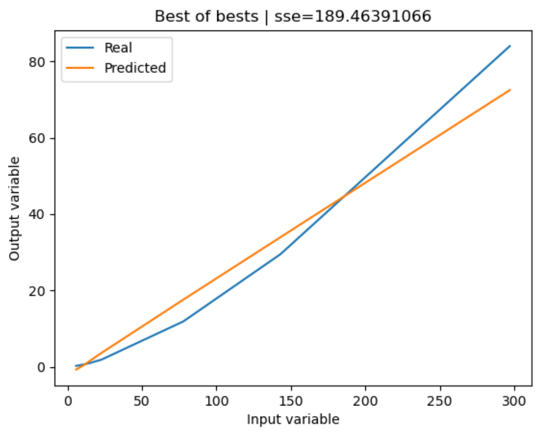

Figure 5: Fit for the resulting expression by grammar-based GP without normalization

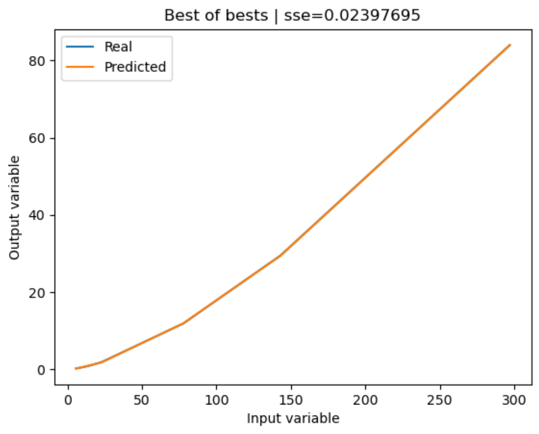

Figure 6: Fit for the resulting expression by grammar-based GP with normalization

As we can see, for the same configuration, the SSE is way higher in the case of non-normalization. Also, we can't even distinguish the real and predicted lines in the figure 6 This scenario also happened for tree-based GP, but we omit that for questions of verbosity. It is a fact that we wasted a lot of time and resources training the GP algorithms without normalization while obtaining poor results. Fortunately, the normalization of data appeared suddenly in our mind while comparing the solar system dataset to the TRAPPIST-1 dataset.

6 Genetic Programming Approaches

Genetic programming is a machine learning technique that involves generating a population of candidate solutions represented as computer programs. These programs are then evolved through a process inspired by biological evolution, which includes selection, crossover, and mutation. The fitness of each candidate program is evaluated based on how well it performs a designated task or solves a specific problem. The fittest candidates are then selected for breeding, and their genetic material is combined to create new candidate programs. This process is repeated over multiple generations until a stopping criterion is met. In the context of regression, genetic programming is commonly used to evolve mathematical functions that approximate the relationship between input and output variables.

Genetic programming approaches for machine learning can face two common challenges: **bloat** and **overfitting**. Bloat refers to the tendency of evolved programs to become excessively large and complex and it is exclusive to tree-based genetic programming algorithms, while overfitting occurs when the evolved program becomes too closely tailored to the training data, which will make it perform poorly for unseen data. To address these challenges, techniques such as limiting program complexity, implementing regularization, and using cross-validation can be employed. Additionally, the choice of genetic operators, representation, and fitness function can impact the likelihood of bloat and overfitting, making them important considerations in the design of genetic programming algorithms.

6.1 Hyperparameter optimization

In our study, we performed hyperparameter optimization for one tree-based genetic programming algorithm and one grammar evolution-based algorithm. We used Bayesian optimization with the Hyperopt module of Python for the optimization process.

To ensure a comprehensive exploration of the parameter space while avoiding excessive time and resource consumption, we formalized an execution plan. Additionally, we fixed some parameters to ensure fairness and commensurability in the selection process.

This fine-tuning allowed us to be more certain that the used configurations for both GP approaches would allow a more accurate comparison of both in performing over the solar system dataset. Even though, due to the **No Free Lunch Theorem**, no algorithm performs better than others for all classes of problems, it can perform better in a certain class. However, it does not make sense to compare two algorithms in this class if they are, speaking in a colloquial manner, bad and poorly configurated

For the solar system dataset, we used the following fixed parameters:

Parameter	**Value**
Dataset	solar system
Generations	75
Fitness function	SSE
RNG Seed	1
Optimizer evaluations	100

Table 5: Fixed Parameters for Bayesian Optimization

By fixing these parameters, we aimed to focus the optimization process on the variable parameters that have the most significant impact on the performance of the algorithm. This approach allowed us to compare the different candidate algorithms fairly and to identify the most promising ones for further evaluation.

The other part of the execution plan, which specifies the intervals and options for parameters like mutation rate, crossover rate, population size, etc. varies according to the GP approach and will be demonstrated in the ahead sections.

