In the name of God

Genetic Programming for Symbolic Regression

Artificial Intelligence Course Project

Student Name: Ilya Jahed

Student Number: 402521126

Course: Artificial Intelligence

Field: Artificial Intelligence

University: Iran University of Science and Technology

Academic Year: 2024 – 2025

Genetic Programming for Symbolic Regression

1. Introduction

This project implements Symbolic Regression using Genetic Programming (GP) in Python.
The objective of symbolic regression is to automatically discover a mathematical expression that best approximates a target function, without assuming any predefined model structure.

Unlike classical regression methods—where the functional form is fixed and only parameters are optimized—symbolic regression simultaneously searches for:

the structure of the mathematical expression, and the numerical constants within that structure.

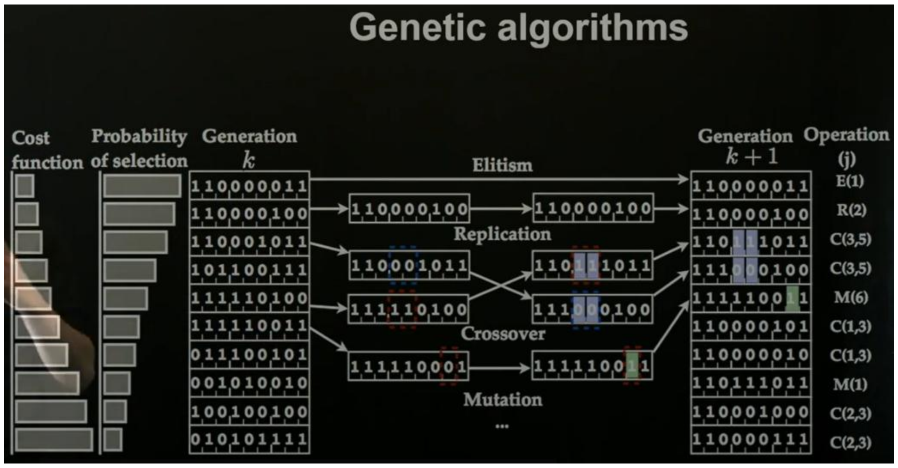
Genetic Programming is particularly well suited for this task because it evolves executable programs represented as expression trees, enabling the discovery of entirely new functional forms.

The implementation follows standard academic Genetic Programming practices, with strong emphasis on:

numerical robustness, modular design, reproducibility and evolutionary stability.

---

2. Background: Genetic Algorithms



A Genetic Algorithm (GA) is an evolutionary optimization technique inspired by natural selection.

A GA operates on a population of chromosomes, where each chromosome represents an encoded candidate solution to a problem.

Core Components of a Genetic Algorithm:

Population Initialization
A set of candidate solutions is randomly generated.

Fitness Evaluation
Each chromosome is evaluated using a fitness (cost) function, which measures solution quality.

Selection
Chromosomes are selected probabilistically, favoring individuals with better fitness.

Elitism
The best-performing individuals are copied directly to the next generation without modification.

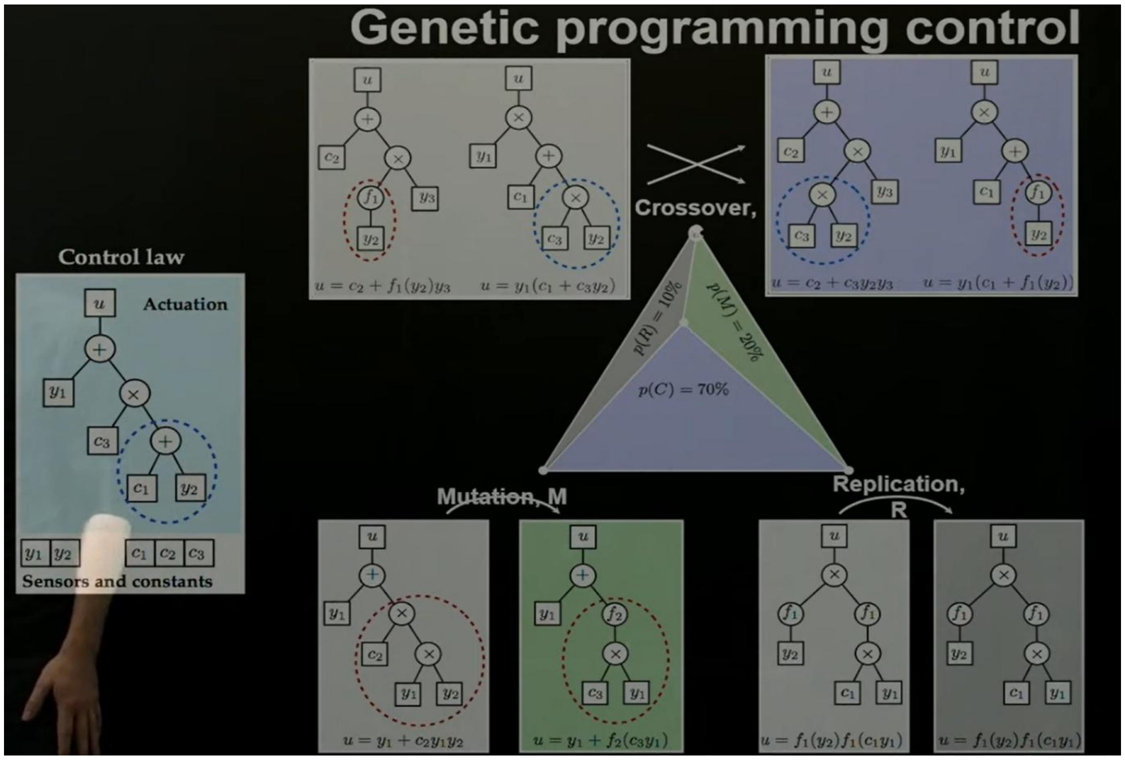
Crossover (Recombination)
Parts of two parent chromosomes are combined to produce offspring, exploiting existing good solutions.

