Stochastic optimization algorithms 2025 Home problems, set 2, Python version

Jarl Dang - 200108279191

October 2025

1 Problem 2.1

The Travelling Salesman Problem (TSP) was solved using an Ant Colony Optimization (ACO) algorithm, which models the collective behaviour of ants as they search for the shortest route between food sources and their nest. Each artificial ant incrementally constructs a tour based on two main factors. The first is the pheromone intensity along each edge, which represents the accumulated experience of previous ants. The second is the visibility, a heuristic value defined as the inverse of the distance between cities, which favours shorter connections. After all ants complete their tours, the pheromone trails are updated, with higher pheromone levels assigned to shorter routes, guiding future ants towards better paths.

Over successive iterations, the colony converges toward shorter paths through the combined effect of exploration (via random route construction) and exploitation (via pheromone reinforcement). The algorithm terminated after 259 iterations with a best path length of 99.79, just below the predefined target length of 100. This path is likely not the global optimum since we stopped the program when we reached an arbitrary limit instead of letting it converge. The final path obtained is illustrated in Figure 1, showing the sequence of cities corresponding to the best tour found.

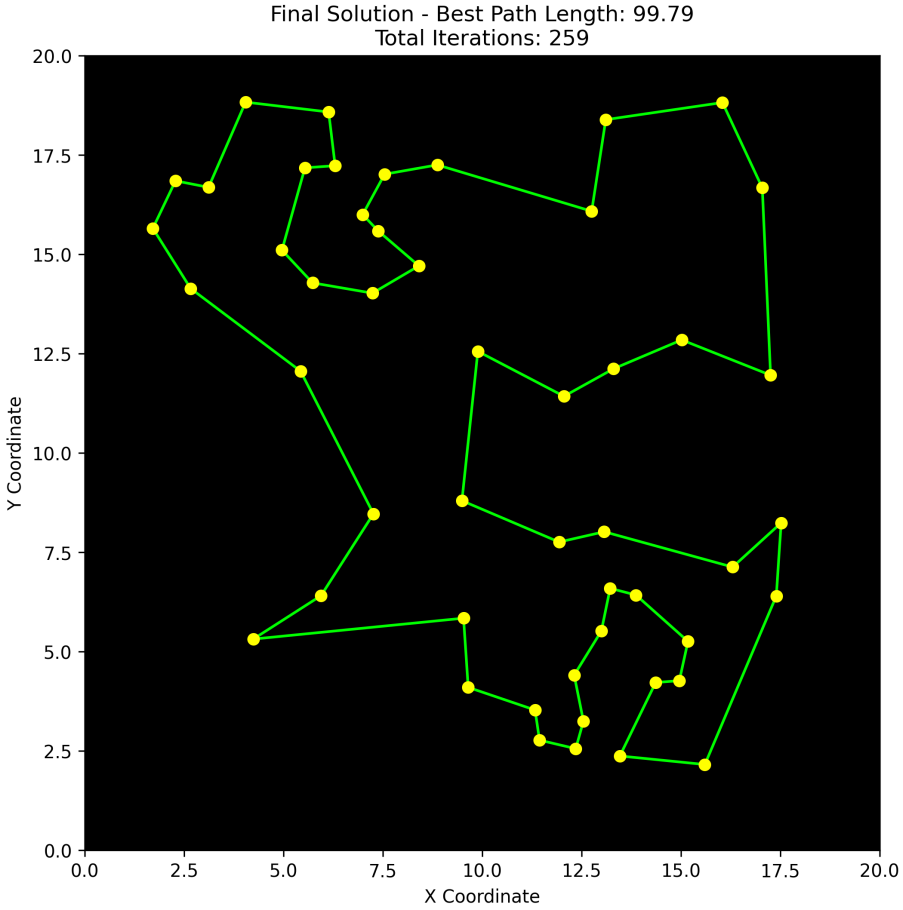

Figure 1: The plot of the final solution showing the best path found.