Operator	**Function**	**Pseudocode**
$+$	Addition	`return` $x + y$
-	Subtraction	`return` $x - y$
$*$	Multiplication	`return` $x * y$
$/$	Protected Division	`if abs`$(y) \leq 1e{-}3$: `return` 1 `else:` `return` x/y

Table 6: Used function set

Tree-based and grammar-based GP both require the definition of a function set. This function set lists the operators that are permitted to operate over the operands. In tree-based GP, the function set is typically defined as a list of functions that take one or more operands as input and produce a single output.

In grammar-based GP, the function set is used in the production of the grammar. Each non-terminal symbol in the grammar is associated with a set of functions that can be used to expand that symbol. These functions are then applied to the operands to create a new expression.

6.2 Tree-Based Approach

Tree-based genetic programming is a machine-learning technique that represents candidate solutions as trees. The nodes in the tree correspond to mathematical operators or constants, and the leaves resemble input variables or constants. The goal is to evolve trees that approximate the relationship between input and output variables, with fitness evaluated based on the error between predictions and actual values. Genetic operators such as crossover and mutation are used to create new offspring, and subtree crossover and hoist mutation can also be used. The fitness function can be adapted to the problem at hand, and the tree-based approach can represent complex relationships and interactions between variables. However, the approach can suffer from limitations such as overfitting and difficulty in interpretation and visualization, and the choice of genetic operators and fitness functions can heavily influence performance.

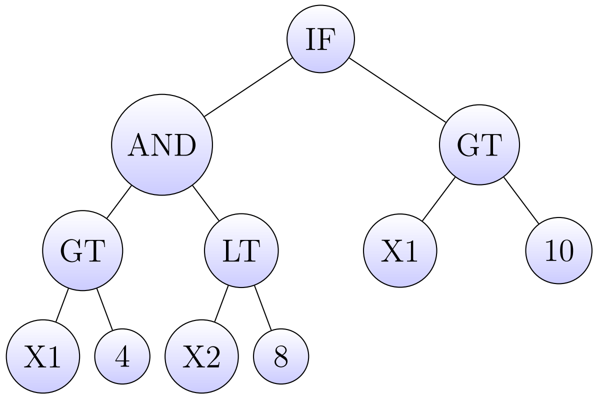

Figure 7: Example of a GP tree applied to a certain Assembly logical program

As referred to before, bloat is a major issue in tree-based genetic programming that results in the emergence of excessively large and redundant trees, making them less readable and more difficult to process and store in memory. This leads to longer training times, hindering the performance of the algorithm. To address this problem, one common solution is to penalize longer trees during the selection process. However, we took a different approach and periodically injected a certain number of **random foreign individuals** into the population. These foreign individuals typically have a low depth

(3-6) and help to compensate for the bloated individuals, maintaining a balance in the population. This approach also preserves the positive aspect of bloat, which is its ability to prevent disruptive effects of variation operators (crossover and mutation) on the individual's code, similar to how redundancy in DNA helps prevent genes from being altered by external factors such as radiation.

It is important to mention that the injection of random foreigners also helped to maintain and improve diversity in the population, which is key for the algorithm to not be trapped in local minima.

6.2.1 Hyperparameter optimization configuration

Below follows a table that lists the execution plan used to optimize most of the hyperparameters used in our tree-based genetic programming approach.

Hyperparameter	**Distribution**	**Values**
Population size	quniform	[100, 500], step=50
Mutation rate	uniform	[0.01, 0.5]
Crossover rate	uniform	[0.5, 0.9]
Random foreigners injected size	uniform	[0.1, 0.5]
Random foreigners injection period	quniform	[5, 50], step=5
Tournament size	quniform	[2, 5], step=1
Elite size	uniform	[0.1, 0.3]

Table 7: Hyperparameters Varied by Bayesian Optimization

`quniform` is a function provided by the `hyperopt` library in Python that generates a distribution for a hyperparameter that is uniformly distributed in a specified range, with a specified step size. The function takes three arguments: the name of the hyperparameter, the minimum value, the maximum value, and the step size.

`uniform` is also a function provided by `hyperopt`, but instead of using a fixed step size, it generates a distribution for a hyperparameter that is uniformly distributed in a specified range, with values drawn at random from that range. The function takes two arguments: the name of the hyperparameter, the minimum value, and the maximum value.

6.2.2 Variation operators

In tree-based genetic algorithms, two primary variation operators are used: subtree crossover and node mutation, which were employed in this work

Subtree crossover Subtree crossover is a type of crossover operator used in tree-based genetic algorithms. It involves selecting two parent trees and swapping subtrees between them to create two new offspring. The crossover point is selected randomly from within the subtrees of each parent.