Mutation
Small random changes are introduced to chromosomes to increase diversity and promote exploration.

Iteration
By repeatedly applying these steps, successive populations move heuristically toward optimal solutions.

---

3. From Genetic Algorithms to Genetic Programming:



Key Difference Between GA and GP

Genetic Algorithms assume a fixed mathematical structure and only optimize parameters.
Example:

$$\boldsymbol{u = ax + by + c}$$

Genetic Programming does not assume any predefined formula.
Instead, it evolves the entire expression itself, including operators, variables, and constants.

This means GP can discover entirely new equations, not just tune coefficients.

Genetic Programming Summary:

GP = GA principles + tree representation

Population → symbolic expressions (trees)

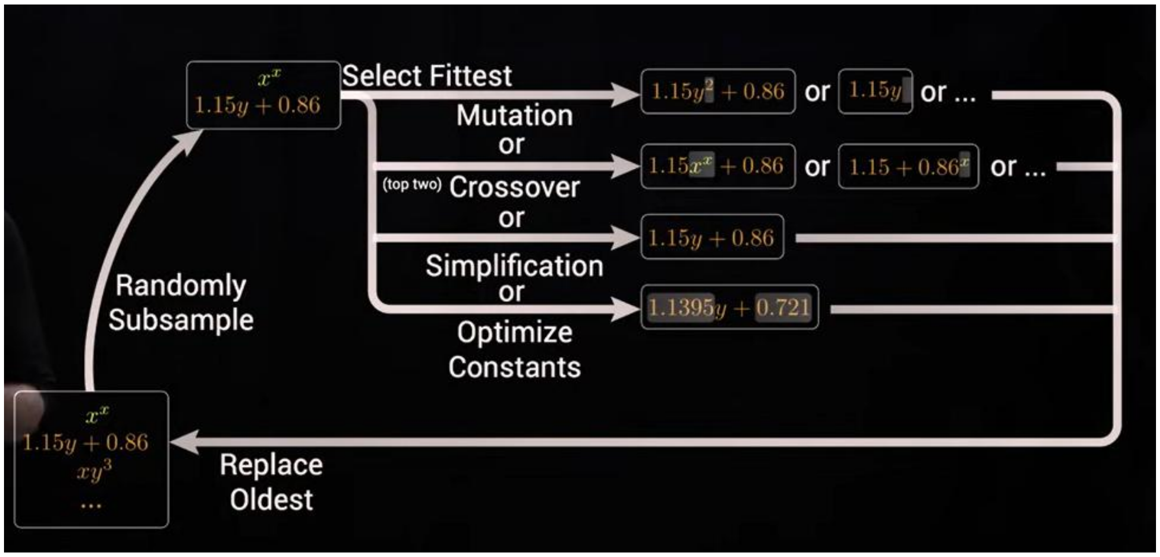
Crossover → subtree crossover

Mutation → subtree mutation

Selection → based on fitness (MSE)

Parameters (constants) → evolved automatically

This is an overview on symbolic regression:



Now we start talking about implementation details:

- Expression Tree Representation

In Genetic Programming, each candidate solution (chromosome) is represented as an expression tree.