2 Problem 2.2

Plot Explanation

The plot visualizes the contours of the **Himmelblau function**, which is commonly used as a benchmark in optimization. It is a hard function to find the minima for, since its surface contains several saddle points and multiple local minima. Thus making it easy for any form of gradient descent approach to get stuck. The contour plot in Figure 2 clearly illustrates this complex landscape, with the contours forming four distinct basins corresponding to the known global minima. The function is defined as:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

Contour lines in the plot represent different function values, illustrating the landscape and multimodal nature of the function. The red stars on the plot indicate the **known minima** of the Himmelblau function. These minima are well-known and are located at approximately:

- $(3.0,\ 2.0)$
- $(-2.805,\ 3.131)$
- $(-3.779,\ -3.283)$
- $(3.584,\ -1.848)$

Each minimum is annotated on the plot, showing where the function achieves its lowest values. This visualization is useful for understanding the complexity of the function and for evaluating optimization algorithms.

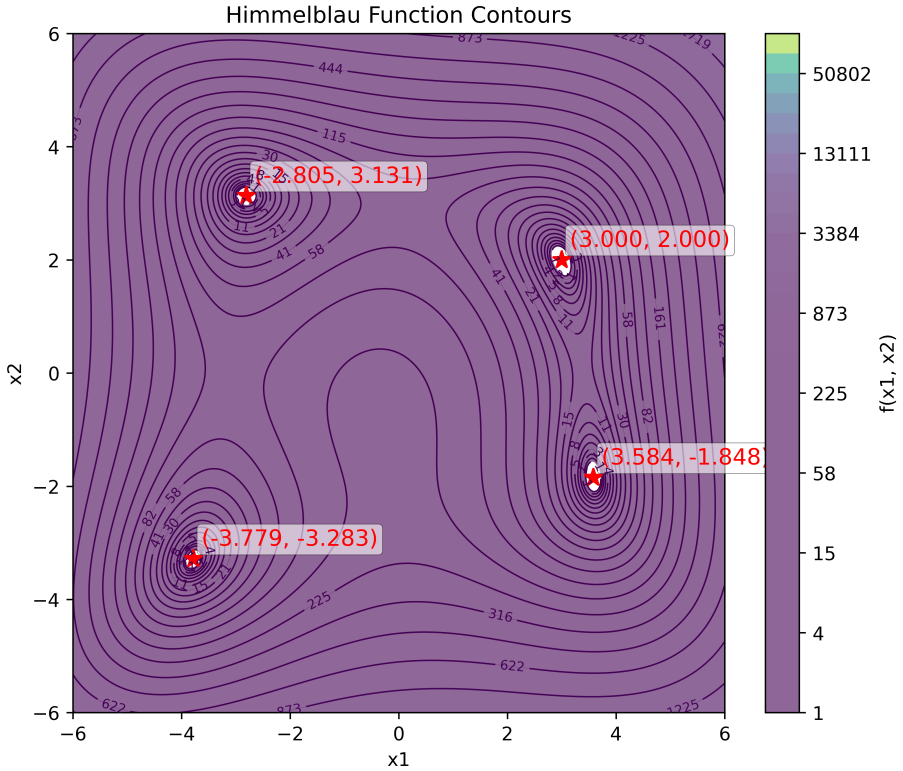

Figure 2: Contour plot of the function f with the previously known minima shown as red stars along with their x values.

Particle Swarm Optimization and Results

PSO operates by simulating a population (or swarm) of particles that move through the search space. Each particle adjusts its position based on both its own best-known position and the best position found by the entire swarm. This interaction allows the algorithm to dynamically balance **exploration** (searching new areas) and **exploitation** (refining known good regions).

In our experiments, we initialized a swarm of particles within the search space and ran the PSO algorithm for a fixed number of iterations. After each run, we recorded the best position and function value found by the swarm. By repeating this process multiple times, we obtained a set of results that demonstrate the optimizer's ability to locate the minima of the Himmelblau function. The minima found by the PSO can be seen in table 2. Since PSO is stochastic I had to run the program several times in order to ensure that I found all the minima.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{f(x_1, x_2)}$
3.584429	-1.848145	0.000000
-3.779299	-3.283189	0.000000
-2.805120	3.131316	0.000000
2.999996	2.000008	0.000000

Table 1: Found minima using PSO and their respective function values for given (x_1, x_2) points.

3 Problem 2.3

Truck Model Implementation

Configurations

The parameter configurations for the program can be seen in the table below.