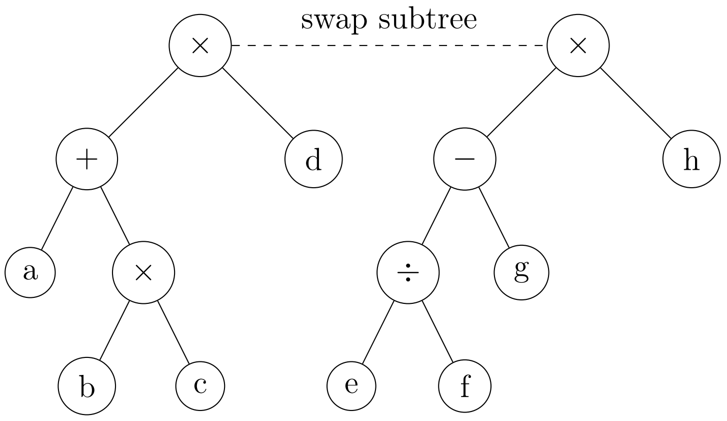

Figure 8: Subtree crossover

Node mutation Node mutation is a type of mutation operator used in tree-based genetic algorithms. It involves selecting a random node in a tree and replacing it with a new subtree generated randomly. This operator allows the exploration of new regions of the search space by creating variations in the tree structures.

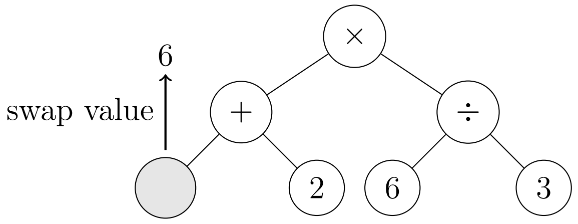

Figure 9: Node mutation

6.3 Grammar-based Genetic Programming Approach (Grammatical Evolution)

Grammatical Evolution is a genetic programming approach that utilizes context-free grammar to define candidate solutions' syntax. It separates the representation from the genetic operators and fitness function for increased flexibility. The genome is a string of symbols generated by the grammar and decoded into a program using symbol-instruction mapping. The genetic operators include mutation and crossover, which can be applied to the genome without affecting program syntax or semantics. The fitness function is typically based on performance in solving a problem or task. Grammatical Evolution can handle variable-length programs and generates syntax-correct programs.

It is important to note that the choice of grammar and mapping can heavily influence algorithm performance, and genome-program separation can hinder program interpretation and visualization.

6.3.1 Hyperparameter optimization configuration

Below follows a table that lists the execution plan used to optimize most of the hyperparameters used in our grammar-based genetic programming approach.

Hyperparameter	**Distribution**	**Values**
Population size	`quniform`	[500, 1000], step=100
Genotype size rate	`quniform`	[128, 512], step=32]
Mutation rate	`uniform`	[0.01, 0.4]
Crossover rate	`uniform`	[0.3, 0.8]
Elite size	`uniform`	[0.1, 0.3]
Tournament size	`quniform`	[20, 100], step=20

Table 8: Hyperparameters Varied by Bayesian Optimization

The population maximum size is much higher in the case of this grammar-based approach. That is because it is much more efficient than the tree-based approach since it uses a linear structure, the grammar itself, and not a non-linear structure such as the tree. We also opted to not implement the injection of random individuals because we observed in an ad-hoc manner that grammar evolution is a higher performer when it comes to promoting and maintaining population diversity than the tree-based GP. In addition, in grammar-based GP, the problem of bloat is non-existing.

6.3.2 Context-free Grammar

start	→ **expr**
expr	→ **op** '(' **expr** ', ' **expr** ')' | **var**
op	→ `mult` | `add` | `sub` | `div_prot`
var	→ x[0] | 1.0

Table 9: Context-free grammar used to feed the grammar-based GP algorithm

The context-free grammar used for the grammatical evolution algorithm is pretty straightforward and intuitive. It allows the operations of sum, subtraction, multiplication, and protected vision to be performed. The terminals are the input variable ($X[0]$) and 1.0. We think that maybe this 1.0 might influence the range of input values for which the genetic algorithm would perform better, and maybe that's why min-max normalization works very well because every normalized value is within the range [0...1]. However, this is just a side note and something maybe for further work.

7 Selection Mechanisms

Selection is an important step in genetic algorithms that determines which individuals from the current population will be chosen to generate the next generation. There are two main types of selection mechanisms: parents selection and survivors selection. The ones explained ahead were the ones used in both tree-based and grammar-based genetic programming algorithms. For the sake of simplicity, we did not include this hyperparameter for optimization, even though we implemented more mechanisms than the ones below listed.

7.1 Parents Selection

Parents selection is the process of selecting individuals from the current population that will be used to create offspring for the next generation. There are several popular selection methods used in genetic algorithms, including:

Tournament selection: In this method, a random subset of the population is chosen and the individuals in the subset compete against each other. The winners of the competition are chosen as the parents of the next generation.

7.2 Survivors Selection

Survivor selection is the process of selecting individuals from the offspring and current population that will make up the population for the next generation. There are several popular selection methods used in genetic algorithms, including:

Elitism: This method selects the best individuals from the current population and passes them directly to the next generation. This ensures that the best solutions are preserved across generations. It is important to note that the elite size must be moderate or we risk losing diversity in the population, because if high, many old individuals would always be present in the current population and the offspring generated would be very similar in terms of genotype and therefore also in fitness.

8 Fitness Functions

Fitness functions were used to evaluate the performance of individual solutions in the genetic algorithm and guide the search for better solutions. The fitness function should measure how well the solution solves the problem at hand.

