# 

**Assessment 3**

**A Genetic Programming Approach for Symbolic Regression**

**Unit Coordinator : Umair Ullah Tariq**

**Class : Tutorial 1**

**Date : 9 June 2025**

**Student Number Student Name**

**12223508 Furkan Tekkartal**

**COIT29224**

**Evolutionary Computation**

**2025 Term 1**

Table of Contents

[0](#_Toc200394202)

[1 Introduction 2](#_Toc200394203)

[2 Execution Instructions 3](#_Toc200394204)

[2.1 Prerequisites 3](#_Toc200394205)

[2.2 Running the Code 3](#_Toc200394206)

[2.3 Output 3](#_Toc200394207)

[2.4 GitHub Repository 3](#_Toc200394208)

[3 Methodology: A Genetic Programming System 4](#_Toc200394209)

[3.1 Program Representation 4](#_Toc200394210)

[3.2 Population Initialization 4](#_Toc200394211)

[3.3 Fitness Evaluation 4](#_Toc200394212)

[3.4 Parent Selection 5](#_Toc200394213)

[3.5 Genetic Operators 5](#_Toc200394214)

[3.6 System Parameters and Termination 5](#_Toc200394215)

[4 Experimental Results and Analysis 6](#_Toc200394216)

[4.1 Execution Log 6](#_Toc200394217)

[4.2 Analysis of Convergence 7](#_Toc200394218)

[4.3 Analysis of the Final Evolved Solution 7](#_Toc200394219)

[4.4 Identifying and Mitigating Bloat 8](#_Toc200394220)

[5 Comparative Analysis of Evolutionary Algorithms 9](#_Toc200394221)

[5.1 Genetic Programming (GP) 9](#_Toc200394222)

[5.2 Particle Swarm Optimization (PSO) 9](#_Toc200394223)

[5.3 Evolution Strategies (ES) 9](#_Toc200394224)

[5.4 Summary of Differences 9](#_Toc200394225)

[6 Coding Practices 10](#_Toc200394226)

[7 Conclusion 11](#_Toc200394227)

[8 References 12](#_Toc200394228)

# Introduction

Genetic Programming (GP) is a sub-field of Evolutionary Computation that creates executable computer programs as solutions to problems. Drawing inspiration from biological evolution, GP starts with a population of unfit, randomly generated programs and iteratively improves them over generations. This is achieved through applying genetic operators such as crossover and mutation, guided by a fitness function that measures a program’s performance on a given task.

This project focuses on the application of GP to symbolic regression. Unlike traditional regression methods which require a predefined model structure, symbolic regression searches for the mathematical expression itself. Given a set of input-output data points, the goal is to discover an underlying function that accurately describes their relationship. This makes GP a powerful tool for automated model discovery in fields where the governing equations are unknown.

The primary objectives of this project were to:

* Implement a modular Genetic Programming system in Python from scratch.
* Apply the system to solve a symbolic regression problem using a provided dataset with two independent variables (x, y).
* Analyse the performance of the algorithm and the characteristics of the evolved solution.
* Compare the principles of Genetic Programming with other evolutionary paradigms like Particle Swarm Optimization (PSO) and Evolution Strategies (ES).

# Execution Instructions

## Prerequisites

System Requirements:

* Python 3.x environment.
* The graphviz Python library (pip install graphviz).
* A system-level installation of the Graphviz software, with its bin directory added to the system’s PATH as explained in the “GraphViz for Image Creation” pdf file (Moodle -> Week 11 -> Additional Resources)

The project is organized into a modular structure to separate concerns:

* **main.py**: The main script that runs the evolutionary process.
* **problem.py**: Defines the problem specifics (functions, terminals, data).
* **tree.py**: Contains the GPTree class and initialization logic.
* **operators.py**: Contains the selection, crossover, and mutation functions.
* **visualization.py**: Handles the generation of the tree diagram.
* **symbolic\_regression\_data.csv**: The input dataset.

## Running the Code

* A screenshot of a computer

  AI-generated content may be incorrect.Run windows PowerShell on project directory.

Image 1: PowerShell in project directory

* Run the program.

python main.py

## Output

* Console logs show progress and results.
* A screenshot of a computer program

  AI-generated content may be incorrect.Visualization plot is saved to the same directory.

