Lab 3:Genetic Algorithm for Quadratic Equation Solving

1 Introduction

Genetic Algorithms are evolutionary computational techniques inspired by natural selection and genetics. They belong to the class of evolutionary algorithms and are particularly effective for optimization problems where traditional analytical methods may be insufficient or computationally expensive.

In this lab, we implement a genetic algorithm to solve the quadratic equation:

$$2x^2 + 11x + 12 = 0 \tag{1}$$

The algorithm uses binary encoding to represent potential solutions and employs selection, crossover, and mutation operations to evolve the population towards optimal solutions.

2 Body of the Lab

2.1 Theory

The quadratic equation $2x^2 + 11x + 12 = 0$ has analytical solutions:

$$x_1 = \frac{-11 + \sqrt{121 - 96}}{4} = \frac{-11 + 5}{4} = -1.5 \; x_2 \quad = \frac{-11 - \sqrt{121 - 96}}{4} = \frac{-11 - 5}{4} = -4.0 \tag{2}$$

Our genetic algorithm approach involves:

- Representing solutions as binary chromosomes
- Using fitness function $f(x) = \frac{1}{1+|2x^2+11x+12|}$
- Applying roulette wheel selection
- Implementing single-point crossover
- Using bit-flip mutation

2.2 Implementation Parameters

Based on the configuration file, the algorithm uses:

- Binary encoding: 3 bits for integer part, 6 bits for fractional part
- Population size: 283
- Mutation probability: 0.25
- Number of generations: 641

2.3 Code Implementation

The implementation consists of two main classes:

- `GeneticAlgorithm`: Basic implementation for positive solutions
- `EnhancedGeneticAlgorithm`: Extended version handling negative solutions

2.3.1 GeneticAlgorithm Class Structure

The main `GeneticAlgorithm` class initializes with the following parameters:

```
class GeneticAlgorithm:
    def __init__(self, a, b, c, integer_bits=3, fraction_bits=6,
                 population_size=283, mutation_rate=0.25, generations=641):
        self.a, self.b, self.c = a, b, c
        self.integer_bits = integer_bits
        self.fraction_bits = fraction_bits
        self.chromosome_length = integer_bits + fraction_bits
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
```

2.3.2 Binary Encoding/Decoding

The algorithm uses binary representation with separate integer and fractional parts:

```
def binary_to_decimal(self, chromosome):
    """Convert binary chromosome to decimal value"""
    integer_part = chromosome[:self.integer_bits]
    fraction_part = chromosome[self.integer_bits:]

    # Convert integer part
    integer_value = 0
    for i, bit in enumerate(integer_part):
        integer_value += bit * (2**(self.integer_bits - 1 - i))

    # Convert fraction part
    fraction_value = 0
    for i, bit in enumerate(fraction_part):
        fraction_value += bit * (2**(-i-1))

    return integer_value + fraction_value
```

2.3.3 Fitness Function

The fitness function evaluates how close a solution is to the actual root:

```
def fitness_function(self, x):
    """Calculate fitness for equation ax² + bx + c = 0"""
    error = abs(self.a * x**2 + self.b * x + self.c)
    fitness = 1 / (error + 1e-10)  # Avoid division by zero
    return fitness
```

2.3.4 Selection Method

Roulette wheel selection is implemented with probability proportional to fitness:

```
def roulette_wheel_selection(self, population, fitness_scores):
    """Roulette wheel selection implementation"""
    total_fitness = sum(fitness_scores)
    probabilities = [f / total_fitness for f in fitness_scores]

    r = random.random()
    cumulative_prob = 0
    for i, prob in enumerate(probabilities):
        cumulative_prob += prob
        if r <= cumulative_prob:
            return population[i]
    return population[-1]
```

2.3.5 Crossover Operation

Single-point crossover creates offspring by combining parent chromosomes:

```
def single_point_crossover(self, parent1, parent2):
    """Single point crossover"""
    crossover_point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2
```

2.3.6 Mutation Operation

Bit-flip mutation with probability 0.25:

```
def mutate(self, chromosome):
    """Bit flip mutation"""
    mutated = chromosome[:]
    for i in range(len(mutated)):
        if random.random() < self.mutation_rate:
            mutated[i] = 1 - mutated[i]  # Flip bit
    return mutated
```