The first function, negative sigmoid, is a minimization fitness function, which means that it tries to minimize the fitness value of a solution. It calculates the error between the predicted output and the actual output and returns a negative sigmoidal value between 0 and -1 that reflects the error. The lower the error, the lower the negative sigmoidal value, and therefore the higher the fitness of the solution.

$$\text{sigmoid}(predicted, real) = -\frac{1}{1 + \sum_{i=1}^{n} |predicted_i - real_i|} \tag{3}$$

The second function, SSE (Sum of Squared Errors), is a minimization fitness function, which means that it tries to minimize the fitness value of a solution. It calculates the sum of the squared differences between the predicted output and the actual output, which is a common way to measure the error in regression problems.

$$\text{sse}(predicted, real) = \sum_{i=1}^{n} (predicted_i - real_i)^2 \tag{4}$$

The third function, MSE(Mean Squared Error), is similar to SSE, but it divides the sum of squared errors by the number of data points to calculate the mean error. This is often used in regression problems to compare the performance of different models.

$$\text{mse}(predicted, real) = \frac{1}{n} \sum_{i=1}^{n} (predicted_i - real_i)^2 \tag{5}$$

The fourth function, RMSE (Root Mean Squared Error), is a variation of MSE, where the mean error is first calculated, and then the square root of the mean error is taken. This function is useful for interpreting the performance of a model in the same units as the original data.

Despite implementing all of these four fitness functions being implemented, we opted to use only the SSE for both tree and grammar-based GP algorithms. This is because it's always desirable to have a differentiable function as a guiding/objective function, which doesn't happen with the implemented sigmoid, because of the absolute operator. Also, it doesn't hide the dispersion of the error as the MSE or RMSE do since they have the mean within their formula, which is only a locality measure.

$$\text{rmse}(predicted, real) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(predicted_i - real_i)^2} \tag{6}$$

Overall, these fitness functions provide different ways to evaluate the performance of solutions in a genetic programming algorithm, depending on the specific needs and goals of the problem at hand.

9 Implementation Notes

In our implementation of the genetic programming algorithms, we started with a base code that was provided to us. However, we organized it in a modular way, using classes to separate the different components of the algorithm. This made it easier to add and test new features, as well as to reuse code in different experiments.

To improve the evaluation of the algorithms, we added primitives to the code that allowed us to test and evaluate them more effectively. These primitives included different fitness functions, genetic operators, and termination conditions.

We also introduced multiprocessing to speed up the tests. By using multiple processors, we were able to run experiments more quickly and efficiently, which allowed us to explore a wider range of parameter settings and problem domains.

Overall, our implementation was designed to be flexible, modular, and easy to extend. By separating the different components of the algorithm and adding new primitives and features, we were able to explore the performance of the genetic programming algorithms on different datasets and problem domains and compare the effectiveness of different variants of the algorithms.

9.1 Libraries Used

In our implementation, we used the following main libraries:

- **NumPy:** A Python library used for numerical computations, providing a high-performance multidimensional array object and tools for working with these arrays. We used NumPy to accelerate vectorial calculus.
- **SymPy:** A Python library used for symbolic mathematics, providing tools for symbolic manipulation of mathematical expressions. We used SymPy to simplify expressions and display them.
- **Matplotlib:** A Python library used for creating static, animated, and interactive visualizations in Python. We used Matplotlib to display graphs.
- **Hyperopt:** A Python library used for hyperparameter optimization, providing a flexible and extensible framework for optimizing machine learning models. We used Hyperopt for hyperparameter optimization.
- **Random:** A Python library used for generating pseudorandom numbers. We used the Random library for the stochastic mechanisms.

10 Results

In this section, we will present the results obtained after performing the main experiments of this work, which are the selection of a tree-based and grammar-based genetic programming algorithm through Bayesian optimization of their hyperparameters as well as the comparison of both through statistical inference, namely, statistical tests.

10.1 Hyperparameter Optimization

Below, we can observe the results obtained after the execution of Bayesian optimization for both GP approaches.

Metric	**Value**	**Units**
Elapsed Time	2:54:10	seconds
Time per Trial	104.51	seconds
Best Fitness	1.14984472114e-06	-

Hyperparameter	**Value**
Population size	350
Crossover rate	0.886824
Mutation rate	0.219956
Elite size	0.133939
Tournament size	3
Random foreigners injected size	0.209002
Random foreigners injection period	5

Figure 10: Bayesian Opt. results for the tree-based GP algorithm

Metric	**Value**	**Units**
Elapsed Time	1:34:59	seconds
Time per Trial	57.00	seconds
Best Fitness	2.93720524187e-06	-

Hyperparameter	**Value**
Genotype size	512
Population size	900
Crossover rate	0.715041
Mutation rate	0.034032
Elite size	0.165956
Tournament size	80

Figure 11: Bayesian Opt. results for the grammar-based GP algorithm

The selected tree-based GP algorithm had a better performance in terms of the best fitness, with a value of $1.14984472114e - 06$, compared to the grammar-based GP algorithm with a value of $2.93720524187e-06$. However, the tree-based GP algorithm took longer to execute, with an elapsed time of 2 hours and 54 minutes, compared to the tree-based GP algorithm, which took only 1 hour and 34 minutes. Additionally, the time per trial of the tree-based GP algorithm is higher than the tree-based GP algorithm, which means it took longer for the algorithm to evaluate each set of hyperparameters.