Parameter/Constant	Value	Description
Neural Network Configuration		
N_I	3	Number of input neurons
N_H	5	Number of hidden neurons
N_O	2	Number of output neurons
W_{MAX}	10.0	Maximum absolute weight value
$SIGMOID\_C$	2.0	Sigmoid activation parameter
Genetic Algorithm Configuration		
Population Size	300	Number of individuals in population
Mutation Rate	1/chromosome length	Probability of mutation per gene
Crossover Rate	0.8	Probability of crossover
Elitism	1	Number of elite individuals preserved
Tournament Size	3	Number of individuals in tournament selection
Max Generations	200	Maximum number of generations
Truck Model Constants		
Mass (M)	20000.0	Truck mass [kg]
Base Engine Brake Coeff (C_b)	3000.0	Engine brake coefficient
Max Brake Temp (T_{max})	750.0	Maximum brake temperature [K]
Temp Cooling Tau (τ)	30.0	Brake cooling time constant
Temp Heating Ch (C_h)	40.0	Brake heating coefficient
Ambient Temp (T_{amb})	283.0	Ambient temperature [K]
Gravity (g)	9.80665	Gravitational acceleration [m/s^2]
Gear Factors	(7.0, 5.0, ..., 1.0)	Gear ratios for 10 gears
Simulation Parameters		
Max Distance	1000.0	Maximum simulation distance [m]
Max Time	3600.0	Maximum simulation time [s]
Velocity Min (v_{min})	1.0	Minimum allowed velocity [m/s]
Velocity Max (v_{max})	25.0	Maximum allowed velocity [m/s]
Time Step (dt)	0.1	Simulation time step [s]
Max Simulation Steps	100000	Maximum steps per simulation

Table 2: Configuration and constant values used in the optimization and simulation code.

Constants and Imports

The file begins by importing necessary modules: `math`, `enum`, `typing`, `numpy`, and a custom function `get_slope_angle` from `slopes.py`. Several physical and simulation constants are defined, such as gravitational acceleration (`GRAVITY`), gear ratios (`GEAR_FACTORS`), and parameters for brake temperature dynamics and simulation control.

Gear Enumeration

A `Gear` class is defined as an `IntEnum` with values from G1 to G10, representing the truck's gears. Each gear has an associated gear factor, accessible via the `factor` property, which indexes into the `GEAR_FACTORS` tuple.

Brake Force Functions

Two functions model the braking forces:

- `engine_brake_force`: Computes the engine brake force as the product of a base coefficient and the current gear's factor.
- `foundation_brake_force`: Calculates the service brake force based on the truck's mass, brake pedal pressure, and brake temperature. If the brake temperature exceeds a threshold, the force decays exponentially.

Gravity Component

The `gravity_component` function computes the downhill force due to gravity, given the truck's mass and the slope angle in degrees.

Truck Class

The core of the file is the `Truck` class, which encapsulates the truck's physical properties and simulation state.

Initialization

The constructor initializes the truck's mass, engine brake coefficient, brake temperature parameters, ambient temperature, and simulation time step. State variables include position, velocity, brake temperature (as a delta above ambient), and simulation time.

Properties and Gear Control

Properties provide access to the truck's mass, current gear, and total brake temperature. Methods for gear control include `set_gear`, `shift_up`, and `shift_down`, allowing the gear to be changed within valid bounds.

State Reset

The `reset` method reinitializes the truck's state, including position, velocity, gear, and brake temperature.

Physics Wrappers

Thin wrapper methods compute the current engine and service brake forces, slope angle, and gravity force at the truck's position.

Net Force Calculation

The `net_force` method calculates the net force acting on the truck as the difference between the gravity force and the sum of the service and engine brake forces.

Brake Temperature Update

The `update_temperature` method updates the brake temperature according to whether the pedal is pressed (heating) or released (cooling), ensuring the temperature does not fall below ambient.

State Update

The `update_state` method advances the simulation by one time step, updating velocity, position, brake temperature, and time.

Simulation Loop

The `simulate` method runs the simulation loop, interacting with a controller function to determine pedal pressure and gear changes. It records the history of the truck's state and checks for constraint violations (velocity limits, brake temperature). The method returns a dictionary containing the simulation history and performance metrics.