Internal nodes represent operators (+, -, *, /, sin, cos, etc.)

Leaf nodes represent terminals:

the input variable x and numeric constants

Example:

The expression: $\boldsymbol{x+3}$

is represented as:

```
  +
 / \
x   3
```

- Evaluation

Expression evaluation is implemented recursively:

Terminal nodes return either the input value or a constant.

Operator nodes evaluate child subtrees and apply the operator.

This recursive structure naturally supports genetic operators such as crossover and mutation.

- Safe Operators and Robust Evaluation

During evolution, many randomly generated expressions may be mathematically invalid: division by zero, square root of negative values, overflow or undefined results.

Crashing the algorithm due to a single invalid individual would halt evolution. Instead, safe operators are used.

Safe Division: Division by zero returns a bounded constant instead of raising an exception.

Safe Square Root: The square root is applied to the absolute value to avoid domain errors.

Design Philosophy: invalid individuals are not removed explicitly. Instead:

they receive poor fitness values, natural selection eliminates them over generations.

This ensures: numerical stability, uninterrupted evolution, preservation of exploration.

- Random Expression Tree Generation

A probabilistic random expression tree generator is implemented with explicit depth control.

Key Features

Maximum depth constraint (max_depth)

Recursive tree construction

Guaranteed syntactic validity

Configurable probability model

Operator Sets

Unary operators: sin, cos, sqrt

Binary operators: +, -, *, /, pow

Probability Model

Decision	Probability
Operator vs Terminal	70% / 30%
Unary vs Binary	40% / 60%
Variable vs Constant	70% / 30%

This balance promotes diversity while keeping expressions meaningful.

- Initial Population Generation

The evolutionary process begins with an initial population of randomly generated expression trees.

Each tree represents a different symbolic expression.

Tree depth is limited to prevent excessive complexity.

No fitness evaluation occurs at this stage.

The goal is to maximize structural diversity.

- Dataset Generation

Since no dataset is provided, synthetic datasets are generated.

Supported Target Functions

$$\boldsymbol{f_1(x) = x^2 + 2x + 1}$$

$$\boldsymbol{f_2(x) = 0.2x + \sin(3x)}$$

$$\boldsymbol{f_3(x) = x^3 + \log(x + 1)}$$

Sampling Strategy

Uniform sampling of input values

Domain restrictions for logarithmic functions

Optional Gaussian noise

2000 samples per dataset

Datasets are stored in CSV format for portability.

- Fitness Evaluation

Fitness is defined using Mean Squared Error (MSE):

$$\text{Fitness} = \boldsymbol{\frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2}$$

Lower fitness values indicate better solutions.

Robust Fitness Handling

If an individual produces: NaN, infinity or undefined output, then it is assigned a large penalty fitness value.

This guarantees: algorithmic stability, uninterrupted evaluation, evolutionary pressure toward robust expressions.

- Parent Selection: Tournament Selection

Tournament selection is used to select parents.

Procedure is this:

1. Randomly select k individuals

2. Compare their fitness values.

3. Select the individual with the lowest MSE.

Tournament size is set to k = 3, balancing: exploration, exploitation, robustness to noisy fitness values.

- Elitism

Elitism preserves the best individuals across generations meaning that top 5% of individuals are copied directly to the next generation.

It Ensures high-quality solutions are never lost.

Improves convergence stability.

- Subtree Crossover

Crossover is implemented as subtree exchange.

Key Properties

Parents are cloned before modification.

Random nodes are selected as crossover points.

Subtrees are swapped via recursive reconstruction.

Maximum depth constraints are enforced.

Retry limits prevent infinite recursion.

This follows standard Genetic Programming literature.

- Mutation Operator

Mutation is implemented as subtree mutation.

Steps:

1-Clone parent tree.

2-Select a random node.

3-Generate a new random subtree.

4-Replace the selected node.

5-Enforce depth constraints.

6-Retry if necessary.

Benefits: Maintains diversity, Prevents premature convergence,Avoids destructive side effects

- Main Evolutionary Loop

Each generation performs:

Fitness evaluation

Elitism

Parent selection

Crossover

Mutation

Population update

This loop repeats for a fixed number of generations.

- Tree Depth Control and Bloat Prevention

Tree depth is strictly limited to prevent bloat.(Bloat refers to uncontrolled growth of trees without fitness improvement.)

Depth control enforces a trade-off between: accuracy, simplicity, interpretability,

computational efficiency.