Comparing the hyperparameters of the two algorithms, we can see that the grammar-based GP algorithm had a large genotype size of 512, which may mean that the higher it is, the more capable the algorithm to detect nonlinear relations between the given data. The population size of the grammar-based algorithm was also larger at 900 compared to the tree-based algorithm's population size of 350. This superiority was imposed from the beginning, when defining the execution plan, due to the higher computational demand of the tree-based genetic algorithm. The crossover rate of the tree-based algorithm was higher at 0.886824 compared to the grammar-based algorithm's crossover rate of 0.715041. On the other hand, the mutation rate of the tree-based algorithm was also higher at 0.219956 compared to the grammar-based algorithm's mutation rate of 0.034032. This was also somehow forced by the execution plan: we intended from the beginning to promote diversity in the tree-based GP populations via hypermutation.

It is difficult to predict the result of a statistical test without more information about the distribution of the fitness values obtained by the two algorithms. However, given that the best fitness value of the tree-based algorithm was lower than the grammar-based algorithm, it is possible that a statistical test could reveal a significant difference between the two algorithms.

10.2 Approaches comparison

To compare the two GP alternatives, we ran each algorithm 30 times with 75 generations, using the hyperparameters that were optimized by Bayesian optimization. By running each algorithm multiple times, we can reduce the effects of random variation and ensure that the results are more representative of the algorithm's true performance. Additionally, by running each algorithm with the same hyperparameters, we can ensure a fair comparison between the two algorithms.

It's worth noting that by running each algorithm 30 times, we can potentially use the Central Limit Theorem (CLT) to make inferences about the performance of the algorithms. The CLT states that the distribution of sample means approaches a normal distribution as the sample size increases. Therefore, if the sample means for each algorithm are normally distributed, we can use statistical tests to determine if there is a significant difference between the means of the two algorithms.

However, it's important to note that the CLT may not be applicable if the sample means are not normally distributed or if other underlying factors could affect the results. Therefore, we should be cautious in concluding based solely on the results of the statistical tests and should also consider other factors, such as the practical significance of any observed differences.