Gear Change Application

The `apply_gear_change` method applies gear change requests from the controller, shifting up or down as appropriate.

Neural Network Controller Implementation

Constants and Imports

The file imports `numpy`, `math`, and several types from `typing`. It also imports a decoding function from `run_encoding_decoding_test.py`. Constants are defined for the neural network weight range (`W_MAX`), truck control limits (`V_MAX`, `V_MIN`, `ALPHA_MAX`, `T_MAX`), and the sigmoid activation parameter (`SIGMOID_C`).

NeuralNetworkController Class

The main class, `NeuralNetworkController`, implements a feedforward neural network for controlling the truck's brake pedal and gear changes.

Initialization

The constructor takes the number of input, hidden, and output neurons, as well as an optional chromosome for weight initialization. If a chromosome is provided, weights are decoded using the `decode_chromosome` function; otherwise, weights are initialized randomly. The network uses bias neurons in both the input and hidden layers.

Activation Function

The `activate` method applies a sigmoid activation function with a configurable parameter c, clipping input values to prevent numerical overflow.

Forward Pass

The `forward` method computes the neural network's output for a given input vector. Inputs are normalized and a bias is added. The method performs matrix multiplication to propagate activations through the hidden and output layers, applying the activation function at each stage.

Control Logic

The `control` method serves as the interface for truck simulation. It normalizes the velocity, slope angle, and brake temperature according to the specified limits. The neural network outputs two values: brake pedal pressure (clamped to [0, 1]) and gear change. The gear change output is mapped to -1 (downshift), 0 (no change), or 1 (upshift) using threshold values. Additional logic enforces sensible gear changes based on velocity and time constraints, such as forcing upshifts at low velocity and downshifts near maximum velocity, and ensuring a minimum time interval between gear changes.

Controller Factory

The function `create_controller_from_chromosome` constructs a controller function from a chromosome, returning a callable that produces control signals for the truck simulation.

Genetic Algorithm Optimization Implementation

Imports and Configuration

The file imports standard libraries for numerical computation, randomization, timing, file I/O, plotting. It also imports the truck simulation model and neural network controller factory.

Type Aliases

Type aliases are defined for population and fitness function types to improve code readability.

GeneticAlgorithm Class

The `GeneticAlgorithm` class implements a genetic algorithm for optimizing neural network weights used in truck control.

Initialization

The constructor sets population size, chromosome length, neural network architecture parameters, mutation and crossover rates, elitism, tournament selection size, and random seed. The population is initialized with random chromosomes, and statistics for fitness history and best individuals are prepared.

Population Management

- `_init_population`: Initializes the population with random chromosomes.
- `_evaluate_population`: Evaluates all individuals using the provided fitness function, sorts them by fitness, and tracks the best individual.
- `_selection`: Implements probabilistic tournament selection, favoring higher-fitness individuals.
- `_crossover`: Performs uniform crossover between two parents to produce offspring.
- `_mutate`: Applies Gaussian mutation to genes, clamping values to the valid range.

Evolution Process

The `evolve` method runs the genetic algorithm for a specified number of generations. Each generation involves evaluating the population, recording fitness statistics, applying elitism, and generating new individuals through selection, crossover, and mutation.

Fitness History Plotting

The `plot_fitness_history` method visualizes the best and average fitness over generations, saving or displaying the plot.

Fitness Function Creation

The `create_fitness_function` function constructs a fitness function for the genetic algorithm. It simulates the truck on a set of slopes using a controller derived from a chromosome, and computes fitness based on average speed, distance traveled, pedal usage, and penalties for constraint violations such as overheating or excessive velocity.

Optimization Routine

The `run_optimization` function sets up the truck and neural network parameters, creates training and validation fitness functions, and initializes the genetic algorithm. It runs the optimization loop, evolving the population and evaluating the best chromosome on the validation set each generation. The best chromosome is periodically saved to file for later use, and fitness history is plotted at the end.