Image 2: Running the code

## GitHub Repository

<https://github.com/furkantekkartal/COIT29224_Asg3>

# Methodology: A Genetic Programming System

The GP system was developed in Python, with its architecture designed to be modular and reflective of the core concepts taught in the course.

## Program Representation

Each individual solution in the GP population is represented as an expression tree. This is a standard and effective representation for mathematical formulas.

* **Internal Nodes:** These are operators from the defined function set (+, -, \*, /).
* **Leaf Nodes (Terminals):** These are the inputs to the expression, consisting of the variables x, y, and ephemeral random constants.

This representation is managed by the GPTree class within the tree.py module in our project.

## Population Initialization

A diverse initial population is critical for a successful evolutionary search. The system employs the ramped half-and-half initialization method. This strategy generates a variety of tree structures by combining two techniques:

* **Full method:** Creates trees where all leaf nodes are at the same maximum depth, resulting in “bushy” structures.
* **Grow method:** Creates trees of more variable shapes and sizes by allowing terminals to be selected before the maximum depth is reached.

This approach ensures the initial gene pool contains a wide range of program sizes and complexities, providing a robust starting point for evolution.

## Fitness Evaluation

The fitness function quantifies the performance of each individual program. For this problem, a program’s quality is inversely related to its error. The fitness is calculated as the sum of absolute errors over all 49 fitness cases in the dataset.

Fitness = ∑ | Predicted\_Result - Actual\_Result |

A lower fitness score indicates a smaller error and thus a better solution. The function calculate\_fitness in main.py implements this logic. To ensure stability, the function assigns an infinite penalty to any expression that results in a mathematical error (e.g., overflow), effectively removing it from the gene pool.

## Parent Selection

To select individuals for reproduction, the system uses fitness proportionate selection, commonly known as roulette wheel selection. This probabilistic method assigns a selection probability to each individual that is proportional to its fitness.

To handle the inverse relationship between error and fitness (lower error is better), the fitness scores are first inverted. A small epsilon (1e-9) is added to the denominator to prevent division by zero for perfect solutions. This implementation, found in operators.py, ensures that fitter individuals are more likely to be selected as parents, while still maintaining genetic diversity by allowing less fit individuals a chance to reproduce.

## Genetic Operators

New solutions are created through the application of two primary genetic operators:

* **Crossover**: The system uses subtree crossover, where two parent trees exchange randomly selected subtrees. This operator allows for the combination and propagation of useful “building blocks” (sub-expressions) throughout the population. It is the primary mechanism for exploring new solutions.
* **Mutation**: Subtree mutation introduces new genetic material by replacing a randomly selected subtree with a newly generated random tree. This operator is crucial for maintaining diversity and preventing the population from converging prematurely to a suboptimal solution.

## System Parameters and Termination

Key parameters for the evolutionary run are defined in main.py. For reproducibility in testing and reporting, the system uses a fixed random seed. This makes the stochastic process deterministic, ensuring that the same results are produced on every execution.

*# From main.py*

*# A random seed ensures that the run is reproducible.*

RANDOM\_SEED = 2

def evolve():

random.seed(RANDOM\_SEED)

*# ...*

The evolution terminates under one of two conditions:

1. A solution is found with a fitness score less than 0.1.
2. The maximum generation limit of 100 is reached

# Experimental Results and Analysis

## Execution Log

A screenshot of a computer program

AI-generated content may be incorrect.The following console output shows the successful execution of the GP system. It highlights the progressive reduction in fitness errors across generations.

Image 3: Console Output

## Analysis of Convergence

The execution log clearly demonstrates the learning capability of the GP system. The best fitness score starts at 100.6667 and is systematically reduced over the 100 generations, with notable improvements occurring in distinct leaps (e.g., dropping from ~81 to ~29 in generation 3).

This pattern is characteristic of a successful evolutionary search process, where the algorithm discovers and propagates beneficial genetic material. The final error of 0.2139 indicates a high degree of convergence towards a good solution.

## Analysis of the Final Evolved Solution

The best individual found by the algorithm is a highly accurate model of data.

* **Final Fitness (Sum of Absolute Errors)**: 0.2139
* A black and white image of a tree

  AI-generated content may be incorrect.**Final Tree Diagram**: The structure of the evolved program is visualized below.