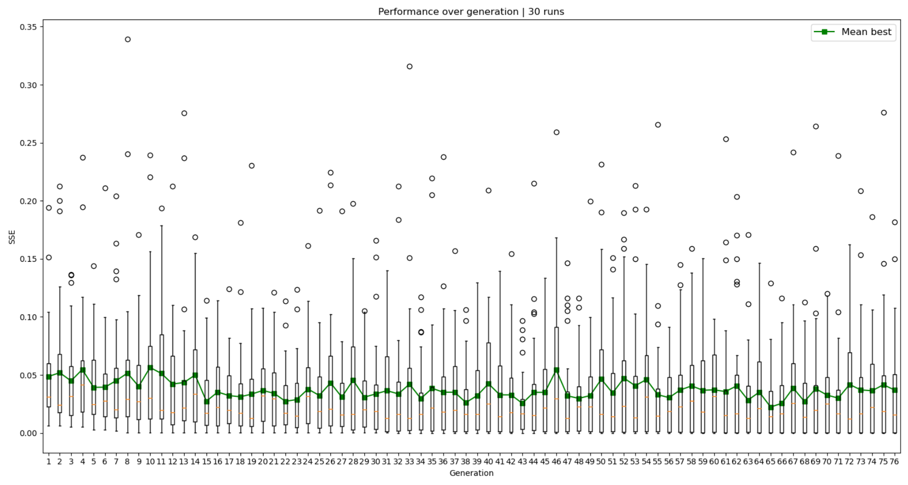

Figure 12: Boxplot for the Best Individuals over 30 runs for the tree-based GP

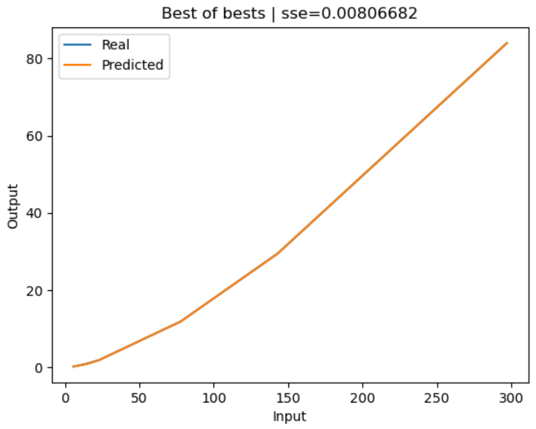

Figure 13: Best of the bests over 30 runs for the tree-based GP

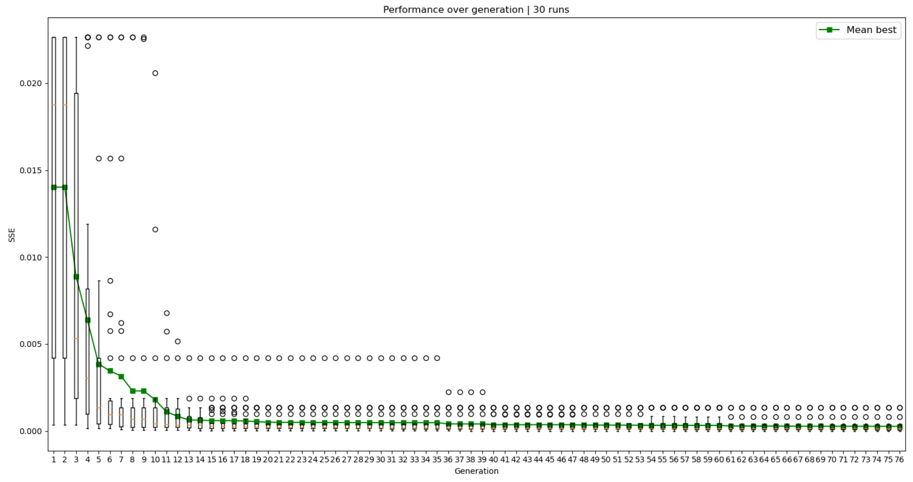

Figure 14: Boxplot for the Best Individuals over 30 runs for the grammar-based GP

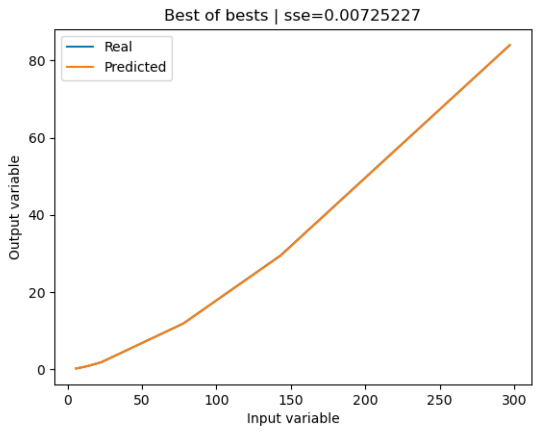

Figure 15: Best of the bests over 30 runs for the grammar-based GP

The plot for the tree-based GP algorithm shows that many runs converge prematurely, which can be seen by the high number of runs that end with a high fitness value. Additionally, the plot has many outliers, indicating the instability of the algorithm. One possible reason for this behavior is that the hyperparameters chosen by the Bayesian optimization might not be suitable for all runs. Therefore, the algorithm could get stuck in a suboptimal solution, leading to premature convergence.

In contrast, the plot for the grammar-based GP algorithm shows a more stable and normal convergence behavior. The fitness values decrease steadily over time, with fewer outliers compared to the tree-based GP plot. Moreover, the grammar-based GP algorithm seems to converge to a lower mean fitness value than the tree-based GP. This suggests that the grammar-based GP algorithm is more effective in finding a good solution compared to the tree-based GP algorithm.

The best of the bests tree-based GP fit aligns flawlessly with the original line, boasting an SSE of 0.00806682. However, we must note that the expression is overly verbose, making it challenging to interpret and prone to errors. As a result, we made the executive decision to discard it from the final analysis, focusing instead on more quantitative measures.

The best-of-the-best model generated by the Grammar-Based GP was able to perfectly align with the original line, resulting in an SSE of 0.00725227.

Overall, the results suggest that the grammar-based GP algorithm is a better alternative than the tree-based GP algorithm. However, it's important to note that statistical tests need to be conducted to determine whether the difference in performance between the two algorithms is significant, which is what will be presented shortly.

10.3 Statistical Analysis

We will conduct a brief statistical analysis by considering the final best fitness values of each algorithm. This analysis will help us gain insights into the properties of the distributions they came from. We will also determine whether we can apply parametric statistical tests to compare both alternatives.

To gain more insights into the properties of these distributions, we can calculate some descriptive statistics such as the mean, median, standard deviation, skewness, and kurtosis. We can also perform some tests to check for normality, such as the Shapiro-Wilk test or the Kolmogorov-Smirnov test.

Based on these descriptive statistics and tests, we determined whether the distributions are normal and whether they have similar variances. If the distributions are normal and have similar variances, we can apply parametric statistical tests such as the t-test or ANOVA to compare the means of the two distributions. If not, we need to use non-parametric tests such as the Mann-Whitney U test or the Kruskal-Wallis test.

Statistic	**Value**
Minimum	1.149800e-06
Maximum	0.181844
Mean	0.036935
Median	0.015926
Variance	0.002205
Standard Deviation	0.046961
Skewness	1.540938
Kurtosis	1.679234
Q1 (25th percentile)	0.000397
Q2 (50th percentile)	0.015926
Q3 (75th percentile)	0.050664

Table 10: Descriptive statistics for the final bests of the tree-based GP algorithm.