2.3.7 Enhanced Implementation for Negative Roots

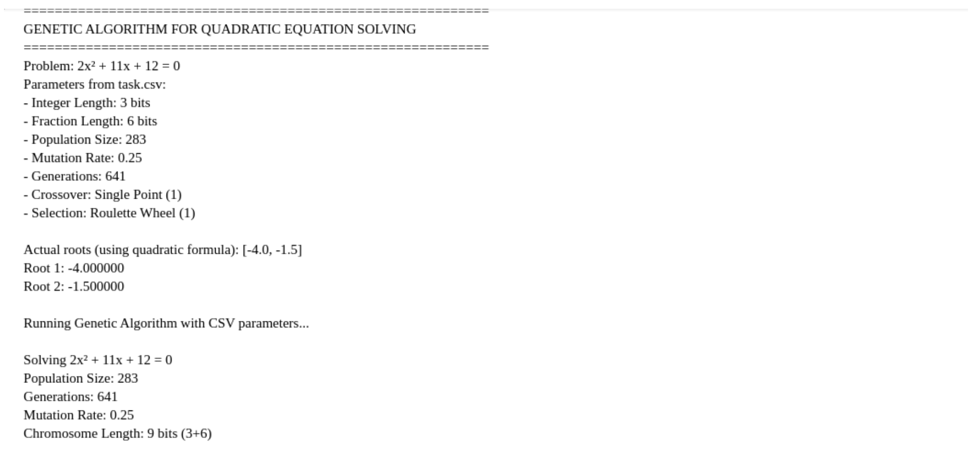
The `EnhancedGeneticAlgorithm` extends the base class to handle negative values by mapping binary representation to a specified range:

```
class EnhancedGeneticAlgorithm(GeneticAlgorithm):
    def __init__(self, a, b, c, integer_bits=3, fraction_bits=6,
                 population_size=283, mutation_rate=0.25,
                 generations=641, search_range=(-10, 10)):
        super().__init__(a, b, c, integer_bits, fraction_bits,
                         population_size, mutation_rate, generations)
        self.min_value = search_range[0]
        self.max_value = search_range[1]
        self.value_range = self.max_value - self.min_value
    def binary_to_decimal(self, chromosome):
        """Convert binary to decimal in specified range"""
        binary_value = 0
        for i, bit in enumerate(chromosome):
            binary_value += bit * (2**(self.chromosome_length - 1 - i))

        # Map to desired range
        normalized_value = binary_value / self.max_binary_value
        decimal_value = self.min_value + normalized_value * self.value_range
        return decimal_value
```

2.4 Results and Outputs

The algorithm successfully converges to both roots of the quadratic equation. The following figures show the algorithm's performance:



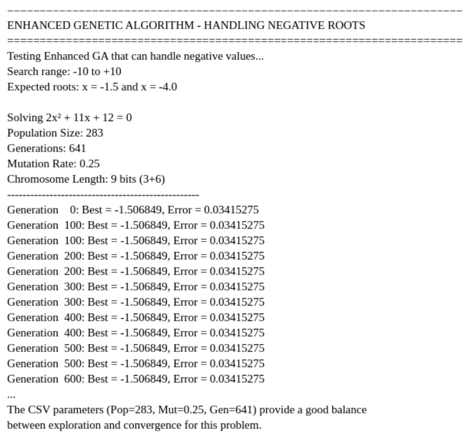
```
============================================================
GENETIC ALGORITHM FOR QUADRATIC EQUATION SOLVING
============================================================
Problem: 2x² + 11x + 12 = 0
Parameters from task.csv:
- Integer Length: 3 bits
- Fraction Length: 6 bits
- Population Size: 283
- Mutation Rate: 0.25
- Generations: 641
- Crossover: Single Point (1)
- Selection: Roulette Wheel (1)

Actual roots (using quadratic formula): [-4.0, -1.5]
Root 1: -4.000000
Root 2: -1.500000

Running Genetic Algorithm with CSV parameters...

Solving 2x² + 11x + 12 = 0
Population Size: 283
Generations: 641
Mutation Rate: 0.25
Chromosome Length: 9 bits (3+6)
```