3.1 Slope Data

The slopes are defined in `slopes.py` and include training, validation, and test datasets. Each slope is characterized by a varying angle, modelled using sinusoidal and cosine functions. The datasets ensure the controller generalizes well to unseen slopes. The slopes were created with the restrictions in mind and also designed to be similar to the slopes given in the task. It was challenging to design the slopes, considering the training set is quite narrow I had to balance between introducing unforeseen slopes and reinforcing typical slopes.

3.2 Choice of Fitness Measure

The fitness measures were carefully designed to align with the objectives of the project. It was a lot of testing to figure out what behaviour should be rewarded and punished in order to provide good results, as well as balancing between safety and speed. It began with a standard fitness measure as the one given, namely a fitness score based on the average velocity times the distance travelled. I then tried to improve upon this by rewarding the speed even more, but this yielded results in which the truck went too fast and broke the constraints. So then I heavily penalized the truck for breaking the constraints. But then it became too scared to break because of breaking the temperature constraint in steep slopes, so I added a term to reward proper brake usage and staying close to the threshold. This configuration of penalties and rewards gave good results, but there were also a lot of tinkering with the parameters on how hard to punish and how much to reward certain behaviours.

Neural Network Controller (GA)

- **Safety:** The penalty for brake temperatures exceeding the maximum ensures the controller prioritizes safety.
- **Efficiency:** The reward for maintaining high average speed encourages efficient operation.
- **Pedal Usage:** The reward for using the brake pedal near the maximum velocity ensures the controller operates within safe limits while minimizing unnecessary braking.

This multi-objective fitness function was designed in order to balance safety and efficiency.

3.3 Results and Observations

Neural Network Controller

- The GA successfully optimized the controller to handle various slopes while avoiding overheating and maintaining efficiency.
- The fitness function's balance between safety and efficiency led to robust solutions that generalized well to unseen slopes.
- Steep slopes that have a low degree of variation pose a significant challenge for the program. This could depend on the slope not being physically feasible to complete within the constraints, thus leading to V_{max} being broken instead of the temperature threshold to maximise fitness.

3.4 Conclusion

The implementation demonstrates the effectiveness of stochastic optimization algorithms for solving complex control. The carefully designed fitness measures played a crucial role in guiding the optimization process toward safe, efficient, and interpretable solutions.

3.5 Plot

The steady increase in both training and validation fitness over generations indicates that the population is evolving towards better solutions. Occasional plateaus or fluctuations may reflect periods of stagnation or exploration, which are typical in evolutionary algorithms. The gap between training and validation fitness can highlight potential generalization issues. Overall, the plot demonstrates the effectiveness of the genetic algorithm in improving solution quality and helps identify when further optimization yields diminishing returns. When running the program with more neurons the model tended towards overfitting, meaning that the blue graph was significantly better than the red graph.

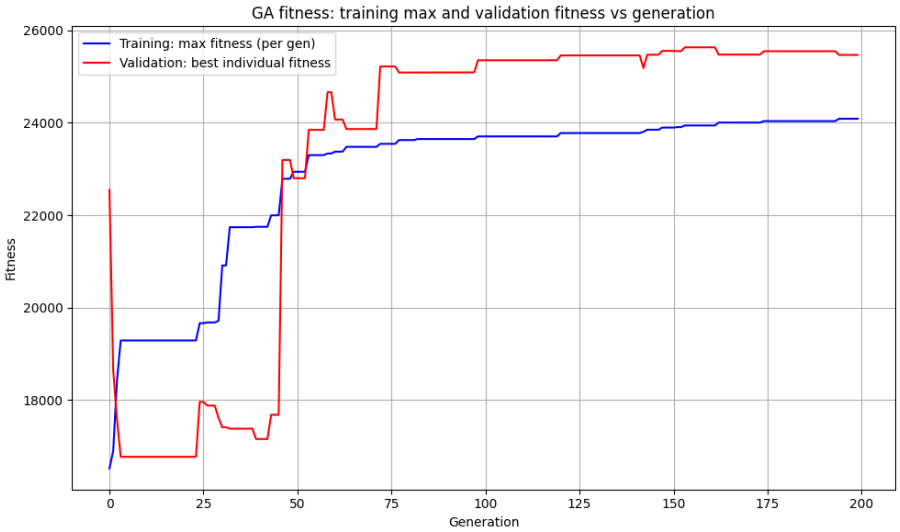

Figure 3: Fitness history as a function of the generation. The blue line indicates the best fitness for the training data set and the red line indicates the best fitness for the validation data set.