According to the Kolmogorov-Smirnov test, the data is not normally distributed with a p-value of 1.5×10^{-8} at a significance level of 0.05. Similarly, the Shapiro-Wilk test also rejects normality with a p-value of 3.0×10^{-5}. This indicates that non-parametric tests should be used instead of parametric tests to compare this algorithm with others.

The minimum fitness value achieved by the tree-based GP algorithm was 1.149800e-06, and the maximum was 0.181844. The mean fitness value was 0.036935, which suggests that the algorithm was able to find reasonably good solutions. However, the skewness of 1.540938 indicates that the distribution is highly skewed towards smaller fitness values, which might suggest premature convergence or instability. Additionally, the kurtosis of 1.679234 suggests that the distribution has heavier tails than a normal distribution, which could be due to the presence of many outliers.

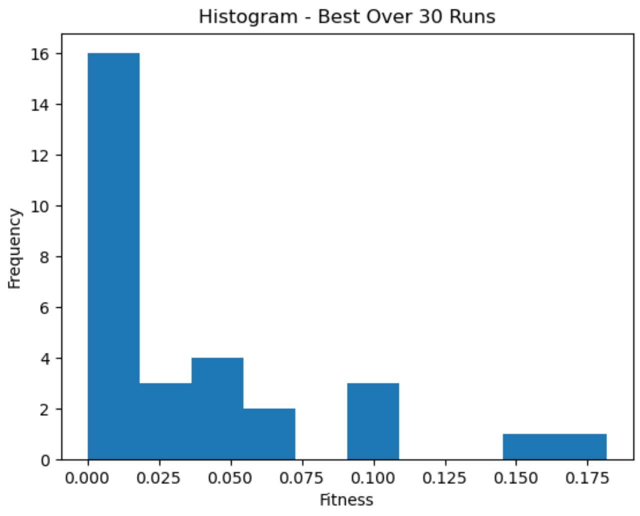

Figure 16: Histogram for the Best Individuals over 30 runs for the tree-based GP

The histogram of the final best fitness values for the tree-based GP algorithm confirms what we previously observed, a highly skewed distribution that resembles an exponential distribution, which is far from being normally distributed. This is in line with the results of the normality tests we performed, which showed that the data is not normal at a significance level of 0.05. The high skewness of the distribution indicates that the algorithm may have experienced premature convergence, as some runs may have converged to suboptimal solutions, while others may have gotten stuck in local optima. Additionally, the presence of many outliers suggests that some runs may have encountered numerical stability issues or other unforeseen problems. Overall, these results indicate that the distribution of the final best fitness values for the tree-based GP algorithm is highly non-normal and non-parametric, which may limit the applicability of some statistical tests for comparing this algorithm with others, in this case, the grammar-based one.

Statistic	**Value**
Minimum	1.0337e-06
Maximum	0.001357
Mean	0.000266
Median	0.000159
Variance	1.167e-07
Standard Deviation	0.000342
Skewness	2.187075
Kurtosis	4.219228
Q1 (25th percentile)	5.992e-05
Q2 (50th percentile)	0.000159
Q3 (75th percentile)	0.000353

Table 11: Descriptive statistics for the final bests of the grammar-based GP algorithm.

As per the Kolmogorov-Smirnov and Shapiro-Wilk tests, the data does not follow a normal distribution. The p-values obtained were extremely low (1.0×10^{-8} and 1.1×10^{-6} respectively) at a significance level of 0.05. Hence, it is recommended to use non-parametric tests for comparing this algorithm with others.

The grammar-based GP algorithm's minimum fitness value was 1.0337e-06, and the maximum value was 0.001357. The mean fitness value was 0.000266, indicating that the algorithm discovered reasonably good solutions. However, the skewness value of 2.187075 reveals that the distribution is highly skewed towards smaller fitness values, indicating possible premature convergence or instability. Besides, the kurtosis value of 4.219228 suggests that the distribution has much heavier tails than a normal distribution.

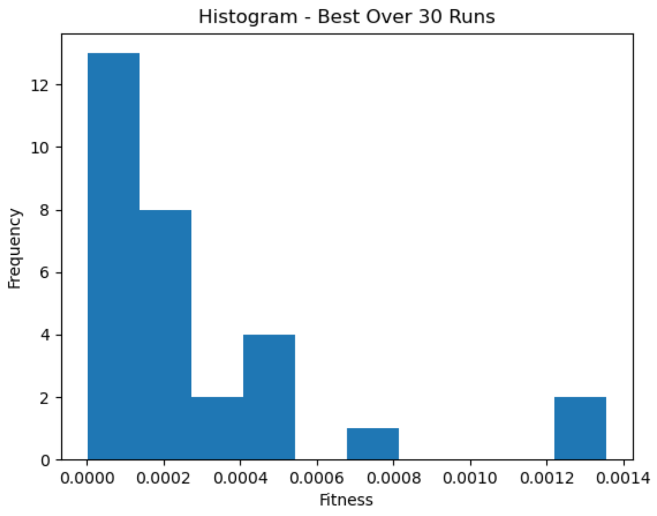

Figure 17: Histogram for the Best Individuals over 30 runs for the grammar-based GP

The histogram displayed in Figure 17 for the best individuals over 30 runs of the grammar-based GP algorithm confirms our earlier observations, highlighting a highly skewed distribution that resembles an exponential distribution. This outcome aligns with the results of the normality tests that we conducted, indicating that the data is not normally distributed at a significance level of 0.05.