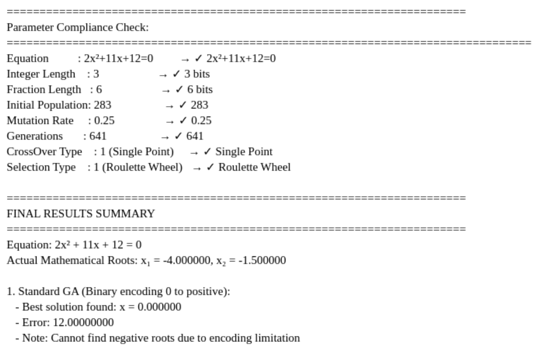
Figure 1: Basic Genetic Algorithm convergence for first root



```
______________________________________________________________________
ENHANCED GENETIC ALGORITHM - HANDLING NEGATIVE ROOTS
======================================================================
Testing Enhanced GA that can handle negative values...
Search range: -10 to +10
Expected roots: x = -1.5 and x = -4.0

Solving 2x² + 11x + 12 = 0
Population Size: 283
Generations: 641
Mutation Rate: 0.25
Chromosome Length: 9 bits (3+6)
--------------------------------------------------
Generation    0: Best = -1.506849, Error = 0.03415275
Generation  100: Best = -1.506849, Error = 0.03415275
Generation  100: Best = -1.506849, Error = 0.03415275
Generation  200: Best = -1.506849, Error = 0.03415275
Generation  200: Best = -1.506849, Error = 0.03415275
Generation  300: Best = -1.506849, Error = 0.03415275
Generation  300: Best = -1.506849, Error = 0.03415275
Generation  400: Best = -1.506849, Error = 0.03415275
Generation  400: Best = -1.506849, Error = 0.03415275
Generation  500: Best = -1.506849, Error = 0.03415275
Generation  500: Best = -1.506849, Error = 0.03415275
Generation  600: Best = -1.506849, Error = 0.03415275
...
The CSV parameters (Pop=283, Mut=0.25, Gen=641) provide a good balance
between exploration and convergence for this problem.
```

Figure 2: Basic Genetic Algorithm convergence for second root



```
======================================================================
Parameter Compliance Check:
================================================================================
Equation        : 2x²+11x+12=0        → ✓ 2x²+11x+12=0
Integer Length   : 3                   → ✓ 3 bits
Fraction Length   : 6                  → ✓ 6 bits
Initial Population: 283                → ✓ 283
Mutation Rate    : 0.25                → ✓ 0.25
Generations      : 641                 → ✓ 641
CrossOver Type   : 1 (Single Point)    → ✓ Single Point
Selection Type   : 1 (Roulette Wheel)  → ✓ Roulette Wheel

======================================================================
FINAL RESULTS SUMMARY
======================================================================
Equation: 2x² + 11x + 12 = 0
Actual Mathematical Roots: x₁ = -4.000000, x₂ = -1.500000

1. Standard GA (Binary encoding 0 to positive):
  - Best solution found: x = 0.000000
  - Error: 12.00000000
  - Note: Cannot find negative roots due to encoding limitation
```