4 Problem 2.4

Configurations of parameters

Below is a table highlighting all the parameter values for the program.

Parameter	Value	Description
Population and Chromosome		
Population Size	250	Number of individuals in population
Min Chromosome Length	10	Minimum instructions per initialised chromosome
Max Chromosome Length	30	Maximum instructions per initialised chromosome
Instruction Size	4	Genes per instruction
Max Instructions	100	Maximum instructions allowed
Variable Registers	4	Number of variable registers
Constant Registers	1	Number of constant registers
Constant Values	{1.0}	Values for constant registers
Genetic Operators		
Crossover Probability	0.8	Probability of crossover
Tournament Size	5	Tournament selection size
Mutation and Adaptation		
High Error Threshold	0.20	RMSE threshold for high mutation
Low Error Threshold	0.01	RMSE threshold for low mutation
Mutation Rate (High Error)	6.0	Mutation rate at high error
Mutation Rate (Low Error)	0.5	Mutation rate at low error
Stagnation Generation Limit	75	Generations before mutation heat-up
Mutation Heat Up Factor	1.5	Multiplier for mutation rate on stagnation
Mutation Cool Down Factor	0.9	Multiplier for mutation rate on improvement
Improvement Threshold Ratio	0.001	Relative improvement threshold
Operators		
Operators	+, -, *, /	Supported arithmetic operations
Fitness and Error		
Max Error Value	1×10^{10}	Value for invalid solutions
Target RMSE	0.01	Stopping criterion for evolution

Table 3: Configuration and parameter values for the Linear Genetic Programming algorithm.

Results

All the results have been given by the test_lgp_chromosome.py program. The computed function, found by the LGP program, from the best chromosome which fitted the functional data can be seen in the equation (1) below.

$$g(x) = \frac{-x^3 + x^2 - 1}{-x^4 + x^2 - 1} = \frac{x^3 - x^2 + 1}{x^4 - x^2 + 1} \tag{1}$$

Numerator coefficients:

$$\left[a_3 = -1, \quad a_2 = 1, \quad a_1 = 0, \quad a_0 = -1\right]$$

Denominator coefficients:

$$\left[b_4 = -1, \quad b_3 = 0, \quad b_2 = 1, \quad b_1 = 0, \quad b_0 = -1\right]$$

The calculated error, according to the standard equation for root mean square deviation, for $g(x)$ compared to the data is zero ($e = 0$). Thus we have found the exact equation which describes the given data points.