11 Statistical inference and tests

In statistical hypothesis testing, parametric tests assume that the data follow a specific distribution, such as the normal distribution, and that the variances of the populations being compared are equal. However, if these assumptions are not met, non-parametric tests can be used instead. In this case, since normality did not apply, constant variance was not needed to test, because one of the mandatory criteria of parametric tests was already violated. Therefore, we resorted to non-parametric tests, namely the Wilcoxon test. The Wilcoxon test is a non-parametric test used to compare two related samples. It is often used as an alternative to the paired t-test when the data does not meet the assumptions of normality and equal variances. The Wilcoxon test is based on the ranks of the differences between the paired observations. The null hypothesis of the Wilcoxon test is that there is no difference between the two populations, while the alternative hypothesis is that there is a difference. Mathematically, the Wilcoxon test involves the following steps:

- Calculate the differences between the paired observations.
- Rank the absolute values of the differences.
- Calculate the sum of the ranks for the positive differences and the sum of the ranks for the negative differences.
- Calculate the test statistic, which is the smaller of the two sums of ranks.
- Calculate the p-value using a reference distribution, such as the standard normal distribution or the Wilcoxon signed-rank distribution.

The significance level used in our Wilcoxon test was 0.05, which means that we reject the null hypothesis if the p-value is less than 0.05. In our case, the p-value was 2.8434237746031825e-05, which is much smaller than 0.05. Therefore, we reject the null hypothesis and conclude that there is a significant difference between the two approaches. Since the two samples come from different populations, it is most likely that the grammar-based GP approach is superior to the tree-based GP approach on the solar system dataset. However, it is important to note that this conclusion is specific to this dataset and may not generalize to other datasets or problems. Additionally, it is worth noting that non-parametric tests are generally less powerful than parametric tests, so it is possible that a parametric test would have yielded a different result if the assumptions of normality and equal variances had been met.

12 Conclusion

In this project, we explored the use of two different genetic programming approaches - tree-based GP and grammatical evolution (GE) - to discover physical laws, with a specific focus on proving Kepler's Third Law using a dataset of planetary distances and periods.

Our results showed that GE was generally more effective than the tree-based GP approach in terms of convergence speed and overall fitness performance. We attribute this to the structured search space and less disruptive crossover operator used by GE. We used Bayesian optimization to perform a grid search for optimal parameters for both GP approaches, and normalization was crucial to obtain good results due to the sparseness of the dataset.

To compare the final best models of each approach, we performed a Wilcoxon non-parametric test since normality could not be assured. The test confirmed that the two approaches performed differently, and most likely the obtained GE is superior. This conclusion was further supported by our complementary analysis of the box plots and best of the best fits.

In conclusion, our findings suggest that the grammatical evolution approach is a promising option for genetic programming in the discovery of physical laws and other structured problems. Further experimentation and analysis could help to determine the best approach for different problem domains and datasets. Specifically, one avenue for further research could be to explore how to extend the GE approach to handle more complex datasets with higher-dimensional inputs. Additionally, other non-parametric tests could be used to further validate our results, and the use of different datasets could be explored to test the robustness and generalizability of our approach.

References

[1] Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection.* MIT press.

[2] Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, 324(5923), 81-85.

[3] Palmer, J., & Goldberg, D. E. (1994). Genetic algorithms and scientific discovery. *Communications of the ACM*, 37(5), 30-44.

[4] Helmuth, T., Spector, L., & Matheson, J. E. (2015). The effectiveness of tree-based crossover in genetic programming. In *Genetic programming theory and practice XIII* (pp. 103-120). Springer, Cham.

[5] O'Neill, M., & Ryan, C. (2003). Grammatical evolution: Evolutionary automatic programming in an arbitrary language. *Kluwer Academic Publishers.*

[6] Wikipedia contributors. (2023, May 9). Evolutionary Computation. In *Wikipedia, The Free Encyclopedia.* Retrieved May 12, 2023, from `https://en.wikipedia.org/wiki/Evolutionary_computation`

[7] Wikipedia contributors. (2023, May 9). Grammatical Evolution. In *Wikipedia, The Free Encyclopedia.* Retrieved May 12, 2023, from `https://en.wikipedia.org/wiki/Grammatical_evolution`

[8] Koza, J. R. (2018). Grammatical evolution. In *Encyclopedia of Evolutionary Biology* (pp. 132-137). Elsevier. Retrieved from `https://www.sciencedirect.com/science/article/abs/pii/S0957417418301751`