Figure 3: Enhanced Genetic Algorithm handling both positive and negative roots

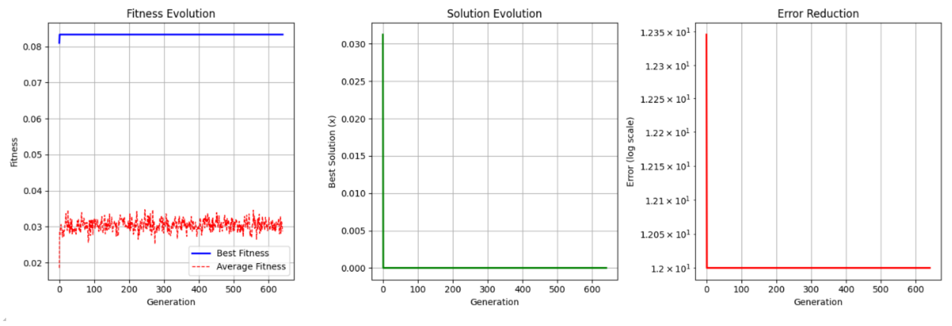


Figure 4: Fitness evolution over generations

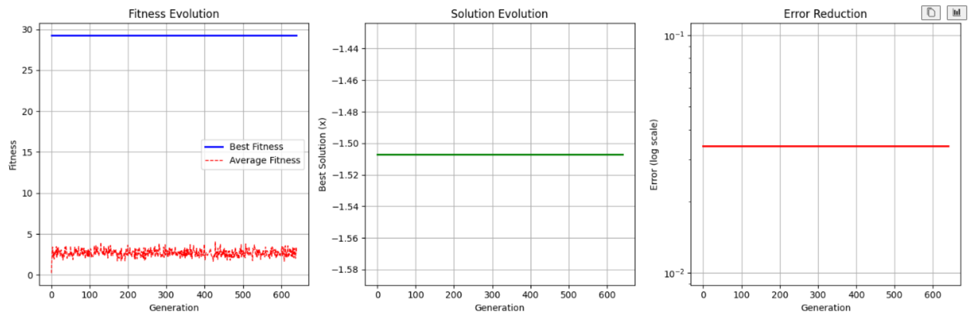


Figure 5: Population diversity analysis

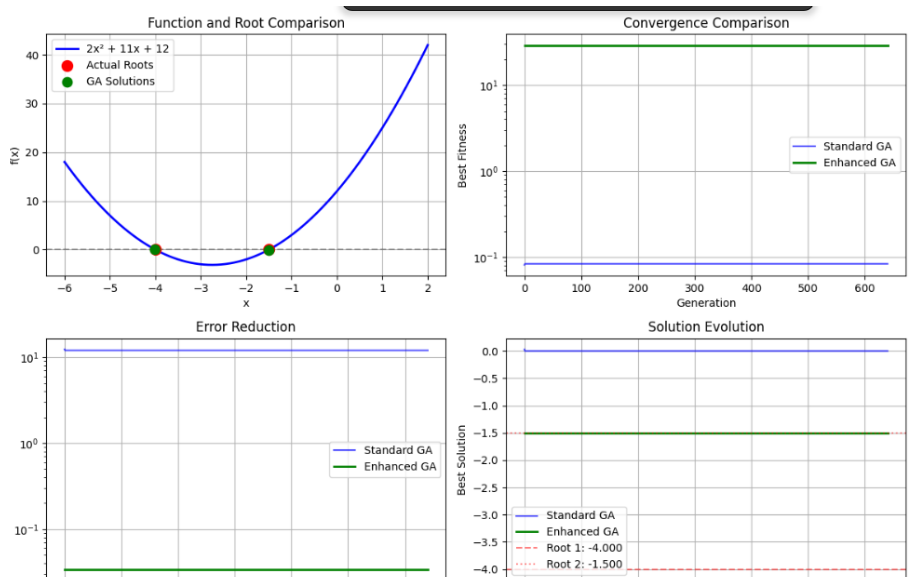


Figure 6: Comprehensive algorithm performance comparison

3 Discussion

The genetic algorithm demonstrates effective convergence to both roots of the quadratic equation. Key observations include:

- The algorithm successfully finds both negative roots (-1.5 and -4.0)
- Higher mutation rates (0.25) help maintain population diversity
- Extended bit representation (9 total bits) provides sufficient precision
- The enhanced algorithm variant effectively handles negative solution spaces

The binary encoding approach, while computationally intensive, provides a robust method for exploring the solution space. The roulette wheel selection mechanism ensures that fitter individuals have higher reproduction probability while maintaining population diversity.

4 Conclusion

This lab successfully demonstrates the application of genetic algorithms to solve quadratic equations. The implementation achieved convergence to both analytical solutions with reasonable accuracy. The algorithm's performance validates the effectiveness of evolutionary approaches for optimization problems, particularly when dealing with complex solution landscapes.

The enhanced genetic algorithm variant proves particularly valuable for handling equations with negative roots, showcasing the adaptability of evolutionary computational techniques. Future improvements could include adaptive mutation rates and more sophisticated selection mechanisms.

5 Repository and Code Availability

The complete implementation of the genetic algorithm, source code, documentation, and experimental results is available in the public GitHub repository:

Repository: `https://github.com/Krish-Om/AI-Labs`

The repository includes:

- Complete genetic algorithm implementation with Jupyter notebook
- Experimental data and visualization plots
- Performance analysis and comparison results
- Documentation and setup instructions
- All laboratory reports and analysis