Image 4: Tree Diagram

## Identifying and Mitigating Bloat

While highly accurate, the final solution is extremely large and complex, as seen in the tree diagram. This is a classic example of bloat, a common phenomenon in GP where program size increases over generations without a proportional gain in fitness. Bloat can occur as the algorithm discovers that adding neutral or redundant code (introns) can protect effective sub-expressions from being destroyed by crossover.

The presence of bloat in the final result is a key finding. It highlights the trade-off between solution accuracy and interpretability. In a practical application, this bloated expression would be unusable. To address this, a technique called parsimony pressure could be introduced. This involves modifying the fitness function to penalize larger trees, encouraging the evolution of solutions that are both accurate and concise.

# Comparative Analysis of Evolutionary Algorithms

This section compares Genetic Programming with two other major evolutionary paradigms.

## Genetic Programming (GP)

GP is unique in that it evolves executable programs, typically represented as trees. It excels at problems where the solution’s structure is unknown, such as symbolic regression. The search is driven by structure-modifying operators like subtree crossover and mutation.

## Particle Swarm Optimization (PSO)

PSO is a population-based algorithm inspired by the flocking behavior of birds. It works with a population of particles, each representing a candidate solution as a vector in a continuous space. Particles adjust their “flight” based on their own best-found position and the best position found by the entire swarm. PSO is highly effective for numerical optimization but is not naturally suited for discovering symbolic structures.

## Evolution Strategies (ES)

ES also evolves vectors of real-valued numbers. Its core feature is self-adaptation, where strategy parameters that control evolution (e.g., mutation strength) are co-evolved alongside the solution itself. Advanced methods like CMA-ES (covered in Week 9) are extremely powerful for optimizing complex, non-linear functions by adapting the search distribution to the problem’s landscape.

## Summary of Differences

Table 1: Comparison Table of methodologies

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Genetic Programming (GP)** | **Particle Swarm Optimization (PSO)** | **Evolution Strategies (ES)** |
| **Representation** | Hierarchical Trees (Programs) | Position Vectors (Particles) | Real-valued Vectors |
| **Primary Analogy** | Genetic Evolution | Social Behaviour (Flocking) | Genetic Evolution |
| **Key Operators** | Crossover & Mutation | Velocity/Position Updates | Self-Adapting Mutation |
| **Best Suited For** | Symbolic Regression, Program Synthesis | Continuous Function Optimization | Continuous Function Optimization |

For this assignment, GP was the most appropriate methodology as the primary challenge was to discover the symbolic form of an unknown mathematical function.

# Coding Practices

To ensure a high-quality and maintainable implementation, the project was structured with a strong emphasis on modularity, as recommended by both course materials and prior feedback.

* **Modular Design:** The project is organized into multiple Python modules, each with a distinct responsibility (e.g., problem.py for problem definition, operators.py for evolutionary operators) as per commented my previous assessment result. This separation of concerns improves readability and makes the code easier to test and extend.

For example, the operators.py module encapsulates all functions related to creating new individuals:

*# Snippet from operators.py*

*def select\_parents\_roulette\_wheel(population, fitnesses):*

*# ...*

*def crossover(parent1, parent2):*

*# ...*

*def mutation(individual):*

*# ...*

* **Classes and Methods:** The core GPTree data structure is implemented as a class, encapsulating data (the node’s content) and behaviour (e.g., calculating size, collecting nodes).
* **Documentation:** All modules and functions are documented with docstrings explaining their purpose, and inline comments clarify key algorithmic steps.
* **Code Conventions:** The code adheres to standard Python naming conventions and style guidelines for clarity and consistency

# Conclusion

This project successfully demonstrated the implementation and application of a Genetic Programming system to a symbolic regression problem. The system evolved a highly accurate mathematical model from a given dataset, achieving a final error score of 0.2139.

The key findings include not only the effectiveness of GP in finding accurate solutions but also the observation of practical challenges like program bloat. This highlights the importance of balancing the search for accuracy with the need for solution simplicity. By structuring the code in a modular fashion and documenting the process thoroughly, this project meets the core requirements of the assessment and provides a solid foundation for further exploration into more advanced GP techniques.

# References

COIT29224 Week 9-12 Lecture Notes and Tutorial Materials.