- Visualization

At each generation:

the best individual is visualized using Graphviz,

tree evolution can be inspected,

bloat or degeneration can be detected.

- Final Evaluation

After evolution:

the best individual is evaluated on sample inputs then predictions are compared with true target values.

Results demonstrate that the evolved symbolic expressions closely approximate the target functions.

Now lets talk about hyperparameter tuning:

I have done some changes in the code to have stable datasample so we can tune our hyperparameters with a consistent dataset.

Experiment 1 – Effect of Population Size

Objective:

The objective of this experiment is to analyze the impact of population size on the performance of the Genetic Programming (GP) algorithm for the symbolic regression task.
Population size is a critical hyperparameter in evolutionary algorithms, as it directly affects population diversity, exploration capability, convergence behavior, and computational cost.

Experimental Setup:

In this experiment, all algorithm parameters were kept constant, and only the population size was varied.
The following population sizes were evaluated:

Small population: Pop = 20

Medium population: Pop = 50

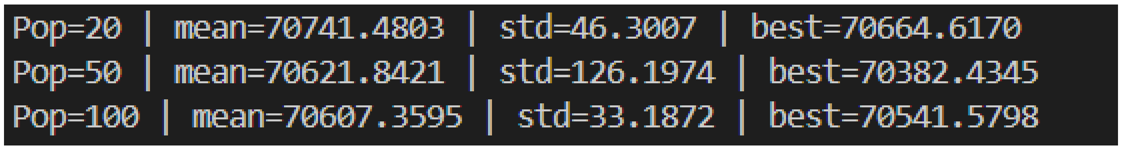
Large population: Pop = 100

For each population size, the algorithm was executed five independent runs to reduce the effect of randomness inherent in evolutionary processes.
At the end of each run, the final best Mean Squared Error (MSE) was recorded as the performance metric.

Experimental Results:

The statistical results obtained from multiple independent runs are summarized in Table 1.

Table 1 – Effect of Population Size on Final MSE



```
Pop=20 | mean=70741.4803 | std=46.3007 | best=70664.6170
Pop=50 | mean=70621.8421 | std=126.1974 | best=70382.4345
Pop=100 | mean=70607.3595 | std=33.1872 | best=70541.5798
```

Results Analysis

1. **Effect of Population Size on Solution Quality**
   Increasing the population size from 20 to 100 results in a reduction in the average MSE.
   This indicates that larger populations provide greater genetic diversity, allowing the algorithm to explore the search space more effectively and discover better symbolic expressions.

2. **Stability and Robustness**
   The standard deviation for population size 100 is significantly lower compared to population size 50.
   This demonstrates that larger populations lead to more stable and reliable performance, with less sensitivity to random initialization.

3. **Best Observed Performance**
   The lowest (best) MSE value was achieved with a population size of 50.
   This suggests that medium-sized populations can occasionally produce superior solutions, although their performance is less consistent across runs.

4. **Computational Trade-off**
   Increasing the population size improves solution quality and stability but also increases computational cost.
   Therefore, there exists a trade-off between **performance quality** and **execution time**.

5. **Discussion**

The results confirm that population size plays a significant role in the performance of Genetic Programming.
Small populations tend to suffer from premature convergence due to limited diversity, while larger populations maintain diversity for a longer period, leading to more consistent convergence behavior.

Although the population size of 50 achieved the best individual result, the population size of 100 demonstrated the most balanced performance in terms of average error and robustness.

Conclusion: Based on the experimental results, a population size of **100** is selected as the preferred configuration for this project, as it provides a favorable balance between solution quality, stability, and computational efficiency.

Experiment 2 – Effect of Operator Ablation (Removal of Addition Operator)

Objective

The objective of this experiment is to investigate the impact of removing a fundamental arithmetic operator from the function set on the performance and stability of the Genetic Programming (GP) algorithm.
Specifically, the addition operator (+) was removed to analyze its role in symbolic expression construction and overall regression accuracy.