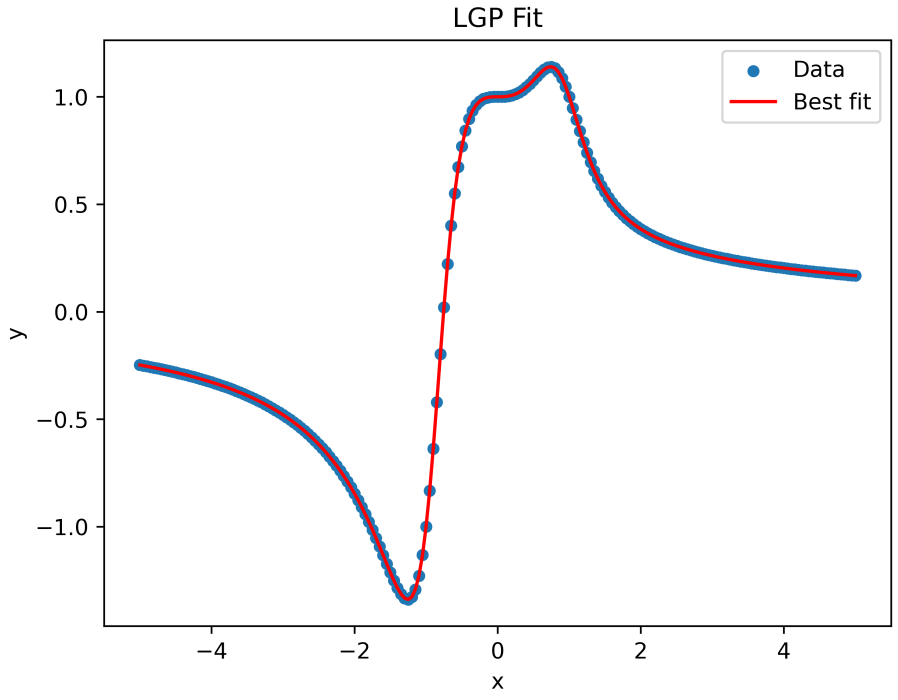

Figure 4: Symbolic regression using Linear Genetic Programming (LGP): The red curve shows the best-fit rational polynomial discovered by LGP $g(x)$, closely matching the target data points (blue dots).

Algorithm Description

The Linear Genetic Programming (LGP) approach works by evolving a population of simple computer programs, each made up of a sequence of instructions. These instructions use a set of registers, split into variable and constant types. The main evolutionary steps are the standards steps in a GA, namely selection, evaluation, crossover, and mutation.

To keep the population from getting stuck and losing diversity, we use an adaptive mutation rate. If the population stops improving for a while, we increase the mutation rate to shake things up and help the search escape local optima. If things are going well and we see regular improvements, we lower the mutation rate to focus on refining the best solutions. This way, the mutation rate adapts to what the search needs at any given time.

For selection, we use standard tournament selection to pick parents, and we always keep the best individual from each generation (elitism) so that good solutions aren't lost. For the tournament selection a tournament probability of 0.8 was chosen. Two point crossover happens with a set probability (0.8), and we throw away any offspring that end up with too many instructions, which helps keep solutions simple and manageable.

Register Configuration and Effects

The number of registers in Linear Genetic Programming (LGP) directly influences the complexity and flexibility of the evolved programs. In our implementation, we use four variable registers and one constant register. The LGP program uses the following register configuration:

- **Number of variable registers:** 4
- **Number of constant registers:** 1
- **Constant register values:** [1.0]

Variable Registers Variable registers serve as storage for intermediate results during program execution. After experimenting with different number of variable registers the number 4 yielded good results. Suggesting that it enables the representation of polynomials and rational expressions, while keeping the search space manageable. Too few variable registers would restrict the program's ability to model complex relationships, while too many could lead to slower convergence and less interpretable solutions.

Constant Registers Constant registers provide fixed numerical values that can be used in evolved expressions. In our setup, we use only one constant register. This choice simplifies the search space and encourages the algorithm to rely more on the input variable and arithmetic operations, often resulting in simpler and more generalizable solutions.

Using a single constant register works well when the target function does not require multiple distinct constants. It reduces the risk of overfitting and makes the evolutionary process more efficient. However, if the underlying function requires several different constants, having only one constant register may limit the program's ability to fit the data accurately. In such cases, the evolved solutions may lack flexibility and fail to capture certain relationships present in the data. In this case, we see that that all the coefficients are simple values.

I ran the program with several different constant registers with values that allow us to do various operations, such as scaling, sign inversion, null and keep the value as is. These values are represented below with $C = 4$:

$$[1.0,\ -1.0,\ 2.0,\ 0.0]$$

But the one that yielded the best results and converged the fastest was with just one constant register with value 1, likely because of the facts as mentioned earlier.

Special Operators

- **Adaptive mutation rate:** The mutation probability is not fixed, but dynamically adjusted throughout the evolutionary process. If the population shows no significant improvement in fitness for a specified number of generations (stagnation), the mutation rate is increased ("heated up") to encourage exploration and help escape local optima. Conversely, when the algorithm detects substantial progress (a new best solution), the mutation rate is decreased ("cooled down") to promote exploitation and fine-tuning of promising solutions. This adaptive mechanism helps balance exploration and exploitation, reducing the risk of premature convergence and improving overall search efficiency.

- **Elitism:** To ensure that the best-found solution is not lost during the evolutionary process, elitism is utilised. This means that the best individual from the current generation is always preserved and carried over to the next generation, guaranteeing that solution quality does not degrade.

- **Chromosome length control:** To prevent the evolution of overly complex or bloated solutions, offspring with more than 100 instructions are discarded and not added to the population. This operator encourages the evolution of simpler, more interpretable programs and helps maintain computational efficiency.