Experimental Setup

In this experiment, two configurations of the GP algorithm were compared:

- **Full operator set**: including the addition operator (+)
- **Reduced operator set**: excluding the addition operator (+)

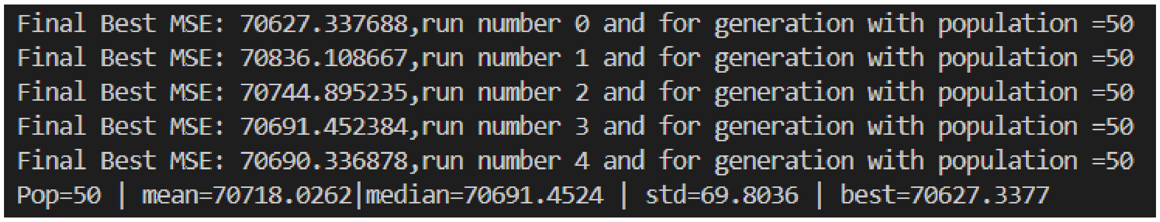
All other parameters, including population size, tree depth, number of generations, and genetic operators, were kept constant.
A population size of **50** was used, based on prior experimentation.
For each configuration, the algorithm was executed **five independent runs** to account for the stochastic nature of Genetic Programming.
The final best Mean Squared Error (MSE) from each run was recorded, and statistical measures including **mean, median, standard deviation, and best MSE** were computed.

Experimental Results

The statistical performance results for both configurations are summarized in **Table 2**.

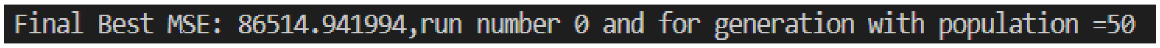
Table 2 – Effect of Addition Operator Removal on Final MSE

With +:

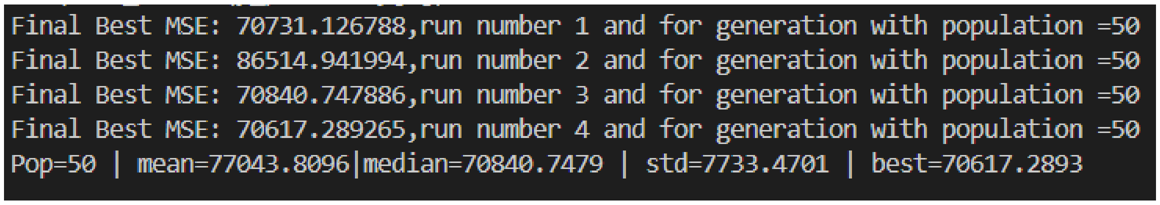


```
Final Best MSE: 70627.337688,run number 0 and for generation with population =50
Final Best MSE: 70836.108667,run number 1 and for generation with population =50
Final Best MSE: 70744.895235,run number 2 and for generation with population =50
Final Best MSE: 70691.452384,run number 3 and for generation with population =50
Final Best MSE: 70690.336878,run number 4 and for generation with population =50
Pop=50 | mean=70718.0262|median=70691.4524 | std=69.8036 | best=70627.3377
```

without +:



```
Final Best MSE: 86514.941994,run number 0 and for generation with population =50
```



```
Final Best MSE: 70731.126788,run number 1 and for generation with population =50
Final Best MSE: 86514.941994,run number 2 and for generation with population =50
Final Best MSE: 70840.747886,run number 3 and for generation with population =50
Final Best MSE: 70617.289265,run number 4 and for generation with population =50
Pop=50 | mean=77043.8096|median=70840.7479 | std=7733.4701 | best=70617.2893
```

Results Analysis

1. Effect on Typical Performance

The configuration including the addition operator achieved a lower median MSE compared to the configuration without it.
Since the median is robust to outliers, this indicates that the majority of runs perform better when the + operator is available.

2. Stability and Robustness

Removing the addition operator resulted in a substantial increase in the standard deviation of MSE values.
This demonstrates that the reduced operator set leads to unstable convergence and higher sensitivity to random initialization.
In contrast, the presence of the + operator produces consistent and reliable performance across runs.

3. Best Observed Performance

Although the configuration without the + operator achieved a marginally lower best MSE in one run, this improvement was not consistent across runs.
This suggests that the observed improvement is likely due to stochastic effects rather than a systematic advantage.

4. Role of the Addition Operator

The addition operator plays a critical role in enabling flexible symbolic composition.
Its absence restricts the expressive power of the GP trees, making it more difficult to construct accurate approximations of the target function.

Discussion

The results indicate that operator ablation can significantly affect both performance and robustness in Genetic Programming.
While the removal of the + operator may occasionally lead to competitive solutions, it substantially degrades overall stability and typical performance.
This highlights the importance of maintaining a sufficiently expressive function set to support reliable evolutionary search.

Conclusion: Based on the experimental findings, the inclusion of the addition operator is essential for stable and robust symbolic regression performance.
Although the reduced operator set achieved a slightly better best-case result, the full operator set demonstrated superior median performance and significantly lower variance.
Therefore, the addition operator was retained in the final configuration of the GP system.

The end.